Facilitating Experiential Learning in Networking Education

Jaber Daneshamooz*
jaber@ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

Sanjay Chandrasekaran
sanjaychandrasekaran@umail.ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

Sylee Beltiukov
rbeltiukov@ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

Walter Willinger
wwillinger@niksun.com
Northwestern University
Evanston, IL, USA

Arpit Gupta
arpitgupta@ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

Abstract

Generative models now answer questions in seconds, in networking as across computer science, and students increasingly trust the answers without checking them. Checking an answer typically means running an experiment, and setting one up by hand often takes days, leaving little time for the reasoning that matters. Teaching that reasoning is now a central job of a networking course. Students build this reasoning skill through experiential learning: they explain why a system behaves as it does by predicting an outcome, generating data to test it, and reconciling the two. *Pramana*, a composable backend for empirical networking research, makes that data cheap and fast to generate. A student describes an experiment in natural language, and Pramana sets up the network, runs the application, and returns analyzable data. We deployed Pramana in a project replicating a congestion-control result in an upper-division networking course, where 46 of 70 students consented to research use. (1) 15% of them caught Pramana running an experiment other than the one they asked for, spotting it from the evidence rather than from the tool's success report. (2) 89% say the project would have taken at least twice as long, or been impossible to finish in time, without Pramana. (3) Retrospective self-ratings of understanding rose from 3.07 to 3.93 out of 5, with 30 of 46 students improving and none declining. We discuss how Pramana generalizes beyond this project, what did not work, and our plans to improve Pramana.

CCS Concepts

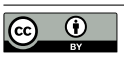
• **Social and professional topics** → **Computer science education**; • **Networks** → *Network experimentation.*

Keywords

networking education; inquiry-based learning; active learning; experiential learning; natural-language interfaces; generative AI

ACM Reference Format:
Jaber Daneshamooz, Sanjay Chandrasekaran, Sylee Beltiukov, Walter Willinger, and Arpit Gupta. 2027. Facilitating Experiential Learning in Networking Education. In *Proceedings of the 58th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2027), February 17–20, 2027, Sacramento, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3857222.3860185

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCSE TS 2027, Sacramento, CA, USA
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3075-7/2027/02
https://doi.org/10.1145/3857222.3860185

1 Introduction

A student with a networking question can now get a fluent answer in seconds. Ask a generative model to configure a link, walk through a protocol, or trace a state machine, and it does so on demand [2, 10]. These are the tasks networking courses have taught and graded for decades. Because a model now performs them (the *how*) cheaply and quickly, assignments that ask only these questions no longer show what a student understands. What a model cannot easily supply, and what a student still needs to learn, is an explanation of a system's behavior (the *why*). Moving a course from teaching how networks work to why they work the way they do requires experiential learning, a form of the active learning that improves outcomes across STEM courses [11].

Answering a *why* question usually takes data: the student forms a hypothesis and tests it. For example, BBR [4] and CUBIC [13] are two congestion-control algorithms, the rules a sender uses to pace how fast it sends packets. To decide whether BBR keeps a link less congested than CUBIC, the student generates traffic under controlled conditions and compares how many packets queue up at the link under each algorithm [12]. Obtaining that data is the costly step. By hand, it means setting up an isolated network, fixing its bandwidth and delay, running an application across it, recording the measurements, and writing the analysis code. None of this work concerns the question the student set out to answer, and it can take days. In a prior offering of our course, students built this infrastructure themselves: setup consumed the term, little time remained for reasoning about results, and one group's load generator saturated the campus wireless network. We report this from instructor recollection; it was never measured, and it motivates our approach but proves nothing about it. The barrier is long-standing: earlier courses in which students reproduced networking experiments found that setting up the infrastructure took most of the time and left little time for reasoning and analysis [21, 22].

This paper examines what lowering that cost, measured in student time, does for learning. It shows how *Pramana* [9], a composable backend for empirical networking research, can facilitate experiential learning in networking courses. Pramana has a *thin waist* design, the hourglass shape that IP gives the Internet's architecture [17]. One narrow interface sits between the experiments a student can ask for, above it, and the places an experiment can run, below it [9]. A student asks for controlled networking data in a

