Deploying Stateful Network Functions Efficiently using Large Language Models

Hamid Ghasemirahni
KTH Royal Institute of Technology
Stockholm, Sweden

Alireza Farshin
NVIDIA
Stockholm, Sweden

Mariano Scazzariello
KTH Royal Institute of Technology
Stockholm, Sweden

Marco Chiesa
KTH Royal Institute of Technology
Stockholm, Sweden

Dejan Kostić
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Stateful network functions are increasingly used in data centers. However, their scalability remains a significant challenge since parallelizing packet processing across multiple cores requires careful configuration to avoid compromising the application's semantics or performance. This challenge is particularly important when deploying multiple stateful functions on multi-core servers. This paper proposes FlowMage, a system that leverages Large Language Models (LLMs) to perform code analysis and extract essential information from stateful network functions (NFs) prior to their deployment on a server. FlowMage uses this data to find an efficient configuration of an NF chain that maximizes performance while preserving the semantics of the NF chain. Our evaluation shows that, utilizing GPT-4, FlowMage is able to find and apply optimized configuration when deploying stateful NFs chain on a server, resulting in significant performance improvement (up to 11×) in comparison to the default configuration of the system.

CCS Concepts: • **Networks** → **Network servers**; • **Computing methodologies** → *Information extraction.*

Keywords: Intra-Server Load Balancing, Stateful Network Functions, LLMs, Static Code Analysis, RSS Configuration.

ACM Reference Format:
Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Marco Chiesa, and Dejan Kostić. 2024. Deploying Stateful Network Functions Efficiently using Large Language Models. In *4th Workshop on Machine Learning and Systems (EuroMLSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3642970.3655836

EuroMLSys '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0541-0/24/04
https://doi.org/10.1145/3642970.3655836

1 Introduction

Recent advances in networking devices, including the emergence of multi-hundred Gbps NICs [31] and powerful programmable switches [21], have boosted the network links capacity. Concurrently, state-of-the-art systems suggest storing packet data in locations outside of the CPU, such as programmable switches [14] and RDMA servers [37]; hence, transmitting only packet headers to software Network Functions (NFs) for processing. Consequently, NFs now face unprecedented packet rates ($> 10^8$ packets per second (pps)), necessitating highly optimized systems to support this workload effectively [10]. Utilizing Receive-Side Scaling (RSS) [19] to dispatch traffic among multiple cores on a server is the typical approach to scale software NFs. However, this approach introduces multiple challenges, particularly when the NF is stateful and must maintain state *per flow*. To process a packet, stateful NFs are required to retrieve the stored state from memory, which becomes costly when the needed information is not present in the higher levels of CPU caches or when multiple CPU cores access it simultaneously. To address this challenge, state-of-the-art systems propose solutions to reduce the processing cost by (*i*) using advanced data structures to minimize the average number of memory accesses per packet [4, 13], (*ii*) eliminating the inter-core transfer of data by sending all packets of a flow to the same core [34], and (*iii*) reducing the state sharing overhead using transactional memory [43]. However, these existing solutions do not scale when deploying a chain of stateful NFs on a server, as finding system bottlenecks requires a *detailed* understanding of the semantics and software structure of deployed NFs, which is not feasible with traditional approaches.

The performance of Large Language Models (LLMs) shown in various areas [16, 27] has resulted in many proposals to use LLMs for generating code [15], assisting programmers [35], and optimizing compilers [8]. Motivated by this trend, we investigated applying LLMs to solve the problem of configuring complex NF chains by analyzing software implementations of NFs and extracting information about the software's behavior, semantics, and system-level performance. The extracted information is used to optimize

the software's infrastructure, deployment configuration, and execution pipeline. Relying on LLMs enables us to have a framework-agnostic code analyzer, which is easier to adopt than a domain-specific parser. We introduce FlowMage that utilizes an LLM (*i.e.,* GPT [32], Code Llama [36], and Gemini [39]) to efficiently deploy a chain of stateful NFs on a commodity server and improve performance. More specifically, FlowMage (*i*) analyzes the source code of NFs, (*ii*) extracts meta-information about the implementation of each NF, and (*iii*) automatically calculates & applies an optimized configuration just before deploying the chain. We demonstrate that the meta-information provided by the LLM allows FlowMage to effectively reduce the overheads associated with concurrent memory space accesses, thereby improving the chain's throughput.

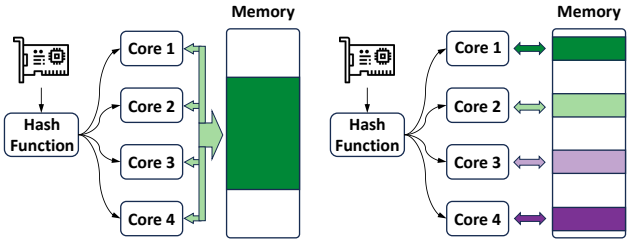
Contributions. To the best of our knowledge, we are the first to (*i*) show the benefits of employing LLMs in the context of packet processing and (*ii*) address the challenges of deploying stateful NF chains with multiple flow granularities. In a nutshell, we:

- Evaluated LLMs ability to extract information about the logical behavior of code, its semantics, and the utilized data structures;
- Developed FlowMage to *automatically* find and apply the optimal configuration *via LLMs* before deploying a chain of stateful NFs;
- Evaluated FlowMage and showed the benefits of optimally deploying stateful NFs on multi-core servers.

2 Motivation

Linear scaling of stateful NFs is a key challenge when deploying networking applications on commodity servers. The overhead of fetching state information from memory is a well-known barrier at high packet processing rates. Therefore, minimizing memory accesses and maintaining state information in higher levels of CPU caches is essential to reduce this overhead [11, 13].

