Pramana: A Composable, Domain-Specific Backend for Empirical Networking Research

Jaber Daneshamooz
UC Santa Barbara

Eugene Vuong
UC Santa Barbara

Alagappan Ramanathan
UC Irvine

Manni Moghimi
UC Santa Barbara

Haarika Manda
UC Santa Barbara

Satyam Kumar
IIT Delhi

Snithik Thode
IIT Delhi

Satyandra Guthula
UC Santa Barbara

Sylee Beltiukov
UC Santa Barbara

Dongsu Han
KAIST

Tarun Mangla
IIT Delhi

Sangeetha Abdu Jyothi
UC Irvine

Walter Willinger
Northwestern University

Arpit Gupta
UC Santa Barbara

Abstract

Empirical networking research often tests a hypothesis with data the study itself generates, so accelerating this research requires shortening the time from forming a hypothesis to generating that data. Consider whether a bulk BBR download fairly shares its bottleneck with a Google Meet call. Testing this requires configuring a realistic bottleneck link, running the BBR transfer and the Meet call together, and collecting the call's quality metrics. Today a researcher builds this setup from scratch for each new idea; this delay from idea to data is the *ideation-to-data gap*. AI agents already propose hypotheses faster than researchers can generate data to test them. Closing the gap needs a backend that generates the data for each new hypothesis on demand, without a new setup.

We envision Pramana, a composable *backend* for networking research designed as a thin waist, with research intents above it and execution substrates below. At its core, the *intent specification* separates the data a study needs (the *intent*) from where that data is generated (the *substrate*) and how it is produced (the *mechanism*), so one specification can target any substrate. We build a corpus of 255 data-generation intents from 66 published papers and find that the intent specification expresses all of them, while no existing tool covers more than 11%. Our proof-of-concept implementation covers 27% of these intents. For CCAnalyzer, one of the 66 papers, the implementation regenerated the data for one trial of each network setting and congestion-control algorithm in the paper's main experiment. We lay out what implementing the full design requires and why it needs a community effort.

CCS Concepts

• **Networks** → **Network performance evaluation**; *Network measurement*; *Network experimentation.*

Keywords

Data generation, network experimentation, intent specification, reproducibility, research acceleration

ACM Reference Format:
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 *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859953

1 Introduction

Empirical networking research tests hypotheses against data, so its speed depends on how long it takes to go from an idea to data that can confirm or refute it. AI agents now propose hypotheses far faster than anyone can test them [19, 27, 30, 39, 47], and testing them has become the bottleneck [22].

An empirical study draws on four recurring activities [24]: re-creating a published result, extending it until it breaks, fixing what broke, and establishing a new one. Each activity needs data, and a researcher requests that data by issuing an *intent*. In networking, an intent specifies what data to generate, while a *substrate* specifies where to generate it (e.g., a laptop, the cloud, or a testbed). An intent names the bottleneck's capacity and queue, the cross-traffic pressure on it, and the *workflows*, the programs that run on its end hosts. A



This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859953

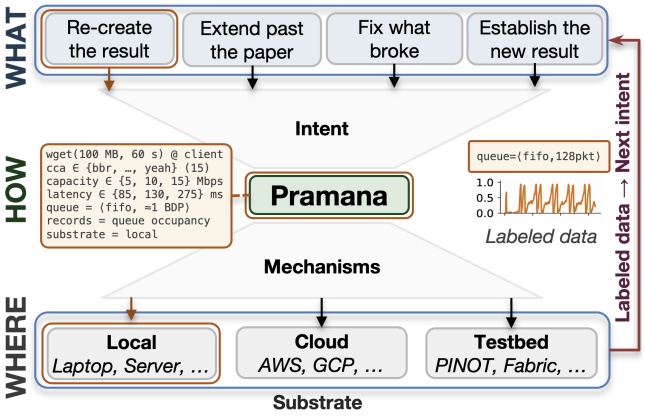


Figure 1: Data-generation thin waist.

workflow either runs a command-line tool, such as an `iperf3` bulk download, or is an *application workflow* that drives a real application through a session, such as Google Meet. Workflows send traffic through the bottleneck, record what the study measures, or do both. A packet capture is a *telemetry workflow*, and so is the Meet call when it reports its video statistics. Each workflow that sends traffic carries the delay and loss of its own path and the congestion control of its sender. Every intent fills these same fields with different values. Stating an intent takes a few sentences, but realizing it means writing code against several tools. A few groups amortize that cost with verticals built for one domain, such as Pantheon [49] for congestion control. Intents span too many domains to build one vertical for each, so most researchers rebuild the setup per study. We call the resulting delay between an idea and its data the *ideation-to-data gap*.

Over the past fifteen years, related efforts have narrowed this gap. Mininet [20, 25], netUnicorn [4], NetReplica [11, 12], and NetGent [14] each turned one concern (topology, node mapping, bottleneck conditions, and application workflow) into a reusable, replayable artifact. Yet each tool by itself is insufficient: NetReplica cannot drive an application, and NetGent cannot shape a bottleneck. Running a real Google Meet call against a BBR bulk download over one shared, shaped bottleneck, for instance, takes several of these tools at once. The gap persists because these tools make a researcher write the intent and the substrate into the *mechanism*, the code that produces the data. Composing them still forces a researcher to re-encode the intent in one platform's mechanism, so the intent is tied to that platform and must be rewritten for the next one. A thin waist (Figure 1) would place one specification between research intents and execution substrates and compose these capabilities behind it.

