

A System for Fast, Accurate, and Intuitive Querying of Large-Scale BGP Datasets

Thomas Holterbach
bgproutes.io

Thomas Krenc
IJ Research Laboratory

Thomas Alfroy
bgproutes.io

KC Claffy
CAIDA/UC San Diego

Abbas Mohsenpour
UCLouvain

Cristel Pelsser
UCLouvain

chat.bgproutes.io

ABSTRACT

BGP data collected from operational routers by platforms such as RIPE RIS or RouteViews is essential to address connectivity issues and analyze the global Internet's structure. However, the vast and ever-growing volume of this data makes deriving useful insights challenging: Existing APIs and dashboards offer limited perspectives, while analyzing large archives of MRT files is slow and cumbersome.

We present *ChatBGP*, a domain-specific chatbot that turns plain-English questions about BGP data into Python code that runs swiftly and returns accurate answers. *ChatBGP* is enabled by four contributions: (i) a measurement study showing that BGP data is highly redundant and therefore highly compressible; (ii) a database scheme that exploits this redundancy to compress BGP data while enabling fast retrieval of specific data elements; (iii) an expressive API that exposes this database through a simple interface; and (iv) a prompt-engineering scheme that guides ChatGPT to synthesize optimized Python code that uses this API to accurately answer user queries, while retrieving only the data needed to answer each query.

With *ChatBGP*, gaining insights into BGP data becomes effortless and significantly faster—e.g., delivering results in seconds instead of hours with existing tools. Network operators can rapidly diagnose connectivity issues, including security threats, researchers can analyze larger datasets while improving reproducibility, and students can engage with BGP monitoring in a more intuitive and interactive way.

CCS CONCEPTS

• Networks → Network monitoring;

KEYWORDS

Internet measurement, BGP, Routing Security

ACM Reference Format:

Thomas Holterbach, Thomas Alfroy, Abbas Mohsenpour, Thomas Krenc, KC Claffy, and Cristel Pelsser. 2026. A System for Fast, Accurate, and Intuitive Querying of Large-Scale BGP Datasets. In *Proceedings of the 2026 ACM Internet Measurement Conference (IMC '26)*, October 12–16, 2026, Karlsruhe, Germany. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3777912.3839829>



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

IMC '26, October 12–16, 2026, Karlsruhe, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2327-8/26/10.
<https://doi.org/10.1145/3777912.3839829>

1 INTRODUCTION

“Which AS neighbor of AS3333 is using AS path prepending?”, “Did prefix 8.0.0.0/12 experience a peak of withdrawals yesterday?” These are typical questions network operators may ask daily to resolve critical BGP issues. Similarly, researchers studying the global structure of the Internet may ask “What is the average length of an AS path traversing a Tier-1 AS?”. The BGP data collected by platforms like RIPE RIS [2] and RouteViews [112] serves as a gold mine for answering such questions. This data is sourced from hundreds of independent operational BGP routers, a.k.a Vantage Points (VPs), which network operators voluntarily configure to export their BGP route updates to these collection platforms.

However, deriving insights from large volumes of BGP data is challenging and often requires narrowing the analysis to a subset of the data, compromising the quality of inferences [6, 7]. For historical analyses, users must write scripts to download, decompress, and parse many (sometimes tens of) gigabytes of data archived in compressed MRT files [17]. Even studying an isolated incident requires downloading and processing many files spanning hundreds of VPs and prefixes to distill the relevant pieces. Alternatively, public and commercial actors provide APIs and dashboards (e.g., RIPEstat [89]) to deliver key insights into the collected data. But such APIs require computing, storing, and indexing massive quantities of metadata alongside the raw data [41], which exacerbates data management and sustainability challenges.

The BGP data deluge—exacerbated by the recent adoption of BMP for data collection [33]—is an inherent aspect of the evolving digital ecosystem, but its relentless growth puts two important stakeholders—network defenders and scientific researchers—at a decided disadvantage. As the saying goes: attackers have to succeed only once, but defenders have to succeed all the time (to keep attackers out). Specifically, malicious actors can exploit the existing barriers to processing gigabytes of BGP data for anomaly detection, giving them a significant advantage when using BGP as a vector for infrastructure compromise. More generally, network researchers face the same asymmetric resource landscape—the infrastructure and its complexity grows faster than our scientific understanding of it. Thus, in our view, one of the most ethically-aligned uses of AI technologies is to shift this balance back toward defenders

and scientists—those motivated to shed transparency on network dynamics.

ChatBGP. We present *ChatBGP*, a domain-specific chatbot that processes user queries about BGP data written in plain English and generates optimized Python code that executes swiftly and delivers accurate answers. *ChatBGP* offers two transformative capabilities. First, it handles a wide range of queries of BGP data, including complex ones requiring analyses or even plotting of data on graphs. Second, *ChatBGP* scales effectively, making it an ideal system to accommodate the expected growth in the number of VPs [5, 7].

ChatBGP runs at `chat.bgproutes.io`. It stores several weeks of BGP data collected from literally all public BGP data collection platforms—bgproutes.io [32], RIPE RIS [2], RouteViews [112], PCH [93], and CGTF [20]—encompassing more than 5,000 vantage points (VPs), and its API handles hundreds of queries per day from researchers and operators.

Impact. We believe *ChatBGP* establishes new standards for BGP data storage and sharing, with the potential to become the de facto tool for analyzing BGP data. *ChatBGP* will accelerate operational security beyond what state-of-the-art tools offer today. For instance, a forged-origin BGP hijack detection system like DFOH [34] can identify a forged AS path, but without contextual insights, deeper investigations and timely detection of related compromises—such as serial hijacking [110]—is challenging. With *ChatBGP*, network defenders can query and analyze in a few minutes the updates observed within a specific timeframe around the detected hijack and that include resources (e.g., IP prefixes or ASes in the AS path) associated with the hijacking AS. Researchers can use *ChatBGP* to process more extensive datasets efficiently, fostering reproducibility—a persistent challenge in our community [11]. Finally, *ChatBGP*'s intuitive interface makes it an excellent tool for teaching BGP monitoring, aligning with emerging trends in AI-driven education [79].

Key contributions. Advances in LLMs have spurred the emergence of domain-specific chatbots that provide insights into datasets, including applications in networking [8, 42]. However, designing *ChatBGP* presented unique challenges due to the nature of BGP data, its substantial volume, and the complexity of user queries. To address these challenges, we first identify *longitudinal redundancy* in BGP data and design a *BGP data storage scheme* that exploits this redundancy to losslessly compress the data. Our scheme not only minimizes storage requirements but also enables fast and flexible retrieval (e.g., using regular expressions), and is compatible with both OLTP databases (i.e., row-oriented), and OLAP databases (i.e., columnar-oriented), making it useful across various deployment settings. Second, we curated a set of common BGP-related questions from mailing lists and research papers to inform the design of *four API endpoints*, which can be composed programmatically (e.g., in a Python script) to answer a wide range of relevant BGP questions quickly and concisely. Finally, we perform *domain-specific prompt engineering* of ChatGPT so that it generates Python code that uses our API and its optimization parameters to answer BGP-related questions accurately and without retrieval of irrelevant data, to maximize speed.

Outline The structure of this paper is as follows:

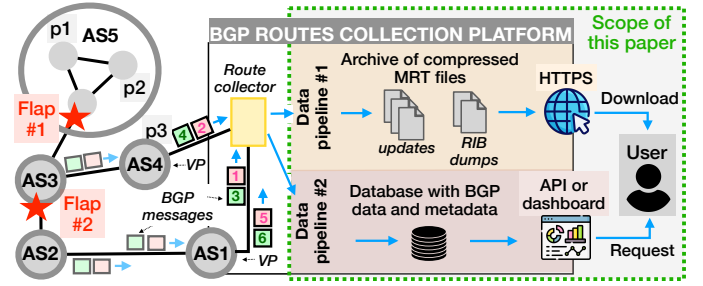


Figure 1: Current BGP data collection, storage and distribution pipelines. Blue arrows show all steps the data follows.

- We provide background on how BGP data is shared and analyzed today (§2), along with key requirements for a system that distributes BGP data to users (§3).
- We demonstrate that existing methods for accessing BGP data fail to meet these requirements (§4).
- We introduce *ChatBGP* and outline its workflow (§5).
- We detail *ChatBGP*'s three components: its BGP data storage method (§6), expressive and simple API (§7), and its domain-specific prompt-engineering scheme (§8).
- We evaluate *ChatBGP*, showing that it accurately answers a wide range of queries, often orders of magnitude faster than state-of-the-art techniques (§9), and that its design is sound across diverse deployment settings.
- We show how *ChatBGP* is deployed and used in practice (§10).

2 BACKGROUND

BGP data collection platforms. These platforms connect, often through *route collectors*, to many BGP routers (also called peers or VPs), collect the BGP updates that those routers export (usually via BGP, but recently also through BMP [33]) and share the data. RIPE RIS (RIS) [2] and RouteViews (RV) [112] are among the most widely-used public platforms, peering with hundreds of routers. Fig. 1 illustrates how these platforms operate using a scenario where AS1 and AS4 act as VPs. ASes are represented as gray circles, while routers are depicted as gray nodes (a circled gray node represents an AS with a single router). AS4 originates prefix p3, while AS5 originates p1 and p2. For simplicity, we assume no BGP policies are configured and focus on prefixes and AS paths, omitting other attributes such as community values. We also assume BGP has converged and analyze the BGP update messages (referred to as *updates*) exported by AS1 and AS4 following two link-flap events: one between AS3 and AS5, and another between AS2 and AS3. A flap refers to a link going down and being restored shortly afterward.

Types of BGP data. RIS and RV publish two types of data.

BGP updates are routing messages that exchange reachability information between two BGP routers. These messages include three main *components*: the *Network Layer Reachability Information* (NLRI) field which lists prefixes that are newly reachable through the sending router; the *BGP attributes*, which store characteristics about the reachable routes (e.g., AS path or community values); and the *Withdrawn Routes* field, which lists prefixes that are no longer reachable through the sending router. For example, after the second flap, AS2

sends a BGP update to *AS1* with *p1*, *p2* and *p3* in the withdrawn route field, i.e., these prefixes are no longer reachable through *AS2*. Once the connectivity is restored, *AS2* advertises the reachable prefixes via two separate BGP updates. The first update received by *AS1* includes both *p1* and *p2* in the NLRI field as they share identical attribute values: their AS path is identical: *AS2-AS3-AS5*. The second update includes *p3* in the NLRI field with the AS path *AS2-AS3-AS4*. All these BGP updates are sent to the BGP data collection platform in real time, either over a dedicated BGP session or via BMP [103].

RIB dumps capture a snapshot of a VP's Routing Information Base (RIB) at a specific moment in time. They include BGP attributes (e.g., AS path, community values) for every prefix observed by the VP. As of Dec. 2024, if a VP has visibility into all IPv4 prefixes announced (a.k.a. a "full feeder"), its RIB contains ≈ 1 million entries, one for each prefix [22].

Storing and sharing BGP data. There are two common methods for storing and sharing BGP data (Fig. 1).

Archive of compressed MRT Files. MRT is a standardized format designed to store routing information from various protocols, including BGP (updates and RIB dumps) [10].

- BGP updates in MRT files contain raw messages received by the router, preserving native BGP optimizations, e.g., grouping prefixes that share identical attribute values (Fig. 2). RIS (resp. RV) compresses these files using gzip (resp. bzip2) and archives them every five (resp. 15) minutes.
- RIB dumps in MRT format store one entry per prefix without applying any optimization. RIS (resp. RV) compresses these files using gzip (resp. bzip2) and archives them every two (resp. eight) hours.

Both RIS and RV archive BGP updates and RIB dumps on a per-route-collector basis, i.e., data from multiple VPs is mixed within the same MRT files (as in Fig. 2). Recently though, RIS has started offering per-VP MRT files, but these are only retained for a couple of weeks, limiting their usefulness for long-term historical analysis [98]. Note that the MRT format includes all BGP attributes. However, only the AS path and community values are relevant for BGP monitoring globally. The others (e.g., local-pref or MED) have only a local scope and we did not consider them in developing *ChatBGP*.

Users can download, decompress, and parse MRT files to extract BGP data. This process can be streamlined using libraries like *bgpstream* and *bgpkit* [14, 92]. For a more interpretable representation, users can display updates and RIB dumps prefix-by-prefix using tools such as *bgpreader*, which is part of the *bgpstream* toolkit [92]. In the scenario depicted in Fig. 1, this representation returns eight lines during the first flap (four withdrawals and four announcements) and six lines during the second flap (three withdrawals and three announcements), as shown in Fig. 2.

API or Dashboard. A more customized approach involves storing BGP data—or metadata such as aggregated statistics—in a database and providing access to users via an API or a dashboard. For instance, RIPEstat [89] and the RV API [101] offer an API with multiple endpoints, allowing users to efficiently retrieve insights from collected BGP data, such as AS neighbor lists, prefix counts or RIB entries. Commercial platforms like ThousandEyes and Kentik that

run their own BGP route collection platform offer proprietary APIs and dashboards for accessing and analyzing their BGP data.

3 REQUIREMENTS

We outline the key requirements for a system like *ChatBGP*, which aims to efficiently store and provide BGP data to users.

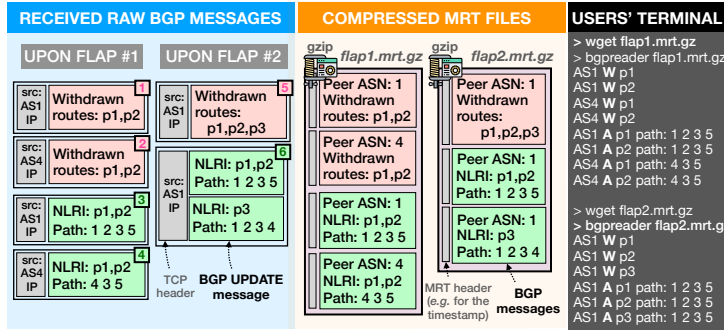
Flexible: The system must support a wide range of queries about BGP data. BGP data serves a variety of purposes, e.g., network operators use it to monitor reachability to their prefixes and researchers use it to map the AS-level topology. In Table 7 (Appendix), we list 39 BGP-related questions commonly asked by operators and researchers, with a sample highlighted in Table 1. We derived these questions from mailing lists and research papers using a methodology described in §8. This list shows that BGP data is used to address a wide range of questions spanning all BGP data components (e.g., prefix, AS path, community values). Regardless of the query's focus, as long as it pertains to BGP data (either raw BGP messages or RIB dumps) and does not require external data, the system must enable users to answer their questions.

Fast: The system must deliver fast responses. Timely responses are essential for network operators to address critical BGP issues, such as attacks or misconfigurations, which can otherwise result in connectivity losses for users and revenue losses for companies. Additionally, fast query responses empower researchers to scale their experiments and produce more reliable results, instead of being forced to sample to save resources [7].

