Auditing the Ask Astro LLM Q&A app

Today, we present the second of our open-source AI security audits: a look at security issues we found in an open-source retrieval augmented generation (RAG) application that could lead to chatbot output poisoning, inaccurate document ingestion, and potential denial of service. This audit follows up on our previous work that identified 11 security vulnerabilities in YOLOv7, a popular computer vision framework.

Specifically, we found four issues in Ask Astro, a retrieval augmented generation (RAG) open-source chatbot application modeled after Venture Capital firm A16Z's reference architecture for RAG applications. RAG is one of the most effective techniques for enhancing a large language model (LLM) with information not contained in its training data set using a context knowledge base.

In this blog post, we review the RAG architecture as deployed in Ask Astro and then dive deeply into our technical findings, which can be classified along two high-level streams:

- Architectural issues: Lack of manual moderation or document deletion capability allows attackers to poison the chatbot's output with harmful information, echoing recent academic literature, most notably Carlini et al. (2023).
- Implementation faults: multiple implementation bugs could compromise the accuracy of document ingestion (Split-view poisoning through GitHub Issues, GraphQL injection in Weavite client) or threaten financial denial of service (prompt injection in question expansion prompt).

To conclude, we provide several best practices that can help RAG deployments avoid issues like these. If your project could use a similar checkup, please contact us.

Quick navigation links:

About Ask Astro

Ask Astro is an open-source chatbot that provides technical support for Astronomer, an orchestration tool for Apache Airflow workflows. It is fully automated and requires no administration or management after deployment.

There are two primary reasons why Ask Astro was a good candidate for this type of audit. First, the project is actively maintained and has a high-quality codebase and sophisticated design that demonstrates what developers can achieve using a modern ML development stack. Considerable effort has also been undertaken to create clear documentation and write automated tests.

Second, the project's primary purpose is as a community education tool. It is structured and documented as a RAG reference implementation and advertises its adherence to A16Z's reference architecture for RAG applications. Moreover, its implementation uses a representative sample of popular tools for constructing RAG applications:

- Weaviate, a vector database that stores document embeddings;
- Langchain, a Python-based framework for LLM programming; and
- Apache Airflow, a workflow orchestration system used in Ask Astro to manage document retrieval and processing.

Ask Astro will likely be a starting point for many new RAG developers. Thus, many other RAG applications will likely follow a similar design and encounter the same challenges as Ask Astro.

The application has a relatively narrow attack surface. It comprises the two main workflows diagrammed in Figure 1: document ingestion and generating responses to user questions.

