Toward Reproducing Network Research Results Using Large Language Models

Qiao Xiang*, Yuling Lin*, Mingjun Fang*, Bang Huang*, Siyong Huang*,
Ridi Wen*, Franck Le†, Linghe Kong◇, Jiwu Shu*°
*Xiamen University, †IBM Research, ◇ Shanghai Jiao Tong University, °Minjiang University

ABSTRACT

Reproducing research results is important for the networking community. The current best practice typically resorts to: (1) looking for publicly available prototypes; (2) contacting the authors to get a private prototype; or (3) manually implementing a prototype following the description of the publication. However, most published network research does not have public prototypes and private ones are hard to get. As such, most reproducing efforts are spent on manual implementation based on the publications, which is both time and labor consuming and error-prone. In this paper, we boldly propose reproducing network research results using the emerging large language models (LLMs). We first prove its feasibility with a small-scale experiment, in which four students with essential networking knowledge each reproduces a different networking system published in prominent conferences and journals by prompt engineering ChatGPT. We report our observations and lessons and discuss future open research questions of this proposal.

CCS CONCEPTS

• **Networks** → **Network performance evaluation**; • **Computing methodologies** → **Machine learning**;

KEYWORDS

Networking systems, Large language models

ACM Reference Format:

Qiao Xiang, Yuling Lin, Mingjun Fang, Bang Huang, Siyong Huang, Ridi Wen, Franck Le, Linghe Kong, Jiwu Shu. 2023. Toward Reproducing Network Research Results Using Large Language Models. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), November 28–29, 2023, Cambridge, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626111.3628189

1 INTRODUCTION

Reproducing network research results have both significant education and research values. For education, it completes students' learning process on computer networks with lecture attendance and textbook reading, in accordance with the usual process of science study worldwide, *e.g.*, educators at Stanford University assign reproduction projects in their networking classes [37]. For research, it ensures the results are accurate and trustworthy, gives researchers a hands-on opportunity to understand the pros and cons of these results, and motivates more innovations, *e.g.*, IMC 2023 introduces a replicability track for submissions that aims to reproduce or replicate results previously published at IMC [11].

The best practice for people to reproduce a published network research typically involves one of three approaches. First, rerun a publicly available prototype provided by the authors (*e.g.*, [38]) or other people who implement it based on the publication (*e.g.*, [43]). Second, if no public prototype is available, people may contact the authors to ask for a private prototype. Third, if no prototype is available, people need to manually implement one following the publication.

The best practice to reproduce network research results has limitations. All three approaches in the best practice are limited for various reasons. First, not much published research comes with a publicly available prototype. Our study shows that even in prominent networking conferences such as SIGCOMM and NSDI, only a small number of papers provide publicly available prototypes from the authors. From 2013 to 2022, only 32% and 29% of papers in SIGCOMM and NSDI, respectively, provide open-source prototypes. Although some non-authors implement prototypes and release them to the public [8], the number of such prototypes is even smaller. Second, the authors sometimes are reluctant to share a private prototype for various reasons (*e.g.*, patent filing, commercial product, policy, and security).

As such, without ready-made prototypes, the dominant way for people to reproduce the results of a published networking paper is to manually implement its proposed design. Although this "getting-hands-dirty" approach provides people precious experience in understanding the details of the paper, in particular the pros and cons of the proposed design, the whole process is both time and labor consuming and error-prone. In the long run, it is unsustainable because network research results are becoming more and more complex. If people are spending more time trying to reproduce the published results, they will have less time for critical thinking and innovation. One may think this is not a unique issue for the networking community, but a prevalent one for the whole computer science discipline. However, the situation is more severe for networking research. For example, based on our private conversations with several research groups and

HotNets '23, November 28–29, 2023, Cambridge, MA, USA

ACM ISBN 979-8-4007-0415-4/23/11…$15.00
https://doi.org/10.1145/3626111.3628189

results from a reproduction class [37], it may take a fresh graduate student one week to reproduce a machine learning paper by manual implementation, but one or two months to reproduce a networking paper.

Proposal: reproducing network research results using large language models (LLMs). In this position paper, we make a bold proposal to reproduce network research results by prompt engineering the emerging LLMs. Such a proposal, if built successfully, can benefit the networking community from multiple perspectives, including (1) substantially simplifying the reproduction process, (2) deepen our understanding of network research results and help efficiently identify their missing details and potential vulnerabilities (*e.g.*, hyper-parameters and corner-case errors) and (3) motivating innovations to improve published research. It could even help improve the peering review process of networking conferences [30], partially realizing the vision of a SIGCOMM April Fools' Day email in 2016 [1].

