Using LLMs to Automate Network Data Collection Workflows

Gen Tamada, Tim Qin, Wesley Truong, Adyah Rastogi

Abstract

The ability to collect real-time datasets is becoming increasingly important as research in networking systems is relying heavily on large datasets. Current existing networking datasets (e.g., CIC-IDS-2017, MAWILab) are either outdated or gathered in simulated environments not representative of the user experience [4]. In this paper, we wanted to explore a way to automate the data collection process in a way that simulates human behavior. With the advent of LLMs and agents, we decided to create a tool that can be called so that an LLM can run data collection with minimal human intervention. In our implementation, we have gotten the LLM workflow to start data collection from YouTube and Twitch autonomously. The only prerequisite from the researcher is to provide a one-time recording of their actions taken when recording data and the LLM can use that information to reproduce the data collection process. The implementation details can be found at https://github.com/Ononymous/llm-network-data-collector.

1 Introduction

In this paper, we propose an AI-driven workflow that makes use of the actions a human takes while collecting data to learn how to collect data. For this, we opted to use the Playwright automation library. This library creates readable code transcripts of human actions on a browser, which enables the LLM to adapt to human actions and provides for it to learn from HTML text for more accurate interactions with a website. In the first step of our data collection process, a workflow guideline is generated from the recording's transcript by Playwright, which is used by the LLM according to the specified prompts in order to execute an autonomous data collection script. Below is an outline of the contributions of this paper:

- We design a **prompt-engineering framework** that instructs the LLM to augment recorded code with selector inference, randomized actions, and data hooks that eliminate manual selection and enable automated data extraction
- We implement a data collection library using an LLM, Google Gemini, which generates a workflow based on parameters that can be configured for metrics such as playback bitrate and livestream details. Users simply prompt the LLM with a website to collect data from and the intervals at which information should be collected.
- We validate our approach on primarily two different sites, YouTube and Twitch, to demonstrate that LLM scripting is tolerant to common UI changes and requires a minimal amount of human changes.

2 Background and Motivation

Modern machine learning models for network analysis, monitoring, and anomaly detection are highly dependent on high quality, representative datasets [2]. Traditional datasets like CIC-IDS-2017 and MAWILab are often static, out-of-date, or collected under controlled laboratory conditions that fail to capture the inconsistent and dynamic nature of real-world user behavior [4]. As a result, models trained on these datasets suffer from poor generalization and are prone to shortcut learning and out-of-distribution drift when deployed in production environments [3]. In particular, streaming platforms such as Twitch exhibit a wide range of network and playback behaviors which are not adequately reflected in existing data. Thus, there is a pressing need for an autonomous, scalable, and adaptive data collection framework that can (a) emulate realistic user interactions and (b) be robust to evolving web interfaces without frequent human intervention [1].

2.1 Existing Approaches and Their Limitations

Manual Script Authoring. The traditional way to collect web-based telemetry is to write handcrafted Playwright, Selenium, or PyAutoGUI scripts. While these tools provide fine-grained control, every UI change, no matter how minor, invalidates locators (CSS selectors, XPaths), causing scripts to break. Maintaining dozens or hundreds of scripts across multiple streaming platforms scales poorly and may be unfeasible. Moreover, purely manual scripts rarely incorporate randomness or error recovery logic, resulting in data that lacks diversity and realism.

Passive Dataset Collection. While traditional passive telemetry data collection remains a foundational technique for network observability, enabling tasks like fault diagnosis and performance characterization, its efficacy is often crippled by inherent limitations. As described by Mohan et al. [5], the indiscriminate capture of traffic

generates an immense volume of data, leading to prohibitive storage costs and significant analytical burdens. This data slog renders real-time analysis difficult and often obscures the very actionable insights network operators seek. These challenges motivate a paradigm shift from static, exhaustive collection to intelligent, targeted acquisition. Our research, therefore, creates expedited telemetry collectors through user-steered prompt generation, allowing more targeted telemetry collection without the time-consumption of previous manual collection setups, sidelining the scalability and analysis bottlenecks of conventional passive methods.

