Testing LLM-Generated Distributed Protocol Code

Brendan Coyne
Boston University
Computer Science
Boston, USA
bmcoyne@bu.edu

Ankush Das
Boston University
Computer Science
Boston, USA
ankushd@bu.edu

Abstract

Large Language Models (LLMs) are increasingly used for code generation, including complex distributed systems. However, their reliability in stateful and failure-prone environments remains unclear. In this paper, we present a structured evaluation framework for assessing LLM-generated implementations of distributed protocols, including Two-Phase Commit, Ring Election, and Raft. We introduce a simulation environment that injects realistic network faults such as message loss, delay, and duplication. Our results show that while LLMs perform well on simpler protocols and code completion tasks, they struggle with complex consensus algorithms and exhibit inconsistent debugging behavior. We further analyze the relationship between model size and performance, finding that parameter count alone does not predict effectiveness. These findings highlight the need for systematic testing and verification in agentic code generation systems.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Networks** → *Network protocols*; • **Software and its engineering** → *Designing software*.

Keywords: large language models, distributed systems, protocol verification, code generation, reliability

ACM Reference Format:
Brendan Coyne and Ankush Das. 2026. Testing LLM-Generated Distributed Protocol Code. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Principles of Agentic Engineering (PAgE '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3819802.3820581

1 Introduction

Large Language Models (LLMs) have rapidly grown in popularity, particularly in the realm of generating code. Modern systems such as ChatGPT and Google Gemini are used by programmers not only for simple scripting tasks, but also for more complex software development. Programmers typically use LLMs for debugging [3, 6], system design and program synthesis [1], and even complex multi-step programming workflows [5]. Recent work highlights that LLMs are increasingly being integrated into software development workflows, supporting tasks such as code generation, debugging, and testing [6]. As these models scale in size and training data, their ability to produce relevant code has improved significantly [1], raising important questions about their reliability in more complex and state-dependent programming environments.

However, there can always be some drawbacks to using LLMs for coding that could present some major risks. Generated code can appear correct and readable on the surface, while containing subtle logical errors [3]. Models are also prone to hallucinations, where outputs appear plausible but contain incorrect implementations or misuse of APIs [3, 8]. Prior work has shown that LLM-generated code may contain logical defects, performance issues, and security vulnerabilities that are not easily detected through standard testing methods [3, 8]. Such issues are especially concerning in safety-critical domains and protocols, where incorrect behavior can lead to severe consequences, such as failures in autonomous systems or medical devices. Bugs in distributed systems are typically very difficult to detect due to their dependence on timing and ordering, along with partial system failures [7]. The chances of such errors occurring makes LLM an ideal stress test for evaluating reliability.

Given these challenges and issues, there is a clear need for frameworks so LLM-generated code can be systematically tested, evaluated, and debugged correctly. Existing evaluation benchmarks for LLM-generated code primarily focus on small-scale programming tasks and do not fully capture the complexity of real-world systems [4, 8]. An ecosystem in which we would not only be able to identify correctness issues, but also provide insights into the limitations of each current model when applied to complex programming tasks would be critical into building functioning and complete code. Evaluating an LLMs performance is crucial for determining whether they can be trusted to generate high and production-quality code, particularly for systems where the correctness of said code is essential. In this work, we propose a structured approach to assessing LLM performance on distributed protocol implementations, focusing on both correctness and robustness.

The purpose of this research is to evaluate the ability of Large Language Models (LLMs) to write, interpret, and reason about protocol code. It has been studied that Large



This work is licensed under a Creative Commons Attribution 4.0 International License.
PAgE '26, Boulder, CO, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2717-7/2026/06
https://doi.org/10.1145/3819802.3820581

Language Models can handle simpler coding tasks [1, 3], but it remains less understood whether they can perform correctly on more complex and state-dependent systems such as distributed protocols [3, 5]. Throughout the research process, multiple different LLMs were tested on protocol implementations in order to evaluate their ability to generate, debug, and refine code in failure-prone environments.

Distributed protocols are foundational to computing infrastructure because they enable collections of independent nodes to coordinate reliably in the presence of concurrency, failures, and network uncertainty. Core services—such as storage systems, databases, and cloud platforms—depend on protocols for consensus, replication, and fault tolerance to maintain consistency and availability. Without such mechanisms, systems would be unable to provide strong guarantees about data integrity or service continuity under realistic operating conditions. As deployments grow in size and heterogeneity, well-designed distributed protocols become critical abstractions that encapsulate complexity, allowing engineers to reason about system behavior while ensuring robustness against failures and dynamic environments.

