Distilling distributed system specifications with Large Language Models

Mike He
Princeton University

Zhendong Ang
National University of Singapore

Ankush Desai
Amazon Web Services

Aarti Gupta
Princeton University

Abstract

Formal specifications are critical for reasoning about the correctness of distributed systems and for enabling runtime monitoring. While recent advances have focused on automatically *learning* such specifications, the challenge of *distilling* meaningful and non-trivial specifications from a large, noisy set of candidates remains largely unaddressed. In this position paper, we propose an approach for specification distillation: identifying the most critical specifications that merit the overall system correctness. We design a four-metric rating framework that quantifies importance of system specifications to the underlying distributed system and leverages the reasoning capabilities of Large Language Models to rank learned specifications following our rating framework. We conducted preliminary experiment on distilling specifications learned by PInfer for 11 open-source and 3 proprietary benchmarks to demonstrate effectiveness of our approach.

1 Introduction

Modern distributed systems operate at massive scales, coordinating thousands of nodes across global infrastructures while maintaining various consistency guarantees. The correctness of these systems hinges on their adherence to *formal* specifications that define desired system behaviors and exclude behaviors that lead to errors. Yet, through our extensive experience using the P modeling framework to guarantee industrial-strength distributed services at AnonCompany[1], the traditional approach of manually developing these specifications has become a bottleneck in the development process, consuming significant engineering effort. Many prior efforts attempted to address this challenge by analyzing execution traces or actively monitoring system states to automatically *learn* specifications [3, 6, 14].

While these techniques alleviate the developers' burden of formalizing correctness, their applicability for large-scale, complex distributed systems are hindered by learning an unreasonably large number of specifications. The data-driven approaches adopted by these frameworks often rely on searching and learning with a grammar [3, 6, 14] that creates an exponentially large search space including many trivial specifications. For instance, without suitable user intervention, Dinv [6] learns a million of properties for Raft [11]. A recent work, PInfer [3], alleviates this issue by focusing on a more restricted grammar and using logic-based pruning procedures to remove redundant specifications subsumed by others. It additionally utilizes a model-checking-based fuzzer [4] to falsify the learned specifications with a P [5] model. Despite these efforts, PInfer still learns hundreds of specifications for complex systems. Working with these tools then transforms the burden from formalizing correctness to identifying "critical" correctness specifications that benefit downstream tasks (e.g., verification) from large sets of learned specifications. We refer to this challenge as the *specification distillation* problem.

1.1 Challenges and Our solution

We propose to address the specification distillation problem leveraging the reasoning capabilities of Large Language Models (LLMs). Specifically, we identify two key challenges in this problem:

Challenge (C1): "Criticalness" of specifications does not have a formal definition. Even the informal statements about criticalness vary across contexts. For instance, for consensus protocols such as Paxos [9], the uniqueness of decisions is critical, whereas various database systems concern more about atomicity and consistency.

Challenge (C2): Distilling specifications requires expertise of the system design and its intended behaviors. This can be a burden for the user or require specialized analysis tools.

To address **(C1)**, we propose a *four-metric rating framework* that measures criticalness of specifications along their generalizability, criticality, distinguishability, and visibility. These metrics capture different aspects of specification importance, from universal applicability to user-facing impact. Note that these metrics are extensible and can be customized. To address **(C2)**, we leverage LLM agents to analyze system implementations and apply our rating framework to *rank* specifications. We conducted preliminary experiments on distilling specifications learned by PInfer for 11 open-source and 3 proprietary benchmarks. Our results show that using our technique with a state-of-the-art LLM, all specifications

[1] Name elided for anonymity

identified by developers are found in top-10 distilled candidates in 8 out of 14 benchmarks. Additionally, we identify critical properties of proprietary protocols overlooked by the developers.

2 Background

2.1 P language and modeling framework

P [4, 5] is a state-machine-based programming language for formal modeling and analysis of distributed systems. A P program comprises state machines communicating asynchronously with each other using *events* comprised of typed data values. The machines run concurrently, receiving and sending events and updating the local states.

