NetGent: Agent-Based Automation of Network Application Workflows

Jaber Daneshamooz* **Eugene Vuong‡** **Laasya Koduru*** **Sanjay Chandrasekaran*** **Arpit Gupta***
*University of California Santa Barbara ‡California State University, East Bay

Abstract

We present *NetGent*, an AI-agent framework for automating complex application workflows to generate realistic network traffic datasets. Developing generalizable ML models for networking requires data collection from network environments with traffic that results from a diverse set of real-world web applications. However, using existing browser automation tools that are diverse, repeatable, realistic, and efficient remains fragile and costly. *NetGent* addresses this challenge by allowing users to specify workflows as natural-language rules that define state-dependent actions. These abstract specifications are compiled into nondeterministic finite automata (NFAs), which a state synthesis component translates into reusable, executable code. This design enables deterministic replay, reduces redundant LLM calls through state caching, and adapts quickly when application interfaces change. In experiments, *NetGent* automated more than 50+ workflows spanning video-on-demand streaming, live video streaming, video conferencing, social media, and web scraping, producing realistic traffic traces while remaining robust to UI variability. By combining the flexibility of language-based agents with the reliability of compiled execution, *NetGent* provides a scalable foundation for generating the diverse, repeatable datasets needed to advance ML in networking.

1 Introduction

Machine learning for networking has become an increasingly active area of research for tasks such as QoE inference and optimization of networked applications. A persistent barrier is access to realistic, labeled application data at scale [1]. Unlike vision or NLP, networking datasets often *cannot* be scraped or passively collected: they must be *generated* by executing live application workflows (e.g., streaming a video, joining a meeting, browsing social media) so that traffic, logs, and user interactions reflect real deployments. Today, researchers commonly rely on browser-automation scripts (e.g., Selenium [24], PyAutoGUI [36]) to repeat experiments and scale data collection. However, authoring and maintaining such scripts is long, manual, and brittle, especially for complex, multi-step tasks across diverse sites.

As a result, prior work frequently narrows the scope to a small set of applications or use cases when generating datasets [4]. Yet building generalizable ML models requires collecting across *many* applications, inputs (e.g., different videos), and network environments. To keep up with this demand, data collection pipelines must repeatedly execute the same workflows under varied conditions while remaining robust to evolving user interfaces and behavior [8].

Consider a concrete example: automate Disney+ to open ESPN, select the first video, and move the playback slider to the five-minute mark. Even this simple task varies: a user may or may not be logged in; profile selection may be required; the ESPN entry point and page layout change over time; ads or PIN prompts may appear. Such variability makes it difficult to design automation that is both robust and repeatable, especially when scaled to thousands or millions of runs across diverse network conditions for ML training and evaluation.

Requirements. This example illustrates six interrelated requirements for networking data generation: (1) *diversity* across applications and platforms; (2) *repeatability* so identical inputs yield identical outcomes across many runs and network conditions; (3) *complexity* to capture dynamic, non-linear, multi-step interactions; (4) *robustness* to survive frequent UI changes; (5) *realism* to mimic human behavior and avoid undesired bot detection; and (6) *efficiency* to minimize token usage and workflow-generation time. Meeting all six simultaneously is non-trivial; improving one dimension often degrades another.

Why existing approaches fall short. Web/GUI agents and script-based automation each solve a subset of these needs. Agentic approaches (e.g., ReAct [46], Reflexion [33]) emphasize online planning and self-reflection, but incur high token costs per execution and remain unreliable on long horizons—GPT-4 agents achieve $\approx 14\%$ success on WebArena versus $\approx 78\%$ for humans [47]; Mind2Web [9], VisualWebArena [18], and BrowserGym [5] further document these gaps. Scripted frameworks (Selenium, Playwright, PyAutoGUI) provide efficient replay but are notoriously flaky under UI drift; empirical studies attribute failures to asynchronous waits, DOM instability, and timing issues [21, 39, 26]. More specialized web scraping tools, such as for the broadband-plan querying tool (BQT [26, 17]), trade generality for robustness, but are still fragile to UI changes. None of these simultaneously delivers diversity, repeatability at scale, robustness, realism, and efficiency.