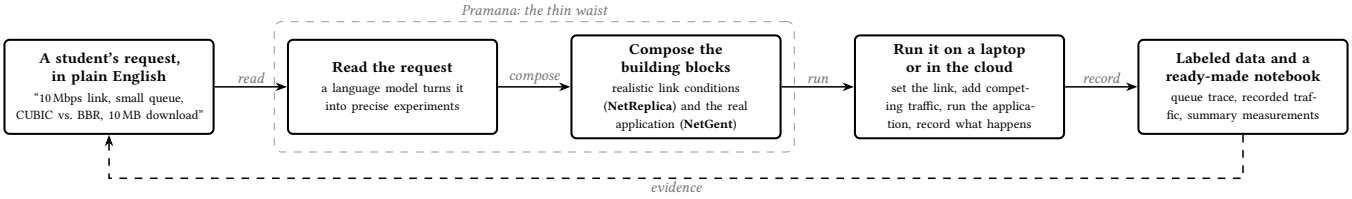


Figure 1: How Pramana turns a student's natural-language request (left) into labeled data with a ready-made notebook (right). Pramana composes the systems that set the experiment's link conditions and its application; the student writes the request, reads the evidence (dashed return), and leaves configuring those systems to Pramana.

natural-language request, and Pramana composes reusable systems behind that interface to return the data, realistic and labeled, in minutes instead of days, on a laptop or in the cloud. When data is this cheap to generate, a student can afford a three-step *predict-measure-reconcile* loop: **predict** what a system will do, **measure** it by generating data, and **reconcile** the prediction with the evidence. Understanding forms in the reconcile step, where a gap between what the student expected and what the data shows forces the question why. Figure 1 shows the loop for one request, a file download under CUBIC over a slow link. Pramana returns a labeled dataset with a ready-made notebook, and the student reads the evidence and reruns the experiment with one setting changed. Pramana runs the experiments, and the student does the predicting and the reconciling. Most work on generative AI in computing education defends assessment against the model; here Pramana's language model writes the specification of an experiment that students then analyze.

This paper makes two contributions. **(1) A pedagogical thesis.** We argue that the AI era moves networking education from recalling architectural choices to deriving them empirically, that the predict-measure-reconcile loop is the mechanism, and that paper replication is one instance of a broader design space. Because Pramana has a thin-waist design, new uses can grow above it, such as experiments other than congestion-control replication, and new places to run below it, such as a shared testbed like FABRIC [1, 9]. Section 5 discusses which other course assignments it supports today. **(2) A classroom deployment and user study.** Pramana served as the data-generation backend for a congestion-control replication project [20] in our networking course for upper-division students, and we surveyed the 46 consenting students, following the methodology of a related networking-education study [5]. We describe the problems students encountered (§3.6) and report results on students' reasoning from evidence, on cost (time saved), and on learning. First, 15% of students noticed that Pramana had run a different experiment from the one they requested, and they found the mismatch in the logs, traces, and measurements while Pramana reported success. Second, 89% estimated that, without Pramana, they would have needed at least twice the time or could not have finished. Third, looking back, students rated their grasp of the replicated paper 0.87 points higher after the project than before, on a scale of 1 to 5 (a large effect, Cohen's $d_z = 0.98$); 30 of 46 rated it higher and none lower. Section 5 describes the objective measures a follow-up study would add.

2 Pramana in the Classroom

Experiential learning. A recent first-principles framework for networking education teaches students to derive why a system is designed as it is through analysis [5]. This paper pursues the experiential form of the same skill: the student chooses a scenario worth testing, generates data for it, and explains the result. That form becomes practical once generating the data is no longer slow and costly, and Pramana makes it fast and cheap [9].

What Pramana composes. Pramana composes two systems, each of which separates one part of an experiment. NetReplica sets a link's fixed properties (capacity, delay, and how it buffers and drops packets) separately from the competing traffic on it [6, 7]. It replays that traffic from real production traces, so a student can hold the link fixed and dial the congestion up or down. NetGent turns a natural-language description of an application session, such as opening a player and skipping ahead five minutes, into automation that drives the real application and keeps working when its interface changes [8]. From netUnicorn [3], the first system we built with a thin-waist design, Pramana borrows the separation of what an experiment does from where it runs. Mininet, the de facto choice, runs a whole emulated network with real programs on one laptop [15], but the network is tied to that machine, its links carry no competing traffic, and it does not drive realistic application sessions such as a video call. Wiring such systems together by hand means knowing which one answers which question; with Pramana, one request drives all of them.