Example: Simple Client-Server Protocol. To illustrate the main concepts in a P model, consider a simple client-server protocol, where the client sends a request to the server, and then the server sends back a response. The P model is shown in Figure 1. In this model, the `Request` and `Response` declarations (lines 2-3) specify the names and payload types of the events. For instance, the payload of `Request` has a reference to the client and a request ID. Each state machine (lines 5, 23) has associated states (lines 8, 16, 24) and local variables (lines 6, 7). A state may have an entry function (line 9) executed upon entry. After executing the entry function, the machine tries to dequeue from its event buffer and execute the corresponding *event handler*. For instance, in `WaitResponse` state, the machine has a handler for `Response` (line 17). Machines can communicate using `send` (line 12, 26) and change its state using `goto` (line 13).

An explorer tool PChecker [4] can execute the model and record *event traces* capturing all message exchanges between machines. These event traces serve as input to PInfer for specification learning.

2.2 PInfer: learning specifications from event traces

PInfer [3] is a framework that learns specifications of distributed systems from event traces. Given event traces recorded by PChecker, PInfer automatically discovers specifications encoded as first-order logic formulas over events that characterize system behavior. Specifically, PInfer uses the following formula template to guide specification learning:

$$\phi : (\forall \vec{e}_i)^+ . G(\vec{e}_i) \rightarrow (\exists \vec{e}_j)^* . W(\vec{e}_i, \vec{e}_j) \wedge H(\vec{e}_i, \vec{e}_j) \quad (1)$$

where G are *guards* stating control conditions of occurrence of events $\vec{e}_i$, W are *witnesses* that express *existence* of a certain event $\vec{e}_j$, and H are *hypotheses* that hold under G and W.

Specification Learning Example. For the example above, PInfer learns specifications such as the following. We denote a *Request* as $\mathcal{R}$ and a *Response* as $\mathcal{S}$.

$$\forall e_0 : \mathcal{S}. \exists e_1 : \mathcal{R}. e_1 \prec e_0 \wedge e_0.reqId = e_1.reqId \quad (2)$$

$$\forall e_0, e_1 : \mathcal{S}. e_0.reqId = e_1.reqId \rightarrow e_0 = e_1 \quad (3)$$

$$\forall e_0 : \mathcal{R}, e_1 : \mathcal{S}. e_0.reqId = e_1.reqId \rightarrow e_0 \prec e_1 \quad (4)$$

Figure 1. Simple Client-Server protocol in P

```
1  # Event declarations with payload types
2  event Request: (clt: Client, reqId: int);
3  event Response: (reqId: int);
4  # Client state machine
5  machine Client {
6    var server: Server;
7    var rid: int;
8    start state SendRequest {
9      entry (srv: Server) {
10       server = srv;
11       rid = randomId();
12       send server, Request, (clt=this, reqId=rid);
13       goto WaitResponse;
14     }
15   }
16   state WaitResponse {
17     on Response do (payload: (reqId: int)) {
18       assert(payload.reqId == this.rid);
19     }
20   }
21 }
22 # Server state machine
23 machine Server {
24   start state Serving {
25     on Request do (req: (clt: Client, reqId: int)) {
26       send req.clt, Response, (reqId=req.reqId);
27     }
28   }
29 }
```

These specifications capture the following properties:
Eqn 2: Every response corresponds to a prior request, where $\prec$ denotes the traditional happens-before [8] relation.
Eqn 3: Responses are unique per request ID
Eqn 4: Requests always happen before responses

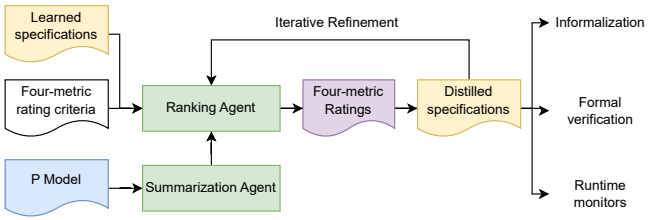
For more complex protocols, the formula template (Eqn 1) can create larger search spaces, which may lead PInfer to learn hundreds of specifications from event traces. We aim to distill these learned specifications to a smaller subset.