Easy: The system must be easy to use. Misconfigurations are frequent when (re)configuring BGP [82]. Simple, automated tools for analyzing BGP data can streamline monitoring and reduce errors. Moreover, a user-friendly design can support education by enabling students to access and analyze real BGP data during practical exercises without requiring extensive programming or networking knowledge.

Scalable: The system must be scalable and cost-efficient in terms of storage. Studies have shown that current platforms need to deploy significantly more VPs to enhance BGP visibility [7]. This visibility gap motivated the design of a new generation of BGP data collection platforms (e.g., *bgproutes.io* and *bgp.tools*) that aim to expand coverage by adopting automated open peering policies (as opposed to traditional platforms such as RIS and RV [4, 13] that more selectively peer) and using new collection pipelines leveraging BMP [33] that allow expanded visibility. The system must therefore handle a large volume of data to accommodate the anticipated growth of these BGP collection platforms.

Cleanable: The system must be able to remove bogus data. Some VPs generate noisy updates [37], because of bugs [19, 86], failures [21] or even adversarial inputs intended to mislead BGP monitoring systems [28, 80, 83]. While identifying the root causes or deciding whether to retain such data lies outside the system's scope, the system must be able to remove data from VPs deemed faulty, without impacting the service or data integrity from other VPs.



Q3	How many ASes appear always at the end of the AS path and never in the middle?
Q9	What are the paths where AS x is at the origin, i.e., the last AS?
Q11	What is the percentage of single hop AS paths?
Q12	Who are AS's x neighbors?
Q17	When did AS x appear in the routing table?
Q19	What time was the last update observed for prefix p for every VP?
Q24	Did the last AS in any AS paths for prefix p change?
Q27	Can we see prefix p?
Q33	Rank the VPs based on the number of updates they collected.
Q36	Which updates hold community c?

Table 1: A sample of ten BGP-data-related questions from the 39 in Table 7 (colors explained in §4).

Figure 2: How BGP updates are received, stored, and presented. Numbers attached to each raw message match those in Fig. 1.

4 MOTIVATION

We highlight the advantages and limitations of the two methods used today to distribute BGP data: MRT-based archives (§4.1) and APIs/dashboards (§4.2).

4.1 Analyzing archives of MRT files

✓ **Advantages: flexible and scalable.** MRT file archives offer flexibility as MRT preserves all information from original BGP packets (Fig. 2), allowing users to answer a wide range of queries. They are also scalable in terms of storage requirements as (i) MRT preserves native BGP optimizations that group prefixes with identical attribute values in the same NLRI field and (ii) recurring attribute values over time enable significant storage savings when compressed with tools like gzip. As of Nov. 2023, the MRT archive of RIS—spanning data from 1999 to the present—occupies ≈50TB [41], a substantial yet manageable volume with commodity servers—even if more VPs are deployed.

✗ **Drawbacks: Hard to clean, hard to use, and slow.** When MRT files contain data from multiple VPs, as is the case for RIS and RV, the data is intermingled, making it difficult to isolate and remove data from a single faulty VP without reconstructing a large part of the archive.

Another significant drawback of MRT-based archives, even when MRT files are built on a per-VP basis, is the inability to retrieve specific bits of the BGP data, e.g., only the bits that involve a particular prefix. Users must download and process entire MRT files, including irrelevant data, to extract specific BGP insights. Tools like BGP2GO [18], which help users locate files containing specific resources, and libraries such as bgpstream [92] and bgpkit [14], which automate downloading, decompression and parsing, simplify and optimize the workflow. However, they do not eliminate this fundamental inefficiency. A recent survey among researchers highlighted the slowness caused by MRT archives [7]. We now further substantiate this issue through experimental analysis.

Experiment settings. We selected ten questions that cover all categories in Table 7, ensuring broad representation of queries, while limiting the overall experiment duration. We divide the ten questions (Table 1) into three categories.

- **Green:** questions without parameters and for which we used data from 35 VPs and for a 1-month period (July 2024).

- **Light blue:** questions that look for RIB entries at a specific time and thus have one parameter: the number of VPs.
- **Yellow:** questions that analyze BGP updates and depend on two parameters: the number of VPs and the time interval.

Other parameters, such as the prefix, AS number, community value, or a precise date (when no time period is required) are chosen randomly when they are applicable. For each question, we wrote a Python script that retrieves MRT files from RIS and RV, and process them to produce the answer. We used bgpstream and its pybgpstream Python library [92] for automated downloading, decompression, and parsing, and we used all its available features, such as filters on BGP attributes and VPs, to minimize processing overhead and reduce execution times. We parallelized the analysis across multiple cores whenever feasible. We used ten cores as increasing their number beyond this point was not helpful since execution time is mainly constrained by the data downloading process. We used a 16-core and 62GB RAM cloud-based VM with a 100Gb/s Internet connection. We ran our code using varying numbers of VPs (2, 10, 35, and 200) and time intervals (1 hour, 1 day, 10 days, and 100 days) covering July to Oct. 2024. For experiments with many VPs (e.g., 200), we used short time periods (e.g., 1 day) to limit the duration of our experiments. For Q17, we optimized the Python script by implementing a binary search algorithm, enabling location of the first occurrence of an AS without the need to sequentially scan the entire dataset. The selected VPs were the most valuable ones from RIS and RV, as determined by MVP [6]. We share the Python scripts we wrote to answer these questions in §B.

Experiment results. Table 2 presents the time required to answer the ten questions. Q19 was time-consuming to answer (6.1 hours) as the Python script iterates backward through all updates in the MRT files to locate the most recent update for the specified prefix, repeating this process for all 35 VPs. Despite our binary search optimization, Q17 was also slow requiring 29 minutes. For the other questions, the time logically increased with both the number of VPs and the length of the time interval. Q3, Q11, Q12, and Q27 required a few to several tens of minutes to answer, depending on the number of VPs. These questions require downloading the MRT RIB dump files preceding the specified time and then applying the updates to reconstruct the RIB at the specific time. Q3 and Q11 took longer (up

Q17	29m, 35GB											
Q19	6.1h, >130G											
	# of VPs: 2				# of VPs: 10				# of VPs: 35			
Q3*	2m, 435MB				12m, 2GB				31m, 6GB			
Q11*	2m, 496MB				8m, 2GB				20m, 4GB			
Q12	1m, 1GB				3m, 3GB				4m, 7GB			
Q27	1m, 739MB				2m, 2GB				5m, 5GB			
	Time interval				Time interval				Time interval			
	1 hour	1 day	10 days	100 days	1 hour	1 day	10 days		1 hour	1 day		Time interval
Q9	17s, 64MB	5m, 1GB	50m, 14GB	7.9h, 127GB	29s, 311MB	7m, 7GB	1.2h, 52GB		36s, 612MB	10m, 10GB		16m, 13GB
Q24	15s, 65MB	5m, 1GB	49m, 14GB	9.0h, 127GB	44s, 412MB	7m, 5GB	1.2h, 52GB		32s, 511MB	11m, 9GB		20m, 19GB
Q33	21s, 83MB	5m, 2GB	50m, 14GB	7.9h, 127GB	24s, 311MB	9m, 5GB	1.1h, 46GB		40s, 557MB	18m, 10GB		19m, 19GB
Q36	20s, 86MB	5m, 1GB	52m, 15GB	8.2h, 127GB	22s, 311MB	6m, 5GB	1.1h, 49GB		33s, 466MB	13m, 13GB		20m, 21GB

Table 2: Time (in red) and data volume (in blue) required to answer a subset of ten questions from those listed in Table 1. Questions marked with a * could not be parallelized due to excessive memory usage.

to an hour) because we could not parallelize them due to excessive memory usage (e.g., these questions require processing full RIB dumps, and storing them consumes memory). As expected, the execution time for *Q9*, *Q24*, *Q33*, and *Q36* increased linearly with the length of the time period. While these queries are relatively fast for short time intervals (e.g., 1 hour), they took over an hour to process when the time period exceeded 10 days (e.g., *Q24* took 9 hours).

Most of the script execution is spent downloading and processing many MRT files—often to locate a few relevant entries. Data volume can reach several gigabytes, and sometimes, hundreds of gigabytes, slowing execution of the scripts.

4.2 APIs and dashboards

✓ **Advantages: Easy and Fast.** By design, APIs and dashboards are inherently user-friendly and optimized for speed. For example, RIPEstat and the RV API typically delivers responses within seconds [89, 101], and commercial dashboards are reputed to be fast to ensure customer satisfaction.

✓ **Drawbacks: Poor scalability and limited flexibility.** To provide swift responses, these APIs require storing BGP data in multiple formats, including precomputed metadata. While this approach improves query performance, it can significantly increase storage requirements compared to compressed MRT archives. As of Jan. 2023, RIPEstat used approximately 800TB of storage—16× more than the size of their MRT archive [41]. This creates substantial data management challenges, increasing both costs and infrastructure complexity [105]. As a result, RIS (and also RV) has deliberately restricted peering with new VPs to control storage demands [3, 4, 12], a strategy in tension with recent recommendations for a substantial increase in VP deployment to reduce BGP visibility gaps [7]. Commercial actors like Kentik have also recognized the scalability challenges of storing network data (including BGP data) in traditional databases like PostgreSQL and providing it to users [39, 40].

Due to these scalability constraints, existing APIs and dashboards generally provide a limited set of endpoints. While these endpoints sometimes allow flexibility into which metric to return using filters, their overall functionality remains constrained [31, 87, 89, 111]. For example, RIPEstat’s *AS Path Length* endpoint provides insights aggregated at the collector level and only for updates related to a single AS. Thus, many questions about AS paths, such as *Q11* and *Q7*, cannot be directly answered. Our manual evaluation of

RIPEstat’s API endpoints (Table 7) reveals that 23 out of 39 questions cannot be answered through the API. Addressing these queries requires writing custom scripts to process additional data, often sourced from MRT files, thus increasing processing time, as shown in Table 2.

5 SYSTEM OVERVIEW

ChatBGP aims to meet all the requirements listed in §3. It already runs in production, and provides insights into BGP data collected from all public platforms: bgproutes.io [32], RIS [2], RV [112], PCH [93], and CGTF [20], totalling over 5,000 VPs, of which 1,225 are full feeders. At present, our production deployment of *ChatBGP* stores and serves RIB dumps and BGP updates, along with their prefixes, AS paths, and standard community values.

Usage. Fig. 3 (left) shows the workflow of *ChatBGP*. Users can submit plain-English questions about the stored BGP data; *ChatBGP* then generates, within seconds, a Python code snippet that returns the answer, when executed. Alternatively, users may write the Python code themselves. After installing the Python library `pybgproutesapi`, which provides a Python interface to *ChatBGP*’s API, users can directly execute either the generated or manually written code. As shown in §9, the Python code often finishes within a few seconds, significantly faster than processing MRT file archives using tools like `bgpstream` [92]. Besides, *ChatBGP* generates code that returns a correct answer for 36 out of the 39 questions in Table 7—and many more. For instance, *ChatBGP* answers the question: “What is the minimum number of AS hops between AS201341 and AS1853 based on the data collected from VP 193.203.0.63 at 11:29:11 AM on October Fifth, 2025.” with the ≈20-line code snippet shown in Fig. 3 (right) that runs in less than six seconds and returns the correct answer.

With its carefully designed API, which supports simple yet flexible database queries, and its `pybgproutesapi` library, which provides an intuitive Python interface, *ChatBGP* produces concise code snippets e.g., in the median case, solutions to the ten questions in Table 1 require only 20 lines of code (30 in the worst case), that users can easily inspect and verify, even if they are unfamiliar with BGP or MRT intricacies. If needed, users can request refinements, and *ChatBGP* will provide an updated code snippet. For instance, using the example above, the user can write “Do not consider AS

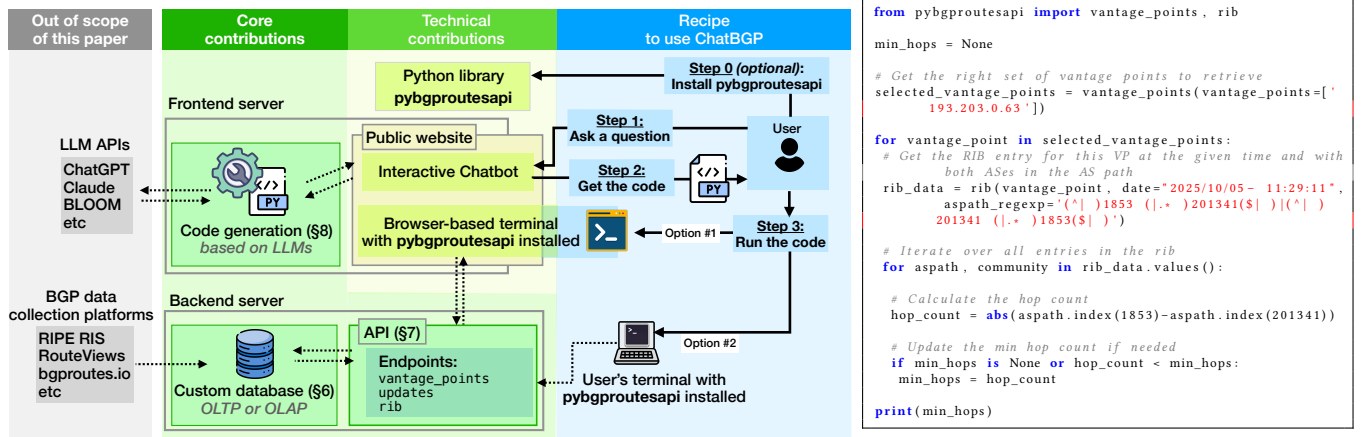


Figure 3: Overview of ChatBGP (top) along with an example of a Python code that it generates and which executes in a few seconds (right).

paths that include a Tier1 AS." and ChatBGP will refine the code snippet accordingly. We illustrate further this process in §E.

Contributions. ChatBGP builds on three key contributions shown in Fig. 3 and described in the next sections.

§6. A measurement study quantifying redundancy in BGP data, together with a database storage scheme that exploits this redundancy to store BGP data in a losslessly compressed format, while enabling fast, fine-grained retrieval of specific data elements (e.g., via regular expressions over BGP attributes) and supporting per-VP data removal.

§7. A carefully designed API and Python interface (pybgproutesapi) that allows users to retrieve exactly the data they need through simple yet highly flexible queries.

§8. A few-shot prompting scheme (a.k.a. *prompt engineering*) that guides ChatGPT to generate optimized Python code snippets leveraging the pybgproutesapi library to quickly answer a wide range of BGP-related queries.