Testing distributed protocol implementations is essential because their correctness depends on subtle interactions between concurrency, partial failures, and nondeterministic network behavior [7]. Message delays, reordering, duplication, and loss create an enormous space of possible executions, making it difficult to systematically explore all relevant interleavings. Moreover, failures such as crashes, partitions, and recoveries can occur at arbitrary points, further amplifying the state space and exposing subtle corner cases. Reproducing bugs is often nontrivial due to the lack of determinism, and many violations manifest only under rare timing conditions.

To evaluate the performance of LLMs on distributed protocol code, we design an experimental framework consisting of two main components: a code generation component and a testing component. The code generation component is powered by LLMs, which are prompted to either generate protocol implementations from scratch or complete partially implemented code. The testing component is responsible for executing and validating the generated code within a controlled simulated environment.

At the core of this environment is a highly customizable *network* that we have implemented, which simulates real-world distributed system conditions such as message loss, duplication, and delay. This allows us to introduce realistic failure scenarios that protocols must correctly handle. The LLM-generated protocol code is prompted to directly interact with this network layer for communication, meaning that its correctness depends not only on producing syntactically valid code, but also on properly handling asynchronous communication and failure conditions. This network essentially acts as a buffer between the sender and receiver processes and is controlled to, for instance, delay, duplicate, and discard messages. We can also model process crashes by dropping all messages emanating from that process.

Table 1. LLMs used in evaluation

LLM Family	LLM Version	Model Size
OpenAI	GPT-4	1.76T
Google	Gemini 2.5 Flash	~200B
DeepSeek	DeepSeek-V3	671B
Mistral.ai	Le Chat	123B
xAI	Grok 4	314B
Claude	Claude 4.6 Sonnet	unknown

The interaction between the two components is iterative. First, an LLM generates or modifies protocol code based on a given prompt. The prompt includes the API of the network and the LLM is instructed to generate code that uses the network API for communication. The generated code is then executed within the testing environment, where its behavior is observed under both normal and faulty conditions. If errors occur, they can either be recorded as failures or fed back into the LLM (in iterative prompting settings) to evaluate its ability to debug and refine its own output. Errors include both compile-time and runtime errors, but also assertion failures that we manually add to the protocol code that check properties like consensus has been reached, or commitment is successful. This setup enables a systematic evaluation of both code generation and reasoning capabilities in complex, state-dependent systems.

We run two different kinds of tests on each protocol. One is *code generation*, where each LLM is given some starter pseudocode and prompted to generate actual protocol code. The other one is *code completion*, where LLMs are given most of the code but some small but important pieces are intentionally omitted. In this paper, we compare multiple LLMs on their ability to perform these tasks and we measure how varying parameter sizes and architectures influence performance. If the LLM is able to successfully generate the desired result, is rewarded a point, and if it error and fails, then it is not rewarded a point. All points are tracked cumulatively across all tests.

1.1 Models for Comparison

For these tests, we used 5 different LLMs as described in Table 1, all varying in popularity and in testing size. Claude does not disclose the number of parameters its models are trained on. These models were selected to represent a range of parameter scales, allowing for us to compare AI-generated answers across multiple levels of complexity. The variations in solutions help highlight how model size and design can influence structure and correctness of their answers. The model size in the final column is what each of the LLMs is trained on. The number indicates the amount of "parameters" used in training that specific LLM. Model parameters represent the learned weights and biases that determine how an LLM processes inputs and generates outputs [2]. Larger

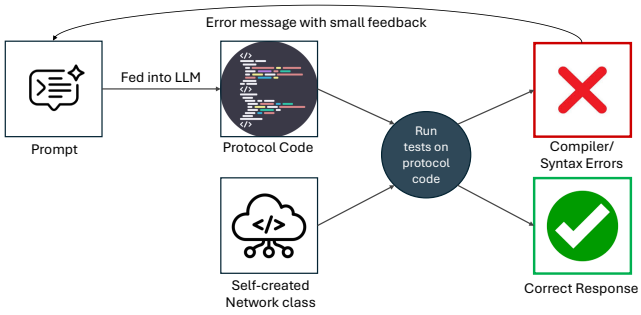


Figure 1. Our Testing Pipeline

models typically have greater capacity to capture complex patterns, although increased scale does not necessarily guarantee improved reliability.