Pramana (Sanskrit for *evidence*) designs this thin waist around the *intent specification*. Pramana applies *progressive disaggregation* [12], separating coupled parts one step at a time so that each separated part can then be specified and changed on its own. The Internet's designers took such a step when they separated TCP from IP, so that applications TCP did not suit, such as real-time speech, could use IP without TCP's reliable delivery [9]. Pramana composes NetReplica's bottleneck conditions, in which cross-traffic recorded on production networks replays through an emulated bottleneck, with NetGent's deterministically replayed application workflows (e.g., YouTube, Zoom, and Twitch). It also borrows netUnicorn's separation of what an experiment runs from where it runs. Pramana adds one more separation: it moves both the what and the where into the specification, out of the mechanism. A researcher can then change the bottleneck, a workflow, or the substrate without changing the other two. Because each of the four activities changes some of these parts and keeps the rest, one specification is designed to serve all four activities on any substrate. A researcher writes the intent in natural language, and a model compiles it into the specification. Only that step uses a language model. Everything below it composes and replays existing artifacts and logs each run's requested conditions.

This paper makes four contributions:

- **The intent corpus.** We build the first corpus of research intents, 255 intents from 66 measurement papers published over a decade. It grounds the intent specification in real demand, can be reused and extended by others, and is the yardstick against which we measure the design, today's implementation, and each specialized tool (§4).
- **The abstraction.** We introduce the *intent specification*, a domain-specific language in which a researcher states the intent (*what*) and the substrate (*where*); the backend supplies the mechanism (*how*). Separating the *what* and the *where* from the *how* is a step of progressive disaggregation that NetReplica, NetGent, and netUnicorn do not take. The language expresses every in-scope intent in the corpus. Because every field maps to a capability of an existing system, we also use it to score what each tool and today's implementation can generate (§3, §4).
- **The architecture.** We design an architecture that composes NetReplica and NetGent behind versioned capability files, in which each service declares what it supports, and that, like netUnicorn, keeps a workflow separate from the node it runs on (§3).
- **A demonstration on published intents.** We test the backend on the corpus intents instead of ones we designed. We measure coverage across the corpus against each specialized tool and regenerate the data of CCAnalyzer's [43] main experiment, one trial per pairing of setting and algorithm.

The intent corpus and the related reproducibility artifact are publicly available [38].

2 Background and Motivation

The four activities. Each study re-creates, extends, fixes, or establishes a result [24], and a recent pair of papers from our corpus (§4) does all four: CCAnalyzer [43] identifies a flow's congestion control algorithm by how it fills a bottleneck queue, reaching 100% accuracy across 15 algorithms; its successor, CClinguist [26], *re-creates* that classifier as a baseline. It *extends* the classifier until it breaks, as CCAnalyzer's accuracy drops to 84.3% at a latency outside its fixed profiles. It *fixes* the break by generating those profiles automatically, recovering accuracy to 91.7%, and it *establishes* new labels on algorithms outside CCAnalyzer's set, including the first ECN-based ones. Each activity needs its own data, and CClinguist's new labels are what a later study would re-create.

Intent, substrate, mechanism. These activities need different data, but each needs controlled data on demand, with every sample labeled by the conditions that produced it (its ground truth). Each request names an intent and a substrate and relies on a *mechanism* (§1). The intent and substrate are easy to state, but the mechanism means standing up the bottleneck, driving the real application, imposing the cross-traffic, and wiring the telemetry, then repeating it all for the next substrate.

The thin waist. An intent is built from bottleneck conditions and workflows, telemetry included, and a node mapping places its workflows on a substrate's nodes. These parts recur across studies and substrates, so a shared backend that builds each once and reuses it for any specification on any substrate shrinks the ideation-to-data gap. Such a backend is the *thin waist* [4] of Figure 1, a narrow layer between many intents and many substrates.

The requirements. Table 1 lists the seven requirements this backend must meet and scores each system against them. The first, *flexibility*, has three dimensions, and the other six (*portability*, *usability*, *fidelity*, *evolvability*, *robustness*, and *scale*) make the backend practical for researchers to run. Each listed predecessor meets at most one dimension of flexibility and exposes a tool-specific interface, and only our composed design meets all seven.

Progressive disaggregation. Each tool in Table 1 applies progressive disaggregation [12] to one part of an experiment. Mininet [20, 25] separates the topology from the hardware, and netUnicorn [4] separates an experiment's tasks from the nodes that run them. NetReplica [12] separates the bottleneck's static settings from its dynamic cross-traffic, and that cross-traffic from the network that recorded it. NetGent [14] breaks an application workflow into reusable states that compose into deterministic replays. Pramana's design separates the intent and its substrate from the mechanism: each field names a capability that a service declares, and the backend picks that service, so a specification names no mechanism.