Artifacts. In §B, we provide a set of artifacts that allow readers to inspect typical code snippets generated by ChatBGP and reproduce some results presented in this paper.

6 BGP DATA STORING SCHEME

Scalability is a key requirement, but storing raw BGP data without optimization leads to massive data volumes (§9) and slow retrieval. Fortunately, BGP data is highly recurrent: identical components frequently repeat over time. This observation allows us to design a lossless compression scheme tailored to BGP data and implementable in both OLTP databases (e.g., PostgreSQL) and OLAP databases (e.g., ClickHouse).

Key observation: recurring patterns in both BGP updates and RIB dumps.

Recurring BGP update component values. Consider the scenario in Fig. 1 and the raw BGP messages exported by AS1 and shown in Fig. 2. Despite the two flap events occurring on different links in the topology, the resulting BGP data from AS1 contains recurring information. Specifically, while the withdrawal messages differ in their *withdrawn routes* (p1, p2 for flap #1 and p1, p2, p3 for

flap #2), the announcements generated by the restoration of the links produce identical BGP updates: an update with the NLRI field containing p1 and p2 and the AS path AS1-AS2-AS3-AS5 appears twice. Although the simple scenario in Fig. 1 does not account for typical path exploration during BGP convergence, our analysis of BGP updates collected by RIS and RV (Table 3), confirms that recurring patterns are prevalent in real-world data. For this experiment, we used one month of BGP data (July 2024) collected from the top 35 VPs identified by MVP [6]. For the NLRI field as well as the AS path and the community value attributes, we categorized cases based on the number of prefixes in the NLRI field, AS path length, and number of community values, respectively. These categories are highlighted in blue, along with the percentage of BGP updates that fall into each category. We found that 10% of NLRI fields containing between two and five prefixes, a case that accounts for 16% of BGP updates (see two blue rows in Table 3), repeat ten times or more (marked with the ★ symbol). Similarly, 25% of AS paths with a length of three to five, which account for 49% of BGP updates, repeat at least eight times (marked with the † symbol). Finally, 5% of community values containing three to five entries, which account for 9% of BGP updates, repeat ≈15k times (marked with the o symbol).

Recurring entries in RIB dumps. BGP routes to most announced IP prefixes remain relatively stable over time, changing primarily during rare events such as reconfigurations or failures [36]. As a result, many entries in consecutive RIB dumps are identical, especially when the dumps are separated by only a few hours or days. Fig. 4 illustrates this stability by comparing, for each VP, the RIB from July 1, 2024, with subsequent RIBs at intervals between them gradually increasing from three to 30 days. We see that the RIBs slightly change over time. For instance, after three days, 97% of RIB entries remain identical in the median case. Naturally, the percentage of identical entries decreases as the time interval between RIBs grows. Yet, even after 30 days, 86% of the RIB entries remain unchanged. The percentage dropped below 60% for only one VP during a short timeframe, possibly due to backup path activation during a failure.

Characteristics of the BGP routes collected in July 2024	# of prefixes in NLRI fields				AS path length			# of community values			
	[0,1]	[2,5]	[6,20]	>20	[1,3]	[3,5]	>5	[0]	[1,2]	[3,5]	>5
	79%	16%	4%	1%	5%	49%	45%	78%	6%	9%	7%
50th per	4	1	1	1	4	3	2	2	8	4	
75th per	17	3	2	1	13	8 [†]	5	24	188	16	
90th per	72	10 [★]	5	3	42	33	15	2k	4k	89	
95th per	270	25	11	5	74	58	35	11k	15k ^o	492	
99th per	2k	120	46	21	365	547	212	332k	134k	6k	
average	83	10	4	2	62	59	28	19k	9k	490	
Estimated compression factor	4x	5x	4x	2x	4x	6x	8x	0x	3x	11x	41x

Table 3: Distribution of the number of identical occurrences for each BGP update component as observed in raw BGP updates on a per-VP basis—i.e., identical component values from distinct VPs are not considered recurrent. Green symbols link table entries to the text.

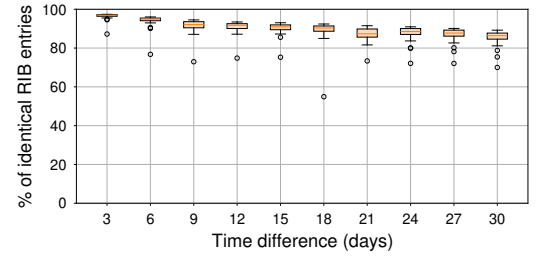


Figure 4: Distribution of the percentage of identical RIB entries across 20 full-feeders VPs (eleven v4 and nine v6).

Lossless BGP data compression scheme. We use two different strategies to compress BGP updates and RIB dumps, which we briefly describe below and detail further in §D.

BGP updates compression. We implement levels of indirection by storing pointers for each component in a BGP update rather than duplicating the values (a.k.a database normalization). These pointers reference a mapping table that associates each pointer with the actual component value. Since component values can consume significant space (e.g., many prefixes in the NLRI field, multiple ASes in the AS path, or many community values), this indirection achieves substantial compression. Table 3 presents the estimated compression factors, defined as the ratio between the size of the original BGP messages and the size achieved using our compression scheme (assuming four-byte pointers). Compression ranges from 2× to 5× for NLRI fields, 4× to 8× for AS paths, and reaches its highest efficiency with community values (3× to 41×), due to their high redundancy.

RIB dumps compression. Instead of storing full RIB dumps for every VP every eight or two hours (as done by RIS and RV, respectively), we store a complete RIB dump at the start of each month and then record partial RIB dumps every eight hours. These partial dumps contain only the entries that differ from the full RIB stored at the beginning of the month. Since over 86% of RIB entries remain unchanged even after a month (Fig. 4), these partial RIB dumps require significantly less storage compared to saving full RIB dumps. Reconstructing a complete RIB at any point in time thus involves three steps: (i) retrieving the most recent complete RIB prior to the given time, (ii) updating it with the changes recorded in the most recent partial dump prior to that time, and (iii), updating it with the updates observed between that partial dump and the given time.

7 API DEFINITION

Defining an API that is both simple to use and expressive enough to capture the wide variety of possible queries is challenging, given the large breadth of the query space. To tackle this problem, we followed a two-step methodology.

Step #1: Identifying the range of questions users are likely to ask. We analyzed operator mailing lists [88] and publications from top networking conferences to gain a comprehensive understanding

of what network operators and researchers seek from BGP data. Our methodology involved the following steps:

Gathering. We automated processing of large volumes of sources while ensuring an unbiased analysis (i.e., without prioritizing specific types of questions). We developed a web scraper to extract the body text of emails from the NANOG mailing list since Jan. 2020 and the abstracts of research papers published at SIGCOMM, NSDI, and IMC since 2020.

Classification. We used ChatGPT to rate the likelihood that each email or abstract involved BGP-related analysis. For example, we prompted ChatGPT with the following request to evaluate paper abstracts: “The following text is the abstract of a paper published in a conference related to networking. Give me a score (from 1 to 10) that this paper uses BGP data.” We classified entries scoring above five as relevant to BGP (Positive cases), while we classified those scoring five or below as irrelevant (Negative cases) and excluded them.

Validation. To ensure the accuracy of our classification, we manually evaluated 5% of all emails and papers since 2020 to establish a ground truth dataset—this involved manually scoring 1200 emails and 41 papers. We considered an automated inference correct if it matched the manual classification. Our automated classification achieved an accuracy of 93% for emails and 91% for papers, where accuracy is $\frac{TP+TN}{TP+TN+FP+FN}$.

Question generation and sanitization. For all NANOG threads and papers rated above five, we asked ChatGPT to summarize the BGP analysis as a high-level question. We discarded questions that required additional knowledge not available in the BGP data as collected by RIS and RV, such as questions about RPKI or those that require running algorithms that are not standardized (e.g., for failure detection and localisation).

Improving clarity. We manually grouped similar questions and refined their phrasing to remove ambiguities and ensure consistency. For instance, we rephrased the question “How many stub ASes are there?” to “How many ASes always appear at the end of the AS path and never in the middle?”, avoiding ambiguity in the definition of a stub AS. However, we retained terms like “Tier 1” and “blackholing communities” as they are well-defined in many sources (e.g., Wikipedia), allowing ChatGPT to interpret and handle them. Our

analysis resulted in a set of 39 questions (Table 7). For each question, we provide at least one reference to an email, research paper, blog post, or tweet originating or discussing the question.

Parameters identification. Each question involves specific resources (e.g., an IP prefix, an AS number, etc.) and may also target a particular date, time interval, or VP. We treated these as parameters (see “Parameters” column in Table 7).

Step #2: Defining primitive functions to query the database.

From the 39 questions, we identified and implemented four primitive functions that provide a high-level, intuitive interface to the database:

- `vantage_points`: Retrieves a list of VPs matching specified criteria provided as parameters (e.g., IP protocol).
- `updates`: Returns a list of BGP updates (at the prefix granularity like with `bgpreader`, see Fig. 2) or a count of updates for a given VP. This function supports multiple parameters e.g., for filtering results by time interval, prefix, AS path, or community values. Flexible filtering is enabled through regular expressions for many parameters.
- `rib`: Fetches the RIB at a given time for a given VP. It supports filtering by prefix, AS path, or community values, with regular expressions available for flexible filtering.
- `topology`: Fetches the AS topology for a given VP.

We then developed an API with four endpoints, each corresponding to one of the identified functions (Table 8). These endpoints generate SQL queries based on the provided parameters, retrieve the relevant data from the database, and return it to the user. We also created the `pybgproutesapi` Python library, which offers Python functions that seamlessly interact with these endpoints. When combined programmatically (e.g., within a Python script), these endpoints enable answers to all 39 questions with remarkably simple programs.

8 CODE GENERATION

Current LLMs are often general-purpose, offering broad applicability but lacking specialization. Additionally, no comprehensive public repository of code snippets exists for answering BGP-related questions, which could be leveraged to specialize an LLM through prompt engineering. Thus, as we experimentally demonstrate in §C, ChatGPT struggles to generate correct `bgpstream`-based code, even with tailored prompt engineering. For example, with the following simple question: “Write a Python code that returns all BGP updates collected by the RouteViews vantage point with IP 45.68.16.3 and for which the AS path contains both AS13335 and AS8919 on July 1, 2024.” ChatGPT-5.1 produces an incorrect code snippet that improperly uses the `bgpstream` library. Moreover, even if the generated code is syntactically correct, it is not optimized for performance, leading to execution times comparable to those in Table 2. We therefore trained ChatGPT to generate domain-specific Python code that uses the `pybgproutesapi` library to retrieve data efficiently from the database and to accurately and swiftly answer BGP-related questions (e.g., those listed in Table 7). We use the following steps.

Generating diverse phrasings. A question about BGP data can be expressed in many ways, and common terms can appear in different forms. For example, “vantage point” can be written as “VP” or “BGP router”, while “IP prefix x.x.x.x/y” could be phrased as “subnet

x.x.x.x with mask length y”. We prompted ChatGPT to rephrase each of the 39 questions in Table 7 into 20 different variations, using alternative wordings. We manually reviewed the generated questions, discarded one that was nonsensical, and used the 779 remaining ones (shared in §B) in the next steps.

Writing correct code snippets. For each of the 779 generated questions, we created (using a mix of manual writing and automation) a Python script that produced the correct answer and uses the `pybgproutesapi` library. Whenever possible, we leveraged parameters of the four primitive functions in the `pybgproutesapi` library to optimize the code. For example, we utilized regular expressions to minimize data retrieval volumes and accelerate script execution. Parameters such as AS numbers, community values, and dates were selected randomly for each question. These optimized code snippets (shared in §B) serve as the ground truth for the next step.

Few-shot prompting. We trained ChatGPT-4o (the most recent version available when we built *ChatBGP*) using the few-shot prompting technique—commonly referred to as “prompt engineering”—a widely adopted method for tailoring LLMs to respond in a domain-specific manner. In the training prompt, we first provided detailed specifications for the four primitive functions in the `pybgproutesapi` library, including their parameters and outputs. Then, we concatenated the 39 questions, randomly selecting from the various phrasings we generated, alongside their corresponding correct and optimized Python code. The resulting training prompt typically contained $\approx 15k$ tokens, well within ChatGPT’s token limits.

9 EVALUATION

We evaluate *ChatBGP* along three dimensions: the scalability and soundness of its database design, and the speed and accuracy of its generated code.

ChatBGP’s BGP data storing scheme exploits data redundancy effectively. We show that the storage scheme reduces data volume and becomes more efficient as the retention period grows.

Methodology. We evaluate *ChatBGP*’s data compression scheme (§6) on PostgreSQL and compare it against three baselines: (i) MRT files compressed with `bzip2`, the current best-case reference for BGP data-volume compression; (ii) a PostgreSQL database that stores MRT data as-is, where entries may correspond to multiple prefixes sharing the same attribute values; (iii) a naive PostgreSQL-based approach that stores updates on a per-prefix basis (similar to the data format produced by `bgpreader` in Fig. 2) and RIB dumps. Our evaluation focuses on data collected by the RIS collector `rrc04` (a rather small collector) during July 2024, to limit storage requirements for the experiment. Fig. 5 shows the results.

Results. The two PostgreSQL-based baselines imply a linear growth in database size over time. Baseline ii (resp. baseline iii) used more than 87GB (resp. 100GB) after 30 days. In contrast, *ChatBGP*’s database size grows at a decreasing rate, as *ChatBGP* combines repeated occurrences of the same data into single entries. For 30 days, *ChatBGP*’s database requires nine times less storage than baseline iii, and is only one-fifth larger than compressed MRT files.

The design of ChatBGP’s BGP data storing scheme is sound. We show that it provides the best tradeoffs for meeting the requirements in §3.

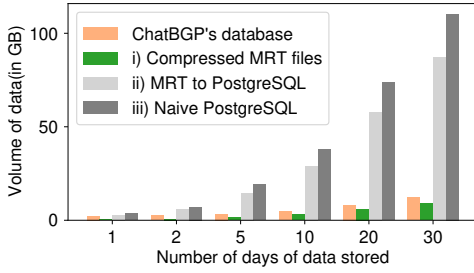


Figure 5: Database size as a function of the number of ingested days, comparing ChatBGP's database scheme with three baselines.