2 Outline of Our Approach

2.1 Pipeline

Our evaluation framework, as depicted in Figure 1, follows a structured pipeline consisting of three main stages:

1. **Code Generation:** Large Language Models are prompted to generate or complete distributed protocol implementations.
2. **Execution:** The generated code is run within a simulated distributed environment.
3. **Fault Injection and Evaluation:** The system introduces failures such as message loss, delay, and duplication to evaluate robustness and correctness.

This pipeline enables systematic testing of both correctness and failure-handling behavior in LLM-generated distributed systems code.

2.2 Protocol Examples

For our testing framework, we start with 3 different protocols to test the LLMs ability of generating viable code. Our choice of protocols reflects protocols of varying complexity both in design and implementation to help test generalizability of LLM capabilities.

1. **Two Phase Commit Protocol**: This protocol used in distributed systems to ensure all participants agree to commit or abort on a transaction initiated by a coordinator. It works in two stages: a *voting* phase and then a *commitment* phase, which is controlled by the coordinator. In the first stage, the coordinator asks each participant to submit a vote (yes/no) on whether they would like to commit the transaction or not. The coordinator then collects the votes; if all participants vote 'yes', the coordinator sends a 'Commit' message to each participant in the second phase. If any participant votes 'no', the coordinator sends an 'Abort' message to each participant.
2. **Ring Leader Election**: This is a leader election algorithm where each participant is arranged in a ring and passes election messages in one direction. Each process randomly generated an ID and sends it along the ring. Each process then receives an ID, compares it with their own ID, and sends along whichever is higher. This way, the process with the highest ID wins and is recognized as the new coordinator/leader for all of the nodes.
3. **Raft Consensus**: This is a leader-based algorithm for maintaining a replicated, ordered log across a set of servers. Time is modeled as monotonically increasing terms, and each server can be in one of three roles: leader, follower, or candidate. Under normal operation, a single leader handles all client requests; followers passively replicate the leader's log. The leader appends each client command as a new log entry and sends it to followers, which include the entry along with the index and term of the preceding log entry. This log matching mechanism ensures that if a follower's log diverges, it is brought back into consistency by deleting conflicting entries and appending the leader's entries. Leader election occurs when followers stop receiving heartbeats within an election timeout. At this stage, a follower becomes a candidate, increments its term, becomes a candidate, and solicits votes via 'RequestVote' messages. Votes are granted based on term freshness and log up-to-dateness, ensuring that a newly elected leader has all committed entries. If a candidate gets a majority of votes, it becomes leader and immediately starts sending heartbeats while other followers reset their election timeouts.

```
def send(self, thread_id, msg):
    if self.failure_drop_msgs():
        return
    with self.lock:
        if thread_id in self.buffers:
            if not self.delay_msgs(thread_id, msg):
                self.buffers[thread_id].put(msg)
                if self.duplicate_msg():
                    self.buffers[thread_id].put(msg)
```

Figure 2. Simplified send function with fault injection

2.3 Network API

Our testing framework is implemented in Python. The core of the framework is implemented in a Network Class that is common across all protocols. The most important functions regarding the Network class are the *send* and *receive* functions, to send and receive messages respectively. The rest of the Network Class contains functions to configure failures.

Figure 2 contains a simplified implementation of the 'send' function that simulates unreliable communication between nodes. First, it checks if the message needs to be dropped; if true, the function exits immediately. Else it enters a `self.lock` block (to ensure thread-safe access to shared data structures).

```
def recv(self, thread_id, timeout=10):
    start = time.time()
    while time.time() - start < timeout:
        with self.lock:
            if thread_id in self.buffers and not self.buffers
                [thread_id].empty():
                return self.buffers[thread_id].get()
        time.sleep(0.1)
    return None
```

Figure 3. Simplified receive function with timeout handling

Inside the lock, it checks whether the given thread id exists in `self.buffers` (which stores message queues per thread). If the thread exists, it checks whether the message needs to be delayed. If not, the message is placed into the corresponding buffer. Then, if the message needs to be duplicated, it is inserted into the buffer a second time.

Figure 3 contains a simplified implementation of the 'recv' function, which attempts to receive a message for a given thread id within a specified timeout period. It starts by recording the current time in start, then enters a loop that continues as long as the elapsed time is less than the timeout time. Inside the loop, it enters a lock and checks whether the given thread id exists in `self.buffers` and whether the corresponding queue is not empty. If both conditions are true, it retrieves and returns the next message using the `get()` function. If no message is available, the function releases the lock and pauses briefly with a sleep. This loop repeats until either a message is received or the timeout expires, in which case the function returns *None* to indicate that no message was received in time.