To scale up stateful NFs, one parallelizes the NF by creating an instance per core and relies on the NIC to evenly distribute traffic among available CPU cores.* However, dispatching packets of the same flow to different cores necessitates a synchronization mechanism among these instances to coordinate parallel accesses to shared data structures, leading to significant overhead due to the long memory access time. To eliminate the overhead of synchronization and sharing of data structures, state-of-the-art solutions rely on a hashing function (*e.g.,* Toeplitz) to dispatch all packets of a flow to the same core, enabling a so-called *shared-nothing* architecture. Figure 1 illustrates the difference between shared and shared-nothing architectures when different CPU cores access the memory. NICs typically† use RSS [19] mechanism to distribute traffic among cores. For each packet, RSS calculates the hash of a set of packet header fields and forwards the packet toward a CPU core based on the hash value (typically, using the hash as an index to a table that contains the assigned CPU cores). However, achieving a shared-nothing architecture is challenging as it requires a careful configuration of RSS to be aligned with the flow definition of the deployed stateful NFs [34]. For instance, a Flow Statistics Counter (FSC) [1] keeps statistical information per TCP/UDP connection, requiring RSS to dispatch packets based on the 5-tuple of packets attributes (*i.e.,* source IP, destination IP, source port, destination port, and protocol) to ensure a shared-nothing architecture. However, a Port Scan Detector (PSD) maintains states by source IP to identify potential port scanning activities, thus requiring packet distribution solely based on the source IP.



(a) Shared. **(b)** Shared-nothing.

Figure 1. Different models for parallelizing stateful network functions where per-flow state data is either (a) shared among cores or (b) private per core.

Manually configuring RSS significantly increases the error rate since (*i*) the operator needs to have a comprehensive understanding of the NF behavior and its requirements, and (*ii*) finding the correct RSS configuration is not always trivial [34]. To address this issue, a few works have proposed frameworks for automatically configuring RSS. Maestro [34] proposes a system that automatically analyzes an NF and generates a new implementation that distributes the workload across multiple cores with respect to the NF's semantics. NFOS [43] proposes a programming framework that helps NF-domain experts to develop and optimize scalable NFs without worrying about concurrency complexities and exploits memory transactions [28] to process packets instead of the common lock-based synchronization.

2.1 What are the Current Problems?

Existing solutions to automatically configure RSS are not widely adopted due to several limitations in analyzing the code or complexities of applying them to different packet processing frameworks, as described below:

*We focus on the run-to-completion model [23].

†We do not consider rule-based flow steering mechanisms [18].

Requiring exhaustive symbolic execution. To configure RSS, proposed systems typically run an exhaustive symbolic execution to detect all possible execution paths and extract the behavior and semantics of stateful NFs [33, 34]. This introduces all of the inherent limitations of symbolic execution, such as statically bounding loops while incurring the large overhead of re-running the symbolic execution when adding/modifying NFs.

Relying on code annotations. Existing systems often necessitate developers to adhere to specific coding conventions [34] and supply annotations [43] for accurate code analysis. This prerequisite poses a notable challenge for integrating these systems into established frameworks due to restrictions on data structure usage, the need for substantial modifications to existing frameworks, and the obligation for developers to comply with specified coding standards.

Supporting a single NF deployement. One of the key aspects of using software-based NFs is the ability to deploy a chain of NFs on a commodity server. However, existing systems mainly focus on single NF deployment; they do *not* provide a solution to automatically configure RSS when deploying a *chain* of stateful NFs. Finding the optimal RSS configuration for a NF chain is challenging, as different NFs in the chain may operate based on different flow definitions. Additionally, in some cases where the flow definition of NFs are mutually exclusive, achieving a shared-nothing model is *theoretically impossible*. For instance, linking a PSD and a Policer, which respectively track state information based on source and destination IP addresses, necessitates some level of state sharing among cores as it is not feasible to ensure a shared-nothing model for both NFs at the same time.

2.2 Machine Learning for Networking Systems

The integration of Machine Learning (ML) models into networking systems represents a significant shift towards more intelligent and efficient network infrastructures [5]. Following that, many systems use ML to improve different aspects of networking, such as traffic prediction [45], routing [26], congestion control [44], security [25], and network management & configuration [29, 30, 38, 41]. Additionally, the recent promising performance of LLMs opens new possibilities to further utilize ML-based networking systems. More specifically, most LLMs are built on deep learning architectures [40] with massive datasets containing text, books, and GitHub repositories, which helps them understand programming languages. Therefore, LLMs are well-suited to generate, refine, or analyze code, as shown in several studies [7, 42].

As a result, several research efforts explore the possibility of employing LLMs for static code analysis [9]. For instance, ChatGPT can understand programming languages [6] and create function summaries with more accuracy than static analysis methods, especially with loops and variable-length data structures [24].

How can LLMs help to efficiently deploy stateful NF chains? The notable abilities of LLMs in code analysis provide an opportunity to address the limitations of state-of-the-art solutions. As opposed to Maestro [34] and NFOS [43], LLMs enables us to propose a *framework-agnostic* solution to analyze the behavior, semantics, and performance of NFs without requiring exhaustive symbolic execution or code annotations. Therefore, in this paper, we explore the potential of using LLMs to extract meta-information about NFs found in packet processing frameworks. The modular architecture of these frameworks often allows for the automatic extraction of code associated with each NF, which is usually concise enough to be directly utilized in LLM prompts for analysis. The next section proposes our LLM-based system, FlowMage, which automatically extracts this meta-information and utilizes them to deploy chains of stateful NFs in an optimized way.

3 FlowMage: Optimized NF Chaining

This section proposes leveraging the promising performance of LLMs to efficiently deploy a chain of stateful NFs on commodity servers. Our solution maximizes performance of the chain while requiring only small changes to the existing well-known frameworks.