Requirement	Mnt	nUn	nRp	nGt	Pramana
Flexibility — the wide top					
Diverse network env	~	✗	✓	✗	✓
Complex app workflow	✗	✗	✗	✓	✓
Node mapping	✗	✓	✗	✗	✓
Portability across substrates	✗	✓	✗	✗	✓
Usability of the intent interface	~	~	~	~	✓
Fidelity of spec to intent	✗	✗	✗	✗	✓
Evolvability of subsystems	✗	✗	✗	✗	✓
Robustness under failure	✗	✗	✗	✗	✓
Scale to research-time sweeps	✗	✗	✗	✗	✓

Table 1: Requirements for a data-generation thin waist and the systems that meet them. ✓: native ~: partial ✗: absent. Mnt: Mininet, nUn: netUnicorn, nRp: NetReplica, nGt: NetGent. Pramana's column scores the design; §3 states what the implementation has.

A motivating example. Prudentia's [31] intent, also in our corpus (§4), simultaneously needs the application, the bottleneck, and the mapping of its workflows to nodes. The *Google Meet* call (WebRTC, up to 1.5 Mbps) plays *"the reference Big Buck Bunny video"* and contends with the BBR download on one shared drop-tail bottleneck of *"approximately 4×BDP"* at 8 or 50 Mbps with a normalized 50 ms RTT, and every sample carries *"video resolution, frames per second, freezes per minute, and high-delay packets"* (those past the ITU 190 ms bar) [31]. NetReplica shapes the bottleneck, replays cross-traffic, and runs client commands, but it cannot join a call or any other interactive session, so the call's quality never appears. NetGent drives the call but has no contended bottleneck to put it behind, and Mininet runs a real client but cannot replay recorded cross-traffic. The researcher therefore builds a per-paper orchestration script that puts both workflows behind one shared bottleneck. NetReplica sets the drop-tail bottleneck, NetGent runs the call, and the script sets BBR on the download's sender through `sysctl`, as our CCAnalyzer script did (§4). The script also sets each workflow's delay, starts the download, the capture, and the call under study, and labels each sample with its conditions. The intent leaves settings implicit: the 4×BDP queue is 200 KB at 8 Mbps and 1.25 MB at 50 Mbps, and a 50 ms RTT means adding whatever delay the substrate's path lacks. Writing the script requires learning each tool's interface and parameters, and every later study repeats that work. To move the study to AWS or replace the call with a Twitch stream, the researcher rewrites the script. We use Prudentia's intent as the running example of what the design must express, and CCAnalyzer's intent as the one whose data we regenerate (§4).

What the abstraction adds. Pramana is designed to replace this described script with one specification. In its grammar (Listing 1), the Meet call is the Foreground, the workflow under study, and the BBR download is a second workflow

$$
\begin{aligned}
\mathit{Evidence} &= \mathit{collect}(\mathit{ExperimentSet}) \\
\mathit{ExperimentSet} &= \mathit{compile}(\mathit{Intent}, \mathit{Answers}) \mid \text{user-written } \mathit{Spec} \\
\mathit{Experiment} &= \langle \mathit{Foreground} \otimes \mathit{Regime} \rangle \times \mathit{iterations} \\
\mathit{Foreground} &= \{\ \mathit{Workflow}_i \ @\ \mathit{node}_i\ \} \\
\mathit{Workflow} &= \mathit{CLI}(\mathit{app}) \ \triangleright\ \mathit{NFA} \text{ with } \mathit{path}, \mathit{cca} \\
\mathit{path} &= \langle \mathit{latency}, \mathit{jitter}, \mathit{loss}, \mathit{reorder}, \mathit{dup} \rangle \quad \text{(per workflow)} \\
\mathit{cca} &= \langle \mathit{algo}, \mathit{stack}, \mathit{version}, \mathit{params}, \mathit{flags}, \mathit{build} \rangle \\
\mathit{Regime} &= \mathit{Static}\langle \mathit{capacity}\downarrow\uparrow, \mathit{queue} \rangle \otimes \mathit{Dynamic}\langle \mathit{pressure} \rangle \\
\mathit{queue} &= \langle \mathit{discipline}, \mathit{size}, \mathit{params}, \mathit{ecn}, \mathit{mode} \rangle \\
\mathit{pressure} &= \mathit{replay}(\mathit{ctp}) \mid \mathit{load}(\mathit{Workflow}^* \mapsto \mathit{Node}^*) \mid \text{both} \\
\mathit{iterations} &= \mathit{int} \mid \mathit{until}(\mathit{sufficient}(\mathit{signal}, \mathit{precision}), \mathit{min}, \mathit{max}) \\
\mathit{identity}(e) &= H(\mathit{workflow@sha}, \mathit{params}, \mathit{path}, \mathit{cca}, \mathit{static}, \mathit{dynamic}) \\
\mathit{collect} &= \mathit{deploy}; \mathit{prepare}; \mathit{verify}; \mathit{run}; \mathit{publish}
\end{aligned}
$$

Listing 1: The intent-specification grammar.

that the Regime, the imposed conditions, loads onto the same bottleneck as Dynamic pressure, with a separate path and congestion control for each workflow. An orchestration script over the same tools reveals a missing capability (e.g., BBR for a remote sender) only in the data it returns. The design checks a specification against the declared capabilities before anything runs. With each field tied to a capability, one grammar scores every tool against the same 255 intents from 66 papers (§4).

3 Pramana: A Data-Generation Thin Waist

We first describe the design; the last two paragraphs state what the implementation runs today.