2.2 Problem Statement and Context

We want to build a system that can automatically navigate a target web application, collect Stats for Nerds data, record associated network packet traces (e.g. via `tcpdump`), and export structured JSON for downstream machine learning tasks. The primary challenges are:

- **Emulating User Interaction:** Websites frequently modify their UI, so making a robust tool that can withstand the changes to these elements is essential for data collection.
- **Diversity and Realism:** To avoid overfitting, the data being accessed must not always be cached on the closest servers of the Content Distribution Network (CDN), meaning some randomization scheme needs to exist for data collection. Also, some web applications provide some data in the front-end that is otherwise not accessible just through packet traces and other third-party tools.
- **Scalability:** Manually authoring and maintaining Playwright scripts for each platform and content type does not scale. We need an approach that can generalize recorded interactions into robust workflows and adapt iteratively to UI changes [6]. The final product should be able to adapt to new data collection environments without heavy manual intervention or lengthy, manual adaptation to the new environment.

3 Implementation

Our implementation journey involved significant iteration on the choice of automation frameworks for interacting with dynamic web applications like YouTube. The key goal was to collect video telemetry with minimal manual intervention, while ensuring stability, scalability, and extensibility for future data collection across multiple platforms.

3.1 Initial Exploration with PyAutoGUI

Our earliest implementation used PyAutoGUI, a Python library for simulating mouse and keyboard input. While

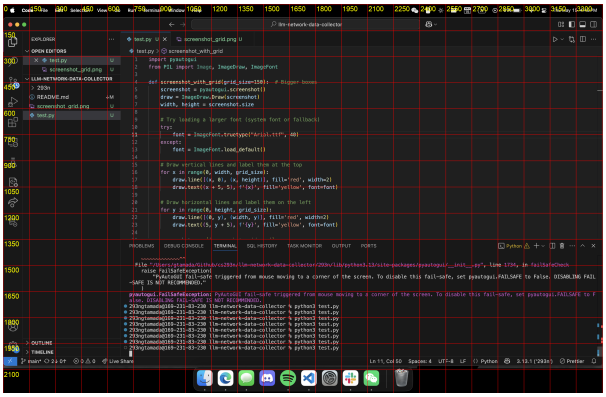

Figure 1: *Overlaying a labeled grid on screenshot for LLM to produce PyAutoGUI code*

PyAutoGUI is intuitive and well-suited for simple desktop automation, it requires hardcoded pixel coordinates to interact with UI elements. This rigidness proved problematic: UI elements on platforms like YouTube frequently change positions due to ads, screen resolution, or layout shifts. We experimented with various enhancements—such as overlaying grid markers on screenshots (as shown in **Figure 1**) and prompting LLMs with annotated images—but these solutions were inefficient and brittle. Moreover, each new test required capturing and processing a fresh screenshot, which introduced excessive latency and complexity.

3.2 Limitations of Selenium

Selenium was our next candidate. It offered DOM-level control and broad community support but fell short in reliability. Selenium-based tests frequently broke due to asynchronous content loading, dynamic element injection, and timing inconsistencies. The flakiness conflicted with our goal of minimal intervention. Its architecture also lacked native support for multiple concurrent browser contexts or seamless ad handling(features critical for replicating user behavior in video platforms).

3.3 Adopting Playwright for Robust Automation

Our successful implementation uses Playwright, a modern browser automation library developed by Microsoft. Playwright supports deterministic, low-latency control of Chromium, Firefox, and WebKit browsers with fine-grained element selectors and built-in support for asynchronous operations.

To scale the process of generating Playwright test scripts and make it more efficient, we developed a custom pipeline called the **Workflow Generator**. This Python-based automation script interacts with the Gemini LLM to synthesize robust, generalized Playwright test scripts from user input and minimal supervision.