FlowMage relies on LLMs to analyze the source code of NFs, where the LLM (*i*) provides meta-information about each network function and (*ii*) compares NFs code complexity in terms of the average processing required per packet. To deploy a chain of stateful NFs, FlowMage undertakes a three-step approach (shown in Figure 2). First, a feature tracker keeps the list of all existing NFs and ensures that the extracted meta-information per NF is always available and updated with the latest changes in the NFs code. Second, it accepts a raw configuration file as input, *i.e.,* the standard method for defining chains and configurations of NFs in most existing frameworks. End users do not need to be concerned about the parallelization configuration in the provided file. FlowMage employs a solver to leverage the extracted data from the LLM to identify an optimal RSS configuration, enhancing the chain's performance. Finally, FlowMage generates a configuration file that encapsulates all vital settings for parallelizing the given NFs and the determined RSS configuration. Each step and the system's essential requirements are explained in the subsequent sections.

3.1 Prompt Formulation

FlowMage harnesses LLMs through a two-step process. In the initial phase (*i.e.,* meta-data extraction), FlowMage is designed to monitor modifications and additions of NFs within the source code. Upon detecting changes to an existing NF's code or the introduction of a new NF, it automatically extracts features of the NF immediately after

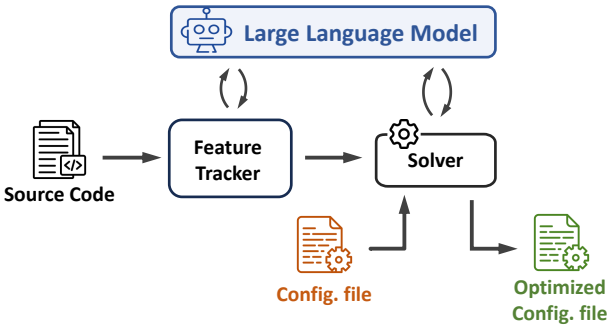


Figure 2. Deployment stages in FlowMage. FlowMage receives a configuration file from the user, utilizes the extracted meta-information per NF, and adds all necessary configurations to optimize the proposed chain.

the source code compilation. This process ensures an up-to-date repository of NF characteristics and stores the extracted data in a special file. The data extracted at this step pertain to the semantics and behavior of the NF, which are crucial for identifying the most effective configuration setup. The features extracted during this step include the following:

Statefulness indicates whether an NF stores state information per flow. If an NF is identified as stateless, FlowMage does *not* consider further parameters since parallelization does not present a scalability challenge.

Flow Definition (F) identifies the flow definition used by the NF. Currently, FlowMage accommodates a *subset* of the 5-tuple attributes of packets for flow definition, which encompasses the majority of stateful NFs. Future versions of FlowMage may expand this capability to cover all potential flow definitions (*e.g.,* a bridge that associates MAC addresses with network interfaces).

Read-Write Intensity (I) identifies the frequency at which states are updated. It plays a critical role in configuring the chain since the adverse effects of state sharing are considerably lower in read-intensive applications than in write-intensive ones, owing to the parallelization overhead, such as locks and inter-core data transfer.

Pointer Chasing (P) indicates the presence of one or more pointers within the state data structure. Pointers lead to irregular memory accesses during packet processing, which can increase system overhead and cause notable performance degradation, especially when states are shared among multiple cores.

To accurately extract the specified information from an NF, FlowMage sends a comprehensive prompt containing (i) a general introduction outlining the expected information from the LLM (T_G), (ii) a section detailing the specific information, listing the required features and the definition of each feature (T_C), (iii) a brief directive specifying the desired response format (T_F), which varies for each feature, and (iv) the relevant source code segments associated with

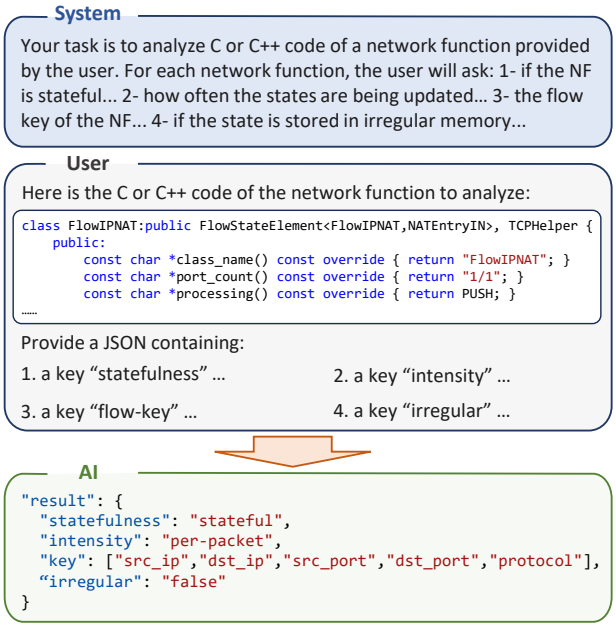


Figure 3. FlowMage automatically crafts a prompt and sends it to an LLM to examine an NF's code and extract features.

the NF, automatically retrieved from the framework's source code by FlowMage (T_S). The final prompt, designed to optimize the accuracy and relevance of the LLM's outputs, is a concatenation of these components ($T_G \| T_C \| T_F \| T_S$). Figure 3 demonstrates a summarized sample of a prompt to examine a stateful NF. In this figure, the top box (*i.e.,* System) contains a message to set the general objectives that the LLM should follow, including T_G and T_C, while the middle box (*i.e.,* User) includes the examining NF's source code (T_S) as well as the desired response format (T_F).

In the subsequent phase (*i.e.,* Code Complexity Evaluation (CCE)), upon receiving a configuration file indicating parameters and chains of NFs, FlowMage employs the LLM to assess and compare the complexity and frequency of state accesses per packet among the stateful NFs. To do so, FlowMage crafts a prompt* and appends it with the source codes corresponding to pairs of NFs, which is then sent to the LLM to determine which NF requires a higher average processing per packet. This procedure is applied to all possible pairs of NFs within the chain to deduce the most optimal configuration. Section 3.2 explains the methodology and mechanisms by which the solver utilizes this information to configure the system.