Proposed approach. We introduce *NetGent*, an AI-agent framework that separates *what* a workflow should do from *how* it is executed. Users provide natural-language state prompts—high-level *trigger–action* rules (e.g., "if on login page, enter credentials," "if viewing profiles, select `snlclient`", "if cookies popup, select `Accept`")—which specify an abstract, non-linear workflow. A *State Synthesis* component compiles these abstract prompts into *concrete states* with application-bound detectors (DOM/text/URL) and reusable executable code. Concrete states are cached in a repository and deterministically replayed by a *State Executor*; when UIs change, *NetGent* regenerates only the affected states from the same abstract prompts. This compile–then–replay design blends the flexibility of language-based synthesis with the efficiency and stability of compiled execution, directly addressing the six requirements above.

Contributions and evidence. We implement *NetGent* and evaluate it across 50+ workflows spanning video-on-demand streaming, live video streaming, video conferencing, social media, and web scraping (similar to BQT). Section 2 details the abstractions and execution model; Section 3 demonstrates that (i) abstract user prompts in natural language expand into hundreds of lines of executable code across diverse applications (extensibility), (ii) caching and replay reduce token cost and make millions of repeat runs economically feasible (efficiency and repeatability), and (iii) UI drift is handled by regenerating only the impacted state (robustness). Together, these results position *NetGent* as a proof of concept for scalable and realistic data generation in networking, complementing controllable platforms such as netReplica [8]—capable of emulating a diverse range of realistic networking conditions. We make *NetGent* and its generated workflows available at `https://github.com/SNL-UCSB/netgent`.

2 System Design

2.1 Architectural Abstractions

NetGent separates *what* a workflow should do from *how* it is executed through three abstractions.

Abstract NFA. Users define an abstract nondeterministic finite automaton (NFA) [27] using natural-language state prompts. Each state prompt specifies *triggers* (conditions that identify the state), *actions* (intended task), and an optional *end condition*. This representation captures non-linear flows (complexity) while keeping intent decoupled from UI specifics (robustness). For example, in a Disney+/ESPN workflow, states may include `login`, `select_profile`, `navigate_to_espn`, `select_video`, and `playback`.

Concrete NFA. During execution, *NetGent* compiles each abstract state into a *concrete state* defined by $\hat{s} = (\textit{detectors}, \textit{code})$: a set of CSS element, text, or URL detectors bound to the current application version, together with reusable executable code. This compiled form enables deterministic replay (repeatability) and cross-run reuse (efficiency). For example, the abstract trigger "if on login page" becomes a detector set (form labels, button text, stable DOM paths) and a short program that types credentials and clicks "Log In."

Cache and Replay. Concrete states are stored in a *State Repository*; a *State Executor* replays their code deterministically. If a detector later fails due to UI drift, only that state is regenerated from the abstract rule (robustness). Common states (e.g., `login`, `select_profile`) are reusable across workflows and apps (efficiency, diversity).

2.2 Workflow Execution Model

Figure 1 illustrates the runtime loop which generates executable code from user prompts.

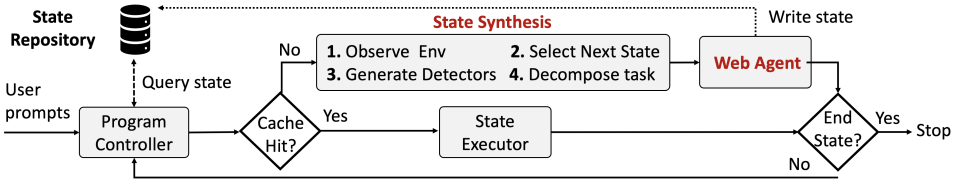

Figure 1: *NetGent* runtime loop progresses from the initial to the end state

Controller queries the cache first. Given the current page (DOM) and the last transition, the *Program Controller* queries the *State Repository*. If a cache hit occurs, the Controller invokes the *State Executor* to replay the stored code in the browser and the workflow advances. This cache-first policy is the core of compile–then–replay and eliminates repeated reasoning (repeatability, efficiency).

Cache miss triggers one-shot synthesis. On a cache miss, the Controller invokes *State Synthesis* (LLM), which performs four steps using the current DOM, screenshot, and user's rules: (1) *Observe* the environment to form a structured view; (2) *Select* the appropriate next abstract state (trigger–action pair); (3) *Generate* concrete detectors that reliably recognize that state; (4) *Decompose* the action into a simplified plan with decomposed tasks. The *Web Agent* executes these decomposed tasks and generates the executable code. The *Concrete State* is then written back to the repository. Only the missing node is synthesized; the abstract NFA and prior states remain intact (robustness, efficiency).