Failures are configured by setting appropriate members of the class. We provide a few examples below:

- **Message Drops and Duplicates**: These are configured by setting two thresholds, namely `drop_rate` and `duplicate_rate` between 0 and 100. Once configured, when a message is sent, we generate a random number and compare it against this threshold to determine whether the message needs to be dropped or duplicated. This way, our failures are randomized, adding more non-determinism to our testing framework.
- **Message Delays:** These are incorporated by modeling two variables: `delay_rate` that controls the probability with which a message should be delayed and `delay_time` that controls the amount of delay. Sending a delayed message triggers the start of a timer, and when the timer expires, the message is inserted into the buffer.
- **Message Corruptions:** Another possible failure with networks is that messages can be corrupted, possibly changing its value. For example, a 'yes' vote in two-phase commit can be corrupted to 'no' and vice-versa. We model these kinds of failures manually since they are protocol-specific. For example, the following code models the failure described above with some probability controlled by `fr`, i.e., failure rate.

```
if __name__ == "__main__":
    num_participants = 5
    # set the failure rates

    network = Network(drop_rate, duplicate_rate,
    delay_rate)
    participants = [Participant(i, network, sr, fr)
    for i in range(num_participants)]
    coordinator = Coordinator(num_participants,
         network)

    coordinator.start()
    for p in participants:
        p.start()

    coordinator.join()
    for p in participants:
        p.join()
```

Figure 4. Bootstrap Code for Two-Phase Commit (part of the LLM prompt)

```
        for i in range(self.num_participants):
            msg = self.network.recv("coordinator")
            if msg is None:
                print("Coordinator: Missing a vote due
                     to message drop! Aborting
                     transaction.")
                decision = "ABORT"
                break
            participant_id, vote = msg

...

            if vote == "NO":
                decision = "ABORT"
```

Figure 5. Pieces of our self-created Coordinator class

```
def failure_switch_votes(self, vote,
participant_id, fr):
    if random.randint(1, 100) <= fr:
        original_vote = vote
        if vote == "YES":
            vote = "NO"
        else:
            vote = "YES"
    return vote
```

2.4 Tests

We evaluated LLMs on two different kinds of tests:

- **Code Generation:** Each LLM is given the complete implementation of the network interface, and is instructed to generate the necessary classes for each protocol party. The LLM is also given a 'main' function that bootstraps the execution, so that it can focus on implementing the main protocol components. Figure 4 provides an example of bootstrapping two-phase commit with 5 participants. It initializes the failure rates, creates the relevant parties and calls the `start()` function on each party that initiates its code.

```
Tests:

(sr = 100): Passed__
(sr = 100, fr = 50): Passed
(sr = 100, dr = 50): Passed
(sr = 100, dupr = 50): Passed
(sr = 100, delay_rate = 50): Passed

Grade: 5/5
```

Figure 6. Example grading system

- **Code Completion:** Each LLM is give the protocol code for one of the parties involved in the protocol with a crucial piece missing. For instance, for the two-phase commit, we commented out the initialization code for the coordinator where it sends the 'Prepare' message to each participant. Another example is where we "forget" to count the votes of the participants to make the decision to either commit or abort.

Code Generation. For this task, our evaluation follows a structured workflow. A prompt is first given to the LLM to generate a specific party class. The generated code is then incorporated into our existing framework and executed within the simulation environment. If the LLM code produces errors, then the errors, along with textual feedback, is fed back into the same LLM conversation, allowing the model to refine its output. This process continues until a correct implementation is produced or the attempt is deemed unsuccessful.

To further analyze LLM behavior and robustness, we evaluate two prompting strategies:

- **Cleared Cache:** In this setting, each prompt is issued independently, with no prior interaction history. The LLM must generate a complete implementation using only the initial prompt, without access to previous errors or corrections. This evaluates the model's ability to produce correct code in a one-shot setting.
- **Non-Cleared Cache:** In this setting, we maintain the conversational context and iteratively provide the LLM with error messages and concise feedback after each failed attempt. This allows the model to learn from its mistakes and progressively refine its implementation. While this approach often reduces the number of attempts to reach a correct solution, it also exposes inconsistencies in model behavior, as LLMs may occasionally reintroduce previously corrected errors.