3 Distillation with LLMs

Distillation problem has mainly two challenges: first, the importance of specifications is not quantitatively defined; second, understanding the specifications requires expertise about the protocol design. To address challenge 1, we introduce a *four-metric rating framework* that evaluates the contribution of each specification to system correctness. For challenge 2, we design a distillation workflow that incorporates our rating framework and leverages the LLMs to *rank* specifications based on the ratings.

Figure 2 illustrates our workflow. We first use a Summarization agent to analyze the P model and extract event, state machine definitions, and event flows between machines. Then, we provide the Ranking agent with the learned specifications, a detailed explanation of our rating framework with illustrative examples, and the analysis of the P model to set up the domain context. The Ranking agent is then prompted to evaluate the specifications following our rating framework and rank them based on the ratings. The set of specifications can be distilled through an iterative process until satisfying user-provided constraints (e.g., number of specifications, minimum or average rating threshold).

Figure 2. LLM-based specification distillation workflow.



3.1 Four-Metric Rating Framework

Our rating framework evaluates each specification ϕ along four dimensions, producing scores in the range $[0.0, 1.0]$ for each metric. The four metrics are derived from intuitions that good specifications should be *generalizable* across different configurable parameters, capturing *correctness* properties and rejecting *undesired* behaviors of the system. For example, different system configurations may involve different number of nodes, network topologies, or workload patterns (e.g., read-heavy, write-heavy, etc.). Additionally, the specifications will be more debuggable if they directly affect outputs that are visible to the end user. Table 1 provides some rating examples for each metric on specifications learned for the Two-Phase Commit [7] protocol.

Generalization Score $G(\phi)$. The generalization score $G(\phi)$ reflects the likelihood that a specification represents a system *invariant* that holds across all valid executions, independent of system configuration or implementation. Generalization score distinguishes between specifications that capture general correctness properties versus those overfitting to specific configurations. Examples are shown in Table 1 ($G(\phi)$ row).

Criticality Score $C(\phi)$. The criticality score $C(\phi)$ evaluates the severity of consequences when a specification is violated, measuring the *blast radius* of potential failures and their impact on system correctness and recoverability. This metric captures whether the violation of a specification would cascade into system-wide failures or remain localized. Specifications with high criticality scores protect against violations that result in unrecoverable system states or compromise essential safety properties. The $C(\phi)$ row of Table 1 shows examples for this metric.

Distinguishability Score $D(\phi)$. The distinguishability score $D(\phi)$ measures how effectively a specification serves as a separator that differentiates between correct and incorrect system behaviors. Specifications with high distinguishability should be *strong* enough to exclude erroneous behaviors, while weak enough to apply to all correct behaviors. Examples are shown in the $D(\phi)$ row of Table 1.

Visibility Score $V(\phi)$. The visibility score $V(\phi)$ measures how directly a specification's properties are observable by end users or system operators. We focus on specifications defined over *events* observed during system execution, where events can be either observable to the user of the system (e.g., responses) or internal to the system (e.g., cluster reconfiguration). High visibility score indicates that the violation of the specification is "closer" to the interface between the system and its users, making it easier to detect and debug. Some examples are shown in the $V(\phi)$ row of Table 1.

Figure 3. Preliminary results: columns show the numbers of specifications in the target set included after distillation to top-k candidates. The total number of specifications learned by PInfer is shown along with benchmark names. The top 11 rows are benchmarks with open-source protocols.