The intent specification. Pramana's abstraction, the *intent specification*, composes the artifacts that the tools of §2 disaggregated. An experiment is then a *composition* of existing artifacts, a *Foreground* of workflows running over a *Regime* of imposed conditions, and the researcher only states the intent. We write the abstraction as a domain-specific language whose grammar Listing 1 gives. A model translates the researcher's natural-language intent, written by hand or extracted from a paper, into this language.

The intent-specification grammar. Listing 1 defines how evidence is produced: *compile*() produces a specification from the intent, the middle rows define its structure, and *collect*, defined in the last row, is the mechanism that realizes it. The bottleneck (*Regime*) is precise and shared, and NetReplica imposes it once for all workflows. Each *path* is approximate and belongs to one workflow, as does its sender's congestion control (*cca*). An *NFA* is a nondeterministic finite automaton over an application session, and a *ctp* is a recorded cross-traffic trace that NetReplica replays. The *until* clause stops a run once a signal the application or tool reports is statistically sufficient, so no stopping rule depends on a computed result.

A specification denotes a distribution over condition-labeled datasets, each sample tagged with the condition vector it realized. These precise semantics let the backend

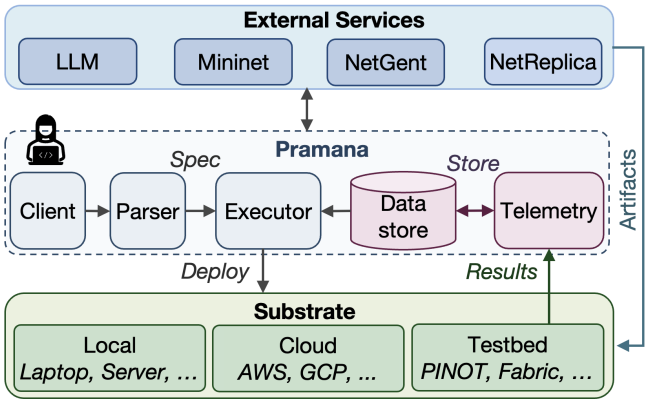


Figure 2: Pramana's architecture.

decide before any run whether the declared services can realize a specification (its *realizability*). Because a specification can name only what some service has declared, realizability reduces to a satisfiability check over those declarations that runs in polynomial time, whereas for open-ended code the same question is undecidable [3, 6]. The semantics also keep everything below *compile* deterministic, since *compile*() is the only step that may invoke a model. Determinism covers how an experiment is built and driven. Live traffic still varies, as a live call differs between runs, and each sample's label records the conditions that run actually realized. The intent specification stops at data generation, because pulling metric extraction into it would fix a set of metrics, narrow the intents a researcher can express, and require the model to reason past *compile*, breaking the determinism below it. Pramana produces the data for the confirmed intent, and the researcher decides whether it supports the hypothesis.

Compiling an intent. For the Prudentia intent of §2, Pramana compiles the researcher's description clause by clause into the grammar of Listing 1. The Meet call, a browser session that plays the reference video, becomes the *Foreground*, a *Workflow* pinned to a node. The shaped bottleneck becomes the *Regime*, whose *Static* part sets the intent's two capacities and the *queue* (a drop-tail FIFO of ≈ 4×BDP). *Dynamic pressure* loads a second *Workflow*, a bulk `iperf3` download whose sender runs *cca* = BBR, onto the bottleneck. Each workflow's *path* carries the normalized 50 ms latency.

The architecture. In the design (Figure 2), the model compiles the intent into a specification, and the Parser checks every field against the capabilities NetGent and NetReplica declare, confirms the translation with the researcher, and gives each experiment a content-hash identity. The Executor is designed to realize each experiment on any substrate: local (a laptop or desktop), cloud (AWS [1], GCP [16]), or a testbed (PINOT [41], RIPE Atlas [34], FABRIC [2], CAIDA Ark [5]). NetGent and NetReplica ship their artifacts straight to the

substrate, and the nodes report results to Telemetry, which stores them.

The design choices. In the design, Prudentia's specification keeps the Meet workflow, the shaped bottleneck, and the node mapping as separate artifacts. A researcher can therefore swap the call for a Twitch stream without touching the bottleneck (*flexibility*) and move the run from a laptop to AWS by changing only the node mapping (*portability*). A researcher expresses intents in natural language; the model only fills in a fixed grammar, so its output can be checked field by field (*usability*). Every service lists its capabilities in a versioned capability file, and the parser rejects any field that names a capability no file lists. It then echoes the translation in plain language for the researcher to confirm, so the researcher can spot mistranslated values and workflows that the check cannot see (*fidelity*). Because these files are versioned, NetGent can add a workflow, such as one that records a new metric, without changing the grammar (*evolvability*). A specification names every input, so each experiment's identity is a hash of those inputs (Listing 1). The backend is crash-only: after any crash, it restarts and reruns only the experiments with no stored result (*robustness*). A failed run keeps its data and a failure label. The Executor fences off a worker that loses contact with it, so that worker cannot disturb another experiment's conditions. When an application's interface changes and a NetGent workflow no longer matches it, NetGent regenerates the workflow offline, and once the researcher confirms it, the workflow runs under a new hash. The hash also lets the backend skip identical experiments, and a persistent worker pool amortizes setup across a sweep (*scale*).