3.2 Solver

Given the meta-information extracted from each NF, FlowMage transforms an input configuration file into

*We show a sample CCE prompt in Appendix A.

its optimized counterpart. This enhanced configuration file addresses all necessary setups to reduce the overhead associated with concurrent access. It contains information about the optimized RSS configuration, NFs that can ensure a shared-nothing model, and NFs with the shared states among CPU cores. To find the optimized configuration, FlowMage conducts an automatic extraction of the list of existing stateful NFs within a chain. Leveraging the stored meta-information for each NF, an optimization problem aimed at minimizing the packet processing cost across the chain can be formulated as follows:

$$\underset{r}{\text{minimize}} \quad \sum_{i=1}^{N} \text{cost}(I_i, F_i, P_i, r) \quad \text{subject to} \quad r \subset R$$

where R represents the set of all packet attributes that the NIC supports as RSS parameters (*e.g.,* the 5-tuple attributes of packets in the current implementation), N denotes the number of stateful NFs in the chain, and the cost function returns the overhead value of an NF with a given set of features (*i.e.,* I_i, F_i, P_i) and a given set of RSS parameters. In scenarios where the solver identifies multiple configurations yielding equivalent cost outcomes, it resorts to a CCE step that utilizes the LLM to compare the code complexity of NFs in terms of the average processing required per packet. The decision criterion in such cases favors the configuration that assigns the more complex NF to operate under a shared-nothing model. This approach ensures that NFs with intricate processing demands benefit from an architecture that minimizes contention and maximizes performance. However, the CCE step is not mandatory and might be disabled if including the source codes of both NFs in a single prompt risks exceeding the LLM's token limits. In such scenarios, FlowMage treats all NFs as having equivalent processing complexity per packet.

The current version of FlowMage employs a rudimentary cost function that ranks features according to their performance impact, configuring the RSS to ensure that NFs receiving the highest scores are allocated to a shared-nothing architecture. Future versions of FlowMage will introduce a more advanced cost function designed to accommodate a broader spectrum of scenarios and further refine the optimization of packet processing chains.

3.3 Implementation

As previously highlighted, one of the significant advantages of FlowMage is its generality and the minimal modifications required for integration with existing packet processing frameworks. A crucial criteria for a networking framework to be compatible with FlowMage is ① the ability to configure RSS via the input configuration file and ② support the operation of each stateful NF in both shared and shared-nothing models.

To validate the feasibility and effectiveness of FlowMage, we implemented it as a proof of concept on top of FastClick [3], a state-of-the-art packet processing framework. To meet the first criterion, FlowMage is designed to accept RSS parameters directly from the configuration file, ensuring that RSS is configured during the system's initialization phase. Addressing the second criterion—facilitating stateful NFs to function in both shared and shared-nothing environments—FlowMage introduces a base class from which all stateful elements are inherited. This class is designed to read the sharing state of an NF from the configuration file, employing a locking mechanism to regulate concurrent accesses in instances where NFs share state among multiple cores, thus guaranteeing thread safety and maintaining operational integrity.*

FastClick implements a hierarchical, object-oriented architecture for stateful NFs, complicating the process of source code tracking due to the necessity of detecting and monitoring changes across the hierarchy. To simplify the integration of FlowMage across various architectures, we introduce a customizable function that allows for modifications to tailor the code extraction process to fit the structure of the targeted networking framework. Integrating FlowMage into other packet processing frameworks requires modifying this function to align with the framework's structural conventions. Additionally, FlowMage contains framework-agnostic Python scripts to implement feature tracker and solver components. The solver component currently relies on a simple rank-based algorithm to find the optimal configuration; however, switching to a more complicated cost function may require using constraint-based solvers. The source code is available at hamidgh09/FlowMage.

4 Evaluation

This section evaluates FlowMage and demonstrates that LLMs can accurately analyze the behavior of NFs as well as provide some relevant data about the complexity of NFs. Furthermore, we assess the impact of leveraging extracted information on the performance of NF chains with diverse properties.

Experimental Setup To assess the performance of FlowMage, we conducted experiments in a testbed containing two commodity servers interconnected via a 32 × 100-Gbps Edgecore Networks DCS800 Wedge 100BF-32X switch, powered by an Intel® Tofino™ASIC [20]. One server acts as the traffic generator and the other as our Device Under Test (DUT), which runs various chains of stateful NFs. The DUT is equipped with NVIDIA ConnectX®-6 NICs [31] and Intel® Xeon® Gold 6346 CPUs @ 3.10 GHz, featuring 32-KiB per-core L1 instruction, 48-KiB data, and 1.3-MiB per-core L2 caches, alongside a 36-MiB shared Last Level Cache (LLC) with 12 cache ways. To simulate high traffic load on the DUT, the Tofino switch clones incoming packets

*Appendix B illustrates a sample FastClick configuration file.

with varying source IP addresses. To ensure the experiments reflect realistic conditions, we employ CAIDA trace files to represent sample network traffic. Given the inherently low packet rate of these traces, we segment the trace into smaller windows and replay multiple windows concurrently. This approach increases the offered load while preserving the intrinsic characteristics of the traffic. For the experiments involving LLM prompts, we employed the OpenAI API, Google Vertex-AI API, and CodeLlama-34B-Instruct run on an NVIDIA A100 80 GB GPU. To have a fair comparison, we use LangChain [17] to interact with different models.

4.1 Accuracy of Code Analysis

Our initial evaluation concentrated on the capability of current LLMs to analyze the semantics of existing NFs. Given the constraints on the maximum number of input tokens (per prompt) in prevalent models, we utilized OpenAI's GPT-3.5-Turbo, GPT-4-Turbo, CodeLlama-34B-Instruct, and Gemini-1.0-Pro models. These models were selected based on their capacity to handle extensive prompts, thereby accommodating NFs source code within a prompt.