If the LLM passes all the tests, then it is graded with full points. More generally, the score depends on the number of passed tests. Figure 6 contains a snippet describing our grading system. We execute the protocol first without any failures, and then each successive test contains *one* failure scenario (e.g., delay, drop, duplicate) and checks if assertions are passing with that failure. Ultimately, the grade is computed simply based on the number of tests passed.

Code Completion. For this task, a small portion of the existing protocol implementation is removed and replaced with a placeholder comment '`// ADD CODE`'. The LLM is provided with the remaining code and must infer the code of the missing section. Unlike code generation, this task is evaluated in a single pass/fail setting. The LLM generates the missing code once, after which the completed implementation is reintegrated into the system and tested using the same evaluation framework.

As Section 3 with demonstrate, LLMs consistently performed well when asked to produce correct implementations in a single attempt. This suggests that models are more reliable when completing localized logic within an existing structure rather than generating full protocol implementations from scratch. This also suggests and confirms our intuition that the most effective way to use LLMs is to break down the code generation task into smaller subtasks, and then use LLMs selectively to complete each subtask. Not only does this result in more modular and compositional code, but it also makes it easier for the user to control the generation process such that it is more readable and understandable.

For this task, the LLMs are graded off the first trial only and based on whether all tests passed or failed. Unlike Code Generation, where the points depended on the number of passing tests, with Code Completion, the test is graded directly as a Pass/Fail based on whether or not the protocol passed all the tests.

3 Results

As a first evaluation criteria, we had an iterative back-and-forth conversation with an LLM to gradually arrive at a correct implementation (one that passes all the tests). The conversation starts with a very simple high-level description of the protocol without even providing the LLM with the network API. So, obviously the code produced would not compile in the first attempt due to incorrect usage of the network API. Our subsequent prompts would simply include the errors obtained from compiling the generated protocol code. Once the LLM started using the network API correctly, it would continue to make runtime errors by not handling failures correctly, e.g. message delays or drops. Hence, our subsequent prompts would include the name and textual description of the failing tests. Figure 7 shows an example error obtained by using the network API incorrectly and the subsequent prompt provided to the LLM.

Table 2. Number of Prompts for 2PC Conversation

LLM	Participant	Coordinator
GPT-4	14	11
Gemini 2.5	16	15
DeepSeek-V3	18	21
Le Chat	18	20
Grok 4	16	15

```
TypeError: send() takes 3 positional arguments but
    4 were given
Coordinator: Committing transaction
Network: Timeout waiting for message at 3
Participant 3: Failure due to missing decision
    message
Network: Timeout waiting for message at 1
Participant 1: Failure due to missing decision
    message
'''
Example Prompt:
The send function should only contain the
    participant number and its decision.
Ex: for p in range(self.num participants): self.
    network.send(p, decision)
'''
```

Figure 7. Example runtime error and prompt

Table 2 describes the results of LLMs on back-and-forth conversation for the two-phase commit protocol, particularly how many prompts were required to generate the participant and coordinator code respectively. Noteworthy here is that passing of tests was *not monotonic*. In other words, as the LLM tried to self-correct the generated code, often it would start failing tests that were previously passing. This led to extra iterations until the generated code was passing all tests.

Tables 3 and 4 describe the performance of LLMs on the two-phase commit, ring leader election, and Raft consensus protocols. Each column represents a task and shows the number of tests passing for that task. In Table 3, the tasks are as follows:

- **Part**: The LLM is given the Network and Coordinator code, and asked to generate the Participant code;
- **Coord**: The LLM is given the Network and Participant code, and asked to generate the Coordinator code;
- **P/C**: The LLM is given the Network code, and asked to generate both the Participant and Coordinator code;
- **Compl**: Complete a missing piece in Coordinator class (vote counting)

The first 3 tasks are Code Generation tasks and the last one is Code Completion. Almost all LLMs scored perfect on the Part and Coord tests, but struggled on the P/C test. And all LLMs score perfectly on the Code Completion tasks. In Table 4, RingN and RaftN are Code Generation tasks for each protocol, and RingC and RaftC are Code Completion tasks. The LLMs performed moderately well for RingN and RingC, but struggled on the Raft tests, especially RaftN.