Realistic execution and termination. To enhance realism and evade bot detection, our web agent integrates browser stealth, human-like interaction, and network stealth (details in Appendix §A.2). An end state is declared when an application-level condition holds (e.g., for ESPN, a `<video>` element is playing and time is advancing). Otherwise, the Controller loops to the next state.

Concrete example. Starting at the Disney+ homepage, the Controller hits cached `login` and `select_profile` states on subsequent runs; on the first run these are synthesized once. Navigating to the ESPN hub and clicking the first video may trigger ads or a PIN prompt; the NFA branches handle these cases by synthesizing (once) a `type_pin` or `skip_ad` state and writing them to the repository. Playback detection serves as the end state, after which *NetGent* records the successful trace and terminates.

3 Evaluation

We evaluate *NetGent* against the requirements introduced in §2, mapping experiments to the abstractions in §2.1 using Gemini 2.5 Flash [12] with a temperature of 0.2. Details of the APIs and frameworks employed are provided in §A.1. All evaluations and executions were conducted on a MacBook Pro (Apple M3 Pro chip, 11-core CPU, 14-core GPU, and 18 GB RAM).

Diversity across applications. A central goal of *NetGent* is to keep user effort low while scaling to diverse applications. We hypothesize that prompt length serves as a proxy for user effort, while the lines of generated code reflect the automation complexity that would otherwise need to be implemented manually. We therefore measured the size of user prompts and the length of generated Python code across 50+ workflows spanning five domains: video-on-demand streaming, live video streaming, video conferencing, social media, and web scraping. Table 1 in the appendix provides the list of these applications, along with their evaluation based on code generation time, number of tokens used, and dollar cost. Each workflow requires only a 100–200 word prompt to generate code that spans hundreds of lines. This large expansion factor demonstrates that small, uniform specifications suffice to produce substantial executable workflows. Moreover, the same prompt structure generalizes

across platforms within a category (e.g., Hulu and Disney+), showing that *NetGent* is easily extensible to new applications with minimal effort.

Efficiency and repeatability. We next examine efficiency and repeatability within a specific workflow. The system leverages the compile–then–replay method to achieve repeatability by reusing stored concrete states, while caching reduces token usage by avoiding redundant LLM calls.

We focus on the ESPN workflow, where user interactions include `login`, `select_profile`, `playback` and other related actions. Running this workflow without any stored concrete NFA consumes 278k tokens per run, translating to \$0.098 at \$0.35 per million tokens[1]. Executing this workflow one million times without reusing a concrete NFA would cost roughly \$98,000 in LLM usage alone. By contrast, the compile–then–replay approach reuses stored states, eliminating per-run LLM costs and ensuring deterministic execution. This avoids redundant LLM and API calls, allowing a single generated workflow to be reused across multiple runs.

However, due to UI changes and other factors, periodic updates are necessary to handle new or modified states. Assuming a system with 10 states, generating each new state requires ≈ 42.6k tokens ($\approx \$0.015$) on average. Generating workflows for all 10 states from scratch incurs a one-time cost of $\approx \$0.15$. If the workflow drifts weekly over a year (52 weeks), updating all states would cost $\approx \$7.8$. With caching, only changed states need updating. Assuming one state update per week, the annual LLM cost drops to $\approx \$0.78$. Caching thus minimizes redundant LLM calls and enables deterministic replay of previously synthesized states. This demonstrates that compile–then–replay with state caching ensures deterministic behavior while making large-scale execution economically viable.

Robustness under UI drift. Finally, we evaluate robustness to interface changes. The hypothesis is that *NetGent* can localize regeneration to only the affected state, avoiding costly re-synthesis of the full workflow. We perturb the ESPN workflow, requiring a PIN for profile access. In this case, only the affected state (`type_pin`) was regenerated, while other states such as `login`, `select_profile` and `navigate_to_espn` were replayed from cache without modification. Using the caching method, regenerating the affected state required only $\approx$ 20k tokens and the entire process took 216 seconds, compared to $\approx$ 375k tokens and 406 seconds if done from scratch. This bounded overhead confirms that state-level regeneration is sufficient. Even when multiple states change, the unaffected portions of the workflow remain intact. Such robustness ensures that workflows remain usable despite frequent UI drift in production applications. See Figure 2 in the appendix for the full ESPN workflow.