The generator, shown in **Figure 2**, works as follows:

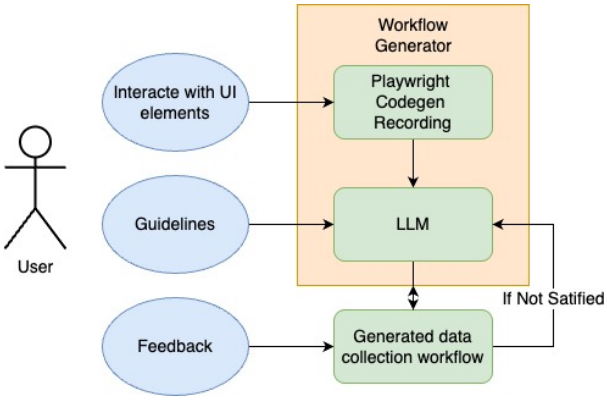

Figure 2: *Workflow Generator pipeline*

1. **Input Collection:** The user provides either a Playwright recording file or a target web application URL and base filename. If no recording is provided, the generator uses `playwright codegen` to allow the user to interactively record a test, saving the result in a local script.
2. **Guideline Integration:** The user can supply a guideline text file to steer how the test should behave (e.g., what elements to click, what data to collect). These domain-specific instructions enhance the LLM's ability to adapt the test to telemetry collection or complex behaviors.
3. **LLM-based Generalization:** The core of the workflow sends the recorded test code and user-supplied guidelines to Gemini 2.5 via API. The LLM then generates a new Python Playwright test script that incorporates structure, randomness, error handling, and user-driven instrumentation (e.g., collecting telemetry or skipping ads).
4. **Execution and Feedback:** The newly generated test script is executed directly. If a feedback file is provided (e.g., user notes or error logs), the generator resends the current test code and feedback to Gemini for iterative refinement. This loop continues until the user confirms the test is satisfactory.

This modular, feedback-driven workflow allows domain experts or researchers to build and refine complex test cases without writing extensive automation code. Importantly, it eliminates the need to prompt the LLM with a new screenshot or DOM structure each time, because the important HTML elements are already captured by the recorded Playwright script.

We validated this workflow by successfully generating agents for both YouTube and Twitch telemetry scraping, with minimal manual intervention and robust handling of edge cases (e.g., ads, offline streams, popups).

4 Evaluation

4.1 YouTube Telemetry Pipeline

For YouTube, our agent performs the following:

1. Generates a random multilingual query using a curated noun list.
2. Navigates to the top video result and opens the video page.
3. Right-clicks the video player and selects "Stats for nerds" from the context menu.
4. Watches the video for a randomly chosen duration while collecting telemetry once per second.
5. Injects random user actions such as pausing or skipping ahead to simulate realistic sessions.

This pipeline collects structured statistics such as buffer health, resolution, framerate, and network activity. The results are serialized into a local JSON file for future use.

4.2 Twitch Telemetry Pipeline

To demonstrate generalizability, we extended our agent to Twitch. We leveraged a third-party site that generates random embedded Twitch streams. Our Twitch pipeline includes additional decision logic for stream filtering and telemetry activation:

1. Load a random stream from an external preview site.
2. Handle audience warning dialogs (e.g., "Intended for certain audiences") by locating and clicking "Start Watching".
3. Skip the stream if the streamer is offline or if ads are detected by scanning for banner text elements.
4. Open the "Settings" menu within the video iframe, navigate to "Advanced", and enable the "Video Stats" panel.
5. Sample key statistics (e.g., resolution, latency, framerate) once per second for a fixed time window.

4.3 Data Collection Results

We wanted to showcase the type of data we were able to extract in this section. The data extracted in **Figure 3** is from Twitch's video stats. Twitch has good data formatting and not much was needed to improve upon its readability. As can be seen below, relevant information such as download bitrate, bandwidth estimate, protocol, and buffer size are all included.

We also wanted to include the collected YouTube data in **Figure 4**. The formatting is not as clean as Twitch, but it illustrates differences in data transparency between different websites. Finding an efficient way to restructure collected data is a task that can be adapted for a future iteration of our tool.