Our primary evaluation metrics are: (*i*) the accuracy of each LLM, (*ii*) the associated costs, and (*iii*) the number of tokens required for extracting the features delineated in Section 3.1. To enhance the robustness and validity of our findings, our sample set of NFs includes existing elements and plugins within both the FastClick [1] and VPP [12] frameworks. The test set comprises 18 modules containing 8 stateless and 10 stateful FastClick element and VPP plugins representing a spectrum of common networking functions and applications. To verify the consistency of the outcomes provided by the LLMs, we set the temperature of models to 0.001 and repeat each prompt 10 times. Table 1 demonstrates the accuracy of each LLM in evaluating these elements and plugins.* The results indicate that GPT-4 Turbo stands out for its high accuracy, highlighting its effectiveness for integration within FlowMage.

Given the modular architecture of current packet processing frameworks, as discussed in Section 3, the code extracted for almost all† NFs generally fits within the prompts without surpassing the input token limits imposed by each model. In our experiments, the average token count utilized for analyzing each NF stood at 7164, with the smallest being 1168 tokens for a stateless MAC address swapper in FastClick, and the largest reaching 31570 tokens for processing an Access Control List (ACL) implemented in VPP. This efficiency in token usage translates to a very low processing cost, with the most extensive NF codebase costing $0.31 per analysis using GPT-4 Turbo. The price is similarly low for other models.

*The detailed results are available in Appendix C.

†Except one that is discussed in Appendix C.

Table 1. Accuracy of LLMs in analyzing FastClick elements and VPP plugins. Each cell denotes the number of correct assessments out of 18 NFs in total. Llama refers to CodeLlama-34B-Instruct, and Gemini refers to Gemini-1.0-Pro.

Attribute	OpenAI GPT		Llama	Gemini
	4 Turbo	3.5 Turbo		
Statefulness	18/18	15/18	11/18	15/18
Flow definition	10/10	8/10	4/10	8/10
R/W intensity	9/10	7/10	3/10	6/10
Pointer chasing	9/10	6/10	4/10	6/10

How reliable is the estimate of code complexity? To assess the precision of LLMs in evaluating the code complexity between pairs of stateful NFs, we formed 30 distinct NF pairs and utilized GPT-4-Turbo for the complexity comparison. Our results show that GPT-4-Turbo consistently identified the NF with more codebase complexity in all scenarios. The experiments had an average token usage of 13661, peaking at 46107 and bottoming out at 4379. Note that, we limited our analysis to GPT-4-Turbo exclusively as each prompt included the source code for two NFs, thereby exceeding the token capacity of the other models (*i.e.,* GPT-3.5 Turbo and Gemini-1.0-Pro) or causing significantly high response time (when using CodeLlama-34B-Instruct) in some scenarios.

4.2 FlowMage's Performance Improvement

To highlight the benefits of FlowMage, we set up several chains of stateful NFs on our DUT. Our objective is to compare the highest throughput achieved when employing FlowMage's optimizations against those obtained using a standard deployment configuration, where all NFs share states across cores to represent state-of-the-art systems. For these experiments, we rely on the unmodified version of FastClick (*i.e.,* vanilla FastClick).

In our initial experiments, we create a chain combining a Policer and a Source IP Tracker. The Policer maintains state information for each destination IP address, while the Source IP Tracker monitors traffic data for each source IP address. This chain serves as an illustrative example due to (*i*) its simplicity, featuring only two NFs, (*ii*) the mutually exclusive flow definitions of the NFs preventing a shared-nothing model for both simultaneously; and (*iii*) the minimal per-flow data storage requirements of the two NFs, which allows us to assess the effectiveness of FlowMage in minimizing state sharing overhead.

Figure 4 demonstrates the maximum throughput achieved by FlowMage in comparison to vanilla FastClick across a range of CPU core allocations. Our results show that vanilla FastClick saturates at 60 Mpps upon scaling up to

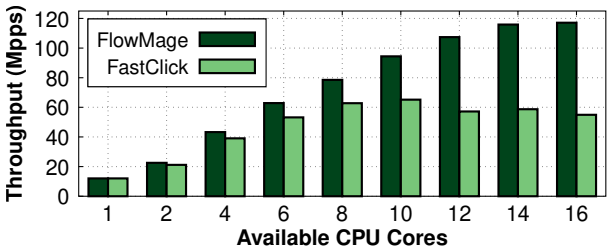


Figure 4. FlowMage improves the performance of a chain of network functions consisting of a Policer and a Source IP Tracker by up to 2× by efficiently deploying the chain.

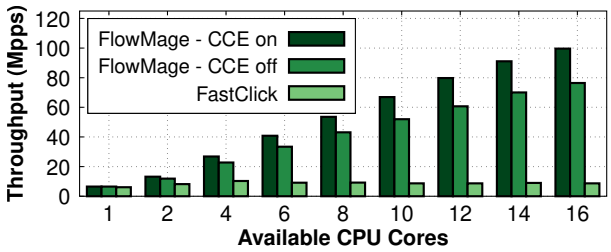


Figure 5. FlowMage can successfully find the efficient RSS configuration when deploying a chain of NFs composed of a Policer, a FSC, and a PSD. Using CCE further amplifies the performance improvements achieved by FlowMage.

8 cores. This is attributed to the fact that as more cores are introduced, the system requires more coordination for state access among these cores, leading to throughput saturation.

In addition to the previous experiment, we executed a more intricate scenario by chaining three stateful NFs—a Policer, a FSC, and a PSD—on our DUT. This experiment highlights FlowMage's ability to identify the most optimized configuration and shows the impact of employing the LLM for complexity comparison within the solver component.

The Policer and the PSD maintain flow states per destination and source IP addresses, respectively, whereas the FSC keeps data per 5-tuple attributes of packets. Leveraging the meta-information extracted for each NF, FlowMage detects that dispatching packets according to the source IP address among cores facilitates a shared-nothing architecture for the PSD and FSC, although the Policer requires state sharing across cores. Conversely, utilizing the destination IP address as the RSS key allows the Policer and FSC to adopt a shared-nothing model, leaving the PSD to share states. Confronted with two equally viable solutions, if the CCE feature in the solver component is enabled, the solver automatically uses the LLM to compare the complexity of the Policer and PSD codebases. This comparison enables FlowMage to select the optimal RSS configuration, favoring the NF with the more complex codebase.