Summary. Across all experiments, *NetGent* satisfies the requirements of §2: prompts abstract away application-specific details to ensure diversity and extensibility; caching enables efficiency and repeatability; and state-local regeneration ensures robustness to UI drift. These results collectively demonstrate that *NetGent* provides a scalable foundation for generating realistic networking datasets across heterogeneous applications.

4 Limitations and Future Work

While *NetGent* demonstrates that abstract NFAs combined with compile–then–replay enable scalable and repeatable workflows, several limitations remain. First, manual workflow verification and failure handling currently require user intervention; automating step-level validation and state-level recovery would enable self-healing and fully autonomous workflows. Second, *NetGent* is limited to web applications; extending the NFA abstraction to desktop environments would broaden applicability. Together, these extensions will make *NetGent* more autonomous, robust, and broadly deployable.

Acknowledgement

This work was supported in part by the National Science Foundation (CAREER Award No. 2443777 and CNS Award No. 2323229) and a research gift from Cisco and Google. Eugene Vuong was supported by the Cal-Bridge Program. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award NERSC DDR-ERCAP0029768.

[1]The total token cost is based on both input and output tokens, each typically charged at a different rate. For simplicity, we represent it here as the total number of tokens.

References

[1] Roman Beltiukov, Wenbo Guo, Arpit Gupta, and Walter Willinger. In search of netunicorn: A data-collection platform to develop generalizable ml models for network security problems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2217–2231, 2023.

[2] Bluesky. `https://bsky.app`, 2025. Accessed: 2025-08-29.

[3] Bright Data - All in One Platform for Proxies and Web Scraping. https://brightdata.com/.

[4] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–25, 2019.

[5] Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. The browsergym ecosystem for web agent research, 2025.

[6] Cisco Systems, Inc. Webex. `https://www.webex.com`, 2025. Accessed: 2025-08-29.

[7] Jaber Daneshamooz, Satyandra Guthula, Jessica Nguyen, William Chen, Sanjay Chandrasekaran, Ankit Gupta, Arpit Gupta, and Walter Willinger. Netreplica: Toward a programmable substrate for last-mile data generation. *arXiv preprint arXiv:2507.13476v2*, 2025.

[8] Jaber Daneshamooz, Jessica Nguyen, William Chen, Sanjay Chandrasekaran, Satyandra Guthula, Ankit Gupta, Arpit Gupta, and Walter Willinger. Addressing the ml domain adaptation problem for networking: Realistic and controllable training data generation with netreplica. *arXiv preprint arXiv:2507.13476*, 2025.

[9] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023.

[10] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving Planning of Agents for Long-Horizon Tasks, April 2025.

[11] Espn. `https://www.espn.com`, 2025. Accessed: 2025-08-29.

[12] Gemini 2.5: Our newest gemini model with thinking. `https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/`. Accessed: 2025-08-12.

[13] Google LLC. Google meet. `https://workspace.google.com/resources/video-conferencing/`, 2025. Accessed: 2025-08-29.

[14] Google LLC. Youtube. `https://www.youtube.com`, 2025. Accessed: 2025-08-29.

[15] Hulu. `https://www.hulu.com`, 2025. Accessed: 2025-08-29.

[16] Jitsi. `https://jitsi.org`, 2025. Accessed: 2025-08-29.

[17] Laasya Koduru, Sylee Beltiukov, Jaber Daneshamooz, Eugene Vuong, Arpit Gupta, Elizabeth Belding, and Tejas N. Narechania. Enabling data-driven policymaking using broadband-plan querying tool (bqt+), 2025. arXiv:2511.05838.

[18] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024.

[19] Langchain. `https://www.langchain.com`. Accessed: 2025-08-23.

[20] LinkedIn Corporation. Linkedin. https://www.linkedin.com, 2025. Accessed: 2025-08-29.

[21] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.

[22] Meta Platforms, Inc. Instagram. https://www.instagram.com, 2025. Accessed: 2025-08-29.

[23] Microsoft Corporation. Microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software#Products-and-services, 2025. Accessed: 2025-08-29.

[24] Michael Mintz and SeleniumBase contributors. Seleniumbase.