			ChatBGP database scheme (§6)		Naive
			Postgres	Clickhouse	Clickhouse
Query response-time results (Best query execution times are shown in green; worst are shown in red)					
Endpoint	Filters	Related questions	Average across 100 runs		
rib	no filter	Q2, Q3, Q4, Q6, Q7, Q10 Q11, Q15, Q16, Q25, Q30, Q34	31.46s	25.81s	40.91s
	one prefix (exact match)	Q5, Q27, Q28, Q39	0.06s	0.88s	0.16s
	many prefixes (exact match)	Q29	0.55s	1.3s	0.19s
	many prefixes (subnet match)		0.94s	1.56s	0.18s
updates	no filter	Q21, Q22, Q24, Q33	7.42s	6.12s	10.32s
	one prefix (exact match)	Q19, Q26	0.01s	0.31s	0.02s
	many prefixes (exact match)		0.23s	0.44s	0.01s
	many prefixes (subnet match)		0.42s	0.73s	0.01s
	many AS paths (regex match)	Q8, Q9, Q17, Q18, Q23, Q35 Q36, Q37, Q38, Q31, Q32	0.48s	0.03s	0.01s
topology		Q1, Q12, Q13, Q14, Q20	40.92s	32.34s	49.9s
Data ingestion and volume results					
Time to ingest one day of data			1154s	1286s	2423s
Data volume			33GB	8.40GB	37.00GB

Table 4: Database benchmark results.

Methodology. We benchmark ChatBGP's PostgreSQL-based storing scheme (§6, §D) against two ClickHouse (i.e., columnar-oriented) baselines: one using ChatBGP's storing scheme and one "flat" baseline that does not, i.e., updates are not decomposed into shared tables and full RIBs are stored every eight hours to match RIS [2]. We omit flat PostgreSQL, which corresponds to the naive PostgreSQL case in Fig. 5 and performs poorly in storage footprint. For each configuration, we ingest one month of BGP data from 50 VPs at RIS collector rrc00, including 20 full feeders. We measure query execution time, ingestion time, and disk space usage. For query execution, we test the three main ChatBGP API endpoints: rib, updates, and topology; we omit vantage_points, which is fast for all configurations. For each endpoint, we evaluate one-day queries that either read all data or use common filters: exact prefix matching over one or multiple prefixes, more-/less-specific prefix matching, and regular-expression matching. These filters cover the search patterns used by most questions in Table 7; for each endpoint and filter, Table 4 also reports the corresponding questions. We report the results in Table 4.

Result #1: ChatBGP improves response time in key scenarios. No evaluated database configuration dominates across all search scenarios. However, we observe a clear trend: the naive ClickHouse baseline performs worst for all three endpoints when queries use no filter, and worse than ChatBGP with PostgreSQL when queries use a single exact-prefix filter. For instance, for a rib query with an exact-prefix match, ChatBGP with PostgreSQL returns the answer in 0.06s on average, compared with 0.16s for naive ClickHouse (2.7× slower). This difference matters, as most questions in Table 7 query endpoints either with no filter or with a single exact-prefix filter, and some of these queries are time-critical. For instance, a looking-glass query (see §10) requires a rib query with a single-prefix match for each VP. When the number of VPs is large, a 2.7× response-time difference can translate into excessive waiting time for users. Naive ClickHouse performs better for queries with more selective filters, such as prefix subnet matching (i.e., more-/less-specific prefix matching) or regular-expression matching.

Result #2: ChatBGP's data storing scheme reduces data volume and

write pressure. Interestingly, using ChatBGP's data storing scheme with ClickHouse reduces write pressure by roughly a factor of two compared to naive ClickHouse and decreases the data volume from 37GB to 8.40GB. ChatBGP with PostgreSQL provides the best write performance and uses slightly less disk space than naive ClickHouse. The fact that naive ClickHouse performs worse on both storage footprint and write pressure highlights the usefulness of ChatBGP's data storing scheme. **Key takeaways.** The right database choice depends on the platform requirements. For ChatBGP, the requirements in §3 justify the scheme in §6. However, if users issue more advanced-filter queries, naive ClickHouse may be preferable. With more hardware, or for smaller deployments (e.g., an internal network deployment rather than a global service), a hybrid architecture is likely best: PostgreSQL with ChatBGP's scheme and naive ClickHouse can run jointly, with a backend query planner routing each query to the expected fastest database. Conversely, if the number of VPs grows to several tens of thousands—which we believe plausible with BMP, as it can export an order of magnitude more data than a BGP session—ClickHouse with ChatBGP's scheme may become preferable because it has the smallest storage footprint.

ChatBGP generates correct code in most cases. We compare ChatBGP against a baseline that always returns the correct code. **Methodology.** The correct baseline corresponds to manually written code snippets for each of the 39 questions in Table 7. We compare the output of these snippets against the output of code generated by ChatBGP using its few-shot prompting scheme (§8) limited to 25,000 tokens. For each question, we randomly select parameter values (e.g., time interval or prefix), while ensuring that at least some data is returned, thereby avoiding trivial empty-result cases. We also excluded the evaluated from the few-shot prompting used by ChatBGP, i.e., we used the 38 other questions to train ChatGPT. This approach simulates a realistic scenario where a user asks ChatBGP a question on which it has not been trained. When querying either ChatBGP or the correct baseline, we also specify the expected output format (e.g., an integer, a Python dictionary, or a list of AS paths), so that outputs can be compared directly and differences in formatting are not mistaken for incorrect answers. We evaluate the generated

Outcome	Jaccard score	Runs (%)
Perfect match	= 1.0	89.74
High similarity	≥ 0.8	2.56
Medium similarity	≥ 0.5	2.56
Incorrect	= 0	5.13
Not answered	–	0.00

Table 5: Percentage of *ChatBGP*-generated code outputs matching the correct baseline, aggregated over ten runs for the 39 questions.

code ten times for each question. For each pair of answers, we compute the Jaccard index: a score of 1 means the two outputs contain the same elements, 0.5 means half of the elements overlap, and 0 means that they share no elements.

Results. Table 5 shows the distribution of the jaccard scores. For $\approx 90\%$ of the runs, the output returned *ChatBGP*’s generate code is identical to the *correct* baseline. For 5%, it is partially correct (e.g., could be lack of precision, or missing elements), and for the remaining 5% is fully incorrect. Most non-perfect cases come from subtle semantic mismatches rather than syntactic failures. For instance, for *Q17*, *ChatBGP* uses a RIB query instead of an update query, leading to an inaccurate timestamp offset of roughly 2h15. For *Q22*, *ChatBGP* queries hourly intervals starting at exact hours (e.g., 01:00:00), whereas the baseline uses intervals starting at 00:59:59, 01:59:59, etc. For *Q36*, *ChatBGP* uses a community regexp that only matches when the queried community value appears at the beginning of the community field, whereas the baseline also matches the value after whitespace-separated community entries.

Further observations. The results suggest that question phrasing and precision matter for *ChatBGP* to generate accurate code. We confirm this in §E, where clearly phrased questions improve code-generation accuracy by 7–13% compared with poorly phrased ones. We also show in §E that the token budget matters, although performance appears to converge after 15,000 tokens. Finally, *ChatBGP* generates concise and intuitive code: the median snippet has only 20 lines of code (30 in the worst case). This makes it easier for users to spot issues and request refinements, as illustrated in §E.

***ChatBGP* generates optimized code that greatly reduces data-processing costs.** We show that *ChatBGP* uses pybgproutesapi filters whenever possible, minimizing the volume of data downloaded. We use data volume as the optimization metric, as it affects execution time and required processing resources.

Methodology. We run the code generated by *ChatBGP* for the 39 questions in Table 7 and with 100 VPs, and compare the downloaded data volume against three baselines. The *best-case* baseline consists of manually written code snippets that use pybgproutesapi with all its applicable filters to minimize data volume. The *ChatBGP w/o opti.* baseline consists of manually written pybgproutesapi snippets that use no optional filters, except the mandatory filters (e.g., VP and date parameters). The *bgpstream* baseline is the volume of MRT files downloaded by bgpstream using all its available filters, as in the experiment reported in Table 2.

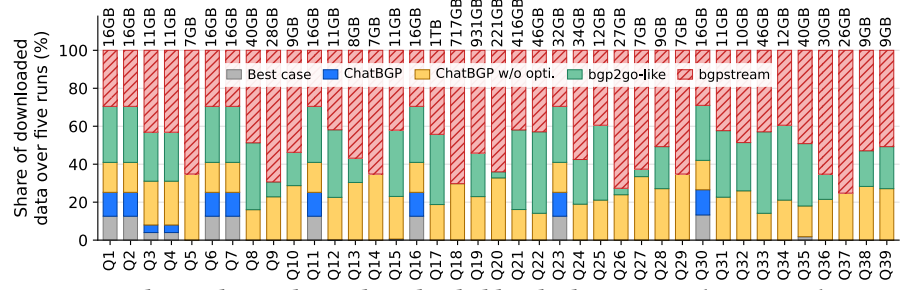


Figure 6: Relative data volume downloaded by the best-case, *ChatBGP*, *ChatBGP* w/o opti, bgp2go-like, and bgpstream, for each question in Table 7 and 100 VPs.

Result #1: pybgproutesapi’s endpoint design is sound and useful.

Fig. 6 shows the relative data volume downloaded by *ChatBGP* and each baseline. For each question, the total data downloaded by *ChatBGP* and all baselines is normalized to 100%, with each segment showing the fraction downloaded by *ChatBGP* or the corresponding baseline. We see a significant gap between *best-case* and the *bgp2go-like* baseline. This highlights that pybgproutesapi’s endpoint definitions and filtering parameters are well chosen: when used carefully, they allow substantially reducing the data downloaded, often down to only a few bytes. Without these filters, *ChatBGP* would perform roughly like the *ChatBGP w/o opti.* baseline, which performs similarly to the *bgp2go-like* baseline, i.e., the state-of-the-art MRT counterpart.

Result #2: *ChatBGP* generates optimized code. Fig. 6 shows that *ChatBGP* downloads the same amount of data as the *best-case* baseline for almost all questions, often orders of magnitude less than the *bgp2go-like* and *bgpstream* baselines (MBs instead of GBs). This demonstrates that *ChatBGP* exploits pybgproutesapi’s filters and generates optimized code snippets. The total downloaded data volume (*ChatBGP* plus all baselines) is shown above each bar to indicate the order of magnitude. Cases where *ChatBGP* and the *best-case* baseline still download GBs of data (e.g., *Q1*, *Q23*, *Q30*) correspond to questions that inevitably require retrieving full RIBs.

***ChatBGP* gives quick answers.** We evaluate the execution time of Python codes that use *ChatBGP*’s API to answer queries.

***ChatBGP* setup.** We deployed *ChatBGP* on a server with 24 cores, 62GB of RAM, an SSD with a 5GB/s write speed, and a 10Gb/s Internet connection. We used PostgreSQL as the backend database and the same sliding-window data retention mechanism as in our production deployment (§10). To simulate real-world workloads during the experiment, we continuously inserted new data from 800 VPs (i.e., roughly the number of VPs from platforms such as RV) while removing older data.

***Client* setup.** For each question in Table 7, we ran the manually written code snippets used to train *ChatBGP* (§8) and measured their execution time. We executed these snippets from the same VM used to evaluate bgpstream [92] in §4 (Table 2), enabling a fair comparison.

Evaluation methodology. As in §4 for Table 2, we categorized questions into three groups—**Green**, **Light blue**, and **Yellow**—based on whether their execution time was evaluated with varying numbers of VPs and time intervals. We used the same VPs as in Table 2, i.e., the ones provided by MVP [6], which include VPs from RIS and

Q17	2m (↓13x) 762B											
Q19	5m (↓65x) 3KB											
	# of VPs: 2				# of VPs: 10				# of VPs: 35		# of VPs: 200	
Q3	15s (↓10x) 17MB				1m (↓12x) 241MB				3m (↓9x) 779MB		16m (↓4x) 8GB	
Q11	15s (↓9x) 20MB				1m (↓8x) 241MB				3m (↓6x) 788MB		15m (↓3x) 8GB	
Q12	14s (↓8x) 4B				25s (↓7x) 536B				1m (↓3x) 3KB		4m (↓2x) 60KB	
Q27	11s (↓9x) 2B				4s (↓42x) 2B				4s (↓78x) 2B		3s (↓153x) 1B	
	Time interval				Time interval				Time interval		Time interval	
	1 hour	1 day	10 days	100 days	1 hour	1 day	10 days		1 hour	1 day	1 day	
Q9	9s (≈) 4B	5s (↓63x) 4B	8s (↓376x) 4KB	11s (↓) 46KB	12s (↓2x) 253B	12s (↓38x) 18B	17s (↓252x) 184KB		44s (≈) 64B	56s (↓11x) 11KB	1m (↓9x) 9KB	
Q24	6s (↓2x) 4B	5s (↓62x) 4B	21s (↓143x) 423B	40s (↓808x) 640B	11s (↓4x) 18B	14s (↓33x) 18B	51s (↓85x) 138B		51s (≈) 64B	1m (↓7x) 547B	3m (↓6x) 4KB	
Q33	5s (↓4x) 13B	5s (↓62x) 17B	12s (↓253x) 19B	22s (↓) 20B	10s (↓2x) 57B	12s (↓46x) 69B	32s (↓121x) 76B		40s (≈) 211B	55s (↓20x) 248B	2m (↓8x) 2KB	
Q36	5s (↓4x) 4B	5s (↓63x) 541B	9s (↓352x) 4B	5s (↓) 4B	10s (↓2x) 18B	10s (↓39x) 18B	10s (↓397x) 18B		41s (≈) 795B	40s (↓20x) 64B	1m (↓14x) 15KB	

Table 6: Time (orange) and data volume (gray) required to query *ChatBGP*’s optimized database and answer each of the ten benchmark questions, compared to the results of analyzing MRT files using *bgpstream* (Table 2).

RV. Other parameters, such as the prefix, AS number, community value, or a date (when no time period is required) are chosen randomly when applicable. Execution times and data volumes may vary across runs due to the random selection of certain parameters, e.g., the same question targeting two different ASes—a Tier 1 versus a stub AS—might return different numbers of updates, leading to variations in execution time and data size. Despite these differences for individual questions, the overall results were consistent across questions within a category. Thus, we focus on a single run of these questions.

Result #1: ChatBGP gives answers in seconds or a few minutes. Table 6 shows the execution times of the Python code snippets querying *ChatBGP*’s database and compares them with the time required to answer the same ten questions by analyzing MRT-based archives using *bgpstream* (Table 2). Table 9 (Appendix) reports execution times for all 39 questions. Across a broad range of questions and parameters, the *ChatBGP*-generated Python code provides answers within a few seconds. For example, *Q3* completes in one minute using ten VPs, and *Q36* takes ten seconds when considering a ten-day time interval and ten VPs. Execution time increases with the number of VPs and with the length of the time interval when its a parameter (yellow questions). However, execution remains rather fast, typically within a few minutes, e.g., with 200 VPs, *Q12* and *Q33* are answered in two to four minutes respectively.

Result #2: ChatBGP provides answers faster compared to MRT-file-based methods. Table 6 reports the speed-up factors (next to the ↓ symbols) for the ten questions used to benchmark *bgpstream*, calculated as the ratio of execution times between *bgpstream* and *ChatBGP*. We can see that the speed-up factor is often greater than 10×, frequently exceeds 100× (e.g., for *Q27* with 200 VPs), and sometimes surpasses 1000× (e.g., for *Q9* with two VPs over a 100-day interval).