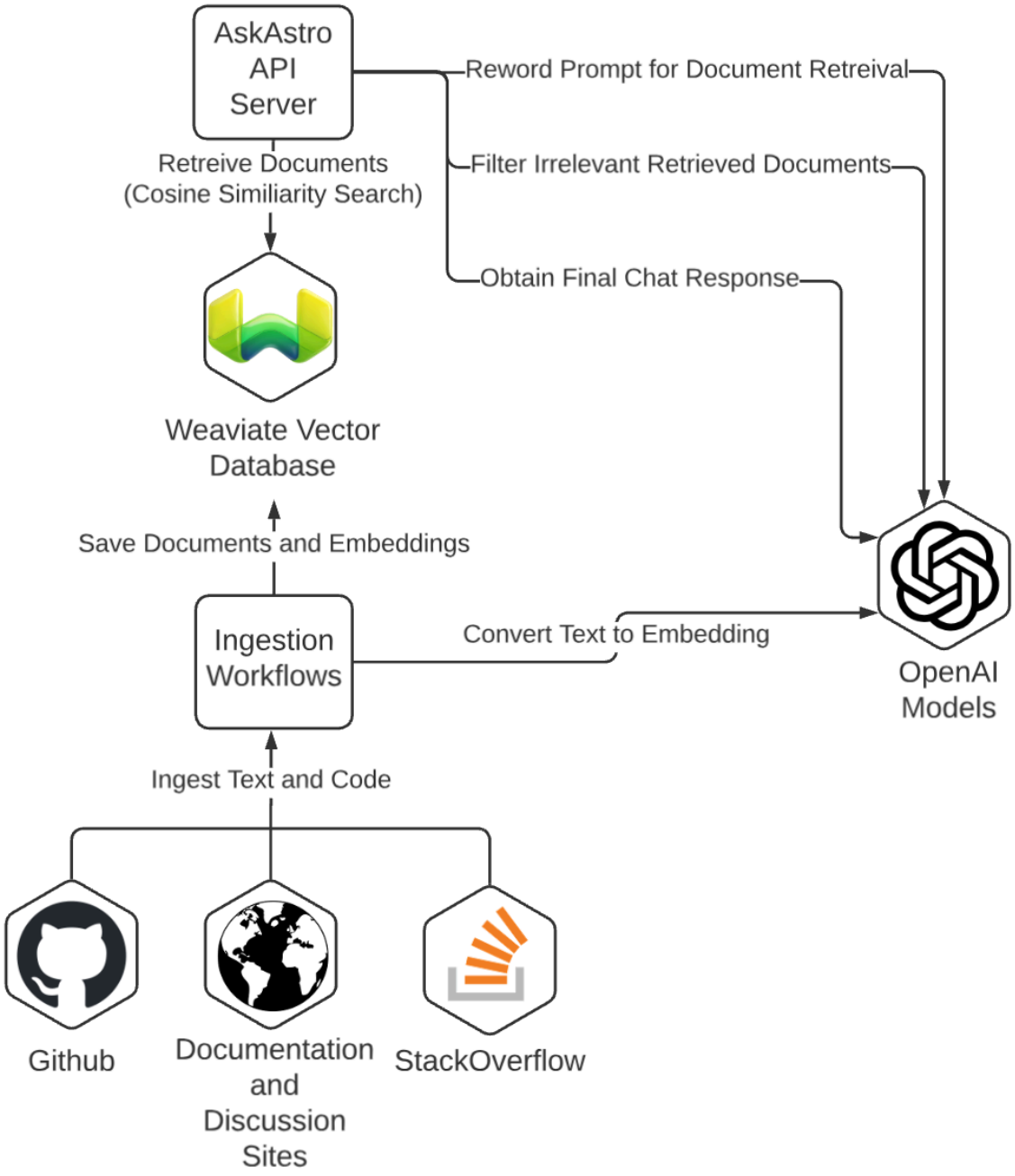

Figure 1: Ask Astro data flow diagram

Document ingestion

Ask Astro uses a series of Apache Airflow workflows triggered through Astronomer to ingest documents from the following sources:

- Official documentation for Apache Airflow, the Astronomer CLI, the Astronomer Cosmos, and the Astronomer SDK
- The official Astronomer blog
- Python source code for contributions to the Astronomer Registry, which contains user-submitted workflow components for Astronomer and Airflow
- Documentation in two GitHub repositories from the OpenLineage project
- GitHub issues for the Apache Airflow repository
- StackOverflow threads with the `airflow` tag

After downloading the source material over HTTPS, Ask Astro pushes it into Weaviate, an open-source vector database. During this step, Weaviate makes an API call to OpenAI to convert the document text into an embedding, which Weaviate saves locally.

Answer generation

When a user submits a question through the API, Ask Astro undertakes a multi-step process to retrieve relevant documents and generate an answer. This process begins by asking the LLM to generate two reworded versions of the original question to aid in retrieving relevant documents from the vector database. These questions are forwarded to Weaviate, which uses a cosine similarity search to retrieve the most relevant documents. Ask Astro then invokes the Cohere Reranker API, a well-known LLM provider, to rerank these documents according to their relevance to the user's original question. An LLM filter then removes documents the model evaluates as irrelevant to the user's question. Finally, the LLM generates a user-facing answer, with the final list of documents packaged into the question's context window.