From one request to labeled data. A student writes a single request, for example "a 10 Mbps link with a small queue, comparing CUBIC and BBR on a 10 MB download" (Figure 1). A language model, guided by a controller [14], compiles the request (what the companion paper [9] calls an *intent*) into a precise experiment specification and splits the comparison into two runs, one per algorithm. For each run, Pramana sets up a fresh sandbox, configures the link and any competing traffic, runs the application, and records what happens. It returns a dataset labeled with each run's requested conditions: the raw packet capture, a trace of how full the link's queue stayed over time, and summary measurements such as throughput, delay, and loss. A ready-made notebook plots the queue traces, in which each algorithm leaves a pattern that students learn to tell apart. Because the specification names every setting, a student can rerun it to get comparable data, or change one setting, say the congestion-control algorithm, while holding the rest fixed.

The version students used. Requests are often incomplete: "a small queue" names no size. The version students used filled each missing value from a file of defaults and recorded the values it chose

with the results, so a student could check them against the paper being replicated or the question being asked. Deciding whether a default fits the hypothesis is itself part of the exercise. That version did not check a request against what the underlying systems support, so it could accept a request it could not carry out and still report success (§3.4).

Pramana's design and what runs today. A companion paper presents Pramana's design and implementation [9]; this paper reports its first use in a course. The problems students ran into shaped the design: each system publishes a versioned *capability file* listing what it supports, Pramana checks every request against those files, and it shows the student the parsed configuration before the run. Adding an algorithm or an application then changes nothing in how students write requests [9]. No system publishes a capability file yet. The current implementation runs in local containers and on AWS. It shapes one bottleneck link per run (the slowest link on the path, where packets queue), with configurable capacity and buffer size, and sets each application's delay and loss. It offers 15 congestion-control algorithms built into the Linux kernel and 22 application workflows, each a scripted session of a real application such as YouTube or Zoom. It rejects an algorithm outside those 15 and fails on an unknown queue discipline (the rule that decides which queued packet the link sends next or drops), but a request for BBR on a remote sender still runs without BBR and without a warning. It builds no network with more than one link. Against 255 data-generation experiments drawn from 66 networking papers in eight research areas [7], the implementation covers 68 (27%), or at most 87 (34%) when experiments the papers underspecify are counted, while the best existing single tool covers 11% [9].

3 Deployment and Evaluation

Students used Pramana in a paper-replication project, and we surveyed those who consented. We organize the evaluation around the **predict-measure-reconcile** loop the thesis rests on. Three research questions follow the loop's steps, and two cover its outcomes. **RQ1** (Predict): does natural-language framing move students toward stating a prediction (§3.2)? **RQ2** (Measure): does Pramana make generating data cheap (§3.3)? **RQ3** (Reconcile): do students check their predictions against the evidence (§3.4)? **RQ4** (Understanding): do students report understanding the replicated paper better (§3.5)? **RQ5** (What did not work): where does the version students used get in the way (§3.6)?

3.1 Study Design

Students reproduced the central result of CCAnalyzer [20], a study showing that a host can tell which congestion-control algorithm a remote server uses just by watching how the network queue between the host and the server fills and drains over time. The companion paper [9] regenerated the data of CCAnalyzer's main experiment, one trial per network setting and algorithm. Working in small groups, students used Pramana to set up a controlled link, run downloads under three congestion-control algorithms (CUBIC, BBR, and Reno), record how the queue behaved, and compare against the paper. They ran both controlled experiments with a known algorithm and "in the wild" transfers against public servers, where the students control only their own end of the connection. Replication is the one instance we measured; §5 discusses others. We chose CCAnalyzer for four reasons. It has one objective measure of success, classification accuracy, so students can tell whether they reproduced it. Its premise, that a host cannot choose the server's algorithm, can only be discovered empirically, so reconciling a prediction with the data takes real reasoning. Its results depend mostly on collecting the data, which is Pramana's job, and little on analyzing it, which is the students' job; students can then compare their plots directly with the paper's. And it needs only what the version students used already offered: a bottleneck link, a choice of congestion-control algorithms, and queue measurements.

The assignment, for reuse. The assignment's core settings are the link's bandwidth, its delay, and the congestion-control algorithm, and Pramana offers more (§2). Students also varied the queue size (32, 64, and 128 packets), and some compared queue disciplines such as CoDel for bonus credit; the survey did not cover this bonus work, and the replicated paper used neither queue disciplines nor competing traffic. In Part 1, students fix a known algorithm (CUBIC, BBR, or Reno) and write a request setting those values, then record how full the queue stayed and repeat under a second algorithm under identical conditions. In Part 2, they infer which algorithm a public server uses from the same queue pattern. Deliverables are a short report plus the requests and notebooks that regenerate the figures. Adopters need a Linux machine per group, since BBR is unavailable on macOS (§3.6).