Result #3: ChatBGP’s speed comes from downloading and processing little data. Table 6 reports the data volumes downloaded by the generated Python snippets, and Table 9 (Appendix) also reports data-reduction factors for these ten questions (highlighted in light red), computed like speed-up factors but using data volume. For nearly all questions, the data-reduction factor exceeds 10×, meaning that *ChatBGP* downloads at most 1/10th of the data required by *bgpstream*. For most questions requiring BGP updates (i.e., those marked in yellow), the data-reduction factor exceeds 1000× (indicated by the ↓ symbol). As expected, speed-up and data-reduction factors are strongly correlated: the less irrelevant data *ChatBGP* downloads,

the larger the speed-up. However, as the number of VPs increases, data-reduction and speed-up factors decrease, because MRT files contain a larger fraction of relevant data. This provides a practical guideline: MRT processing is reasonable when queries require nearly all available data, whereas *ChatBGP* is better suited for selective queries that retrieve specific subsets of BGP data.

10 CHATBGP IN THE REAL WORLD

ChatBGP operates efficiently. We run *ChatBGP* on a commodity server equipped with an AMD EPYC 9454P 48-core processor, 192 GB of RAM, and a 20 TB SSD hosting PostgreSQL. *ChatBGP* continuously ingests new data from 5,000 vantage points (the ones from RIS [2], RV [112], bgproutes.io [32], PCH [93], and CGTF [20]), remove outdated data, and handle user queries.

Backend server load. Currently, *ChatBGP* receives approximately 15,000 API queries per day from dozens of users (excluding the requests performed by our CI/CD pipeline). While we cannot always determine who originates these requests, we know that the users include both network operators and researchers, most of whom do not collaborate with us. This relatively high number of requests is partly explained by our rate limits, which require users to split some large queries into multiple smaller ones. Over the last five days, only 0.008% of these requests resulted in an error, while all others were successfully answered. Note that most of these errors were not due to a lack of server resources, but rather to malformed queries or rate limiting. A small number of heavy hitters (fewer than five) generate the majority of requests, whereas most users issue only sporadic queries. All API endpoints are queried regularly, although the updates endpoint dominates due to a single heavy user using it with the AS-path regular-expression filters. Despite this activity, the system load remains low: The average 15-minute CPU load is only 11.64%. Disk activity also remains well within the system’s I/O capacity, with around 5,400 reads/s (≈ 99 MB/s) and 650 writes/s (≈ 16 MB/s), and extremely low I/O latencies (0.11 ms read, 0.07 ms write). This confirms that the current workload is far from saturating the hardware and that the server could accommodate additional VPs and query volume.

Disk space usage. Our server stores 45 days of data collected from 5,000 VPs, all queryable by users. This dataset consumes only ≈9TB of disk space. With our 20TB SSD, we therefore expect to retain roughly three months of data. Larger-scale deployments or longer retention periods (e.g., multiple years) are naturally possible thanks to *ChatBGP*’s data storage scheme, which, aside from high compression rate, allows data, and thus query load, to be distributed

across multiple servers. In particular, data can be partitioned across servers on a per-VP basis: each VP has its own dedicated set of database tables (§6), and the primitive functions `updates` and `rib` operate per VP (§8). In §11, we further discuss how existing *lossy* compression strategies can be integrated into *ChatBGP* to extend data retention without increasing disk space usage.

ChatBGP is useful. We give three examples.

Automating Graph Generation. *ChatBGP* can generate code for plotting figures based on BGP data using tools like the `matplotlib` Python library. For instance, we generated Fig. 4 using *ChatBGP* (the exact query is detailed in §B). With only minor adjustments to improve the visual design of the plot, we were able to generate Fig. 4 in ≈ 15 minutes (code available in §B). This capability to rapidly generate figures for BGP data analysis could enhance the flexibility of existing static dashboards (e.g., RIPEstat) and facilitate reproducibility [11].

Giant looking glass. A key BGP observability tool is the looking glass [24, 91, 96], which shows the RIB entry for a prefix across many VPs, or its absence when reachability is missing. Useful features for a looking glass include, (i) retrieving data from as many VPs as possible (ii) being fast (iii) supporting advanced filtering options e.g., AS-path or community regular expressions, and (iv) inspecting RIB entries in real time and at any point in time. With *ChatBGP*, users can implement a looking glass with all these features in just a few lines of code. In our production deployment, we provide such a dashboard that displays entries for all 5,000+ VPs in ≈ 10 s.

Setting the BGP maximum prefix. The *BGP maximum prefix* option restarts or shuts down a session when a peer announces more prefixes than a configured threshold, helping detect misconfigurations and limit route leaks. One network operator uses *ChatBGP* to set this limit on its route server: a daily 100-line Python script counts, for each IXP member, RIB entries whose AS path contains the member’s ASN, considering only VPs in Tier-1 ASes. The operator then uses these counts to estimate each member’s expected route announcements and configure the corresponding limits.

11 RELATED WORK AND TAKEAWAYS

Tools for analyzing large BGP datasets. `bgpstream`[92] and `bgpkit`[14] are libraries for downloading and parsing MRT archives. BGP2GO [18] is a service that returns a list of MRT update file URLs containing a specified BGP update component value. This allows users to minimize the total data they need to download when searching for a single prefix, ASN, or community. However, BGP2GO only indexes MRT update files from RV, limiting its coverage. Recently, RIPE has begun publishing BGP data in the Parquet format, enabling SQL-based analyses directly on the data [23]. While this speeds up local processing and integrates well with data-science tools, users must still download large Parquet files, even when most of which are irrelevant to their query.

Takeaway. While *ChatBGP* excels at retrieving specific portions of the data, downloading and parsing MRT archives remains useful for large-scale queries requiring *all* updates or full RIB dumps from many VPs (such as *Q3* and *Q11*). *ChatBGP* is still faster in these cases (see Table 6), but downloading MRT files avoids putting sustained pressure on the I/O resources of our servers. In our production

deployment, we therefore use rate limiting to guide users toward our API for queries requiring narrow, targeted slices of the data, while directing bulk, high-volume data transfers to MRT archives instead.

BGP compression techniques Zhu et al. proposed a *lossless* compression method that merges multiple consecutive overlapping RIB dumps into one table [117], although it does not support retrieving RIBs at arbitrary points in time. Recent *lossy* BGP data compression techniques aim to reduce storage requirements by identifying and eliminating redundancy in BGP data [6, 7].

Takeaway. These *lossy* approaches can be seamlessly integrated into *ChatBGP*’s storage scheme to enhance scalability and cost efficiency. For example, MVP detects groups of redundant VPs [6]; when combined with *ChatBGP*, only the data from a single representative VP in each group could be retained long term, while the rest are discarded after a short retention period. Similarly, some VPs are inherently noisy and disproportionately increase storage usage [37]; storing data from these VPs only temporarily would still support meaningful analyses while enabling long-term, low-cost data retention.

Translating high-level queries into code and SQL queries. LLMs have demonstrated great potential in converting high-level natural language queries into actionable database commands (a.k.a Text-to-SQL [1, 115]) using prompt engineering (e.g., [27, 35]). Akin to *ChatBGP*, some works extend beyond query translation to include code generation for more flexible data analysis and visualization [116]. Yet, these models are often general-purpose, whereas *ChatBGP* is tailored to questions about BGP data. Moreover, beyond generating SQL queries, *ChatBGP* produces code that is not only syntactically correct but also performance-optimized.

Chatbots for querying networking data. ChatIYP is a chatbot for the Internet Yellow Page [8, 26]. Kotaru et al. [42] proposed a system enabling network operators to query domain-specific databases containing local network metrics such as logs and flow-level data. While these systems share some design principles with *ChatBGP*—such as gathering operator questions or employing prompt engineering—they are not specialized for raw BGP data. BGP data introduces unique challenges, including managing large volumes of data and supporting diverse, complex queries, all while meeting requirements for speed, accuracy, and scalability.

12 CONCLUDING REMARKS

We do not claim *ChatBGP* solves all BGP data analysis challenges, but it bridges the gap between advanced AI-driven tools, such as LLMs, and BGP data, opening a new frontier that can transform BGP research, operations, and education. We have already taken the concrete step of integrating *ChatBGP* into an existing public BGP route collection platform. Our next aspiration is to adapt the approach we developed for *ChatBGP*’s to the integration of other datasets, both BGP-related (e.g., RPKI-ROA), as well as other infrastructure data (e.g., RIPE Atlas [95] or IYP [26]). We optimistically hope such techniques can shift the playing field in favor of Internet infrastructure security, stability, and resilience.

Ethics. This work does not raise any ethical issues.

ACKNOWLEDGEMENTS

Part of this work was funded by NGI Zero Core, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program. This research was also supported by the European Union and Wallonia through the "Wallonia 2021-2027" program, by the Association Française pour le Nom-mage Internet en Coopération (AFNIC), and by the Silicon Valley Foundation for Cisco (CG1318167 and CG1348196). This material is based on research sponsored by the National Science Foundation (NSF) OAC-2131987 and CNS-2120399 and the views do not necessarily represent those of NSF.