The implementation today. Our proof of concept connects to NetReplica and NetGent through their service interfaces, runs in local containers and on AWS, and regenerated CCAnalyzer's traces through a short script that set each run's parameters (§4). It offers a single shaped bottleneck for each run's traffic, 22 NetGent application workflows, and 15 kernel congestion-control algorithms. It shapes no second link, though 38 in-scope intents need several (§4). For CCAnalyzer's runs, the implementation records the requested conditions and the queue trace, with its own bandwidth and latency probes off during capture so that their traffic does not enter the trace. The implementation has none of the design's robustness and scale mechanisms, so each experiment waits 44 s for a fresh worker, a crashed campaign cannot resume, and identical experiments run again. Under bursts of concurrent workers, the single telemetry service limits capture-heavy studies.

Running the Prudentia intent. For Prudentia's intent, the compile step fills each unstated field with a default and reports that interpretation back with the results. NetReplica shapes the drop-tail bottleneck on the substrate worker. NetGent, however, runs one workflow per run, either from its library or, for a shell tool such as `iperf3`, generated on request. So the Meet call and the `iperf3` download have not run together on the link, and the implementation reports none of the call metrics that §2 lists. BBR can be set two ways, through the kernel and per socket. The implementation applies both, and only for a sender inside the substrate. It rejects an algorithm that is not included in the 15 offered ones or that the kernel cannot load, and it fails when asked for an unknown queue discipline. The only combination of fields it checks is a queue setting that contradicts the buffer size, so a request for BBR on a remote sender runs without BBR and without a warning. The design's check needs a capability file from each service, and no service publishes one today.

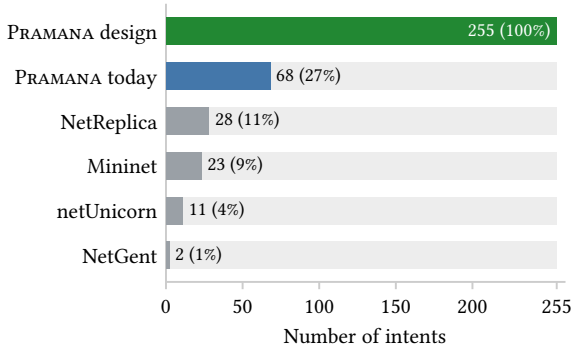


Figure 3: In-scope intents that each system covers.

> Through a FIFO bottleneck sized near one BDP, at 5, 10, or 15 Mbps and 85, 130, or 275 ms of RTT, download a 100 MB file with `wget` for 60 s from a server running each of the 15 Linux congestion-control algorithms, five samples each, and record when packets enter, leave, and drop at the bottleneck.

$$\begin{aligned}
\textit{Experiment} &= \langle \textit{Foreground} \otimes \textit{Regime} \rangle \times 5 \\
\textit{Foreground} &= \{\ \texttt{wget}(100\,\text{MB}, 60\,\text{s})\ @\ \textit{client},\ \texttt{http_server}\ @\ \textit{server} \\
&\qquad \text{with } \textit{cca} \in \{\text{bbr, bic, cdg, cubic, …, yeah}\}\ (15)\ \} \\
\textit{Regime} &= \textit{Static}\langle \textit{capacity} \in \{5, 10, 15\}\ \text{Mbps},\ \textit{queue} = \langle \text{fifo}, \approx 1\,\text{BDP} \rangle \rangle \\
\textit{path} &= \langle \textit{latency} \in \{85, 130, 275\}\ \text{ms RTT} \rangle \\
\textit{records} &= \text{queue occupancy at the bottleneck}
\end{aligned}$$

Listing 2: An intent and its specification. Top: the intent we extract for the runs that test CCAnalyzer's classifier in its main experiment (1 of its 11). Bottom: the specification it compiles to (Listing 1): 9 network settings × 15 algorithms × 5 samples = 675 runs.

4 Evaluation

We ask three questions: (i) whether the grammar expresses every intent we collected, and how its coverage compares with existing specialized tools, (ii) how much of the corpus the implementation covers today, and (iii) what regenerating one published dataset costs.

Experimental setup. Our unit of analysis is the *experiment*, the data collection behind one measured result. We sample papers by CCS concept and keyword, such as *Network measurement*, *Transport protocols*, and *congestion control* across SIGCOMM, IMC, and NSDI over the past decade. We build the corpus by hand from these papers, identifying each paper's results and, for each result, the experiments that produced it, and record them in a Result×Experiment matrix. We write down each experiment's intent as natural-language text, the way the paper states it. An intent is in scope when its experiment generates its own traffic, which we call *endogenous* data generation, excluding simulation, passive capture, and offline computation. Of 387 extracted intents, 255 (66%) are in scope and form the in-scope corpus, one intent per experiment. CCAnalyzer [43], for example, reports 26 data-bearing figures and tables, and they come from 11 experiments. Its headline result (100% classification accuracy) rests on one of them, 675 runs over nine network settings and 15 congestion-control algorithms. Prudentia [31] reports its findings from 8 experiments, one of which pits a Meet call against competing downloads, the intent of §2. Listing 2 shows CCAnalyzer's intent as we extract it and its specification. A grammar axis is a group of related fields, such as the bottleneck or the application workflow, and most intents (192, 75%) need two or more. For each in-scope intent, one reader records its needs from the paper and a second checks every entry against the same text; a deterministic rule then compares them with a capability sheet that cites each system's code.