Instrument and sample. The course materials and the survey called Pramana's interface the "Agentic Thin Waist" platform, so survey items about "the platform" refer to Pramana. An end-of-project survey has seven sections, and its items are numbered by section and question: background (1), the replicated paper before the project (2), adoption and usability (3), the natural-language interface (4), task load (5), understanding and effort after the project (6), and open-ended responses (7). Sections 3 and 4 are five-point Likert items (agreement ratings from 1, strongly disagree, to 5, strongly agree). Table 1 lists the rated items we analyze; the ten-point task-load (5.1, 5.2) and recommendation (6.5) items are reported in §3.6. After consent filtering, 46 of the 70 students in the course remain ($n = 46$). Respondents are not novices: 85% report prior network-experiment experience and median self-rated Python proficiency is 4 out of 5. Two respondents reported not using Pramana; we retain them under the consent-based rule and report a sensitivity analysis (§3.6) that leaves the same descriptive pattern. Our institution's Institutional Review Board determined the survey exempt (45 CFR 46.104(d), categories 1 and 2); participation was voluntary, and we analyze only consenting students' responses.

Descriptive analysis. Our analysis follows Daneshamooz et al. [5]. Each five-point item is tested with a one-sample t-test against the neutral midpoint of 3 ($df = 45$); items 3.4 and 4.4 are negatively worded and reverse-coded. Because items differ in how far they clear neutral, we read the ranking of effects in Figure 2 rather than any single item, and treat the per-item t-tests as descriptive rather than independent confirmatory hypotheses. The pre/post pair uses a paired t-test and a Wilcoxon signed-rank test. Free-text responses are coded per respondent, and quotes carry respondent labels (R1–R46); a category is reported only if it reaches a pre-stated $\geq 10\%$ ($\geq 5/46$) threshold.

Table 1: Survey items, abbreviated and numbered by section and question (§3.1). Items 3.x and 4.x are Likert items (1 = strongly disagree, 5 = strongly agree); 2.1 and 6.1 are rated on five points; 6.2 and 6.4 are ordinal multiple-choice items.

Adoption & usability	
3.1	I would like to use the platform again
3.2	The platform was easy to use
3.3	I felt confident using the platform
3.4	I needed a lot of help before I could get going
3.5	The platform saved me time vs. raw tools
3.6	The platform improved my ability to replicate the paper
3.7	I would use the platform for future work
Natural-language interaction	
4.1	Describing my experiment in natural language felt natural
4.2	The platform usually understood what I wanted
4.3	When it misunderstood me, I could correct it
4.4	I often had to rephrase the same intent repeatedly
Understanding & effort	
2.1	Before starting, how well did you understand the paper you were replicating?
6.1	After completing this task, how well do you understand the paper you replicated?
6.2	How much did the platform help you learn, compared to just reading the paper?
6.4	Without the platform (raw tools), how much longer would replication have taken?

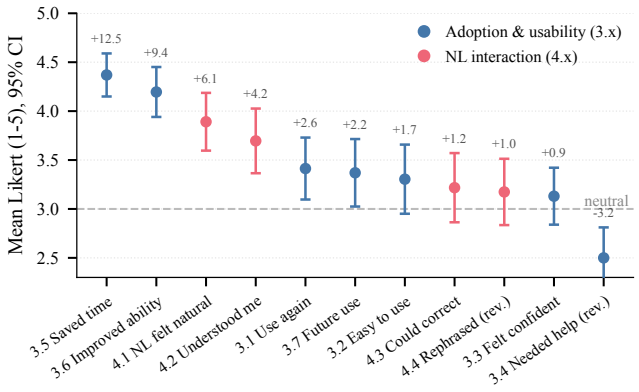


Figure 2: The eleven Likert items, ordered left to right by descending t-statistic against the neutral midpoint (3.0), with 95% confidence intervals; the value above each point is its t-statistic.

3.2 Predict: Framing in Natural Language

Students wrote their requests in natural language, and the interface handled the common case. "Describing my experiment in natural language felt natural" (4.1) reaches mean 3.89 (t = +6.08) and "the platform usually understood what I wanted" (4.2) reaches 3.70 (t = +4.24); the two correlate (Spearman $\rho = 0.61$) and anchor the left of Figure 2, just behind the saved-time and improved-ability items (3.5, 3.6).

