# Infusing AI to Astronomy Store

Final Report

Juan Manuel Goyes Coral
*Information Systems Management, M.Sc.*
*Technische Universität Berlin*
Berlin, Germany
j.goyescoral@campus.tu-berlin.de

Seungmi Lee
*Information Systems Management, M.Sc.*
*Technische Universität Berlin*
Berlin, Germany
seungmi.lee@campus.tu-berlin.de

Johanes Albert Simohartono
*Computer Engineering, M.Sc.*
*Technische Universität Berlin*
Berlin, Germany
simohartono@campus.tu-berlin.de

Deniz Mert Tecimer
*Computer Science, M.Sc.*
*Technische Universität Berlin*
Berlin, Germany
tecimer@campus.tu-berlin.de

Kerem Yavuz Erkal
*Computer Science, M.Sc.*
*Technische Universität Berlin*
Berlin, Germany
erkal@campus.tu-berlin.de

***Abstract*—**
**This paper presents two AI-integrated approaches—AI-Driven Ticket Generation and Automated API Documentation—to enhance the resilience and efficiency of microservices within a Kubernetes environment. AI-Driven Ticket Generation automates error detection and resolution, significantly reducing downtime and improving system stability. Automated API Documentation leverages AI to analyze network traffic, generating comprehensive API documentation across different communication protocols. The integration of AI services improves system functionality but introduces challenges in cost management, testing, resilience, and ethical considerations. This research underscores the potential of AI to advance microservice architecture while highlighting the importance of careful implementation to ensure sustainability and reliability.**
**Appendix-A1**

***Index Terms*—AI, Cloud Native, Kubernetes**

## I. Introduction

Modern application development has been transformed by the adoption of cloud computing and microservices, enabling the creation of modular, scalable, and highly resilient systems. Microservices-based architecture, characterized by their loosely coupled services, offers improved availability, fault tolerance, and horizontal scalability.[1] However, as these architectures scale, managing the growing complexity becomes a significant challenge, necessitating advanced strategies for deployment, maintenance, and operational oversight. Therefore, to fully leverage the advantages of cloud computing and microservices, it is crucial to implement effective management and monitoring strategies.

One of the key aspects of effective management is promptly addressing errors and incidents that appear during operations. Traditional ticketing systems often rely on manual input, which can be time-consuming and prone to human error.[2] Additionally, the dynamic nature of microservices needs comprehensive management of application programming interfaces (API), including documentation, to ensure that developers have access to current and precise information. [3]

This report addresses these challenges by exploring innovative solutions that leverage generative Artificial Intelligence (AI), specifically Large Language Model (LLM). With the advent of AI, there is a growing recognition of its transformative potential across various industries, including e-commerce. AI offers unparalleled opportunities for enhancing user experience, automating complex processes, and driving data-driven decision-making. By integrating novel AI service into an existing microservices architecture, we aim to enhance application development beyond user experience.

We introduce two distinct approaches using API from OpenAI, a highly advanced and widely recognized LLM. Both approaches emphasize transforming invisible machine-level data into human-like outputs through AI-driven analysis. The first approach is AI-Driven Ticket Generation, which generates actionable tickets based on observed data. The second approach is Automated API Documentation, which involves capturing and analyzing all API traffics to create comprehensive documentation in OpenAPI specification. By integrating LLMs into the microservice application, we aim to improve monitoring capabilities and facilitate the generation of human-readable tickets and API documentation.

The remainder of this paper is organized as follows. II System Overview provides a brief overview of the system architecture used for refactoring. III Refactoring Approaches details two different approaches for integrating AI into the selected microservice architecture, including the research background, previous refactoring steps, refactoring procedures, and the evaluation and applicability of the refactored service. Finally, IV Conclusion presents the conclusions of this work.

## II. System Overview

The existing architecture selected is the OpenTelemetry Demo - Astronomy Store[4]. This is a set of microservices that provide online store functionality. All components of the

system are shown in Figure 1. It is composed of a web application, several backend services in multiple programming languages, and some repositories. Each service uses either gRPC or HTTP to communicate with one another. Additionally, the architecture is configured with observability tools such as Prometheus, Grafana, and Jaeger because the main intention of the project is to introduce developers to observability tools using OpenTelemetry. Besides that, it also offers a load generator script based on Locust. The system is ready to be executed in Kubernetes and each microservice generates a Docker image.