Our proposal is backed by the recent success of applying LLMs to both general [7, 26, 28, 34, 41, 42] and domain-specific code intelligence tasks [18, 24, 40] as evidences. For general programming, Copilot [7] can provide effective code completion suggestions. ChatGPT [9] can complete and debug simple coding tasks when given proper prompts. Rahmani et al. [28] integrate LLM and content-based synthesis to enable multi-modal program synthesis. For domain-specific programming, in particular the network domain, SAGE [40] uses the logical form of natural-language sentences to identify ambiguities in RFCs and automatically generate RFC-compliant code. NAssim [18] uses an LLM to parse network device manuals and generate corresponding configurations for devices. They both focus on well-formatted inputs with a limited range of topics (*i.e.*, RFC and manuals).

In this paper, we go beyond to take a first step to thoroughly investigate the feasibility and challenges of reproducing network research results by prompt engineering LLMs.

A preliminary experiment (§3). We conduct a small-scale experiment, where we choose four networking systems published in prominent networking conferences and journals [14, 38, 43, 45] and ask four students with essential knowledge of networking to each reproduce one system by prompt engineering the free ChatGPT [9], a publicly available chatbot built on GPT-3.5, a representative LLM [6].

The results verify the *feasibility* of our proposal, *i.e.*, each student successfully reproduces the system assigned to her / him via ChatGPT. Their correctness is validated by comparing the results of small-scale test cases with those of the corresponding open-source prototype. The efficiency is evaluated using large-scale datasets. Results show that their efficiency is similar to that of their open-source prototypes.

We learn several lessons from the experiment on how to use LLMs to reproduce networking research results more efficiently. First, provide LLMs with separate, modular prompts to build different components of a system and then put them together, rather than provide monolithic prompts to build the whole system. Second, ask LLMs to implement components with pseudocode first to avoid unnecessary data type and structure changes later. Third, data preprocessing is important to reproduction, but is often missed in the paper. We also learn some guidelines for debugging LLM-generated code, including sending error messages / error test cases to LLMs, and specifying the correct logic in more detailed prompts.

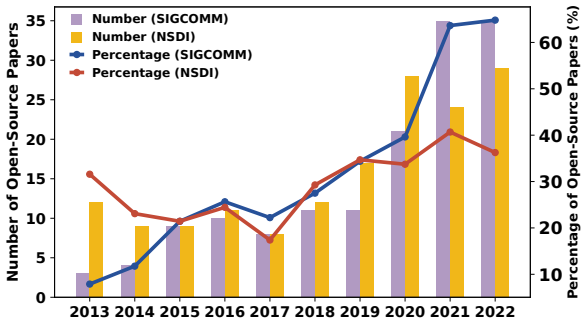


Figure 1: Statistics of SIGCOMM and NSDI papers with an open-source prototype from the authors (2013 - 2022).

Open research questions (§4). We identify several key open research questions regarding LLM-assisted network research results reproduction and elaborate on opportunities to tackle them. They include: (1) how to handle the diversity of these results (2) how to design a (semi-) automatic prompt engineering framework to reproduce these results? (3) how to identify and handle the missing details and vulnerabilities of these results? (4) how to develop a domain-specific LLM for network research reproduction? (5) how to use LLMs to discover optimization opportunities for these results? (6) how to apply this approach of reproduction to promote computer networking education and research?

2 BACKGROUND AND MOTIVATION

We conduct a study on network research reproduction in two prominent network conferences (§2.1) and elaborate on the motivation of our proposal with a simple example (§2.2).

2.1 Background

A statistical study on network research results reproduction. We collect the full research papers in SIGCOMM and NSDI of the past 10 years (*i.e.*, 2013 - 2022). For each paper, we collect: (1) whether the proposed system is open-sourced by the authors; (2) how many other systems are compared in the evaluation; (3) how many of them are open-source prototypes; and (4) how many of them are reproduced by the authors of the proposed system. We do not differentiate publicly available and private prototypes because it is hard to collect this information. Figure 1 plots the open-source statistics of SIGCOMM and NSDI. In total, only 32%/29%/31% of SIGCOMM/NSDI/both papers in the past 10 years release open-source prototypes.