The framing also changed how a subset of participants designed experiments: eight of 46 (17%) describe shifting from low-level commands to a higher-level, one-variable-at-a-time comparison, "the focus shifted to the high-level goal rather than the individual `tc` commands" (R9), though a comparable group reports no change.

3.3 Measure: Cheap Data Generation

The strongest signal in the survey is saved time. "The platform saved me time compared to setting up the experiments by hand" (3.5) is the highest-ranked item at mean 4.37 (t = +12.53; Figure 2), and "the platform improved my ability to replicate the paper" (3.6) is second at 4.20 (t = +9.44). Asked how much more time using raw tools would have cost them, 89% chose at least 2×, including 26% who could not have done it at all; only 7% said "about the same" (Figure 3b). The open-ended responses name the mechanism: 28% credit the automated execution-and-measurement path and 26% credit the natural-language request for letting them focus on design rather than infrastructure (Table 2). For scale, in a 20-run benchmark, Pramana's current implementation compiled a request into a specification in about 2.7 s, and data arrives about three minutes later: a 44 s wait for a fresh worker, a 60 s measurement, and a 71 s telemetry transfer [9].

3.4 Reconcile: Catching Wrong Results

When measurement is cheap, students compare a prediction against the data and chase down the mismatch. The clearest cases concern whether the requested algorithm actually ran. One student asked for CUBIC on a download but found the data looked wrong. Pramana had set the algorithm on the student's own machine, but in a download the server sends the data, so the remote server's own algorithm shaped the queue; as the student put it, "saying a parameter is not the same as controlling it" (R16). Afterward they checked each run directly rather than trusting the tool's report. A second student expected a clear CUBIC-versus-BBR difference, saw none, and concluded the run was invalid (R39). Discovering that one *cannot* dictate a remote server's algorithm is the premise CCAnalyzer builds on, reached here from evidence rather than assertion.

In the coded open-ended responses, 15% of students spotted the platform running something other than what they asked, identifying it from logs and traces instead of the success report; one described "digging back every single log" to find a silent fallback to a default workflow (R32). Asked whether they replicated the key result, a further 39% reported partial replications whose metrics diverged from the paper, the discrepancies that reproduction is meant to make students explain.

Unplanned mismatches. Many of these reconciliations began with gaps in the version students used: it accepted requests beyond its abilities and reported success anyway (§2). We do not claim that catching these requests was by design, but the responses show students using evidence to question the platform's report (§5 returns to this).

3.5 Understanding

In the end-of-project survey, students retrospectively rated their understanding of the replicated paper before starting and after finishing. Understanding rose from 3.07 to 3.93, a mean gain of

Table 2: Qualitative coding of free-text responses ($n = 46$, counts per respondent), reported where a category clears the pre-stated $\geq 10\%$ ($\geq 5/46$) threshold. The automated-execution category includes students who noted there was no measurement instrument to build.

Category	n	%
What helped		
Automated execution and measurement	13	28
Natural-language request lowered the barrier	12	26
Natural language shifted thinking to higher-level design	8	17
Automation enabled rapid iteration	6	13
What did not work		
Environment setup, OS, or kernel barriers	21	46
Silent failures, poor error reporting	7	15
Intent-capability mismatch (wrong run executed)	7	15
Black-box opacity (cannot verify configuration)	5	11

+0.87 (95% CI [0.61, 1.13]; paired $t = +6.67$; Wilcoxon $p < .001$; Cohen's $d_z = 0.98$). The gain is uniform in direction: 30 of 46 students improved and none declined (Figure 3a); §5 weighs what such self-ratings can show. Asked how much the platform helped them learn relative to reading the paper alone, 93% report a positive effect, 20% calling it essential and 52% saying it helped significantly; two learned more from the paper alone (Figure 3c). Asked which concepts the task helped them understand, students most often chose congestion control (83%) and network measurement (67%).

Pre-ratings moderately predict post-ratings (Spearman's $\rho = 0.44$); we detected no link between prior experience with network experiments and the gain ($p = 0.29$) or between Python proficiency and usability ratings ($p = 0.78$).