Figure 5 shows the significant performance improvement (*i.e.,* 8.4×) offered by FlowMage over FastClick in this scenario when various CPU cores are allocated to the processing framework. In the FastClick, RSS dispatches packets based on the 5-tuple attribute, enabling a shared-nothing model for the FSC but necessitating shared states for both the Policer and the PSD. Additionally, the figure illustrates that enabling CCE feature could further improve the throughput by up to 30% (11× in comparison to FastClick) in such scenarios. It is important to note that FastClick's poor performance in this scenario stems from the significant sharing overhead of the PSD, which necessitates sharing relatively large states across cores. The numbers align with the results reported in other studies (*e.g.,* Maestro [34]).

5 Discussion

Would in-context learning or fine-tuning of LLMs affect the system's accuracy? In its current version, FlowMage leverages LLMs to identify features with discrete values, allowing for precise responses without the need for in-context learning or model fine-tuning. Additionally, given the extensive token requirements for each prompt, incorporating sample code for in-context learning is impractical, as it might further strain the token limit or reduce the accuracy. Nevertheless, fine-tuning presents a promising path for enhancing accuracy, especially for features expressed as continuous numbers. For instance, a fine-tuned LLM can be used to (*i*) extract the state size maintained by each NF per flow or (*ii*) predict the average number of irregular memory accesses to process a packet which enhances the accuracy of FlowMage in detecting the optimized RSS configuration. Future iterations of FlowMage can explore the integration of these capabilities, potentially elevating the model's predictive precision and utility in assessing NFs.

Are there other useful information to extract from the source code of NFs? LLMs can analyze not just high-level programming languages but also understand the low-level syntax of network functions, such as assembly or LLVM IR bitcode. This capability enables LLMs to potentially estimate system-level performance metrics for NFs (*e.g.,* the average number of instructions or CPU cycles required per packet), which are challenging to perform when using traditional approaches such as exhaustive symbolic execution. These metrics are essential for a variety of purposes, including efficient resource allocation to NFs [22].

Does RSS configuration affect the load imbalance among CPU cores? In some scenarios where the network traffic contains packets with a limited range of values per 5-tuple attributes, the configuration of RSS can indeed influence the distribution of load across CPU cores. More specifically, if packet dispatching relies on a small subset of 5-tuple attributes, it can potentially lead to load imbalances, partial packet drops, and increased tail latencies. However,

we did not observe this issue in our experiments as we utilized CAIDA trace files with a large diversity of value for all attributes of packets. While the current iteration of FlowMage does not directly tackle these challenges, the framework-agnostic design of FlowMage allows for the incorporation of existing solutions like RSS++ [2] to mitigate potential load imbalances effectively. We leave addressing this issue as a future work.

What are the consequences if the LLM is inaccurate? LLMs are subjected to hallucinations and inconsistent responses. For instance, Section 4.1 showed that LLMs do not always provide correct analysis of NFs. Incorrect responses may cause performance degradations or violate application semantics. More specifically, semantic violations can happen if the extracted flow definition (F) of an NF is a *superset* of the correct flow definition. For example, if an LLM incorrectly identifies the flow definition of a PSD as source IP and source port (instead of the source IP address), the PSD will not be able to track destination port used by a user, as packets with the same source IP address and various source ports end up in different cores, which cause incorrect behavior when using shared-nothing architecture. To minimize the consequences of inaccurate code analysis, integrating an LLM into a production-grade deployment system requires performing additional checks and verification (*e.g.,* keep humans in the loop) to ensure the correctness of the responses generated by the LLM. Moreover, it is possible to combine existing deterministic approaches with LLMs to eliminate the risk of inaccuracies causing semantic violations. For instance, an LLM can be used to guide symbolic execution engines.

6 Conclusion

The rise of LLMs has opened up new opportunities to build and optimize networked systems. This paper demonstrated the benefits of using LLMs to perform code analysis and extract useful information. While we primarily focus on analyzing NFs code, we believe other networking applications could potentially benefit from code analysis capabilities of LLMs. We hope our work motivates further research in this direction.

7 Acknowledgements

The authors wish to thank Gerald Q. Maguire Jr. for his helpful feedback on this work and the anonymous reviewers for their valuable comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

References

[1] Tom Barbette. 2015. GitHub - FastClick. https://github.com/tbarbette/fastclick https://github.com/tbarbette/fastclick.

[2] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. ACM, New York, NY, USA, 318–333. https://doi.org/10.1145/3359989.3365412 http://doi.acm.org/10.1145/3359989.3365412.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Oakland, California, USA) *(ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16. http://dl.acm.org/citation.cfm?id=2772722.2772727

[4] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2021. Combined Stateful Classification and Session Splicing for High-Speed NFV Service Chaining. *IEEE/ACM Trans. Netw.* 29, 6 (dec 2021), 2560–2573. https://doi.org/10.1109/TNET.2021.3099240

[5] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. 2018. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications* 9, 1 (2018), 1–99.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]

[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:2107.03374 [cs.LG]

[8] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. arXiv:2309.07062 [cs.PL]

[9] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2023. Large Language Models for Code Analysis: Do LLMs Really Do Their Job? arXiv:2310.12357 [cs.SE]

[10] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward per-core 100-Gbps Networking. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 17 pages. https://doi.org/10.1145/3445814.3446724

[11] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[12] FD.io. 2017. *Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server*. Technical Report. Cisco, Intel Corporation, FD.io. https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf

[13] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 807–827. https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni

[14] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. 2020. *Parking Packet Payload with P4*. Association for Computing Machinery, New York, NY, USA, 274–281. https://doi.org/10.1145/3386367.3431295

[15] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2201–2203. https://doi.org/10.1145/3611643.3617850

[16] Walid Hariri. 2023. Unlocking the Potential of ChatGPT: A Comprehensive Exploration of its Applications, Advantages, Limitations, and Future Directions in Natural Language Processing. arXiv:2304.02017 [cs.CL]

[17] Chase Harrison. 2024. LangChain. https://github.com/langchain-ai/langchain Accessed 2024-03-23.