Figure 2 plots the statistics of the number of systems-in-comparisons of each paper and how many of them require manual implementation. We observe that the authors spend substantial efforts to manually implement the systems of others. 59.68% of papers compare with at least two other

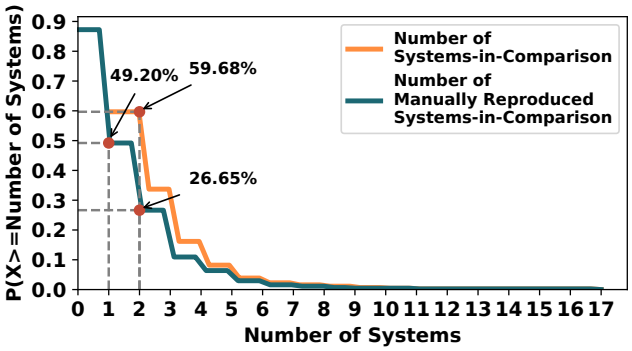


Figure 2: Statistics of the number of compared systems and manually reproduced ones of each paper.

systems. On average, the authors of each paper have to manually reproduce 2.29 systems. 49.20% / 26.65% of the papers have to manually reproduce at least one / two other systems.

Existing reproduction approaches are insufficient. Our study indicates that *although there is a universal need in the community to reproduce network research results, the most dominant way to do so is still manual implementation by following the publication.* Although it would provide people with precious hands-on experiences to better understand the published network research results, it is time and labor-consuming and error-prone. It is unsustainable because network research results, in particular the system ones, are becoming more and more complex. The more time people spend on network research results reproduction, the less time they are left with for critical thinking and innovation.

2.2 Reproducing Using LLMs: Motivation

Inspired by the recent success of LLMs in code intelligence tasks (*e.g.*, code completion [7], debug [9, 41, 42], and synthesis [28, 40]), we propose to prompt engineering of the emerging LLMs to reproduce network research results. As a motivating example, we let three fresh graduate students majoring in computer science interact with free ChatGPT to each develop (1) a UDP server and client that interact to play rock-paper-scissors, (2) an ns-3 simulation of a chain topology with 4 switches and 2 hosts, and (3) an L3 forwarding P4 program, respectively. All three students finished their assignments in less than 12 prompts. For example, after only four prompts with a total of 159 words, ChatGPT generates the correct program for both the UDP server and client, with a total of 93 lines of code (LoC) in Python. Figure 3 shows two snippets of the generated program. These three simple examples show that an LLM can implement network programs and give us the confidence to experiment with more complex network research results.

3 A PRELIMINARY EXPERIMENT

We conduct a small-scale experiment of four participants in summer 2023 to study the feasibility of our proposal.

3.1 Methodology

Participants with basic computer science knowledge. The four participants (referred to as *A* to *D*) are students from three research universities in China. *A* is a first-year master's student majoring in computer science, with a focus

```python
def run_server():
  host = '127.0.0.1'
  port = 12345

  server_socket = socket.socket(socket.AF_INET,
       socket.SOCK_STREAM)
  server_socket.bind((host, port))
  server_socket.listen(1)

  print("Server is running...")

  round_number = 1

  while True:
    client_socket, addr = server_socket.accept()
    print("Connected to", addr)

    while True:
      client_message =
           client_socket.recv(1024).decode('utf-8')
      ...

def run_client():
  host = '127.0.0.1'
  port = 12345

  client_socket = socket.socket(socket.AF_INET,
       socket.SOCK_STREAM)
  client_socket.connect((host, port))

  print("Connected to the server.")

  while True:
    guess = input("Enter your guess (P/R/S for
         paper/rock/scissors, or D to disconnect): ")
    guess = validate_input(guess)
    ...
```

Figure 3: Code snippets of a UDP server and client playing rock-paper-scissors generated by ChatGPT.

on interpretable machine learning. The other three are senior undergraduates who would start their master's program in computer science in fall 2023. During undergraduate, *A*, *B*, and *C* major in computer science, and *D* majors in information and computing science. They all have basic English skills and received basic training in computer science.

Choosing which systems to reproduce. We focus on four systems, two traffic engineering (TE) systems (NCFlow [14] and Arrow [45]) and two data plane verification (DPV) systems (AP [38] and APKeep [43]). All of them are published in top-tier conferences and journals (*i.e.*, SIGCOMM, NSDI, and ToN). We choose them for **three reasons**. First, they are all software systems that can be reproduced in general-purpose programming languages (*e.g.*, Java and Python). Existing LLMs may understand and generate programs in these languages better than those in domain-specific programming languages (*e.g.*, P4 and Verilog). Second, they all run in a centralized controller, which has a simpler architecture and is easier for LLMs to understand than distributed systems (*e.g.*, BGP and Paxos). Third, the papers describing them provide clearly structured details of the systems (*i.e.*, modular components, workflow, definitions, formulations, and pseudocode), making it easier for participants to understand the systems and design proper prompts to interact with LLMs.