The limitations of RAG in adversarial settings

RAG is a powerful way to make LLMs more knowledgeable and more responsive to the needs of a business and its customers. RAG systems also suffer the same well-known flaws as LLMs, such as prompt injection and hallucinations. Additionally, RAG systems depend on the reliability of inputs placed into the vector database. In most non-trivial applications, such as Ask Astro, the documents used to augment the LLM's knowledge base include untrusted documents. The ability to include untrusted documents is not an aberration but a desired feature: people want to do RAG over websites, comments, and user-supplied documents.

Due to fundamental undecidability results, it is impossible for an automated algorithm to flawlessly determine whether a forum post or GitHub comment contains misleading information or is otherwise malicious. Any sufficiently useful RAG system will inevitably index misleading or malicious content.

Academic research on poisoning attacks against hundreds of millions of image-text pairs shows this issue is significant. Many RAG applications use far smaller data sets as input to their vector databases, making poisoning attacks against vector databases economically viable.

Our audit of Ask Astro illustrates how these risks can manifest in practice. We show that attackers can manipulate the application's knowledge base in ways that parallel the two types of poisoning attacks described in Poisoning Web-Scale Training Datasets is Practical by Carlini et al., namely front-running and split-view poisoning:

- Split-view poisoning attacks exploit the mutability of data hosted on the Web by altering a resource in place after the curator or system designer has chosen to introduce it into the system's knowledge base.
- In contrast, front-running poisoning occurs when an attacker with knowledge of the data ingestion schedule posts malicious content just before an ingestion run, only to delete it immediately after ingestion completes.

Findings

[TOB-ASTRO-0001] Data poisoning through source material deletion

Severity: High
Impact: Vector database poisoning leads to inaccurate or malicious answers that are difficult to detect absent manual database review
Scenario: An attacker uses a set of sock puppet accounts to post a complete discussion thread on a community forum just before the system starts an ingestion run. After the ingestion run is complete, the attacker deletes the thread, hiding it from forum moderators. Without any consistent process for propagating source material deletion to the vector database, an attacker who knows the interval at which new documents are ingested can trivially inject arbitrary text into the knowledge base.

Discussion: The absence of any resource deletion check creates a ready-made opportunity for front-running. As implemented, Ask Astro has no safeguards to address inaccurate information in ingested resources and lacks facilities for deleting inaccurate or sensitive documents. The only exception is that Stack Overflow answers with a score of zero or lower are skipped. Community discussions, GitHub issue comments, and source code in the Astronomer Registry are treated as sources of truth no less authoritative than official documentation.

This finding is attributable mainly to Ask Astro's nature as a reference implementation. Understandably, a project of this type would not implement the data moderation processes most organizations need in a production setting.

[TOB-ASTRO-0002] Split-view poisoning through GitHub issues

Severity: Low
Impact: Vector database poisoning via publicly visible source material leads to inaccurate or malicious answers

Scenario: An attacker creates new GitHub issues in the AskAstro repository before document ingestion. When rendered as Markdown, these issues form a forged issues thread with authoritative authorship. The attacker can then insert inaccurate or malicious knowledge into the vector database and make it appear to originate from official sources.

Discussion: The document ingestion routines have two bugs in their processing of GitHub issues. These bugs enable two methods for conducting split-view poisoning attacks against the vector database. When the GitHub issue ingestion routine runs, issues and their comments are downloaded via the GitHub API and concatenated using a rudimentary Markdown template:

```
    issue_markdown_template = dedent(
        """
        ## ISSUE TITLE: {title}
        DATE: {date}
        BY: {user}
        STATE: {state}
        {body}
        {comments}"""
    )
//…
            downloaded_docs.append(
                {
                    "docLink": issue.html_url,
                    "sha": "",
                    "content": issue_markdown_template.format(
                        title=issue.title,
                        date=issue.created_at.strftime("%m-%d-%Y"),
                        user=issue.user.login,
                        state=issue.state,
                        body=issue.body,
                        comments="\n".join(comments),
                    ),
                    "docSource": f"{repo_base}/issues",
                }
            )
```