REFERENCES

- [1] 2024. Awesome-Text2SQL. <https://github.com/eosphoros-ai/Awesome-Text2SQL>.
- [2] RIPE RIS. Raw Data. <https://www.ripe.net/data-tools/stats/ris/>.
- [3] Emile Aben. 2020. Route Collection at the RIPE NCC - Where are we and where should we go? <https://labs.ripe.net/author/emileaben/>.
- [4] Emile Aben. 2023. Two Years of Selective Peering with RIS. <https://labs.ripe.net/author/emileaben/two-years-of-selective-peering-with-ris/>.
- [5] Thomas Alfroy, Thomas Holterbach, Thomas Krenc, KC Claffy, and Cristel Pelsser. 2023. Internet Science Moonshot: Expanding BGP Data Horizons. In *HotNets '23*. ACM.
- [6] Thomas Alfroy, Thomas Holterbach, Thomas Krenc, KC Claffy, and Cristel Pelsser. 2024. Measuring Internet Routing from the Most Valuable Points. In *Technical Report*. arXiv.
- [7] Thomas Alfroy, Thomas Holterbach, Thomas Krenc, KC Claffy, and Cristel Pelsser. 2024. The Next Generation of BGP Data Collection Platforms. In *SIGCOMM '24*.
- [8] Vasilis Andritsoudis, Pavlos Sermpezis, Ilias Dimitriadis, and Athena Vakali. 2025. Poster: ChatIYP: Enabling Natural Language Access to the Internet Yellow Pages Database. <https://arxiv.org/abs/2509.19411>
- [9] APNIC. 2022. BGP in 2022 - the routing table. <https://blog.apnic.net/2023/01/06/bgp-in-2022-the-routing-table/>.
- [10] Lorenzo Ariemma, Mariano Scazzariello, and Tommaso Caiazzi. 2021. MRT#: a Fast Multi-Threaded MRT Parser. In *IFIP/IEEE IM '21*.
- [11] Vaibhav Bajpai, Mirja Kühlewind, Jörg Ott, Jürgen Schönwälder, Anna Sperotto, and Brian Trammell. 2017. Challenges with Reproducibility. In *ACM SIGCOMM Proceedings of the Reproducibility Workshop*.
- [12] Nina Bargisen. 2024. RouteViews Peering Policy. <https://www.routeviews.org/routeviews/2025/01/10/routeviews-peering-policy/>.
- [13] Nina Bargisen. 2025. RouteViews Peering Policy. <https://www.routeviews.org/routeviews/2025/01/10/routeviews-peering-policy/>.
- [14] BGPKIT. 2022. BGPKIT. <https://blog.bgpkit.com/>.
- [15] BGPtools. 2024. bgp.tools Dependency feature. <https://x.com/bgptools/status/175777395115819065>.
- [16] LACNIC Blog. 2023. A Look at LACNIC's IPv6-Only Members. <https://blog.lacnic.net/en/a-look-at-lacnic-ipv6-only-members/>.
- [17] Larry Blunk, Craig Labovitz, and Manish Karir. 2011. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. In *RFC 6396*.
- [18] CAIDA. 2024. BGP2GO. https://blog.caida.org/best_available_data/2024/10/07/streamlining-access-to-bgp-routing-data/.
- [19] Ben Cartwright-Cox. 2023. Grave flaws in BGP Error handling. <https://blog.benjojo.co.uk/post/bgp-path-attributes-grave-error-handling>.
- [20] CGTF. 2025. CGTF Routing Information Share. <https://bgp.cgtf.net/>.
- [21] Pei-chun Cheng, Xin Zhao, Beichuan Zhang, and Lixia Zhang. 2010. Longitudinal study of BGP monitor session failures. *SIGCOMM CCR*. (2010).
- [22] CIDR. 2023. CIDR REPORT. <https://www.cidr-report.org/as2.0/>.
- [23] Ties de Kock. 2025. Querying the DFZ. <https://ripe91.ripe.net/programme/meeting-plan/sessions/22/3Q7AUL/>.
- [24] Hurricane Electric. 2025. Hurricane Electric Super Looking Glass. <https://bgp.he.net/super-lg/>.
- [25] Romain Fontugne, Anant Shah, and Emile Aben. 2017. AS Hegemony: A Robust Metric for AS Centrality.
- [26] Romain Fontugne, Malte Tashiro, Raffaele Sommesse, Mattijs Jonker, Zachary S. Bischof, and Emile Aben. 2024. The Wisdom of the Measurement Crowd: Building the Internet Yellow Pages a Knowledge Graph for the Internet. In *ACM IMC'24*.
- [27] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation.
- [28] Alessandro Giaconia. 2024. Fooling BGP hijack monitors with ease (Master thesis). <https://nsg.ee.ethz.ch/files/public/theses/2024-manipulating-bgp-monitors/thesis-1.pdf>.
- [29] Vasileios Giotsas, Georgios Smaragdakis, Christoph Dietzel, Philipp Richter, Anja Feldmann, and Arthur Berger. 2017. Inferring BGP blackholing activity in the internet. In *ACM IMC'17*.
- [30] Lilia Hannachi and Kotikalapudi Sriram. July 2021. Analysis of Propagation of Regular, Extended, and Large BGP Communities.
- [31] Zach Henderson. 2022. Take a Closer Look with Catchpoint: BGP Monitoring Live Demo. <https://www.youtube.com/watch?v=1kfA648dO8>.
- [32] Thomas Holterbach, Thomas Alfroy, and Cristel Pelsser. 2025. bgproutes.io. <https://www.bgproutes.io>.
- [33] Thomas Holterbach, Thomas Alfroy, and Cristel Pelsser. 2025. Expanding Internet Routing Visibility with BMP. <https://www.youtube.com/watch?v=gRVy0fTxOY>.
- [34] Thomas Holterbach, Thomas Alfroy, Amreesh Phokeer, Alberto Dainotti, and Cristel Pelsser. 2023. A System to Detect Forged-Origin BGP Hijacks. In *NSDI'24*.
- [35] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL.
- [36] Geoff Huston. 2024. BGP in 2023 - BGP updates. <https://blog.apnic.net/2024/01/10/bgp-in-2023-bgp-updates/>.
- [37] Ebrima Jaw, Thomas Krenc, Moritz Muller, kc claffy, Lambert Nieuwenhuis, and Cristian Hesselman. 2025. Noisy Neighbours: Keep the Neighbourhood Quiet. *CNSM*.
- [38] Savvas Kastanakis, Vasileios Giotsas, Ioana Livadariu, and Neeraj Suri. 2023. Replication: 20 Years of Inferring Interdomain Routing Policies. In *ACM IMC'24*.
- [39] Kentik. 2020. NETWORK FLOW DATA INGESTION, STORAGE, AND ANALYSIS. PatentNo:US11,330,002B2.
- [40] Kentik. 2022. The Kentik Data Engine. <https://www.kentik.com/resources/kentik-kde-whitepaper/>.
- [41] Robert Kisteleki. 2023. RIPE NCC Measurement Data Retention Principles. <https://labs.ripe.net/author/kistel/ripe-ncc-measurement-data-retention-principles/>.
- [42] Manikanta Kotaru. 2023. Adapting Foundation Models for Operator Data Analytics (*ACM HotNets '23*).
- [43] Thomas Krenc, Robert Beverly, and Georgios Smaragdakis. 2021. AS-level BGP community usage classification. In *ACM IMC'21*.
- [44] Thomas Krenc, Matthew Luckie, Alexander Marder, and kc claffy. 2023. Coarse-grained Inference of BGP Community Intent. In *ACM IMC'23*.
- [45] Mirjam Kühne. 2021. Interesting Graph - AS Path Lengths Over Time. <https://labs.ripe.net/author/mirjam/interesting-graph-as-path-lengths-over-time/>.
- [46] NANOG Mailing List. 2021. Any2 Los Angeles down again. <https://mailman.nanog.org/pipermail/nanog/2021-January/211770.html>.
- [47] NANOG Mailing List. 2021. AS 701 ? <https://mailman.nanog.org/pipermail/nanog/2021-January/211514.html>.
- [48] NANOG Mailing List. 2021. BGP and The zero window edge. <https://mailman.nanog.org/pipermail/nanog/2021-April/213333.html>.
- [49] NANOG Mailing List. 2021. BGP hijack by AS25478? <https://mailman.nanog.org/pipermail/nanog/2021-November/216357.html>.
- [50] NANOG Mailing List. 2021. DoD IP Space. <https://mailman.nanog.org/pipermail/nanog/2021-April/213309.html>.
- [51] NANOG Mailing List. 2021. Google Geo Location Issues. <https://mailman.nanog.org/pipermail/nanog/2021-July/214151.html>.
- [52] NANOG Mailing List. 2021. PeerinDB refuses to register certain networks [was: Setting sensible max-prefix limits]. <https://mailman.nanog.org/pipermail/nanog/2021-August/214629.html>.
- [53] NANOG Mailing List. 2021. "Tactical" /24 announcements. <https://mailman.nanog.org/pipermail/nanog/2021-August/214560.html>.
- [54] NANOG Mailing List. 2021. Weekly Routing Table Report. <https://mailman.nanog.org/pipermail/nanog/2021-April/213027.html>.
- [55] NANOG Mailing List. 2021. Where to get IPv4 block these day. <https://mailman.nanog.org/pipermail/nanog/2021-August/214469.html>.
- [56] NANOG Mailing List. 2021. Zayo or HE for IP transit. <https://mailman.nanog.org/pipermail/nanog/2021-April/213217.html>.
- [57] NANOG Mailing List. 2022. A few questions regarding about RPKI/invalids. <https://mailman.nanog.org/pipermail/nanog/2022-March/218693.html>.
- [58] NANOG Mailing List. 2022. Amazon peering revisited. <https://mailman.nanog.org/pipermail/nanog/2022-February/217463.html>.
- [59] NANOG Mailing List. 2022. AS 3356 (Level 3) - Community 3356:666. <https://mailman.nanog.org/pipermail/nanog/2021-August/214447.html>.
- [60] NANOG Mailing List. 2022. AS3356 Announcing 2000::/12. <https://mailman.nanog.org/pipermail/nanog/2022-December/221123.html>.
- [61] NANOG Mailing List. 2022. Asia Apac networks. <https://mailman.nanog.org/pipermail/nanog/2022-March/218105.html>.
- [62] NANOG Mailing List. 2022. Authoritative Resources for Public DNS Pinging. <https://mailman.nanog.org/pipermail/nanog/2022-February/217527.html>.
- [63] NANOG Mailing List. 2022. Cogent & Google reachability stable? <https://mailman.nanog.org/pipermail/nanog/2022-June/219564.html>.
- [64] NANOG Mailing List. 2022. Longest prepend(255 times) as path found. <https://mailman.nanog.org/pipermail/nanog/2022-August/220386.html>.
- [65] NANOG Mailing List. 2022. Longest prepend(255 times) as path found. <https://mailman.nanog.org/pipermail/nanog/2022-August/220375.html>.
- [66] NANOG Mailing List. 2022. Newbies Question: Do I really need to sacrifice Prefix-aggregation to do BGP Load-sharing? (the case of Multi-homed + Multi-routers + Multi-upstreams). <https://mailman.nanog.org/pipermail/nanog/2022-October/220826.html>.

- [67] NANOG Mailing List. 2022. RFC: BOGONs over BGP, adding some ranges. <https://mailman.nanog.org/pipermail/nanog/2022-September/220423.html>.
- [68] NANOG Mailing List. 2022. Rogers Outage Canada. <https://mailman.nanog.org/pipermail/nanog/2022-July/219895.html>.
- [69] NANOG Mailing List. 2022. Verizon no BGP route to some of AS38365 (182.61.200.0/24). <https://mailman.nanog.org/pipermail/nanog/2022-July/219996.html>.
- [70] NANOG Mailing List. 2022. What's going on with AS147028? <https://mailman.nanog.org/pipermail/nanog/2022-July/219951.html>.
- [71] NANOG Mailing List. 2022. Yet another BGP hijacking towards AS16509. <https://mailman.nanog.org/pipermail/nanog/2022-August/220321.html>.
- [72] NANOG Mailing List. 2023. Any2 Los Angeles down again. <https://mailman.nanog.org/pipermail/nanog/2021-February/211836.html>.
- [73] NANOG Mailing List. 2023. BGP hijack? <https://mailman.nanog.org/pipermail/nanog/2023-October/223834.html>.
- [74] NANOG Mailing List. 2023. <https://mailman.nanog.org/pipermail/nanog/2023-August/222942.html>. <https://mailman.nanog.org/pipermail/nanog/2023-August/222942.html>.
- [75] NANOG Mailing List. 2023. IPv4 Subnet 23.151.232.0/24 blackholed? <https://mailman.nanog.org/pipermail/nanog/2023-April/221998.html>.
- [76] NANOG Mailing List. 2023. maximum ipv4 bgp prefix length of /24 ? <https://mailman.nanog.org/pipermail/nanog/2023-October/223487.html>.
- [77] NANOG Mailing List. 2023. SDN Internet Router (sir). <https://mailman.nanog.org/pipermail/nanog/2023-January/221291.html>.
- [78] NANOG Mailing List. 2023. Spamhaus flags any IP announced by our ASN as a criminal network. <https://mailman.nanog.org/pipermail/nanog/2023-March/221772.html>.
- [79] Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. 2024. Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education (*SIGCSE 2024*).
- [80] Matthew Luckie. 2014. Spurious routes in public BGP data. *SIGCOMM CCR*.
- [81] Andra Lutu. 2013. The BGP Visibility Scanner. https://labs.ripe.net/author/andra_lutu/the-bgp-visibility-scanner/.
- [82] Doug Madory. 2023. A Brief History of the Internet's Biggest BGP Incidents. <https://www.kentik.com/blog/a-brief-history-of-the-internets-biggest-bgp-incidents/>.
- [83] Lefteris Manassakis. 2023. Suspicious Prefix Announcements? Or RIS Peer Malfunction? <https://www.thousandeyes.com/blog/suspicious-prefix-announcements-or-ris-peer-malfunction>.
- [84] Pedro Marcos, Lars Prehn, Lucas Leal, Alberto Dainotti, Anja Feldmann, and Marinho Barcellos. 2020. AS-Path Prepending: there is no rose without a thorn. In *ACM IMC'20*.
- [85] Stephen McQuistin, Sree Priyanka Uppu, and Marcel Flores. 2019. Taming Anycast in the Wild Internet. In *ACM IMC'19*.
- [86] Oleg Muravskiy. 2020. A Problem with the Processing of RIS Data on RIPEstat: What We Learned. https://labs.ripe.net/author/oleg_m/a-problem-with-the-processing-of-ris-data-on-ripestat-what-we-learned/.
- [87] Anil Murty. 2022. Introducing BGP Monitoring from Kentik. <https://www.kentik.com/blog/bgp-monitoring-from-kentik/>.
- [88] NANOG. 1992. NANOG mailing list. <https://mailman.nanog.org/mailman/listinfo/nanog>.
- [89] RIPE NCC. 2024. RIPEstat: Providing open data and insights for Internet resources. <https://stat.ripe.net/about/>.
- [90] Eugenio Nerio Nemmi, Francesco Sassi, Massimo La Morgia, Cecilia Testart, Alessandro Mei, and Alberto Dainotti. 2021. The parallel lives of autonomous systems: ASN allocations vs. BGP. In *ACM IMC'21*.
- [91] NLNOG. 2025. NLNOG Looking Glass. <https://lg.ring.nlnog.net/>.
- [92] Chiara Orsini, Alistair King, Danilo Giordano, Vasileios Giotsas, and Alberto Dainotti. 2016. BGPStream: A Software Framework for Live and Historical BGP Data Analysis. In *IMC '16*.
- [93] PCH. 2010. Packet Clearing House. <https://www.pch.net/>.
- [94] QRator. 2023. 2023 Internet Segment Reliability Report. <https://qrator.net/blog/details/2023-internet-segment-reliability-rep>.
- [95] RIPE. [n. d.]. The RIPE Atlas measurement platform. <https://atlas.ripe.net/>.
- [96] RIPE. 2025. RIPEstat Looking Glass. <https://stat.ripe.net/lg>.
- [97] RIPE RIS. 2012. Update on AS Path Lengths Over Time. <https://labs.ripe.net/author/mirjam/update-on-as-path-lengths-over-time/>.
- [98] RIPE RIS. 2024. Per-peer dump files. <https://ris.ripe.net/docs/prototypes/per-peer-dumps/>.
- [99] Erik Romijn. 2010. RIPE NCC and Duke University BGP Experiment. <https://labs.ripe.net/author/erik/ripe-ncc-and-duke-university-bgp-experiment/>.
- [100] Matthew Roughan, Simon Jonathan Tuke, and Olaf Maennel. 2008. Bigfoot, Sasquatch, the Yeti and other missing links: what we don't know about the AS graph.
- [101] RouteViews. 2024. RouteViews API. <https://api.routeviews.org/>.
- [102] RIPE routing-wg Mailing List. 2018. [routing-wg] Fwd: Time to add 2002::16 to bogon filters? <https://www.ripe.net/ripe/mail/archives/routing-wg/2018-June/003561.html>.
- [103] John Scudder, Rex Fernando, and Stephen Stuart. 2016. BGP Monitoring Protocol (BMP). RFC 7854.
- [104] Chris Siebenmann. 2022. What are symmetric and asymmetric IP routing. <https://blog.apnic.net/2022/08/01/what-are-symmetric-and-asymmetric-ip-routing/>.
- [105] Felipe Victolla Silveira. 2024. RIPE NCC Measurement Data Retention Principles. https://labs.ripe.net/author/felipe_victolla_silveira/reducing-the-ripe-nccs-data-centre-footprint/.
- [106] Kotikalapudi Sriram and Warren "Ace" Kumari. [n. d.]. Recommendation for Not Using AS_SET and AS_CONFED_SET in BGP. RFC 6472. <https://www.rfc-editor.org/info/rfc6472>.
- [107] Florian Streibelt, Franziska Lichtblau, Robert Beverly, Anja Feldmann, Cristel Pelsser, Georgios Smaragdakis, and Randy Bush. 2018. BGP Communities: Even more Worms in the Routing Can. In *ACM IMC'18*.
- [108] Stephen Strowes. 2019. Visibility of IPv4 and IPv6 Prefix Lengths in 2019. https://labs.ripe.net/author/stephen_strowes/visibility-of-ipv4-and-ipv6-prefix-lengths-in-2019/.
- [109] Georgia Tech. 2024. Global Routing Intelligence Platform. <https://grip.inetintel.cc.gatech.edu/>.
- [110] Cecilia Testart, Philipp Richter, Alistair King, Alberto Dainotti, and David Clark. 2019. Profiling BGP Serial Hijackers: Capturing Persistent Misbehavior in the Global Routing Table. In *ACM IMC'19*.
- [111] ThousandEyes. 2024. BGP Monitoring — Route Monitoring. <https://www.thousandeyes.com/solutions/bgp-and-route-monitoring>.
- [112] Oregon Univ. 2021. Route Views Project. www.routeviews.org/.
- [113] Laurent Vanbever. 2021. What happened to the Internet in 2021 in one print. <https://x.com/lvanbever/status/1475537579843932161>.
- [114] Teun Vink. 2022. An Introduction to IRR Explorer. <https://labs.ripe.net/author/teun/an-introduction-to-irr-explorer/>.
- [115] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanell Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *CoRR* (2018).
- [116] Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. 2023. DataCopilot: Bridging Billions of Data and Humans with Autonomous Workflow. *arXiv preprint arXiv:2306.07209* (2023).
- [117] PEIDONG ZHU, HONG NING, WENPING DENG, KAIYU CAI, JIANQIANG ZHAO, and LITAO ZHOU. Patent CN100555935C 2009. Network monitoring data compression storage and associated detecting method based on similar data set.