Experiment procedure. We mimic the reproduction project in Stanford University CS244 [37]. Each participant is assigned one different aforementioned system and asked to

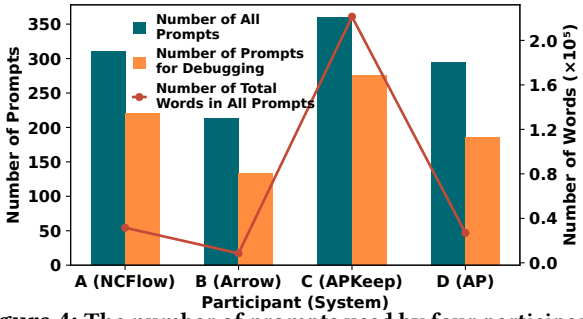


Figure 4: The number of prompts used by four participants.

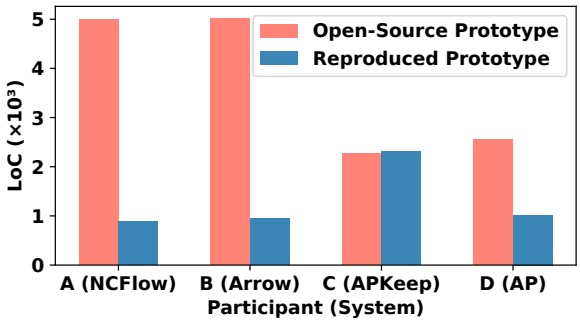


Figure 5: The LoC of open-source / reproduced prototypes.

reproduce it using free ChatGPT in a 25-day period during spare time. They can choose Java or Python for reproducing. They are not allowed to write the implementation code themselves or manually modify the code generated by ChatGPT, nor to copy the prompts of others. They can write their own tests and are encouraged to discuss with other people about the details of the papers and how to interact with ChatGPT. We meet with them online every three to five days to discuss their progress. We do not answer any questions about what specific prompts they should use, but provide them with suggestions from a system designer's perspective. They can also ask questions about the papers, but they do not do so.

Validating the reproduction. Participants validate their reproduction by comparing it with the corresponding open-source prototypes Three systems have open-source prototypes from authors [3–5] and the fourth has a non-author one [8] that is validated in [20]. Each participant uses small-scale tests to examine the correctness of their reproduction. If the output is inconsistent with that of the open-source prototype, she / he manually analyzes the root cause and interacts with ChatGPT to fix the errors. The participant then evaluates its performance on large-scale datasets and compares it with that of the open-source prototype.

3.2 Results

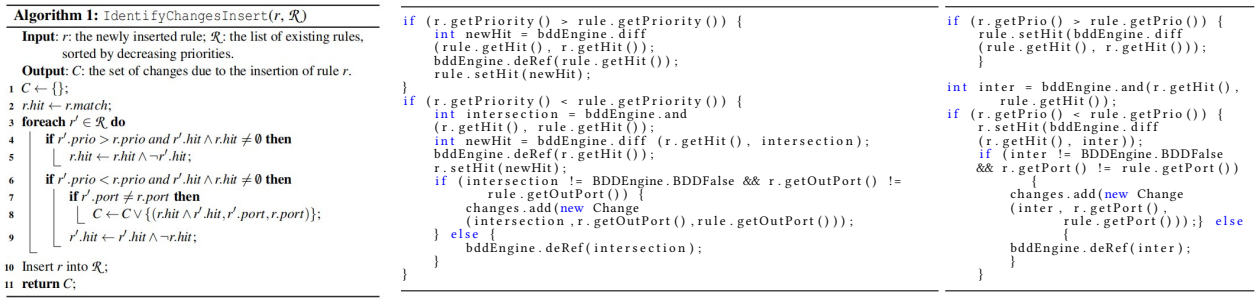
Reproducing network research results via LLMs is feasible. All four participants successfully reproduced their assigned systems. We confirm that all the code are generated by ChatGPT by examining their code and log [15]. Some prompts are based on the pseudocode in the papers. The *correctness* is validated by comparing them with the open-source prototypes in small-scale tests. Figure 4 shows the number of prompts and words each participant used during reproducing. Figure 5 compares the LoC of the reproduced and open-source prototypes. We next present findings on their *performance* by using the datasets in the original papers.

Participant *A*: reproducing NCFlow with high accuracy and performance. *A* evaluates the performance of reproduced NCFlow in 13 TE instances from [14]. The reproduced NCFlow computes the objective function value with a maximal of 3.51% difference from the open-source prototype and a maximal end-to-end computation latency of 6.4 seconds. Although the latency is up to 111x higher, it is due to the choice of LP solvers between two implementations (*i.e.*, the reproduced one uses Pulp [10] and the open-source one uses Gurobi [2]). The LoC of the reproduced prototype is only 17% of that of the open-source one. This is because the open-source prototype has a large portion of code for formatting and parsing the irregular inputs of datasets.