[25] Magnus Müller and Gregor Žunič. Browser use: Enable ai to control your browser, 2024.

[26] Udit Paul, Vinothini Gunasekaran, Jiamo Liu, Tejas Narechania, Arpit Gupta, and Elizabeth Belding. Decoding the divide: Analyzing disparities in broadband plans offered by major us isps. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '23, pages 578–591, New York, United States, 2023. Association for Computing Machinery.

[27] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[28] Reddit, Inc. Reddit. https://www.reddit.com, 2025. Accessed: 2025-08-29.

[29] Ringcentral. https://www.ringcentral.com, 2025. Accessed: 2025-08-29.

[30] Roku, Inc. Roku. https://www.roku.com, 2025. Accessed: 2025-08-29.

[31] Sai Adarsh S, Prasanna Venkateshan, and Prithvi Alva Suresh. Emulating human-like mouse movement using bézier curves and behavioural models for advanced web automation. *IJIRT*, 12(3):1364–1370, 2025.

[32] Yusuf Sani, Andreas Mauthe, and Christopher Edwards. Adaptive bitrate selection: A survey. *IEEE Communications Surveys & Tutorials*, 19(4):2985–3014, 2017.

[33] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[34] Rayadurgam Srikant and Tamer Başar. *The mathematics of Internet congestion control*. Springer, 2004.

[35] Stanford University. Puffer, 2025. Accessed: 2025-08-29.

[36] Al Sweigart. Pyautogui.

[37] Talky. https://about.talky.io, 2025. Accessed: 2025-08-29.

[38] The Walt Disney Company. Disney. https://www.disneyplus.com, 2025. Accessed: 2025-08-29.

[39] Swapna Thorve, Chandani Sreshtha, and Na Meng. An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538. IEEE, 2018.

[40] Tubi, Inc. Tubi. https://tubitv.com, 2025. Accessed: 2025-08-29.

[41] Twitch. https://www.twitch.tv, 2025. Accessed: 2025-08-29.

[42] ultrafunkamsterdam. undetected-chromedriver.

[43] Whereby. https://whereby.com, 2025. Accessed: 2025-08-29.

[44] X. https://x.com, 2025. Accessed: 2025-08-29.

[45] Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V, November 2023.

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[47] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024.

[48] Zoho meeting. `https://www.zoho.com/meeting/`, 2025. Accessed: 2025-08-29.

[49] Zoom Video Communications, Inc. Zoom. `https://www.zoom.com/?lang=en-US`, 2025. Accessed: 2025-08-29.

A Appendix / Supplemental Material

Figure 2: ESPN workflow: log into Disney+, select the account, enter the PIN if required, navigate to ESPN, play the first video, and advance the playback slider to the five-minute mark.

A.1 Language and API Integration

Our system's architecture is built by composing several key technologies. The foundational layer for interacting with LLMs is standardized using core LangChain [19] abstractions, such as `SystemMessage` and `HumanMessage` for model inputs and tool invocation. We specifically access Google's Gemini 2.5 chat models [12] through the VertexAI platform, which is integrated as a backend via the `langchain-google-vertexai` wrapper. While LangChain provides the core communication components, we use LangGraph as the control-flow layer to orchestrate the Browser Agent's complex behavior. LangGraph allows us to define the agent's logic, which is essential for managing its multi-step processes of environment observation, planning, and code generation and execution.

To enhance the Web Agent's capabilities and efficiency, we incorporated several key open-source contributions. Notably, we integrated the DOM Marker code from Browser Use [25] to implement Set-of-Mark (SoM) prompting [45], enabling precise interaction with web elements via visual markers. We also adopted the Planner, Replanner, and Executor prompts from Plan-and-Act [10], allowing the agent to decompose complex tasks into structured planning phases followed by systematic execution. These integrations provide a robust foundation that leverages proven open-source techniques while supporting the unique requirements of *NetGent*, facilitating reliable and scalable automation of complex web workflows.