By implementing both YouTube and Twitch pipelines under the same automation framework, we validated Play-

```
{
  "Download Resolution": "1920x1080",
  "Render Resolution": "1920x1080",
  "Viewport Resolution": "1265x655",
  "Download Bitrate": "3330 Kbps",
  "Bandwidth Estimate": "655 Mbps",
  "FPS": "30",
  "Skipped Frames": "0",
  "Buffer Size": "1.83 sec.",
  "Latency To Broadcaster": "2.04 sec.",
  "Codecs": "avc1.4D0428,mp4a.40.2",
  "Protocol": "HLS",
  "Latency Mode": "Low Latency",
  "Render Surface": "video",
  "Backend Version": "1.41.0-rc.4",
  "Play Session ID": "baabb0bfae877e5114099fa6ed7fe1b9",
  "Serving ID": "9c52a010fdd441ad9ff72164220aa05f"
},
```


Figure 3: *Collected Twitch data (1 per second)*


```
{
  "timestamp_watched": 5.157994985580444,
  "stats": {
    "video_id": "GnVqpmuySTQ",
    "viewport": "791x445*2.00",
    "dropped_frames": "-",
    "current_res": "1280x720@24",
    "optimal_res": "1280x720@24",
    "volume": "100%",
    "normalized_volume": "DRC (content loudness 4.8dB)",
    "codecs": "vp09.00.51.08.01.01.01.01.00 (247)",
    "audio_codec": "opus (251)",
    "connection_speed": "2919 Kbps",
    "network_activity": "0 KB",
    "buffer_health": "18.52 s",
    "live_mode": null,
    "mystery_text": "SABR, s:44c t:0.00 b:0.000-18.517 P L",
    "date": "Fri May 30 2025 20:18:08 GMT-0700 (Pacific Daylight Time)"
  }
},
```


Figure 4: *Collected YouTube data (1 per second)*

wright's flexibility for multi-platform telemetry extraction. Our pipelines gracefully handle transient UI changes, advertisement interruptions, and context-sensitive dialogs—all of which would have posed critical challenges in previous frameworks.

5 Future Works

While our implementation of a data collection pipeline addresses certain issues with automatic data collection, there are still future improvements that can be explored. For one, we had to manually create a parser for our YouTube telemetry data in order to get readable data. Many different sites have different ways of presenting networking data, so a future improvement of indicating to the LLM what specific network features we want to extract can be added. Additionally, since our framework relies on a code generation example of one recording from a manual user, it still technically relies on human intervention and is not completely automatic. Creating a tool to automate this process can be something to look into. Lastly, for highly used video streaming platforms like Netflix or Facebook that require user credentials, we would need to heavily build upon our pipeline to incorporate authorization capabilities, as it may present security issues in a naive implementation.

These enhancements would improve the flexibility and automation of the data collection process and help diversify the type of data collected.

6 Conclusion

In conclusion our method successfully demonstrates a scalable, generalizable method for telemetry data collection. Through the use of an LLM powered collection agent, we are able to generate comprehensive telemetry data that mimics that from real-world scenarios, while requiring minimal upkeep after the initialization of a script through our pipeline.

References

[1] Adamson, R., Osborne, T., Lester, C., and Palumbo, R. Stream: A scalable federated hpc telemetry platform. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). (Cited on page 1.)

[2] Beltiukov, R., Guo, W., Gupta, A., and Willinger, W. 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* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 2217–2231. (Cited on page 1.)

[3] Jacobs, A. S., Beltiukov, R., Willinger, W., Ferreira, R. A., Gupta, A., and Granville, L. Z. Ai/ml for network security: The emperor has no clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery, p. 1537–1551. (Cited on page 1.)

[4] Liu, L., Engelen, G., Lynar, T., Essam, D., and Joosen, W. Error prevalence in nids datasets: A case study on cic-ids-2017 and cse-cic-ids-2018. In *2022 IEEE Conference on Communications and Network Security (CNS)* (2022), pp. 254–262. (Cited on page 1.)

[5] Mohan, V., Reddy, Y. J., Kalpana, K., et al. Active and passive network measurements: a survey. *International Journal of Computer Science and Information Technologies 2*, 4 (2011), 1372–1385. (Cited on page 1.)

[6] Tan, L., Su, W., Zhang, W., Lv, J., Zhang, Z., Miao, J., Liu, X., and Li, N. In-band network telemetry: A survey. *Computer Networks 186* (2021), 107763. (Cited on page 2.)