[18] Intel. 2016. Ethernet Flow Director. http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html

[19] Intel. 2016. Receive-Side Scaling (RSS). http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[20] Intel. 2022. Intel Tofino Series. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[21] Intel Barefoot Networks. 2020. Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs. https://www.barefootnetworks.com/products/brief-tofino-2/

[22] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 517–530. https://www.usenix.org/conference/nsdi19/presentation/iyer

[23] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf

[24] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

[25] Yuancheng Li, Rong Ma, and Runhai Jiao. 2015. hybrid malicious code detection method based on deep learning. *International Journal of Security and Its Applications* 9, 5 (2015), 205–216.

[26] Shih-Chun Lin, Ian F. Akyildiz, Pu Wang, and Min Luo. 2016. QoS-Aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement Learning Approach. In *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, San Francisco, CA, USA, 25–33. https://doi.org/10.1109/SCC.2016.12

[27] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Lin Zhao, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. 2023. Summary of ChatGPT-Related research and perspective towards the future of large language models. *Meta-Radiology* 1, 2 (2023), 100017. https://doi.org/10.1016/j.metrad.2023.100017

[28] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-aid: Detecting and surviving atomicity violations. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 277–288.

[29] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 196–204. https://doi.org/10.1145/3626111.3628183

[30] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What Do LLMs Need to Synthesize Correct Router Configurations?. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/3626111.3628194

[31] NVIDIA Mellanox. 2019. ConnectX®-6 EN IC 200GbE Ethernet Adapter IC. https://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf

[32] OpenAI. 2024. Models. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo Accessed 2024-02-29.

[33] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 372–385. https://doi.org/10.1145/3230543.3230573

[34] Francisco Pereira, Fernando M. V. Ramos, and Luis Pedrosa. 2023. Automatic Parallelization of Software Network Functions. arXiv:2307.14791 [cs.NI] to be published in NSDI24.

[35] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[36] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[37] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1237–1255. https://www.usenix.org/conference/nsdi23/presentation/scazzariello

[38] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 41–47. https://doi.org/10.1145/3626111.3628205

[39] Gemini Team. 2023. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL]

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., San Diego, CA, USA. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[41] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. 2023. Making Network Configuration Human Friendly. arXiv:2309.06342 [cs.NI]

[42] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

[43] Lei Yan, Yueyang Pan, Diyu Zhou, George Candea, and Sanidhya Kashyap. 2024. Transparent Multicore Scaling of Single-Threaded Network Functions. In *19th ACM European Conference on Computer Systems (EuroSys '24)*. ACM, New York, NY, USA, 16 pages.

[44] Ticao Zhang and Shiwen Mao. 2020. Machine learning for end-to-end congestion control. *IEEE Communications Magazine* 58, 6 (2020), 52–57.

[45] Haifeng Zheng, Feng Lin, Xinxin Feng, and Youjia Chen. 2020. A hybrid deep learning model with attention-based conv-LSTM networks for short-term traffic flow prediction. *IEEE Transactions on Intelligent Transportation Systems* 22, 11 (2020), 6910–6920.

A CCE Prompt Structure

As outlined in Section 3.2, in situations where FlowMage identifies multiple viable configurations for RSS, it proceeds with a CCE phase. This phase involves comparing the codebase complexity of NFs and selecting a configuration that benefits the NF requiring the most processing per packet. FlowMage automatically generates prompts that include the source code of NF pairs within a chain and submits these prompts to an LLM. Figure 6 provides an example of such generated prompts.

System

Your task is to analyze C or C++ code of two network functions provided by the user. The user will ask you to estimate which one of the two NFs requires more processing per packet.

User

Here is the C or C++ code of the network function to analyze:

```
class FlowIPNAT:public FlowStateElement<FlowIPNAT,NATEntryIN>, TCPHelper {
    public:
        const char *class_name() const override { return "FlowIPNAT"; }
        const char *port_count() const override { return "1/1"; }
        const char *processing() const override { return PUSH; }
.....
```

```
class IPortScanDetector : public IFlowManager<IPSDState> {
public:
    const char *class_name() const override { return "IPortScanDetector";}
    const char *port_count() const override { return PORTS_1_1;}
    const char *processing() const override { return PUSH;}
.....
```

Provide a JSON containing a single key 'complexity' that contains the name of the NF that requires more processing per packet.

AI

```
"result": {
  "complexity": "IPortScanDetector"
}
```

Figure 6. FlowMage automatically crafts a prompt and sends it to an LLM to compare the codebase complexity of two NFs if it can not decide on the best RSS configuration based on initially extracted attributes of each NF.

B Sample FlowMage's Configuration

To apply the optimized configuration, FlowMage receives a FastClick configuration file as input, automatically identifies the existing elements within the chain, and modifies the configuration of each element to maximize performance. Figure 7 illustrates a sample scenario in which a PSD, a Policer, and a FSC are interconnected in a chain, with 16 cores allocated to the system. In this setup, FlowMage sets the RSS configuration to dispatch packets based on the source IP address (see the modified parameter in the "FromDPDKDevice" element). Additionally, it configures the Policer instances to share states across cores while the others achieve a shared-nothing model.