3.6 What Did Not Work

The same survey records where the version students used fell short: environment setup, error reporting, and verifying what ran. Ease of use (3.2, mean 3.30) and confidence (3.3, mean 3.13) sit at neutral, and "I needed a lot of help before I could get going" (3.4) is the one item on the unfavorable side ($t = -3.23$). Mental demand stayed high (6.30/10), which the reasoning task is meant to be (§5); the rough edges show instead in frustration (6.13/10) and a mixed net recommendation (mean 6.91/10; net promoter −20, more detractors than promoters). The reported friction centers on setup and transparency: 46% of responses cite environment setup, including a software limitation that left BBR unavailable on Macs and blocked the CUBIC-versus-BBR comparison for several groups; 15% cite silent failures with weak error reporting; 15% cite cases where the platform ran a different experiment; and 11% cite not being able to verify what the platform actually did (Table 2). One student called it "a nightmare getting error logs out of the platform" (R42); another spent a long time before tracing failures to the virtual machine rather than their own experiment design (R29).

Two kinds of failure. These failures have different fixes. Some are failures of the tool: the version students used accepted requests it had no way to run, such as one that silently ran a default workflow instead, and reported success. Pramana's design checks each request before running it; the current implementation refuses any algorithm beyond its 15, though a remote sender's algorithm still goes unset without a warning (§2). Others are limits of the real network: a student cannot choose the remote server's algorithm, and CCAnalyzer depends on exactly that limit (§3.4). Because the respondents skew toward strong students, a less-prepared cohort would likely catch fewer of these mismatches, so checking requests before the run matters most for weaker students, who would otherwise have to dig through logs to notice one.

Sensitivity. Excluding the two non-users ($n = 44$) leaves the main results essentially unchanged: "saved time" rises to $t = +12.88$ and "improved ability" to $t = +9.89$.

4 Related Work

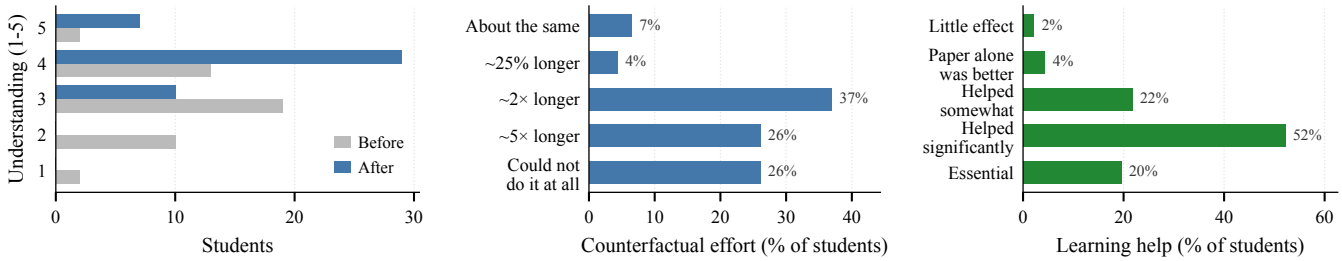
Hands-on networking pedagogy. Reproducing published results is an established way to teach networking. Yan et al. [22] have students reproduce the findings of real networking papers and report that recreating an experiment deepens understanding of its ideas, but that students spend more time rebuilding the setup than analyzing results. Wolosewicz et al. [21] move a graduate course onto a shared research testbed so every student works in the same environment, yet students still stall on onboarding steps such as access keys and resource allocation. In both courses, generating the data stood between the students and the analysis.

Active and experiential learning. Removing this barrier is beneficial, because hands-on experiments are a form of active learning, and active learning improves student outcomes. Prince [18] reviews the evidence for active learning in engineering education, and across 225 STEM studies Freeman et al. [11] report that it raises exam performance and that students in traditional lectures fail about 1.5× as often. The pedagogy is well established; what limits it in networking is the cost of each hands-on experiment.

Generative AI in computing education. Generative AI now bears on that hands-on work, and much of the existing literature treats it as a threat to assessment. Finnie-Ansley et al. [10] show that Codex, an AI code-generation model, outscores most students on introductory exam questions, which undermines standard assessments; Becker et al. [2] warn that ready answers can give students a false sense of proficiency; and Ramirez Osorio et al. [19] find that unsupervised use of AI tools leaves students with weaker skills and mental models. A constructive line uses AI as a tutor: the CS50 "duck" answers student questions and guides them toward their own solutions with pedagogical guardrails [16]. With Pramana, a language model drafts the experiment's specification, Pramana runs it, and the student interprets the measurements; the instructor sets the task, so the use is structured and supervised, unlike the unsupervised use that prior work found harmful. We are not aware of another system used in courses that lets a student generate realistic network data from a request written in natural language, and that ability makes empirical work cheap enough to assign as a regular course activity.