Questions	Parameters	Inspired from	RIPEstat
Autonomous System (AS)			
1 How many ASes are in the routing system ?	IP protocol, date, VP	[22, 54]	✗
2 Identify ASes that are always either at the beginning or at the end of the AS path, meaning that never appear in the middle, and that have at least two different neighbors	IP protocol, date, VP	[15][51, 53][38]	✗
▶ 3 How many ASes appear always at the end of the AS path and never in the middle	IP protocol, date, VP	[54][94][107]	✗
4 How many ASes always appear at the middle of the AS paths and never at one end?	IP protocol, date, VP	[54][9][107]	✗
5 Does AS x appear in an AS path used to reach prefix p ?	date, VP, IP prefix, AS x	[69, 71]	✓
6 Which AS is using AS path prepending, i.e. same AS prepended?	date, VP	[54][97][61, 64][84]	✗
AS path			
7 What is the average length of an AS path in the routing table ?	IP protocol, date, VP	[54][45]	✗
8 Which are the AS paths where AS x appears ?	time interval, VP, AS x	[63, 70, 75]	✗
▶ 9 What are the paths where AS x is at the origin, ie the last AS?	time interval, AS x, VP	[73, 78]	✓
10 How often the AS path from AS x to AS y matches the inverted AS path between AS y to AS x?	date, VP, IP protocol, AS x, ASy	[104][77]	✗
▶ 11 What is the percentage of single hop AS paths ?	date, VP, IP protocol	[85]	✗
Connectivity			
▶ 12 Who are AS's x neighbors ?	IP protocol, date, VP, AS x	[56, 56][34]	✓
13 What is the shortest distance between AS x and AS y in terms of AS hops ?	IP protocol, date, VP, AS x, ASy	[65, 66]	✗
14 Are AS x and AS y connected ?	IP protocol, date, VP, AS x, ASy	[52, 74]	✓
15 Which ASes are connected to AS x without being at the beginning or at the end of the AS path?	IP protocol, date, VP, AS x	[58, 70]	✗
16 Which proportion of the route pass through AS x ?	date, VP, AS x	[25]	✗
Time			
▶ 17 When did AS x appear in the routing table ?	IP protocol, VP, AS x	[90]	✓
18 Since when AS x and AS y are connected ?	IP protocol, VP, AS x, ASy	[47]	✓
▶ 19 What time was the last update observed for prefix p for every vantage point?	IP prefix	[48]	✗
20 What is the most recent day for which AS x and AS y appear disconnected from all the vantage points?	IP protocol, AS x, ASy	[46, 47]	✓
21 Was there a burst of withdrawals i.e., 2 times more withdrawals than the average on other time periods with the same duration	time interval, VP	[62, 68]	✗
22 Give me the number of announcements that were collected for every hour.	time interval, VP	[68]	✓
23 What is the visibility growth in terms number of distinct AS links observed for every VP?	time interval	[100]	✗
IP prefix			
▶ 24 Did the last AS in any AS paths for prefix p change?	IP prefix, VP, time interval	[109][60]	✓
25 How many prefixes can we see?	date, VP, IP protocol	[22, 54][81][29]	✗
26 What are the routing changes for prefix p ?	time interval, VP, IP protocol, IP prefix	[99][49, 72]	✓
▶ 27 Can we see prefix p ?	date, VP, IP prefix	[60, 69, 75]	✓
28 Which ASes originate prefix p ?	date, VP, IP prefix	[114][50, 57, 73]	✓
29 How many bogon prefixes are announced by AS x ?	date, VP, AS x, IP protocol	[67, 102]	✗
30 Among the ASes that announce an IPv4 prefix, what is the proportion that also announce an IPv6 prefix?	date, VP	[16][55]	✗
31 How many /24 AS x is announcing?	date, AS x, IP protocol	[108]	✓
32 Which prefixes are originated by AS x?	date, VP, IP protocol, AS x	[60]	✓
Vantage Point (VP)			
▶ 33 Rank the vantage points based on the number of updates they collected.	time interval, VP	[113]	✗
34 What is the size of the RIB of peer p?	date, VP	[9][76]	✓
Community Values			
35 Which updates announce a blackholing community ?	time interval, VP	[59][29]	✗
▶ 36 Which updates hold community c ?	time interval, VP, community set	[59][43]	✗
37 Which updates use large communities ?	time interval, VP	[30][43]	✗
38 How many distinct community values are related to AS x, i.e how many community values start with x?	date, VP, AS x	[44]	✗
39 What are the community values for prefix p ?	date, VP, IP prefix	[43]	✓
Legend			
▶ Questions marked with a red arrow and highlighted in light red are used in our BGPstream evaluation (Table 2).			
[X] Blue references indicate automated tools such as RIPEstat.			
[X] Green references indicate blog posts, such as those from RIPE or APNIC or posts from social medias (Twitter).			
[X] Orange references indicate emails from mailing lists, primarily from the NANOG mailing list.			
[X] Gray references indicate research papers published in top conferences.			
Rows with a gray background indicates the scope of the following questions.			
✓ Green checks indicate questions that can be answered using the RIPEstat API with a single or a small number of API calls.			
✗ Red crosses indicate questions that cannot be answered using the RIPEstat API or require an excessive number of API calls (e.g., more than 1,000).			

Table 7: List of questions, along with their parameters, related to the BGP data as collected by BGP routes collection platforms such as RIS and RV, and collected using the methodology described in §8.

A USE OF GENERATIVE AI

All Python code snippets produced by *ChatBGP* were generated by ChatGPT; this is the very essence of *ChatBGP*—using LLMs to lower the barrier to working with BGP data. As mentioned in §10, the Python script used to generate Fig. 4 was also produced by *ChatBGP*—and therefore by ChatGPT. Finally, all substantive content of the paper was authored by the researchers; the LLM was used only for stylistic, grammatical, or formatting refinements.

B EVALUATION ARTIFACTS

To facilitate reproducibility, we make several artifacts available in a Git repository:

https://github.com/bgproutes-io/chatbgp_imc26_artifacts.

These include the prompt generator of *ChatBGP* (§B.1) and additional software materials to enable a more thorough evaluation of *ChatBGP* (§B.2).

B.1 *ChatBGP*'s code generator

A Python script is included to test *ChatBGP*'s prompt generation. The script takes three inputs:

- A selection of 39 questions from the 779 generated BGP-related question set;
- Corresponding code for each selected question, generated from the code skeletons, by replacing macros with actual parameter values;
- Specifications of `pybgproutesapi` primitive functions.

With these inputs, the script generates a prompt that we can use for the few-shot prompting. Below, we highlight two considerations for users of *ChatBGP*'s prompt generator.

Scope of queries. *ChatBGP* is specifically designed to answer queries about BGP data as collected by route collection platforms and detailed in Fig. 2. Thus, *ChatBGP* cannot generate accurate code for BGP-related questions requiring external data (e.g., RPKI-ROA) or advanced algorithms (e.g., for business relationship inference). For such cases, the `pybgproutesapi` library remains valuable for retrieving BGP data efficiently, but users still need to manually write their code.

BGP route aggregation. We designed *ChatBGP*'s primitive functions to exclude updates containing AS paths with AS-sets (resulting from route aggregation). This simplifies their specifications—e.g., AS paths can be represented as a list of integers instead of a list of lists—improving code generation accuracy. This design choice does not affect the user experience, as AS-sets are rarely used, have limited relevance (none of the 39 questions in Table 9 involve AS-sets), and are discouraged in practice [106]. Observe that *ChatBGP*'s database still stores routes with AS-sets, allowing them to be retrieved if needed, e.g., through other possible primitive functions.

B.2 Software artifacts

We categorize our artifacts as follows:

Code generation artifacts.

Generated BGP-related questions. We share a JSON file containing the set of questions generated using the methodology described in §8. This dataset includes a total of 779 questions. Each question

in the JSON file has parameter values (parameters are shown in Table 7).

Code skeletons for training ChatGPT. We provide 39 code skeletons used for few-shot prompting, one for each question listed in Table 7. These skeletons include macros (i.e., keywords) designed to be replaced with actual parameter values (which are in the JSON file with the 779 questions) when generating the final code.

Prompt and code for bgpstream-based code generation (§C).

We provide the complete prompt used to implement the prompt-engineering scheme for generating bgpstream-based code with ChatGPT. Additionally, we share the code snippets generated during this process, most of which are incorrect (except for three questions, as detailed in §C).

Artifacts to produce results.

Code for generating results in Table 2. We release Python

scripts utilizing the `bgpstream` library to measure the time required to answer BGP-related questions using MRT-based archives.

Code for generating results in Table 9. We provide Python

scripts that use the `pybgproutesapi` library to evaluate *ChatBGP*'s backend performance. These scripts measure the response time for typical queries and provide insights into the expected user experience with *ChatBGP*.

Code for generating Fig. 4. To generate Fig. 4, we provided *ChatBGP* with the following query: “Create a figure using the `matplotlib` library to visualize the percentage of identical entries between a reference RIB and subsequent RIBs. Identical entries are defined as having the same prefix, AS path, and community values. Use the RIB from July 1, 2024, as the reference (retrieved once) and compare it to RIBs from the following days in three-day intervals up to July 31, 2024. Include data from 20 VPs and parallelize the process across multiple cores to optimize computations for both VPs and dates. Represent the results in the figure using a boxplot for each time step, showing the distribution of percentages across the 20 VPs.” As explained in §10, we made minor modifications to the code generated by *ChatBGP* to enhance the visual design of the figure. We also slightly modified the script by reducing the number of VPs used, allowing it to generate the figure more quickly.

C GENERATING CODE TO ANALYZE BGP DATA IS CHALLENGING

Prompt engineering, including its subset few-shot learning, is a widely used technique for adapting LLMs to specific domains, such as generating code for targeted use cases. In this section, we provide additional detail on how we demonstrated that generating accurate and efficient Python code for retrieving and analyzing BGP data is challenging. More precisely, we experimentally show that applying prompt engineering without carefully defining the API as explained in §7 (i.e., identifying the range of queries to address and defining simple and intuitive primitive functions) and without incorporating the key steps outlined in §8 (i.e., providing carefully crafted code snippets as ground truth), leads to numerous errors.

Methodology. We trained ChatGPT-5.1 using a prompt-engineering scheme to generate Python code that uses the `bgpstream` library for downloading and analyzing BGP data (MRT files). We chose `bgpstream` for three reasons: (i) it represents the state-of-the-art

tool for BGP data analysis, (ii) it is capable of addressing all 39 questions listed in Table 7 when code is generated manually, and (iii) it is accompanied by comprehensive documentation with several examples.

Our prompt-engineering scheme works as follows. First, we include in the prompt the official documentation for the objects and functions provided by the `bgpstream` library, available at https://bgpstream.caida.org/docs/api/pybgpstream/_pybgpstream.html. Next, we incorporate examples from the tutorials on the `bgpstream` website, located at <https://bgpstream.caida.org/docs/tutorials/pybgpstream>, to provide practical use cases. Finally, we identified two GitHub repositories that use the `bgpstream` library and included their code along with their accompanying documentation in the prompt. These two GitHub repositories are <https://gist.github.com/codeout/74ade8e269abd3f4bed66787c63f9c9b> and <https://github.com/estcarisimo/BGP-Collector-Analysis-Toolkit>. We excluded other repositories where `bgpstream`-related lines constituted only a small percentage of the total codebase.

Once ChatGPT-5.1 was loaded with this prompt, we posed the 39 questions listed in Table 7 individually, each phrased identically to the human-generated questions used to evaluate *ChatBGP* (Table 10). The prompt contained approximately 9,000 tokens—a token budget under which *ChatBGP* achieves over 82% accuracy in generating correct code snippets for human-phrased questions (Table 10).

Results. Using this prompt-engineering scheme, ChatGPT-5.1 generated fully correct code snippets for only nine questions (Q8, Q9, Q21, Q22, Q23, Q26, Q34, Q33 and Q36). For most other questions, while the high-level logic was often reasonable, the use of the `bgpstream` library was either incorrect or highly suboptimal (e.g., processing multiple RIB dumps when a single one was sufficient). These issues led to execution failures, incorrect outputs, or substantially increased running times. A recurring problem is that `bgpstream` cannot retrieve a RIB at an arbitrary timestamp—it can only load RIBs at the moments when RIS or RV export them—causing many queries to fail. Another common source of errors was improper filter usage; for example, in Q10, an incorrect AS-path regular expression was applied. In some cases, the returned values were entirely wrong; for instance, in Q37, the regular expression for matching large communities was incorrect, producing erroneous results. For the questions in the *Time* section of Table 7, the generated code was often incorrect because it processed RIB dumps instead updates. Overall, the success rate (23%) is markedly lower than *ChatBGP*’s accuracy (Table 10).

Artifacts. We share the prompt and codes used in the experiment in §B.

D DETAILED BGP DATA ENCODING SCHEME

We now detail the database storage scheme used by *ChatBGP* and introduced in §6. Recall that this scheme is effective because BGP data is highly redundant and therefore exposes high compression opportunities.

Database structure. Fig. 7 illustrates the architecture of *ChatBGP*’s database. The `VANTAGE_POINTS` table lists all VPs along with their characteristics, e.g., the IP address and AS number. Each

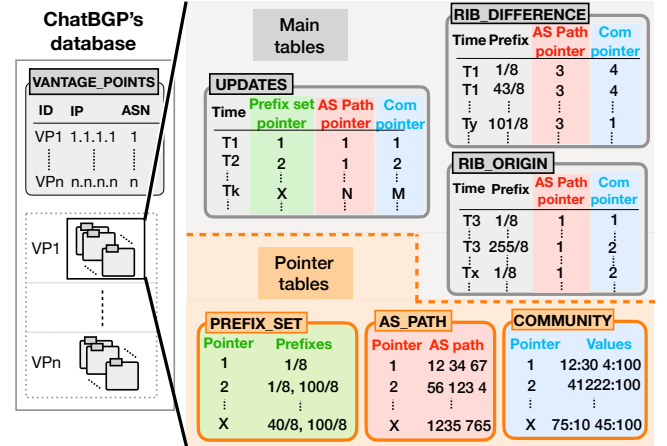


Figure 7: *ChatBGP*’s database structure. Each VP has dedicated tables to store BGP updates and RIB dumps.

VP has its own set of tables: `UPDATES` stores the raw BGP updates received by the VP, while `RIB_ORIGIN` and `RIB_DIFFERENCE` store the full and partial RIBs, respectively, for that VP. In these tables, components such as prefixes, AS paths, and community values are stored as pointers. Pointer tables—`PREFIX_SET`, `AS_PATH`, and `COMMUNITY`—map each pointer to its actual value. In our production setup, which uses PostgreSQL (§10), across all VPs, approximately three quarters of the disk space are occupied by the `UPDATES` tables, followed by the `RIB_ORIGIN` and `RIB_DIFFERENCE` tables. The remaining tables—`PREFIX_SET`, `AS_PATH`, and `COMMUNITY`—consume comparatively little space.

Optimizations for fast data retrieval with PostgreSQL. By default, PostgreSQL scans the table row by row to locate entries matching a query (e.g., BGP updates within a specified time interval). This approach is inefficient for large tables and complex queries such as regular expression matching. *ChatBGP* thus relies on two optimizations to enable fast data retrieval.