Participant *B*: reproducing Arrow with low accuracy due to inaccurate prompts. *B* evaluates the reproduced Arrow in two TE instances from [45]. The computed objective function has an up to 30% difference from that of the open-source prototype. Originally, we thought the root cause was the inconsistency between the paper description and the open-source implementation of the restorable tunnel. After we posted our original results to arXiv [36], the authors of Arrow [45] reached out and helped us confirm that the real reason is the inaccurate prompts *B* sent to ChatGPT, which causes ChatGPT to set the traffic demands and the bounds of some decision variables in the optimization problem to incorrect values. We apologize for our misanalysis and thank the authors of Arrow for pointing it out and helping us fix it.

Participant *C*: reproducing APKeep accurately with comparable latency. *C* compares the reproduced APKeep with a non-author open-source prototype on four real topology datasets to verify loop-free, blackhole-free reachability. In all cases, both prototypes compute the same number of atomic predicates and have approximately the same latency. Both use JDD [33] for binary decision diagram operations and have approximately the same number of LoC.

Participant *D*: reproducing AP accurately but with worse performance due to the choice of different BDD libraries and missing details in the paper. *D* compares the reproduced AP with the open-source one on three real-topology datasets to verify loop-free, blackhole-free reachability. Although both compute the same number of atomic predicates and verification results, the reproduced AP has a substantially worse latency: (1) up to 20x longer time to compute predicates and (2) up to 10^4x longer time to verify reachability). The root cause of the former is the use of JavaBDD, a library with a worse performance of BDD operations than JDD. The root cause of the latter is the missing details of the reachability verification algorithm in the paper. The paper only gives the algorithm on given a path, how to find the predicates reaching d from s. It does not describe how to efficiently find all the predicates reaching d from s from any path (*e.g.*, the authors use a selective BFS traversal in their open-source prototype). Because *D* is not a computer



Algorithm 1: IdentifyChangesInsert(r, $\mathcal{R}$)

Input: r: the newly inserted rule; $\mathcal{R}$: the list of existing rules, sorted by decreasing priorities.
Output: C: the set of changes due to the insertion of rule r.

$C \leftarrow \{\}$;
$r.hit \leftarrow r.match$;
foreach $r' \in \mathcal{R}$ **do**
  if $r'.prio > r.prio$ *and* $r'.hit \wedge r.hit \neq \emptyset$ **then**
    $r.hit \leftarrow r.hit \wedge \neg r'.hit$;
  if $r'.prio < r.prio$ *and* $r'.hit \wedge r.hit \neq \emptyset$ **then**
    if $r'.port \neq r.port$ **then**
      $C \leftarrow C \vee \{(r.hit \wedge r'.hit, r'.port, r.port)\}$;
    $r'.hit \leftarrow r'.hit \wedge \neg r.hit$;
Insert r into $\mathcal{R}$;
return C;

```
if (r.getPriority() > rule.getPriority()) {
    int newHit = bddEngine.diff
    (rule.getHit(), r.getHit());
    bddEngine.deRef(rule.getHit());
    rule.setHit(newHit);
}
if (r.getPriority() < rule.getPriority()) {
    int intersection = bddEngine.and
    (r.getHit(), rule.getHit());
    int newHit = bddEngine.diff (r.getHit(), intersection);
    bddEngine.deRef(r.getHit());
    r.setHit(newHit);
    if (intersection != BDDEngine.BDDFalse && r.getOutPort() !=
            rule.getOutPort()) {
        changes.add(new Change
        (intersection,r.getOutPort(),rule.getOutPort()));
    } else {
        bddEngine.deRef(intersection);
    }
}
```

```
if (r.getPrio() > rule.getPrio()) {
    rule.setHit(bddEngine.diff
    (rule.getHit(), r.getHit()));
    }

int inter = bddEngine.and(r.getHit(),
        rule.getHit());
if (r.getPrio() < rule.getPrio()) {
    r.setHit(bddEngine.diff
    (r.getHit(), inter));
    if (inter != BDDEngine.BDDFalse
    && r.getPort() != rule.getPort())
            {
        changes.add(new Change
        (inter, r.getPort(),
                rule.getPort()));} else
                {
        bddEngine.deRef(inter);
        }
    }
```

(a) Pseudocode in [43]. **(b) Manually implemented code [5].** **(c) ChatGPT-generated code.**

Figure 6: The pseudocode, manually implemented code and ChatGPT-implemented code of APKeep to identify all behavior changes caused by a rule insertion.

science major and unfamiliar with the exponential complexity of path enumeration, D decides to use the algorithm in the paper as a building block to enumerate all the paths from s to d and check the reachable predicates for each path, leading to a much higher verification latency. It could be avoided by stating the use of selective BFS traversal in the paper. We later discuss how we may integrate LLM-assisted reproduction and formal methods to identify such issues.