5 Reflection and Adoption

Checking fluent answers. The result that best captures Pramana's role is the *reconcile* step of the loop: the cases where students caught a wrong answer by checking their prediction against the



(a) Understanding, before vs. after. **(b) Counterfactual effort.** **(c) Learning help vs. the paper alone.**

Figure 3: Student outcomes, with survey items in parentheses. (a) Self-rated understanding of the replicated paper before (2.1) and after (6.1). (b) Counterfactual effort (6.4). (c) Learning help relative to reading the paper alone (6.2).

measured system (§3.4). A generated explanation and a confidently mislabeled experiment share a weakness: both look plausible, and neither has been checked. Pramana makes that check cheap enough to run within a course project.

Threshold and challenge. Pramana lowers the threshold to experiential learning and leaves the challenge in place. The task stayed mentally demanding (6.30/10): students still chose what to test and reasoned about what the measurements meant, the critical-thinking work the course set out to teach (§1). Pramana should come after at least one assignment in which students set up an experiment by hand, such as building a Mininet topology, so that they keep that skill before they use Pramana to run many more experiments.

Beyond this assignment. We evaluated one assignment, a paper replication on a research-grade problem that is demanding for an undergraduate course. The predict-measure-reconcile loop applies to other topics a networking course teaches, within the scope of Pramana's current implementation [9]. A bufferbloat assignment varies the buffer size and measures how queueing delay grows with it (bufferbloat is the delay that oversized buffers add). A queue-management assignment compares a drop-tail queue (it drops arriving packets only when full) with CoDel (it drops packets early to keep queueing delay low) under the same load; some students in our course already ran this comparison for bonus credit. A congestion-control assignment compares the kernel's algorithms across link capacities and delays. An assignment on application quality, such as how a video call's frame rate falls as competing traffic grows, needs one more step. Collectors for a call's frame rate, jitter, and loss and for a video stream's dropped frames run alongside Pramana today, but not yet through its requests. Assignments on networks with several links or on wireless networks need new engineering first. For an assignment with a handful of settings like ours, a few dropdown menus would have sufficed. Natural language is the interface that scales as an instructor adds settings; this study does not show that, but Pramana's design supports it: its grammar already covers settings our assignment never used, such as competing traffic, and each request compiles into a structured specification much like the one the dropdown menus would produce. A further step is *adversarial reproduction*: students search for the conditions under which a replicated method fails, for example a queue discipline or bursty competing traffic that changes the queue pattern CCAnalyzer relies on. Ideally, the search is systematic and needs few runs, instead of an exhaustive sweep. Building the loop into every concept a course teaches is still work in progress, and an instructor adopting it for a new topic should expect to adapt the task and its scaffolding.

Scope of the evidence. This is one course at one institution with a self-selected, consenting cohort, and the data come from a single end-of-project survey. The pre/post ratings are retrospective and self-reported, and no control group ran the project without Pramana. We therefore read the results as a reason to run a controlled study rather than proof that Pramana caused the gains, and the problems in §3.6 bound the reading further. For the next offering, we plan a lightweight objective measure: a five-question quiz on CCAnalyzer before and after the project, or grading the free-response report with the course's existing rubric, ideally with a control group that does the project by hand.

Next steps. Some of the most instructive aspects of the study were accidental: students found that Pramana had run an experiment they had not requested, and learned from it (§3.4). Pramana's design makes that comparison a deliberate step: it rejects a request that no system can run and previews the parsed configuration, so the student checks it against the intended experiment [9]. The full check waits on engineering: each system must publish its capability file. A clarification step that asks the student about a missing value instead of filling in a default, and agents that choose a value from the request's context, are in development, and a hosted deployment would remove the setup friction 46% of students hit (§3.6).

Conclusion. Pramana made generating controlled data cheap enough for a 70-student course to run the predict-measure-reconcile loop within one project, and 15% of the 46 students surveyed caught Pramana's own wrong runs by checking the data. With that loop, a networking course can teach students to explain why a network acts the way it does and to check a model's fluent answer against data they generated and reasoned through.

Acknowledgments. We thank the anonymous reviewers and the students who took part in the study. Researchers at UCSB have been supported by NSF awards 2443777 and 2323229 and by research gifts from Cisco and Google. We used generative AI tools, including Claude (Anthropic), to help draft and edit the text of this paper; the authors reviewed and revised all of it and take full responsibility for its content.