Column indexing: We configure PostgreSQL to create a tree-like structure for specific columns (a.k.a indexing), allowing search operations to execute in an average time complexity of $O(\log(n))$ at the cost of slightly increased memory usage. For *ChatBGP*, we index the *time* and *pointer* columns.

Limited regular expression query space: Indexing is less effective for columns storing strings, e.g., prefixes, AS paths, and community values. Thus, queries requiring regular expression matching on these columns (e.g., the code snippet in §5) must scan the entire table. Fortunately, regular expression matching on BGP updates is applied to the pointer tables that store the actual value of the update components in string format, and which are significantly smaller than the main tables. For instance, after processing 100 days of data (July to Oct. 2024) for a full feeder IPv4 VP, the `AS_PATH` table contains 340k entries— $\approx 21\times$ fewer than the `UPDATE` table. Similarly, the `PREFIX_SET` and `COMMUNITY` table contains $\approx 8\times$ and $\approx 216\times$ fewer entries, respectively, than the `UPDATE` table. The reduced size of these pointer tables allows regular expression matching to quickly execute. Note that regular expression matching on the

prefix of RIB dumps is executed directly on the `RIB_ORIGIN` and `RIB_DIFFERENCE` tables. Yet, the size of these tables is bounded by the number of announced prefixes, i.e., $\approx 1\text{M}$ rows for IPv4 and 222k for IPv6 [22].

Data removal. Since each VP has its own dedicated tables, *ChatBGP*'s operator can remove data on a per-VP basis, e.g., when a VP is deemed unreliable or faulty. Data can also be removed at a finer granularity (e.g., removing specific updates), which is useful for implementing a sliding window to retain only the most recent days of data (see §9).

E SENSITIVITY OF *CHATBGP* CODE GENERATION

The correctness of code generated by *ChatBGP* (evaluated in §9) depends on several factors: (i) the complexity of the question, (ii) the clarity in phrasing of the question, and (iii) the number of tokens used for training ChatGPT.

Methodology. The methodology is similar to the one used in §9 to measure the correctness of generated code (Table 5), except that we now manually assess correctness. This allows us to count code as correct when it is semantically valid for a human user, even if it differs from the programmatic baseline (e.g., using hourly intervals such as 23:00:00–00:00:00 instead of 22:59:00–23:59:00 is considered correct). We also assessed the impact of phrasing precision by using two levels of question clarity. In the first scenario, questions were generated by the older and less capable ChatGPT-3.5, e.g., ChatGPT-3.5 might phrase *Q27* like "Can we access the data for prefix *p*", which is ambiguous. In the second scenario, we rewrote the questions to be more clear and precise. Finally, we investigated the impact of varying the token budget for training to construct the prompt (§8). With a 5k token limit, not all 39 questions and their ground-truth code snippets could be included in the prompt. Using 15k tokens, the default in *ChatBGP*'s design, allowed us to include each question exactly once. Lastly, a 30k token limit enabled inclusion of each question twice.

Results. Table 10 presents the percentage of questions for which *ChatBGP* generated correct Python code and lists the specific question numbers where *ChatBGP* failed to produce correct codes. When a human wrote the question, *ChatBGP* achieved a success rate of 92–95% with 30k and 15k tokens, respectively. Even with poor phrasing, *ChatBGP* generated accurate code snippets over 80% of the time. As expected, increasing the number of tokens used in few-shot learning improved the accuracy of the generated code. Interestingly, the accuracy with 15k tokens was slightly higher than with 30k tokens when questions were phrased by humans. This suggests that the performance of few-shot learning began to converge at 15k tokens—the threshold at which all 39 questions are included at least once in the prompt. Any differences at higher token counts are likely due to the inherent randomness of the probabilistic models underpinning LLMs. Finally, questions leading to incorrect code were often the more complex ones, e.g., *Q10* is challenging to phrase clearly and answer accurately—even for a human, who might question whether sub-paths between the two ASes should be considered—and thus frequently resulted in an incorrect code snippet. Note that even when the generated code snippets are not fully correct, they remain

simple and concise (20 lines in the median case), making it easy for users to spot issues and request refinements (§E).

Asking for refinements. In rare cases where *ChatBGP* generates an incorrect Python code, users can request refinements to correct the error. We illustrate such a case with an incorrect code snippet that *ChatBGP* produced when we evaluated its accuracy in §9 (Table 10), for *Q11*, using 30k tokens, when the question, phrased by ChatGPT 3.5, was: *What is the percentage of single hop AS paths observed on 3/1/24 at 15:21:05 from peers 143.192.143.111, 50.188.246.246, 103.52.15.204, and 2402:1b80::1*. The incorrect code, shown in Listing 1, consists of 30 lines, making it easy to spot the mistake: it incorrectly classifies AS paths of length one as single-hop paths (line 20). This likely results from ambiguities or lack of precision in the question phrasing. In such cases, a user can refine their request by asking: "A single-hop AS path contains exactly two AS numbers, not one. Fix it." and *ChatBGP* then generates the correct code.

```

1 from pybgproutesapi import *
2
3 # Initialize the single hop AS path count and total AS path count
4 single_hop_path_count = 0
5 total_as_path_count = 0
6
7 # Get the right set of vantage points to retrieve.
8 selected_vantage_points = vantage_points(vantage_points=['143.192.143.111', '
9 50.188.246.246', '103.52.15.204', '2402:1b80::1'])
10
11 for vantage_point in selected_vantage_points:
12     # Get the routing information base (rib) for this vantage point.
13     rib_data = rib(vantage_point, date='03/01/2024-15:21:05')
14
15     # Iterate over all entries in the rib.
16     for aspath, community in rib_data.values():
17         # Increment the total AS path count.
18         total_as_path_count += 1
19
20         # Check if AS path is a single hop.
21         if len(aspath) == 1:
22             single_hop_path_count += 1
23
24 # Calculate the percentage
25 if total_as_path_count > 0:
26     percentage = (single_hop_path_count / total_as_path_count) * 100
27 else:
28     percentage = 0
29
30 # Print the percentage
31 print("{:.2f}".format(percentage))

```

Listing 1: Incorrect code generated by *ChatBGP*.

Number of tokens used for the few-shot learning	Percentage of correct code snippets	
	Questions written by chatGPT 3.5	Questions written by a human (us)
7500	77% (wrong for Q10, Q11, Q15, Q28, Q29, Q31, Q32, Q34, Q36)	82% (wrong for Q10, Q13, Q19, Q20, Q24, Q35, Q39)
15000	82% (wrong for Q2, Q10, Q15, Q31, Q34, Q36, Q37)	95% (wrong for Q9, Q15)
30000	85% (wrong for Q10, Q11, Q15, Q20, Q28, Q34)	92% (wrong for Q2, Q10, Q20)

Table 10: Accuracy of *ChatBGP*'s code generation for the questions in Table 7.

Parameter	Type	Required	Default	Description
Endpoint: /vantage_points				
vp_ids	string or null	no	—	Comma-separated list of VP IDs to filter.
vp_asns	string or null	no	—	Comma-separated list of ASNs to filter.
date	string or null	no	—	Date used to retrieve the VPs (YYYY-MM-DDTHH:MM:SS).
date_end	string or null	no	—	If set, check VP up within (date, date_end).
data_afi	integer or null	no	both	AFI: 4 for IPv4, 6 for IPv6; null includes both.
sources	string or null	no	all	Comma-separated collection platforms (e.g., ris, routeviews, bgproutes.io, pch, cgtf).
countries	string or null	no	all	Comma-separated VP IP geolocation country codes (e.g., FR, US, DE).
org_countries	string or null	no	all	Comma-separated org (ASN) country codes.
rib_size_v4	string or null	no	—	RIB v4 filter: operator, value (e.g., >, 900000).
rib_size_v6	string or null	no	—	RIB v6 filter: operator, value (e.g., >=, 100000).
return_uptime_intervals	boolean or null	no	false	If true, include uptime intervals.
Endpoint: /updates				
vp_ids	string or null	yes	—	Comma-separated VP IDs.
start_date	string	yes	—	Start time (YYYY-MM-DDTHH:MM:SS).
end_date	string	yes	—	End time (YYYY-MM-DDTHH:MM:SS).
data_afi	integer or null	no	both	AFI: 4 (IPv4), 6 (IPv6).
return_count	boolean	no	false	If true, return only a count.
chronological_order	boolean	no	true	true: ascending by time.
max_updates_to_return	integer or null	no	—	Upper bound on number of updates.
type_filter	string or null	no	—	Filter by update type.
prefix_filter	string or null	no	—	Filter by prefix.
prefix_exact_match	string or null	no	—	Exact prefix match control.
return_aspath	boolean	no	true	Include AS_PATH in results.
aspath_regexp	string or null	no	—	Regex over AS_PATH.
aspath_exact_match	string or null	no	—	Exact-match control for AS_PATH.
return_community	boolean	no	true	Include communities in results.
community_regexp	string or null	no	—	Regex over communities.
Endpoint: /rib				
vp_ids	string or null	yes	—	Comma-separated VP IDs.
date	string	yes	—	Date used to retrieve the RIB (YYYY-MM-DDTHH:MM:SS).
data_afi	integer or null	no	both	AFI: 4 (IPv4), 6 (IPv6).
prefix_filter	string or null	no	—	Filter by prefix.
prefix_exact_match	string or null	no	—	Exact prefix match control.
return_aspath	boolean	no	true	Include AS_PATH in results.
aspath_exact_match	string or null	no	—	Exact-match control for AS_PATH.
aspath_regexp	string or null	no	—	Regex over AS_PATH.
return_community	boolean	no	true	Include communities in results.
community_regexp	string or null	no	—	Regex over communities.
return_count	boolean	no	false	If true, return only a count.
Endpoint: /topology				
vp_ids	string or null	yes	—	Comma-separated VP IDs.
date	string	yes	—	Date used to retrieve the topology (YYYY-MM-DD or YYYY-MM-DDTHH:MM:SS).
directed	boolean	no	false	If true, return a directed graph; otherwise return an undirected graph.

Table 8: *ChatBGP*'s API endpoints and parameters. There is an AND logic across filters: Multiple filters must all be satisfied for a result to be included (e.g., both `prefix_exact_match` and `aspath_regexp` must match). There is an OR logic within filters: If a filter accepts multiple values, any of them may match (e.g., multiple entries in `prefix_exact_match` or `prefix_filter`).

Q17	2m (13x) 762B (1)	52s, 68B				
Q18		5m (65x) 3KB (1)				
Q19		1m, 143B				
Q20		7m, 2GB				
Q23						
Q1	# of VPs: 2		# of VPs: 10		# of VPs: 35	# of VPs: 200
Q2	14s, 16MB		60s, 240MB		3m, 779MB	23m, 8GB
Q3	15s, 18MB		1m, 241MB		3m, 778MB	25m, 8GB
Q4	15s (110x) 17MB (26x)		1m (12x) 241MB (9x)		3m (15x) 779MB (48x)	16m (148x) 8GB (15x)
Q5	15s, 17MB		1m, 241MB		3m, 779MB	24m, 8GB
Q6	15s, 26MB		1m, 241MB		3m, 792MB	23m, 8GB
Q7	15s, 20MB		60s, 240MB		3m, 780MB	23m, 8GB
Q10	24s, 8B		50s, 36B		2m, 128B	9m, 780B
Q11	15s (19x) 20MB (24x)		1m (38x) 241MB (7x)		3m (48x) 788MB (15x)	15m (138x) 8GB (≈)
Q12	14s (18x) 4B (1)		25s (17x) 536B (1)		1m (13x) 3KB (1)	4m (12x) 60KB (1)
Q13	13s, 4B		25s, 18B		1m, 64B	5m, 390B
Q14	13s, 4B		26s, 18B		1m, 64B	5m, 390B
Q15	12s, 4B		25s, 2KB		1m, 1KB	5m, 13KB
Q16	15s, 16MB		60s, 239MB		3m, 774MB	21m, 8GB
Q25	14s, 14MB		1m, 241MB		3m, 774MB	27m, 8GB
Q27	11s (19x) 2B (1)		4s (142x) 2B (1)		4s (178x) 2B (1)	3s (1153x) 1B (1)
Q28	11s, 47B		19s, 255B		1m, 754B	2m, 8KB
Q29	14s, 4B		31s, 18B		1m, 64B	7m, 390B
Q30	15s, 21MB		1m, 241MB		3m, 787MB	23m, 8GB
Q32	10s, 490B		23s, 275B		1m, 1KB	4m, 60KB
Q34	13s, 11B		29s, 49B		1m, 178B	6m, 1KB
Q38	12s, 4B		24s, 18B		1m, 64B	4m, 390B
Q39	10s, 4B		19s, 265B		1m, 620B	3m, 5KB
	Time interval	10 days	100 days	1 hour	Time interval	Time interval
	1 hour	1 day	10 days	1 hour	1 day	1 day
Q8	6s, 4B	5s, 4B	10s, 266B	11s, 18B	12s, 249B	17s, 2KB
Q9	9s (≈) 4B (1)	5s (163x) 4B (1)	8s (1376x) 4KB (1)	12s (12x) 253B (1)	12s (138x) 18B (1)	17s (1252x) 184KB (1)
Q21	51s, 120B	53s, 150B	1m, 170B	1m, 488B	1m, 590B	6m, 2KB
Q22	5s, 6MB	30s, 216MB	3m, 2GB	12s, 12MB	51s, 305MB	8m, 4GB
Q24	6s (12x) 4B (1)	5s (143x) 423B (1)	40s (1808x) 640B (1)	11s (14x) 18B (1)	14s (133x) 18B (1)	51s (85x) 138B (1)
Q26	6s, 4B	7s, 4B	19s, 4B	11s, 18B	13s, 18B	27s, 2KB
Q33	5s (14x) 13B (1)	5s (162x) 17B (1)	12s (1253x) 19B (1)	10s (12x) 57B (1)	12s (146x) 69B (1)	32s (1121x) 76B (1)
Q35	5s, 1MB	7s, 21MB	29s, 259MB	1m, 67MB	10s, 54B	12s, 350KB
Q36	5s (14x) 4B (1)	5s (153x) 541B (1)	9s (1352x) 4B (1)	11s, 18B	10s (12x) 18B (1)	10s (1397x) 18B (1)
Q37	5s, 5MB	16s, 115MB	2m, 2GB	7m, 5GB	11s, 393KB	12s, 11MB
					34s, 154MB	45s, 72MB
					2m, 1GB	13m, 9GB

Table 9: Time (in red) and data volume (in blue) required to answer the 39 questions listed in Table 7 with *ChatBGP*. We also show the data reduction factors and speed-up factors compared to processing MRT archives with *bgpstream* (Table 2).