3.3 Lessons

After the experiment, we interview the participants to gather their experience and summarize several lessons.

Asking LLMs to implement a system component by component, not the whole system all at once. In the beginning, all participants tried to send ChatGPT prompts like "implement XX that works in the following steps XXX". ChatGPT does not respond well to such *monolithic* prompts. As such, they switch to a top-down approach, which divides the system into components, and for each component, sends ChatGPT more detailed *modular* prompts in sequence to implement, debug and test it. This allows them to reproduce the system successfully. It shows that ChatGPT's capability of understanding system design and implementing it is still limited to small systems and components.

Implementing components with pseudocode first. In a paper, the part closest to the real code is its pseudocode (Figure 6). As such, it would be ideal if ChatGPT could receive it as a prompt to generate the code using it as a basis. However, suppose we first ask ChatGPT to implement the components without pseudocode. When asking it to implement the components with pseudocode later, the generated code often requires substantial changes to data types and structures in the previously generated ones. It is because ChatGPT implements components described by text-based prompts differently from the ones described by pseudocode-based prompts, leading to interoperability issues. Two students find that implementing components with pseudocode first allows ChatGPT to stabilize the key data types and structures and avoid changing them when implementing other components. As such, we plan to design *a pseudocode-like intermediate representation for all components of a system* to improve the efficiency of LLM-assisted reproduction.

Data preprocessing is important to the system, but not to the research paper. Because how to preprocess data is usually not provided by the research paper, participants have to look into the datasets and send ChatGPT data format-related prompts based on their understanding. Thus a generic and automatic data preprocessing solution is another key for improving the efficiency of LLM-assisted reproduction.

Three guidelines for debugging. First, data type bugs can be fixed by sending the error messages to ChatGPT. They can also be avoided by explicitly specifying key variables' data types and structures or at least describing their needed operations and properties in prompts. Second, for simple logic bugs, sending the test causing them to ChatGPT can be an effective approach to repair them. Third, for more complex logic bugs, we can repair them by specifying the correct logic in more detailed, sometimes step-by-step, prompts.

4 OPEN RESEARCH QUESTIONS

Handling the diversity of published network research. This diversity comes from two folds. First, network research papers have very diverse content and level of detail. For example, SIGCOMM and NSDI papers are usually very system-heavy, while other venues (*e.g.*, INFOCOM) focus more on the analytical side. Reproducing them may require different ways of processing and digesting them. Second, networking is an area with many topics (*e.g.*, network architecture, programmable hardware, and distributed systems. Reproducing papers on different topics may require different ways of prompt engineering. Previous studies focus on a limited range of topics and use well-formatted input [18, 40]. They may not apply to the broader context of network research.

One direction to tackle this issue is a unified prompt engineering framework for reproducing network research results [16, 35]. One design is to follow the top-down approach of system development with the following steps: (1) describe to the LLM the key components of the system, (2) describe how components interact and ask the LLM to define the interfaces, (3) provide the LLM with the details of each component to generate the code, (4) test and debug the LLM-implemented component, (5) repeat (3) and (4) for each component, and (6) test and debug the complete system.

Improving the efficiency of reproducing via (semi-) automatic prompt engineering LLMs. Although manual prompt engineering is beneficial for people to better understand the details of published research results, the efficiency of reproducing can be substantially improved by

(semi-) automatic prompt engineering [46]. We plan to focus on automating the aforementioned unified prompt engineering framework. Given a network research paper, our initial design involves (1) using an LLM for natural language (*e.g.*, ChatPDF [12]) to understand the paper and extract its architectures, key components, and workflow, (2) transforming the extracted information into multi-modal prompts (*e.g.*, logic predicates, pseudocode, examples, and test cases) for the overall architecture and each component, (3) sending the prompts for components to an LLM for coding (*e.g.*, FlashGPT3 [34]) to generate, test and debug codes for each component, and (4) sending the prompts for architecture to piece different components into a complete system.

Handling missing details and vulnerabilities in published network research. Due to the space and time limit, the authors may have to omit non-essential technical details in their published research. Such missing details pose additional challenges for reproducing these results, *e.g.*, the experience of participant *D* in our experiment (§3.2). Published network research may also have vulnerabilities, such as inaccurate descriptions of designs and examples (*e.g.*, [20]) and hard-to-tune hyper-parameters (*e.g.*, [43]). They make it challenging for LLMs to reproduce the results correctly (*e.g.*, sending an incorrect example to LLMs). one way to handle these is to encourage the authors to be more thorough during proofreading and provide an appendix on non-essential details and hyper-parameters. Another way is to write papers in a better-structured format (*e.g.*, RFC or system manual).