A.2 Realism: Mimicking Human-Like Interactions

To support realism, we adopted multiple techniques in the web agent so tasks simulate human-like web interactions and avoid bot detection mechanisms. To mitigate automated-behavior detection, our system integrates three anti-bot methods: *(i) browser stealth*, *(ii) movement realism*, and *(iii) network stealth*. For *browser stealth*, we use SeleniumBase [24] with undetected-chromedriver [42] to hide common automation fingerprints. Besides that, repeated logins during automated workflows can trigger service defenses, such as account freezes or forced two-step verification. To overcome this issue, we support persistent user profiles via the `user-data-dir` option. This command makes the browser retain state across sessions, including cookies (login information) and cached site assets. This prevents repeated login flows and creates a continuous session

Table 1: Evaluation of *NetGent* across different workflows, performed entirely without using cached states.

Application Type	Platform	Tokens ($\times 10^3$)	Price ($)	Time (s)
Video Conferencing	Zoom [49]	146.8	0.051	138
Video Conferencing	Microsoft Teams [23]	242.9	0.085	238
Video Conferencing	Google Meet [13]	169.7	0.060	160
Video Conferencing	Zoho Meeting [48]	137.6	0.049	190
Video Conferencing	Jitsi [16]	98.5	0.035	120
Video Conferencing	Whereby [43]	77.1	0.026	78
Video Conferencing	Ring Central [29]	131.9	0.047	139
Video Conferencing	Talky [37]	110.9	0.041	129
Video Conferencing	Webex [6]	107.0	0.037	114
Video on Demand + Live stream	YouTube [14]	131.7	0.047	105
Video on Demand + Live stream	ESPN [11]	278.7	0.098	339
Video on Demand + Live stream	ESPN (PIN Required)	375.4	0.105	406
Video on Demand + Live stream	ESPN (PIN Required)†	20.1	0.006	216
Video on Demand	Disney Plus [38]	249.2	0.088	335
Video on Demand	Hulu [15]	245.0	0.089	386
Video on Demand	Roku [30]	162.2	0.059	187
Video on Demand	Tubi [40]	140.6	0.049	158
Live stream	Twitch [41]	134.6	0.044	90
Live stream	Puffer [35]	109.3	0.038	126
Social Media	X (Twitter) [44]	201.7	0.068	158
Social Media	Instagram [22]	199.1	0.069	180
Social Media	LinkedIn [20]	143.8	0.049	138
Social Media	Reddit [28]	739.5	0.225	82
Social Media	Bluesky [2]	98.1	0.033	79
Web Scraping	BQT [26] (30 ISPs)	145*	0.050*	120*

† This execution uses the cached ESPN workflow (no PIN) and shows *NetGent*'s efficiency under UI drift.
* Approximate average numbers across 30 ISPs.

experience similar to a regular user. Combined with fingerprinting avoidance, this supports a stable, persistent, and human-like browser profile.

To incorporate *movement realism*, we use human-like interaction primitives. Instead of modifying the page DOM directly, the system computes absolute screen coordinates and issues mouse and keyboard events through PyAutoGUI [36]. Mouse movements follow smooth Bezier-curve trajectories [31] rather than instant jumps. Keystrokes are generated with small, variable delays. Scrolling and hovering are restricted to the visible viewport, avoiding sudden jumps and ensuring the agent only interacts with elements a human user can actually see.

Finally, we provide *network stealth* by using a pool of IP addresses from Bright Data [3] to distribute requests across different network origins. This is essential for many web scraping applications [26, 17], as repeated interactions from a single IP increase the likelihood of rate limits or blocking. By varying the network origin, the system reduces such detection signals while maintaining the appearance of normal user traffic. While effective in these settings, this approach is less suitable for services like video streaming that rely on IP-based account tracking to prevent oversharing. Together, these methods allow the system to closely mirror human interactions with applications and the browser, which is critical for generating robust workflows that can be executed repeatedly and reliably without triggering blocking mechanisms.

A.3 Broader Use Cases of *NetGent* for Networking Data Generation

The use cases and importance of *NetGent* extend beyond generating traffic for the control application targeted for data collection. A high-quality, generalizable dataset must be collected in diverse network environments, where one of the most challenging aspects to emulate is cross traffic: Replace it with: traffic from other applications that share the same path or bottleneck link and influence the control application, such as inducing congestion events or triggering ABR algorithms [34, 32]. The main challenge is generating cross traffic that is realistic, diverse, and reactive [7]. A practical approach is to construct a large pool of diverse application workflows and select combinations of them to run alongside the control application. This allows us to emulate realistic environments, such as a home network where one user is streaming video while another is browsing, and evaluating how the control application (e.g., Zoom) behaves under these shared and congested network conditions.