Counting rule. We score every system the same way, Pramana's implementation included. A system covers an intent when its released code, as of the HotNets submission, can create everything the intent needs through the system's own interface, without code written for that intent. Standard command-line tools, such as `iperf3`, count for every system that runs commands on a host. Software that must be compiled for the intent, including a kernel other than the stock one, does not count. Driving an application through its interface is a NetGent feature, so it counts for NetGent and for Pramana through NetGent. If the paper emulates an intent's conditions, a system that recreates them covers it wherever the paper ran it, and replaying recorded cross-traffic through a fixed-rate bottleneck recreates a time-varying capacity. If it measures a real network or runs on phones, the system must reach that network or those devices. If the paper leaves a need unstated, such as the algorithm its server ran, we cannot decide the intent for any system and report it separately. Counting an intent does not mean we have run it, and this section generates data only for CCAnalyzer.

Coverage of existing tools. Figure 3 scores each specialized tool against the corpus, and every miss traces to a grammar axis the tool lacks. NetReplica imposes bottleneck conditions but does not drive an application workflow. netUnicorn places an experiment but does not impose its network conditions, and Mininet executes the application in an isolated namespace but cannot drive it as an application workflow or impose trace-based cross-traffic [12, 18]. The best tool, NetReplica, covers 11% of the corpus and Mininet 9%, and the tools miss different axes, each lacking one that another supplies. The grammar is the union of those axes and expresses all 255 in-scope intents from 66 papers.

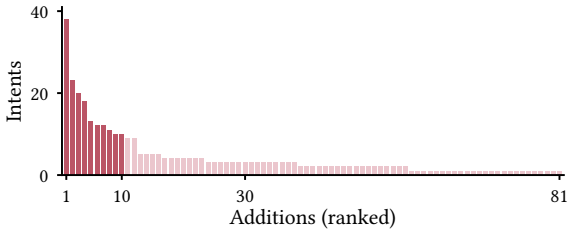


Figure 4: Intents that need each missing addition.

Implementation coverage. Pramana's implementation covers 68 of the 255 intents today (27%). Another 19 are underspecified, so we cannot decide them. Of the 168 intents it cannot cover, 82 need a congestion-control or transport build and 81 need infrastructure it cannot reach (an intent can need both), and half (84) lack more than one addition. They need 81 additions in all, and 56 of them serve more than one intent (Figure 4). The largest is several shaped links for one intent, each with its own capacity, queue, and loss: 30 need two parallel paths for a multipath transport, and 8 need links in series, such as an access link and an interconnect. Mininet builds such topologies but covers 5 of them: of the rest, 26 also lack a software build, such as an MPTCP kernel or a QUIC stack, and 6 a recorded capacity trace on a path. For 12 intents, the shaped links are the only missing addition. Next come two kernels compiled from publicly available code, an out-of-tree MPTCP kernel (23) and a BBRv2 kernel (20), and two connectors that Pramana needs to reach phones (18) and cellular access (13). Added in the order that unblocks the most intents, ten additions raise coverage from 27% to 53%.

Regenerating CCAnalyzer's data. We pick one corpus paper, CCAnalyzer, and Pramana regenerates the queue-occupancy traces its classifier reads in its main experiment, with one trial for each setting and each of the 15 algorithms. In a separate 20-run instrumented benchmark, the compile takes 2.7 s and about 10,000 tokens on a small language model (Gemini 3.1 Flash-Lite). This 2.7 s is the intent-to-specification compile time with artifacts and infrastructure already in place, and it excludes the rest of the researcher's effort. In a user study, 46 students used Pramana to regenerate this result, and 89% reported that they would have needed at least twice the effort to match it by hand [10]. Each run's 60 s measurement and everything after it run unattended. Getting the data *in hand* adds a 71 s telemetry

transfer, which streaming the upload would shorten. In our runs, the script supplied the link parameters, the algorithm, and the download workflow as structured fields that overrode the compile's output. The download's server ran on the host machine, outside the substrate worker, and the driver set that server's algorithm through `sysctl`. A CCAnalyzer author judged the regenerated traces similar to the originals. The sender's segmentation offload also queued packets in bundles of two or more segments, so each queue held at least double its configured number of packets.

5 Related Work

Emulators, testbeds, and verticals. Link and testbed emulators shape a bottleneck's bandwidth, delay, loss, and queue: Dummynet [35] and netem [21] on one link, Emulab [44], ModelNet [42], and DieCast [17] across a topology, Mininet [20, 25] in software, and Mahimahi [29] by record-and-replay. Each shapes the link and leaves the real application and the cross-traffic to the researcher. Verticals such as Puffer [48] and Pantheon [49] fix one application and metric set (adaptive bitrate or congestion control) and accelerated research in that domain. Pramana builds on their fidelity and expresses intents from many domains in one grammar.
Agentic accelerators and language interfaces. Agentic research systems stack hypothesis generation on top, a domain backend in the middle, and raw instruments below. AI Scientist [47] works at the top layer, and systems research suits its propose-and-test loop because a proposed solution can be checked by measuring it [8]. In networking, NetArena scores automation agents against network emulators [51]. None of these systems composes the instruments into a backend that turns an intent into controlled, ground-truth-labeled data. Pramana's compile step draws on semantic parsing and text-to-code generation [7, 50]. Grammar-constrained decoding rules out ill-formed output by construction [15, 37, 45], and a human who confirms the translation catches the errors it cannot [28, 32]. In intent-based networking, the closest work, Lumi, reads its translation of a configuration back to the operator for confirmation [23], and Pramana applies the same check before generating a condition-labeled dataset.