References

[1] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S. Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. 2019. FABRIC: A National-Scale Programmable Experimental Network Infrastructure. *IEEE Internet Computing* 23, 6 (2019), 38–47. doi:10.1109/MIC.2019.2958545

[2] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard – Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE TS)*. doi:10.1145/3545945.3569759

[3] Roman Beltiukov, Wenbo Guo, Arpit Gupta, and Walter Willinger. 2023. In Search of netUnicorn: A Data-Collection Platform to Develop Generalizable ML Models for Network Security Problems. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3576915.3623075

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2016), 20–53. doi:10.1145/3012426.3022184

[5] Jaber Daneshamooz, Sanjay Chandrasekaran, and Arpit Gupta. 2026. A First-Principles Framework for Networking Education. In *ACM SIGCOMM 2026 Education Workshop: Networking Education for the AI Generation (A4NE)*. https://a4ne-workshop.github.io/papers/a4ne26-paper2.pdf Non-archival.

[6] Jaber Daneshamooz, Satyandra Guthula, Jessica Nguyen, William Chen, Sanjay Chandrasekaran, Ankit Gupta, Arpit Gupta, and Walter Willinger. 2026. NetReplica: A Programmable Substrate for Bottleneck-Centric Network Data Generation. arXiv:2507.13476 [cs.NI] https://arxiv.org/abs/2507.13476

[7] Jaber Daneshamooz, Satyandra Guthula, Jessica Nguyen, William Chen, Ankit Gupta, Walter Willinger, and Arpit Gupta. 2026. NetReplica: A Bottleneck-Regime Substrate for Reproducible Network Data Generation. In *Proc. Network Traffic Classification – Reloaded Workshop (NTC-R)*. doi:10.1145/3843199.3857224 To appear.

[8] Jaber Daneshamooz, Eugene Vuong, Laasya Koduru, Sanjay Chandrasekaran, and Arpit Gupta. 2025. NetGent: Agent-Based Automation of Network Application Workflows. arXiv:2509.00625 [cs.NI] NeurIPS 2025 Workshop on Machine Learning for Systems (ML4Sys). https://arxiv.org/abs/2509.00625.

[9] Jaber Daneshamooz, Eugene Vuong, Alagappan Ramanathan, Manni Moghimi, Haarika Manda, Satyam Kumar, Snithik Thode, Satyandra Guthula, Sylee Beltiukov, Dongsu Han, Tarun Mangla, Sangeetha Abdu Jyothi, Walter Willinger, and Arpit Gupta. 2026. Pramana: A Composable, Domain-Specific Backend for Empirical Networking Research. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, Salt Lake City, UT, USA. doi:10.1145/3847290.3859953 To appear.

[10] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proc. Australasian Computing Education Conference (ACE)*. doi:10.1145/3511861.3511863

[11] Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences (PNAS)* 111, 23 (2014), 8410–8415. doi:10.1073/pnas.1319030111

[12] Jim Gettys. 2011. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing* 15, 3 (2011), 96. doi:10.1109/MIC.2011.56

[13] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74. doi:10.1145/1400097.1400105

[14] LangChain. 2024. LangGraph: Building Stateful, Multi-Actor Applications with LLMs. https://github.com/langchain-ai/langgraph.

[15] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*. doi:10.1145/1868447.1868466

[16] Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. 2024. Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE TS)*. doi:10.1145/3626252.3630938

[17] Larry L. Peterson and Bruce S. Davie. 2021. *Computer Networks: A Systems Approach* (6th ed.). Morgan Kaufmann.

[18] Michael Prince. 2004. Does Active Learning Work? A Review of the Research. *Journal of Engineering Education* 93, 3 (2004), 223–231. doi:10.1002/j.2168-9830.2004.tb00809.x

[19] Valeria Ramirez Osorio, Angela Zavaleta Bernuy, Bogdan Simion, and Michael Liut. 2025. Understanding the Impact of Using Generative AI Tools in a Database Course. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE TS)*. doi:10.1145/3641554.3701785

[20] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. 2024. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In *Proc. ACM SIGCOMM*. doi:10.1145/3651890.3672255

[21] Alexander Wolosewicz, Prajwal Somendyapanahalli Venkateshmurthy, and Nik Sultana. 2025. Experience Report: Using the FABRIC Testbed to teach a Graduate Computer Networking course. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE TS)*. doi:10.1145/3641554.3701923

[22] Lisa Yan and Nick McKeown. 2017. Learning Networking by Reproducing Research Results. *ACM SIGCOMM Computer Communication Review* 47, 2 (2017), 19–26. doi:10.1145/3089262.3089266