We may also resort to formal methods (*e.g.*, [17, 29]) for these issues. We could verify and analyze the workflow and algorithms extracted from the paper by either human efforts or the LLMs to search for such issues [40, 44]. If a paper has an open-source prototype, we may comparatively analyze it and the LLM-reproduced one to examine their functionality and performance discrepancy using modular checking [32].

Building domain-specific LLMs for network research reproduction. Such LLMs can substantially improve the scope and efficiency of LLM-assisted network research reproduction. In our experiment, participants reproduce centralized software systems using ChatGPT, a chatbot built on a general-purpose LLM. However, many network research results propose hardware systems [31], hardware-software co-design systems [22] and distributed systems [27]. Not only are they more complex, some of them also require domain specific languages (*e.g.*, P4 and Verilog) that are very different from general programming languages (*e.g.*, Java). General-purpose LLMs may have difficulty reproducing them.

We propose to specifically build a network research reproduction LLM by using network research materials (*e.g.*, papers, codes, and RFCs on various network research topics) as training data. Early evidence supporting the feasibility of such an LLM is the recent success of programming-oriented LLMs [7, 18, 24, 26, 28, 34, 40–42] in providing code completion suggestions, identifying and fixing bugs, and reproducing RFCs. One key challenge is the availability of data. Although there are huge amounts of network traffic data [24], there is substantially less network research code available. One way to tackle this is to integrate data augmentation and static analysis to produce more network research code.

Discovering innovation opportunities from reproducing networking research results. LLM-assisted networking research reproduction could help reveal innovation opportunities. First, it could deepen researchers' understanding on published research results, helping organize their intellectual and critical thinking. Second, analyzing the reproduced prototype using automatic program analysis [39, 44] could expose bottlenecks of the proposed design, leading to system optimization opportunities. Third, although a long shot, it is theoretically feasible to build a deep learning model with open-source and reproduced prototypes of network research as datasets to predict networking innovations, similar to the recent building of AlphaFold [23] in the area of biology. Interpretable AI could be an important tool [21, 25] given their capability to extract logic behind system behaviors.

Promoting computer networking education and research. In the era where AI is the predominant computer science research area, reproducing network research results using LLMs could motivate their interest in networking education and research. First, by interacting with LLMs they get both hands-on experience with the latest AI breakthrough and the opportunity to understand the classic and latest network research results. As such, it is useful for students to strengthen their career skills in both computer networks and AI. Second, as discussed earlier, this process could help students discover innovation opportunities in networking research. Third, it fits into the competency-building model of the recent computer science education paradigm [13, 19]. Fourth, LLM-assisted network research results reproduction could help improve the peering review process of prominent networking conferences [30]. One may recall an April Fools' Day SIGCOMM email in 2016 saying that SIGCOMM will introduce AI to automate the paper review process. Although we believe that this email is still too "forward-looking", our proposal would be an interesting trial to automate part of the review process.

LLM-assisted network research reproduction, in particular a (semi-)automatic framework, also has a negative impact: students may misuse it to finish projects. How to walk the fine line between leveraging LLMs-assisted reproduction to promote networking education and research and misusing it is an important question for both academia and industry.

Acknowledgments. We are extremely grateful for the anonymous HotNets reviewers for their wonderful feedback. We thank the authors of Arrow [45] for their help. Qiao Xiang, Yuling Lin, Mingjun Fang, Bang Huang, Siyong Huang, and Ridi Wen are supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award 62172345, Open Research Projects of Zhejiang Lab 2022QA0AB05, MOE China 2021FNA02008, and NSF-Fujian-China 2022J01004. This work raises no ethical issue.

REFERENCES

[1] 2016. SIGCOMM to use Deep Learning for Paper Selection,An Email from SIGCOMM Mailing List on April 1. https://sigcomm.org/about/mailing-lists/. (2016).

[2] 2018. Gurobi. https://www.gurobi.com. (2018).

[3] 2020. Open-source Prototype of NCFlow. https://github.com/netcontract/ncflow. (2020).

[4] 2021. Open-source Prototype of ARROW. https://github.com/hipersys-team/arrow. (2021).

[5] 2021. Open-source Prototype of Atomic Predicates Verifier. https://gitee.com/gdtongji/atomic_predicates_verifier. (2021).

[6] 2022. GPT-3.5. https://lablab.ai/tech/openai/gpt3-5. (2022).

[7] 2022. Microsoft Copilot. https://github.com/features/copilot/. (2022).

[8] 2022. Open-source Prototype of Flash. https://github.com/snlab/flash. (2022).