Input FastClick Configuration

```
.....
fd0 :: FromDPDKDevice($port, MAXTHREADS 16, NDESC 256)
-> Classifier(12/0800)
-> MarkIPHeader(OFFSET 14)
-> psd :: IPortScanDetector(SIZE 8000000)
-> policer :: IPolicer(SIZE 8000000)
-> flowCounter :: IFlowCounter(SIZE 64000000)
-> td0 :: ToDPDKDevice($port, VERBOSE 99, TIMEOUT -1);
.....
```

FlowMage Output Configuration

```
.....
fd0 :: FromDPDKDevice($port, HASHFUNC "src_ip", MAXTHREADS 16, NDESC 256)
-> Classifier(12/0800)
-> MarkIPHeader(OFFSET 14)
-> psd :: IPortScanDetector(SIZE 8000000, SHARED false)
-> policer :: IPolicer(SIZE 8000000, SHARED true)
-> flowCounter :: IFlowCounter(SIZE 64000000, SHARED false)
-> td0 :: ToDPDKDevice($port, VERBOSE 99, TIMEOUT -1);
.....
```

Figure 7. FlowMage automatically modifies an input configuration file to maximize the chain's performance. Modified parameters are denoted in green.

C Detailed Analysis of LLMs' Accuracy

We provide a detailed report of results obtained from evaluating the accuracy of LLMs in analyzing NFs with diverse characteristics. Table 2 enumerates the FastClick elements and VPP plugins that were included in our study. This selection contains a mix of stateful and stateless modules that are commonly used in various Ethernet-based NFs. Moreover, the table demonstrates the effectiveness of each LLM in identifying stateful NFs, *i.e.,* indicating the precision of each LLM in differentiating between stateful and stateless network functions based on the provided source codes.

Additionally, Table 3 provides a brief report about the correct properties of examined stateful NFs, which serve as the benchmark for assessing the LLMs' accuracy. A comparison of the results across different LLMs, as depicted in Table 4, reveals that GPT-4-Turbo exhibits near-perfect accuracy in detecting the correct properties of the attributes under test in these experiments.

It is important to note that, in certain instances (*e.g.,* the VPP's ACL plugin), the extensive size of the codebase either exceeds the model's input token limit or significantly extends the response time. Such instances are marked with a "-" in the corresponding tables and are considered as failures in our evaluation of model accuracy, reported in Section 4.1.

Table 2. All elements and plugins used in Section 4.1 to evaluate the accuracy of LLMs on detecting statefulness. GPT models refer to the Turbo versions, Llama refers to CodeLlama-34B-Instruct, and Gemini refers to Gemini-1.0-Pro. ✓, ✗, and - represent correct response, wrong response, and failed tests, respectively.

NF (element/plugin)	Framework	Description	Model's Accuracy			
			GPT-3.5	GPT-4	Llama	Gemini
IPortScanDetector	FastClick	Stateful - Counts destination ports each host has touched.	✓	✓	✓	✓
IPolicer	FastClick	Stateful - Limits users' download rate.	✓	✓	✓	✓
ISourceCounter	FastClick	Stateful - Tracks active connections per source IP.	✓	✓	✗	✓
FlowRateLimiter	FastClick	Stateful - Limits packet rate per TCP/UDP connection.	✓	✓	✗	✓
FlowHyperScan	FastClick	Stateful - Flow-based IDS using the HyperScan library.	✓	✓	✗	✓
FlowCounter	FastClick	Stateful - Counts the number of flows & packets per flow.	✓	✓	✓	✓
FlowIPLoadBalancer	FastClick	Stateful - TCP & UDP load-balancer	✓	✓	✗	✓
CheckIPHeader	FastClick	Stateless - Verifies correctness of IP headers	✓	✓	✓	✓
AverageCounter	FastClick	Stateless - Measures historical packet count and rate	✓	✓	✓	✗
BatchStats	FastClick	Stateless - Tracks statistics about received batches.	✗	✓	✓	✓
EtherEncap	FastClick	Stateless - Encapsulates packets in Ethernet header.	✓	✓	✓	✓
EtherMirror	FastClick	Stateless - Swaps Ethernet source and destination.	✓	✓	✓	✓
HashSwitch	FastClick	Stateless - Classifies packets by hash of contents.	✓	✓	✓	✓
EnsureEther	FastClick	Stateless - Ensures Ethernet encapsulated IP packets.	✓	✓	✓	✓
nat64	VPP	Stateful - Network Address and Protocol Translator	✓	✓	-	✓
lb	VPP	Stateful - Maglev-like load balancer.	✓	✓	-	✓
acl	VPP	Stateful - Access Control List	-	✓	-	-
adl	VPP	Stateless - Source address allow/deny list	✗	✓	✓	✗

Table 3. Attributes of stateful elements and plugins used in Section 4.1 to evaluate the accuracy of LLMs.

NF (element/plugin)	Flow Definition	R/W Intensity	Pointer Chasing
IPortScanDetector	src_ip	Per Packet	Yes
IPolicer	dst_ip	Per Packet	No
ISourceCounter	src_ip	Per Packet	No
FlowRateLimiter	5-tuple	Per Packet	No
FlowHyperScan	5-tuple	Per Packet	No
FlowCounter	5-tuple	Per Packet	No
FlowIPLoadBalancer	5-tuple	Per Flow	No
lb	5-tuple	Per Flow	Yes
acl	5-tuple	Per Flow	Yes
nat64	5-tuple	Per Packet	Yes

Table 4. The ability of LLMs to extract attributes of stateful elements/plugins in detail. GPT models refer to the Turbo versions, Llama refers to CodeLlama-34B-Instruct, and Gemini refers to Gemini-1.0-Pro. ✓, ✗, and - represent correct response, wrong response, and failed tests, respectively.

NF (element/plugin)	Flow Definition				R/W Intensity				Pointer Chasing			
	GPT-3.5	GPT-4	Llama	Gemini	GPT-3.5	GPT-4	Llama	Gemini	GPT-3.5	GPT-4	Llama	Gemini
IPortScanDetector	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
IPolicer	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
ISourceCounter	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
FlowRateLimiter	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
FlowHyperScan	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓
FlowCounter	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓
FlowIPLoadBalancer	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✗	✓
lb	✓	✓	-	✓	✓	✗	-	✗	✗	✓	-	✗
acl	-	✓	-	-	-	✓	-	-	-	✓	-	-
nat64	✓	✓	-	✓	✓	✓	-	✓	✗	✗	-	✗