6 Discussion and Conclusion

Checking published evidence. Pramana can help artifact evaluation, whose committees stop at *Functional* or *Reusable* badges. These badges confirm that a paper's code runs but not that its results hold, because re-collecting the data is too costly [36, 46]. Regenerating that data from an intent is the *re-create* activity (§2) applied to another paper. Pramana supplies the data and makes every unstated field an explicit default, possibly unlike the paper's original condition. Whether the data supports the paper's finding remains an analysis question. Airavat [33] generates Internet-measurement analysis pipelines and verifies them against established methodology. Paired with Pramana's regenerated data, it would take a paper's intent to a checked result.
Artifacts. Pramana generates data only from artifacts that already exist. Creating them from a paper's description currently lies outside its scope. That being said, debugging a dysfunctional yet publicly accessible artifact entails mechanical tasks of debugging an existing artifact, setting up its software environment, and validating that it runs. LLM agents have been shown to perform reasonably well for these tasks [40]. We envision NetGent could be extended to take up this task, but it does not support it right now. Though we have not measured the cost, we expect this cost to be one-time per artifact because a finished workflow or build serves every intent that needs it.
Where does AI belong? As models improve, an agent that writes and runs each experiment end to end grows more tempting than Pramana's single, bounded compile step. Does bounding a model with a fixed grammar use AI well, or discard the generality that makes agents useful?
What should shared infrastructure admit? One intent or agent can launch thousands of experiments, yet public infrastructure assumes a researcher who reserves nodes, files requests, and schedules runs. What load limits should RIPE Atlas [34], FABRIC [2], and campus testbeds declare, so agents can check bandwidth commitments, reservation caps, and contention before launching, without human approval?
Conclusion. Pramana is a thin waist for empirical networking research: a researcher states which data to collect and on which substrate, and Pramana supplies the mechanism. Its intent specification can express all 255 corpus intents, while NetReplica, the best existing tool, covers 11%. A language model compiles a plain-language description into the specification, and the backend runs it with NetGent and NetReplica. Today, the implementation covers 68 intents and has regenerated CCAnalyzer's main experiment once per setting and algorithm. Of the remaining intents, 168 require missing capabilities and 19 are underspecified. Building the ten most critical ones would lift coverage from 27% to 53%, but long-term sustainability will only derive from community building. With these capabilities, a researcher or an agent gets the data for a hypothesis by writing its intent instead of building a setup. That ability shortens the ideation-to-data gap for both research [13] and teaching [10].

Acknowledgments. We thank our shepherd, Soudeh Ghorbani, the anonymous reviewers, and the CCAnalyzer authors for their feedback. Researchers at UCSB have been supported by NSF awards 2443777 and 2323229 and by research gifts from Cisco and Google. Researchers at UCI have been supported by NSF award 2443024.

References

[1] Amazon Web Services. [n. d.]. Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/. Accessed 2026.

[2] Ilya Baldin, Anita Nikolich, James Griffioen, Inder Singh 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.

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*.

[4] 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 *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3576915.3623075

[5] CAIDA. [n. d.]. Archipelago (Ark) Measurement Infrastructure. https://www.caida.org/projects/ark/. Accessed 2026.

[6] Eric Hayden Campbell, Hossein Hojjat, and Nate Foster. 2024. Computing Precise Control Interface Specifications. *Proceedings of the ACM on Programming Languages (OOPSLA)* (2024).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. *arXiv preprint arXiv:2510.06189* (2025).

[9] David D. Clark. 1988. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of the ACM SIGCOMM Conference*. doi:10.1145/52324.52336

[10] 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)*. Association for Computing Machinery, Sacramento, CA, USA. To appear. doi:10.1145/3857222.3860185

[11] 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

[12] 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 *Proceedings of the Network Traffic Classification – Reloaded Workshop (NTC-R)*. To appear. doi:10.1145/3843199.3857224

[13] Jaber Daneshamooz, Haarika Manda, Manni Moghimi, Sylee Beltiukov, Walter Willinger, and Arpit Gupta. 2026. Rethinking Network Foundation Models: From Bigger Models to Better Data. In *Network Traffic Classification – Reloaded (NTC-R '26)*. Association for Computing Machinery, Utrecht, Netherlands. To appear. doi:10.1145/3843199.3857225

[14] Jaber Daneshamooz, Eugene Vuong, Laasya Koduru, Sanjay Chandrasekaran, and Arpit Gupta. 2025. NetGent: Agent-Based Automation of Network Application Workflows. https://arxiv.org/abs/2509.00625. arXiv:2509.00625 [cs.NI]

[15] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *EMNLP*.

[16] Google. [n. d.]. Google Cloud Platform. https://cloud.google.com/. Accessed 2026.