[9] 2022. OpenAI ChatGPT. https://openai.com/blog/chatgpt. (2022).

[10] 2022. Pulp. https://pypi.org/project/PuLP. (2022).

[11] 2023. ACM IMC 2023 Call For Papers: Replicability Track. https://conferences.sigcomm.org/imc/2023/cfp/. (2023).

[12] 2023. ChatPDF. https://www.chatpdf.com/. (2023).

[13] 2023. CS2023: ACM/IEEE-CS/AAAI Computer Science Curricula. https://csed.acm.org/. (2023).

[14] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting Wide-area Network Topologies to Solve Flow Problems Quickly.. In *NSDI 2021*.

[15] Anonymous-Authors. 2023. Conversation Log of ChatGPT in the Experiment. https://www.dropbox.com/sh/11tshmzf517juy5/AACLtpzN-Oa-zbJluXNyH7rPa?dl=0. (2023).

[16] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2022. Prompting Is Programming: A Query Language for Large Language Models. *arXiv e-prints* (2022).

[17] Wolfgang Bibel. 2013. *Automated Theorem Proving*. Springer Science & Business Media.

[18] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. 2022. Software-Defined Network Assimilation: Bridging the Last Mile Towards Centralized Network Configuration Management with Nassim. In *SIGCOMM 2022*.

[19] Alison Clear, Allen Parrish, Ming Zhang, and Gerritt C van der Veer. 2017. Cc2020: A Vision on Computing Curricula. In *SIGCSE 2017*.

[20] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. 2022. Flash: Fast, Consistent Data Plane Verification for Large-Scale Network Settings. In *SIGCOMM 2022*.

[21] Yangfan Huang, Yuling Lin, Haizhou Du, Yijian Chen, Haohao Song, Linghe Kong, Qiao Xiang, Qiang Li, Franck Le, and Jiwu Shu. 2023. Toward a Unified Framework for Verifying and Interpreting Learning-Based Networking Systems. In *IWQoS 2023*.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP 2017*.

[23] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly Accurate Protein Structure Prediction with Alphafold. *Nature* (2021).

[24] Franck Le, Mudhakar Srivatsa, Raghu Ganti, and Vyas Sekar. 2022. Rethinking Data-Driven Networking with Foundation Models: Challenges and Opportunities. In *HotNets 2022*.

[25] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. 2020. Interpreting Deep Learning-Based Networking Systems. In *SIGCOMM 2020*.

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv e-prints* (2022).

[27] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *ATC 2014*.

[28] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-trained Language Models and Component-Based Synthesis. *arXiv e-prints* (2021).

[29] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Mit Press.

[30] Scott Shenker. 2022. Rethinking SIGCOMM's Conferences: Making Form Follow Function. *In SIGCOMM 2022* (2022).

[31] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM 2016*.

[32] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *SIGCOMM 2021*.

[33] A. Vahidi. 2020. A BDD and Z-BDD Library Written in Java. https://bitbucket.org/vahidi/jdd. (2020).

[34] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-trained Models. *In PACM PL 2021* (2021).

[35] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. *arXiv e-prints* (2023).

[36] Qiao Xiang, Yuling Lin, Mingjun Fang, Bang Huang, Siyong Huang, Ridi Wen, Franck Le, Linghe Kong, and Jiwu Shu. 2023. Toward Reproducing Network Research Results Using Large Language Models. (2023). arXiv:cs.LG/2309.04716

[37] Lisa Yan and Nick McKeown. 2017. Learning Networking by Reproducing Research Results. *In SIGCOMM 2017* (2017).

[38] Hongkun Yang and Simon S Lam. 2016. Real-Time Verification of Network Properties Using Atomic Predicates. *In TON 2016* (2016).

[39] Rulan Yang, Xing Fang, Lizhao You, Qiao Xiang, Hanyang Shao, Gao Han, Ziyi Wang, Jiwu Shu, and Linghe Kong. 2023. Diagnosing Distributed Routing Configurations Using Sequential Program Analysis. In *APNet 2023*.

[40] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-automated Protocol Disambiguation and Code Generation. In *SIGCOMM 2021*.

[41] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *arXiv e-prints* (2022).

[42] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2022. Automated Feedback Generation for Competition-Level Code. In *ASE 2022*.

[43] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *NSDI 2020*.

[44] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. 2023. Performal: Formal Verification of Latency Properties for Distributed Systems. *In PACM PL 2023* (2023).

[45] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. 2021. ARROW: Restoration-Aware Traffic Engineering. In *SIGCOMM 2021*.

[46] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Models Are Human-Level Prompt Engineers. *arXiv e-prints* (2022).