Table 3. LLM performance on Two-Phase Commit tasks

LLM	Part	Coord	P/C	Compl	Total
GPT-4	5/5	5/5	5/5	6/6	21/21
Gemini 2.5	4/5	4/5	0/5	6/6	14/21
DeepSeek-V3	5/5	4/5	4/5	6/6	19/21
Le Chat	5/5	4/5	0/5	6/6	15/21
Grok 4	5/5	0/5	0/5	6/6	11/21
4.6 Sonnet	4/5	5/5	5/5	6/6	20/21

Part = Participant, Coord = Coordinator, P/C = Participant/Coordinator, Compl = Code Completion

Table 4. LLM performance on Ring Election and Raft tasks

LLM	RingN	RingC	RaftN	RaftC	Total
GPT-4	4/4	5/6	0/4	5/5	14/19
Gemini 2.5	0/4	6/6	0/4	5/5	11/19
DeepSeek-V3	3/4	5/6	2/4	3/5	13/19
Le Chat	3/4	4/6	0/4	2/5	9/19
Grok 4	3/4	5/6	0/4	3/5	11/19
4.6 Sonnet	4/4	5/6	0/4	5/5	14/19

RingN = Ring Node Class; RingC = Ring Code Completion; RaftN = Raft Node Class; RaftC = Raft Code Completion

Table 5. Model size and performance ranking

Rank	Family	Model	Params	Score
1	OpenAI	GPT-4	1.76T	35/40
2	Claude	4.6 Sonnet	unknown	34/40
3	DeepSeek	DeepSeek-V3	671B	32/40
4	Mistral.ai	Le Chat	123B	25/40
5	Google	Gemini 2.5 Flash	~200B	25/40
5	xAI	Grok 4	314B	22/40

Based on Table 5, ChatGPT, Claude, and DeepSeek had the strongest performances, achieving much higher scores across most task categories, particularly in the Two-Phase Commit scenarios. ChatGPT achieved a perfect score in all Two-Phase Commit and Ring Election tasks but failed entirely on Raft. DeepSeek showed more balanced performance by earning partial credit on the Raft test. Other LLMs, such as Gemini and xAI, struggled, especially on 2PC (PC) and Raft tasks, where both models scored zero.

When comparing model size to performance, larger parameter counts do not necessarily correlate with better results. Although GPT-4 (1.76 trillion parameters) achieved the highest score, smaller models such as Mistral (123B) performed much better than expected and even outperformed larger models like Gemini (200B) and Grok 4 (314B). Table 6 highlights an important distinction between performance and parameter count. While GPT-4 achieved the highest overall score, it does so with a significantly larger parameter count, meaning that it has relatively lower efficiency compared to smaller models. In contrast, Mistral's Le Chat delivered strong performance with far fewer parameters, indicating more efficient use of model capacity. This suggests

Table 6. Model efficiency (score per parameter scale)

Model	Params (B)	Score	Score / 100B
GPT-4	1760	35/40	1.99
DeepSeek-V3	671	32/40	4.77
Le Chat	123	25/40	20.33
Gemini 2.5	200	25/40	12.50
Grok 4	314	22/40	7.01

that model architecture, training data, and optimization may play a more significant role than parameter count alone.

4 Conclusion

In this work, we presented a structured framework for evaluating the ability of Large Language Models (LLMs) to generate and debug distributed protocols. By testing multiple models across protocols of varying complexity, including Two-Phase Commit, Ring Election, and Raft, we were able to assess not only the code correctness of each model, but also the presence of any network faults such as message loss, delay, and duplication.

Our findings demonstrate that while LLMs perform well on simpler protocols such as Two-Phase Commit and well-structured tasks like code completion, their struggle is much more significant with more complex systems such as Raft. Additionally, the results show that the model size and parameter count is not a reliable predictor of the performance. Smaller models, such as Mistral's Le Chat, exhibited a stronger efficiency relative to their parameter count, while larger models did not consistently outperform Le Chat and Gemini in the more complex scenarios.

References

[1] Jacob Austin et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[2] Ivan Belcic and Cole Stryker. n.d.. What Are LLM Parameters? https://www.ibm.com/think/topics/llm-parameters. IBM Think.

[3] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Dan Hendrycks et al. 2021. Measuring Coding Challenge Competence with APPS. *NeurIPS* (2021).

[5] Erik Nijkamp et al. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[6] Gabrielle O'Brien. 2025. How Scientists Use Large Language Models to Program. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '25)*. doi:10.1145/3706598.3713668

[7] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 28. doi:10.1145/3158116

[8] Andy K. Zhang et al. 2025. CyBench: A Framework for Evaluating Cybersecurity Capabilities and Risks of Language Models. *arXiv preprint arXiv:2408.08926v4* (2025).

Received 2026-05-05; accepted 2026-05-19