Figure 2: Concatenating issues via a Markdown template

The resulting documents are then stripped of boilerplate text using a series of regular expressions. Several of these regular expressions contain greedy `.*` sequences used with the `re.DOTALL` flag, which makes the dot character class match newlines:

```
issues_drop_text = [
        dedent(
            """                         <\\!--\r
                        .*Licensed to the Apache Software Foundation \\(ASF\\) under one.*under the
License\\.\r
                         -->"""
        ),
        "<!-- Please keep an empty line above the dashes. -->",
        "<!--\r\nThank you.*http://chris.beams.io/posts/git-commit/\r\n-->",
        r"\*\*\^ Add meaningful description above.*newsfragments\)\.",
    ]
```

```
//...
    df = pd.DataFrame(downloaded_docs)

    for _text in issues_drop_text:
        df["content"] = df["content"].apply(lambda x: re.sub(_text, "", x, flags=re.DOTALL))
```

Figure 3: Greedy regular expressions match more than they bargained for.

For example, the last regex in the `issues_drop_text` list will strip any text between the *first* occurrence of the substring `**^ Add meaningful description above` and the *last* occurrence of the substring `newsfragments).` Any time a comment thread contains this boilerplate text, each subsequent commenter can hide the entirety of the preceding thread by adding a new instance of the ending `newsfragments).` delimiter.

Post-processing of issue comments creates a second injection vulnerability that lets attackers fake entire issue threads. After being rendered using the Markdown template, each issue thread is saved in the vector database as a single string. The relevant documents are passed into the LLM's context during question answering via a LangChain "stuff" chain, which concatenates relevant documents. Since context is composed of unstructured text, there is no robust way to separate documents. Thus, attackers can mimic new issue threads by posting issue comments that mimic Ask Astro's issue Markdown template.

Note the power of this technique to bypass some of the mitigations developers might use to distinguish trustworthy data from untrustworthy data. When the document database includes conversations between different users, a straightforward heuristic for identifying the most authoritative statements is to look for an email address or username associated with the vendor that sells the software (Astronomer in the case of Ask Astro). This approach falls apart in the face of this comment forgery vector. If the attacker can forge entire threads of comments, they can forge the author information for each comment as well, defeating the often-recommended mitigation.

This issue has been reported to Astronomer in PR #325 to the ask-astro repository.

[TOB-ASTRO-0003] GraphQL injection in Weaviate client

Severity: Medium
Impact: Retrieval of non-public documents, but only if the Ask Astro vector database shares infrastructure with a non-public database
Scenario: Weaviate's GraphQL schema allows attackers to retrieve documents from two collections in one query. Consider an organization that hosts a public chatbot that draws on public documents, such as API reference material, and an internal chatbot that uses sensitive, private information. An attacker knows this and constructs a specially crafted query against the public-facing chatbot to leak sensitive documents only available internally.

Discussion: The Ask Astro API server uses version 3 of the Weaviate Python client library. All v3 releases of weaviate-client have a bug in the `_sanitize_str` function used to escape parameters to GraphQL queries. Unescaped quotation marks are prefaced with a backslash, and quotation marks that appear to be already escaped are left alone. The following regular expression implements this functionality:

```
value = re.sub(r'(?<!\\)"', '\\"', value)
```

The regex treats any quotation mark preceded by a backslash as adequately escaped. This logic mishandles cases where multiple consecutive backslashes precede a quotation mark. Input containing the substring `\\"` is not transformed because the look-behind assertion fails. In reality, the substring `\\"` is not an escaped quotation mark, but rather an escaped backslash followed by an *unescaped* quotation mark. Interpolating this value directly into a quoted string in a GraphQL query will terminate the string, causing the server to interpret what follows not as part of a string literal, but as query syntax.