	k=10	k=20	k=30	k=40	k=50
2Pc (46)	2/2	2/2	2/2	2/2	2/2
Chainreplication (33)	3/5	5/5	5/5	5/5	5/5
Raft (56)	5/5	5/5	5/5	5/5	5/5
Consensus (28)	1/1	1/1	1/1	1/1	1/1
Distributed Lock (76)	1/1	1/1	1/1	1/1	1/1
Firewall (40)	1/1	1/1	1/1	1/1	1/1
Lockserver (35)	1/1	1/1	1/1	1/1	1/1
Paxos (46)	1/2	2/2	2/2	2/2	2/2
Ring Leader (27)	1/1	1/1	1/1	1/1	1/1
Sharded Kv (19)	1/1	1/1	1/1	1/1	1/1
Vertical Paxos (94)	1/2	2/2	2/2	2/2	2/2
Globalclock (37)	2/3	3/3	3/3	3/3	3/3
Dbleaderelection (47)	3/5	3/5	3/5	4/5	5/5
Mvcc-2Pc (241)	2/10	3/10	6/10	7/10	8/10
Spec Included %	**62.5%**	**77.5%**	**85.0%**	**90.0%**	**95.0%**
Pruned %	**83.0%**	**66.2%**	**51.0%**	**39.5%**	**32.4%**

Overall Rating and Ranking. Given the metric scores $G(\phi)$, $C(\phi)$, $D(\phi)$, and $V(\phi)$ for a specification ϕ, the overall score $S(\phi)$ is computed as:

$$S(\phi) = \lambda_1 \cdot \sqrt{G(\phi) \cdot C(\phi)} + \lambda_2 \cdot D(\phi) + \lambda_3 \cdot V(\phi)$$

where λ_i are hyper-parameters such that $\sum_i \lambda_i = 1$. In particular, $\sqrt{G(\phi) \cdot C(\phi)}$ is a *quality* term ensuring that specifications with high generalization but low criticality (vice versa) do not dominate the ranking. The distinguishability and visibility scores are added linearly, allowing them to contribute to the overall score without overpowering the quality term (controlled via λ_2, λ_3). The Ranking agent is prompted to follow the rating framework and provide the top-k rated specifications. The k value can be decreased iteratively until the user is satisfied with the final result.

Table 1. Example ratings for each metric. each shows the specification formula on the first line. The rating and justification are shown on the second line. $eWriteSuccess$ and $eWriteFailure$ are events sent back to the users.

Metrics	Rating examples in System Prompt
$G(\phi)$	$\forall e_0 : eAbort. \forall e_1 : eCommit. e_0.id \neq e_1.id$ 1: Universally true - abort and commit never occur for the same transaction in ANY execution.
	$\forall e_0 : eCommit. \exists e_1 : ePrepareSuccess. e_1 \prec e_0 \wedge e_0.id = e_1.id \wedge e_0.voter \neq e_1.voter$ 0.5: Generally true but may not hold in single-node deployments.
	$\forall e_0 : ePrepareReq. \exists e_1 : ePrepareFailure. e_0 \prec e_1$ 0: Not generalizable to traces with no failed transaction.
$C(\phi)$	$\forall e_0 : eAbort. \forall e_1 : eCommit. e_0.id \neq e_1.id$ 1: Violation leads to data corruption requiring manual rollback.
	$\forall e_0 : eCommit. \forall e_1 : eCommit. e_0.id = e_1.id \rightarrow e_0 = e_1$ 0.5: Moderately critical - duplicate commits cause inconsistency but may be detectable and recoverable.
	$\forall e_0 : ePrepareSuccess. \exists e_1 : ePrepareReq. e_1 \prec e_0$ 0: Non-critical: the opposite ($e_0 \prec e_1$) may also be true for multiple transactions.
$D(\phi)$	$\forall e_0 : eCommit. \exists_n e_1 : ePrepareSuccess. e_1 \prec e_0 \wedge e_0.id = e_1.id$ 1: Rejects all executions with incomplete prepare successes.
	$\forall e_0 : eCommit. \exists e_1 : ePrepareReq. e_1 \prec e_0 \wedge e_0.id = e_1.id$ 0.6: Catches commits without prepare requests but misses commits with failed prepares.
	$\forall e_0 : eAbort. \exists e_1 : ePrepareFailure. e_0.id = e_1.id$ 0.3: Not strong ePrepareFailureugh to reject executions where the prepare failure happens after the eAbort.
$V(\phi)$	$\forall e_0 : eWriteSuccess. \forall e_1 : eWriteFailure. e_0.id \neq e_1.id$ 1: Operator immediately notices contradictory transaction outcomes.
	$\forall e_0 : eAbort. \exists e_1 : eWriteFailure. e_0.id = e_1.id$ 0.5: Abort and Commit can trigger user-visible events.
	$\forall e_0 : eCommit. \exists e_1 : ePrepareReq. e_1 \prec e_0 \wedge e_0.id = e_1.id$ 0.1: Internal protocol ordering not directly visible to application users.