[17] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. 2008. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI*.

[18] Satyandra Guthula, Jaber Daneshamooz, Charles Fleming, Kesheng Wu, Walter Willinger, and Arpit Gupta. 2025. Netburst: Event-centric forecasting of bursty, intermittent time series. *arXiv preprint arXiv:2510.22397* (2025).

[19] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. *arXiv preprint arXiv:2510.27176* (2025).

[20] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible Network Experiments Using Container-Based Emulation. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. doi:10.1145/2413176.2413206

[21] Stephen Hemminger. 2005. Network Emulation with NetEm. In *Linux Conf Au*.

[22] Kexin Huang, Ying Jin, Ryan Li, Michael Y. Li, Emmanuel Candès, and Jure Leskovec. 2025. Automated Hypothesis Validation with Agentic Sequential Falsifications. *arXiv preprint arXiv:2502.09858* (2025).

[23] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. 2021. Hey, Lumi! Using Natural Language for Intent-Based Network Management. In *USENIX Annual Technical Conference (ATC)*.

[24] Imre Lakatos. 1970. Falsification and the Methodology of Scientific Research Programmes. In *Criticism and the Growth of Knowledge*, Imre Lakatos and Alan Musgrave (Eds.). Cambridge University Press, 91–196.

[25] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*.

[26] Jiahui Li, Han Qi, Ruyi Yao, Jialin Wei, Ruoshi Sun, Zixuan Chen, Sen Liu, and Yang Xu. 2025. CClinguist: An Expert-Free Framework for Future-Compatible Congestion Control Algorithm Identification. In *Proceedings of the ACM SIGCOMM Conference*.

[27] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. *arXiv preprint arXiv:2408.06292* (2024).

[28] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *PLDI*.

[29] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX Annual Technical Conference (ATC)*.

[30] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. *arXiv preprint arXiv:2506.13131* (2025).

[31] Adithya Abraham Philip, Rukshani Athapathu, Ranysha Ware, Fabian Francis Mkocheko, Alexis Schlomer, Mengrou Shou, Zili Meng, Srinivasan Seshan, and Justine Sherry. 2024. Prudentia: Findings of an Internet Fairness Watchdog. In *Proceedings of the ACM SIGCOMM Conference*. doi:10.1145/3651890.3672229

[32] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166.

[33] Alagappan Ramanathan, Eunju Kang, Dongsu Han, and Sangeetha Abdu Jyothi. 2026. Airavat: An Agentic Framework for Internet Measurement. https://arxiv.org/abs/2602.20924. arXiv:2602.20924 [cs.NI]

[34] RIPE NCC. 2015. RIPE Atlas: A Global Internet Measurement Network. *The Internet Protocol Journal* 18, 3 (2015), 2–26.

[35] Luigi Rizzo. 1997. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM SIGCOMM Computer Communication Review* 27, 1 (1997), 31–41.

[36] Damien Saucez, Luigi Iannone, and Olivier Bonaventure. 2019. Evaluating the artifacts of SIGCOMM papers. *ACM SIGCOMM Computer Communication Review* (2019). https://ccronline.sigcomm.org/wp-content/uploads/2019/05/acmdl19-308.pdf.

[37] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *EMNLP*.

[38] SNL-UCSB. 2026. Pramana: Intent Corpus and Reproducibility Artifact. https://github.com/SNL-UCSB/pramana-coverage.

[39] Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. 2025. AI-Researcher: Autonomous Scientific Innovation. *arXiv preprint arXiv:2505.18705* (2025).

[40] Saad Ullah, Praneeth Balasubramanian, Wenbo Guo, Amanda Burnett, Hammond Pearce, Christopher Kruegel, Giovanni Vigna, and Gianluca Stringhini. 2025. From CVE Entries to Verifiable Exploits: An Automated Multi-Agent Framework for Reproducing CVEs. https://arxiv.org/abs/2509.01835. arXiv:2509.01835 [cs.CR]

[41] University of California, Santa Barbara. [n. d.]. The PINOT Project: Programmable Infrastructure for AI/ML for Networking. https://pinot.cs.ucsb.edu. Accessed 2026.

[42] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*.

[43] 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 *Proceedings of the ACM SIGCOMM Conference*. doi:10.1145/3651890.3672255

[44] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*.

[45] Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. *arXiv:2307.09702* (2023).

[46] Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, Roman Beltiukov, Satyandra Guthula, Lisandro Z. Granville, and Arthur S. Jacobs. 2025. When Something Looks too Good to be True, it Usually is! AI is Causing a Credibility Crisis in Networking. *ACM SIGCOMM Computer Communication Review* 55, 1 (2025), 10–15. doi:10.1145/3727063.3727067

[47] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. 2025. The AI Scientist-v2: Workshop-Level Automated Scientific Discovery via Agentic Tree Search. *arXiv preprint arXiv:2504.08066* (2025).

[48] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[49] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *USENIX Annual Technical Conference (ATC)*.

[50] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*. 3911–3921.

[51] Yajie Zhou, Jiajun Ruan, Eric S. Wang, Sadjad Fouladi, Francis Y. Yan, Kevin Hsieh, and Zaoxing Liu. 2026. NetArena: Dynamic Benchmarks for AI Agents in Network Automation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.