Since many applications, including Ask Astro, pass untrusted user input into Weaviate filters, this bug creates a viable injection attack, albeit one with somewhat limited utility. Weaviate's GraphQL schema does not define any mutations—that is, a client can only read data, not write it—so an exploit could not alter the vector database. GraphQL allows clients to combine multiple operations in one request by concatenating them, much like stacked SQL queries, but this technique is not usable against the Weaviate client. The first GraphQL operation generated by the client is anonymous, meaning it does not specify a query name. The GraphQL server cannot combine an anonymous operation with other operations and will reject any GraphQL request containing an anonymous query and any second operation. However, Weaviate's GraphQL schema allows attackers to retrieve documents from two collections in one query, creating a potential data leakage vulnerability.

This finding has been reported as issue #954 in the weaviate-python-client repository on GitHub.

[TOB-ASTRO-0004] Prompt injection in question expansion prompt

Severity: Low
Impact: Excessive resource consumption or financial denial-of-service
Scenario: The first step in answering a question is for GPT-3.5 Turbo to provide two alternate phrasings of the same question. Using prompt injection techniques, the attacker can submit a question that causes the model to generate more than two questions or even reply with an arbitrary string. Attacker-influenced queries could cause the model to produce an inordinately large amount of output in the rewording step, contributing to a denial of service.

Discussion: Finally, we arrive at prompt injection, the most frequently discussed class of LLM bugs. Blocking undesirable classes of LLM output is undecidable and, therefore, unsolvable in

the general case (Glukhov et al. (2023)). Thus, defenses against prompt injection are fundamentally imperfect and prompt injections are bound to happen.

The impact is minimal in this case because the resulting reworded questions aid in retrieving documents from the vector database, not in the final request that answers the user's question. Unlike the previous bugs, this attack cannot be used to solicit false answers from the chatbot. Ask Astro uses the less-expensive GPT-3.5 Turbo for question rewording, reducing this issue's financial impact. However, if a single OpenAI API key grants permissions for both models, that key could still trip a global resource limit, thereby shutting down the entire account. Further, Astronomer.io informed us they use various rate limiting and anti-DDoS measures in production; We recommend similar measures for production deployments.

Going from RAGs to riches

The core challenge of any successful RAG deployment is ensuring the integrity of information introduced into the vector database. Ask Astro ingests data from multiple sources that an attacker could poison with false information. The lack of ongoing integrity verification processes makes it likely that poisoned data would remain in the database even if the original forum post or GitHub issue were deleted.

To address this challenge, we recommend the following best practices:

- Any **RAG application will need tools and processes to audit and maintain the vector database.** Proper audit and moderation tools will help **mitigate the data poisoning risk and aid debugging and evaluation**. Whenever a content moderator deletes an untrusted web source, an automated process should promptly delete it from the database. All updates to content sources, whether trusted or untrusted, should also propagate to the vector database.

- Simply synchronizing the vector database with the underlying live web resources is insufficient. A developer should not offload the responsibility for the vector database's accuracy onto forum moderators and other third parties, since those actors may not have the same goals and motivations as the RAG developers. Therefore, **humans must conduct an ongoing review of the vector database for inaccurate or irrelevant content**. The data review system should track actions taken by human moderators in the data set's provenance and lineage records.

- Ask Astro's GitHub issue processing bug demonstrates that a RAG system's data ingestion process is another potential source of bugs that could affect the quality of the system's output. **Each text parsing or data processing step should be carefully tested with inputs that include a mix of real-world data, edge cases, and simulated attack payloads.**

- Finally, the GraphQL injection bug in the Weaviate library illustrates one of the essential principles in application security: every interface between two system components

carries a set of potential attack vectors that must be understood and mitigated. Moreover, the analysis of these attack vectors must be context-specific. For example, recall that the impact of the GraphQL injection bug depends on what data is stored in the same Weaviate deployment as Ask Astro's vector database. Thus, **thorough threat modeling is an indispensable step for a machine learning application with as many moving parts as a RAG chatbot**.

Getting help

If your organization is designing or building a machine learning system that uses RAG or any other specialized methodology, our security engineers can help with threat modeling, design and infrastructure review, code review, fuzzing, and more. We specialize in the unique intersection of application security and machine learning to provide a holistic security evaluation of your applications. Contact us to see if we're a good fit for you.