4 Preliminary Experiments

We have conducted preliminary experiments to evaluate the effectiveness of our approach. We use the state-of-the-art Claude Sonnet 4 [1] for creating Summarization and Ranking agents. We applied our approach to 11 well-known open-source protocols (including Raft [11], Paxos [9], and Chain Replication [13], etc.) and 3 proprietary protocols written in P [4] language. We construct the target specification set (43 in total) identified by prior literature [2, 9–11] for open-source protocols and specifications composed by development teams for proprietary protocols. We use PInfer [3] to learn specifications as inputs to the distillation process. In this experiment, we set the weights to $\lambda_1 = 0.8, \lambda_2 = 0.1$ and $\lambda_3 = 0.1$ to favor specifications with higher quality terms.

Our results are shown in Figure 3. Notably, *all* specifications of 8 of the 14 benchmarks are identified after distilling to only 10 top-rated candidates. The last two columns shows the percentage of target specifications included and specifications pruned. We also analyzed specifications that were not in our target set but ranked higher by the LLM. We found that these specifications fall into two main categories: (1) specifications that can be expressed by relations between other events, and (2) critical specifications missing from the target set. The first category is less interesting since they can be derived from the known specifications with domain expertise. On the other hand, the second category is more valuable, as it highlights critical specifications that developers may have overlooked. For instance, the top-ranked specification for the proprietary MVCC-2PC protocol ensures that users do not observe inconsistent transaction status between the leader server and shard servers. While this specification was not in the target set identified by the protocol developers, it represents a crucial correctness property for the protocol. This demonstrates a key benefit of our approach: it can help developers discover important specifications that they may have missed during development. Upon validation of such specifications, developers can strengthen protocol specification sets with these properties.

5 Future work

Improving accuracy with Reinforcement Learning. Currently, we hard-code the weights (λ_i) to compute overall ratings. In future work, we would like to consider using Reinforcement Learning or Reinforcement Learning with

Human Feedback [12] to dynamically adjust the weights to better adapt to different protocols.

Towards an automated verification framework. We plan to incorporate our workflow into a broader automated verification framework. In this framework, given a P model and a set of traces, PInfer learns automatically a set of specifications. Then, the distillation process chooses critical specifications that should be further verified with a downstream verifier (e.g., PVerifier [4, 10]). Our rating framework is general and extensible, and we can incorporate additional metrics such as *inductiveness* of specifications to facilitate the verification process.

References

[1] Claude Sonnet 4 — anthropic.com. https://www.anthropic.com/claude/sonnet. [Accessed 07-07-2025].

[2] GitHub - GLaDOS-Michigan/I4: The code base for the I4 prototype, as described in the SOSP '19 paper "I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols" — github.com. https://github.com/GLaDOS-Michigan/I4. [Accessed 11-04-2025].

[3] GitHub - p-org/P at experimental/pinfer — github.com. https://github.com/p-org/P/tree/experimental/pinfer. [Accessed 07-07-2025].

[4] GitHub - p-org/P: The P programming language. — github.com. https://github.com/p-org/P. [Accessed 18-03-2025].

[5] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 321–332, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1149–1159, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Jim Gray. *The transaction concept: virtues and limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[8] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[9] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[10] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[11] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[12] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[13] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 7, USA, 2004. USENIX Association.

[14] Yuan Xia, Deepayan Sur, Aabha Shailesh Pingle, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Srivatsan Ravi. Discovering likely invariants for distributed systems through runtime monitoring and learning. In Krishna Shankaranarayanan, Sriram Sankaranarayanan, and Ashutosh Trivedi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 3–25, Cham, 2025. Springer Nature Switzerland.