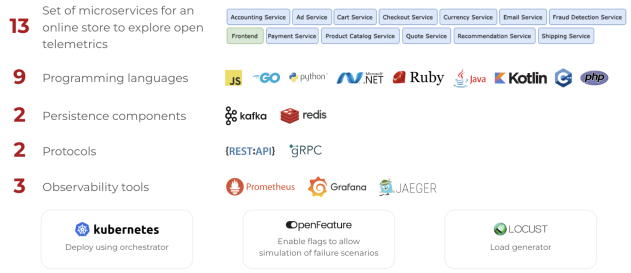


Fig. 1. Existing Architecture - OpenTelemetry Demo Astronomy Store

In addition to the observability tools, there are several feature flags that are useful to simulate some scenarios. They are managed by Flagd service. A flag can be enabled by modifying its value in a JSON configuration file. For example, we can introduce failures in the microservices by generating an error. The metrics and traces generated during the simulated failure contains useful data that for further analysis.

## III. Refactoring Approaches

### A. AI-Driven Ticket Generation

*1) Background:* In modern cloud and microservices environments, the increasing complexity and volume of operations have made error detection and resolution more challenging. Traditional manual processes for ticket generation, which involve human operators identifying and logging incidents, are often slow and prone to errors, leading to delays in detecting and resolving issues. As a result, there is a growing need for automation in incident management, particularly in generating support tickets that help address system failures and maintain reliability.

Recent advancements in Artificial Intelligence (AI) and Machine Learning (ML) have shown promise in automating various aspects of software operations, including ticket generation. By leveraging AI-driven methods, particularly with the help of large language models (LLMs) like OpenAI's ChatGPT, it is possible to automate the ticket generation process, making it faster, more accurate, and consistent.

The primary goal of this approach is to develop an AI-driven ticketing system that integrates with existing observability services. In our case, we are using the trace data from OpenTelemetry Collector. This system automates the creation of support tickets by analyzing trace data and logs to detect errors, summarize relevant information, and generate detailed incident reports. Automating this process through AI can significantly reduce the Mean Time to Detect (MTTD) and, consequently, the Mean Time to Resolve (MTTR), leading to better system availability and reduced downtime. The approach also aims to optimize data processing to handle large volumes of logs efficiently, addressing the limitations of current AI models related to token count and API costs.

*2) Preparation for Refactoring:* The OpenTelemetry Collector is utilized to receive, process, and export telemetry data. It enables the configuration of multiple receivers, processors, and exporters, which are combined to function within a pipeline. The pipelines are specified in the otelcol-config.yml file and ConfigMap named as opentelemetry-demo-otelcol in kubernetes/opentelemetry-demo.yaml file for the Kubernetes environment. Telemetry data is categorized into three types: traces, metrics, and logs, each of which can be managed through distinct and customizable pipelines. By default, the collector uses the OpenTelemetry Protocol (OTLP) receiver to receive telemetry data. The batch processor processes the data into batches, after which the data is exported to several endpoints, i.e. Jaeger, Prometheus, and OpenSearch by the default exporters. For the Kubernetes environment, they also use additional pipelines, namely k8sattributes, which adds Kubernetes-related metadata, memory_limiter which modulates the memory limit, and resource which adds the service pod id.

We modified the processor for traces data. Instead of using batch processing and in addition to the processors used in the Kubernetes environment, two processors are utilised. The first processor should wait up to 11 seconds to collect and group all spans that belong to the same trace before sending them to the next stage. 11 seconds are chosen deliberately as 10 seconds is the threshold for a timeout error in the frontend service, deduced from cypress tests of the service. Hence, waiting 11 seconds suffices to collect all spans of a trace. The traces are filtered based on the span's status code to drop spans without an error. We are interested in the details of the spans containing errors, therefore we also explicitly set the processor to propagate any error messages, instead of ignoring them.

*3) Refactoring Procedure:* A new service called "AI Ticketing" was created, which functions as a bridge between OpenTelemetry Collector [5] and ChatGPT. Here, we implement a JavaScript serverless application using Google Cloud Functions. It listens for HTTP requests containing trace data, processes it, analyzes the trace data containing errors using AI, and then creates an error ticket. The code handles errors gracefully and ensures that only valid requests are processed.

The application sets up an HTTP-triggered function to handle incoming POST requests. It expects the request body to contain trace data in JSON format. The trace data is preprocessed by serializing the JSON to a string format and trimming it.

An OpenAI object is initialized with a provided API key. We curated a prompt as seen in Appendix-C that defines the role and task of the AI model. It is designed to instruct the

model to act as a Support Engineer who analyzes trace data and generates a detailed support ticket. The prompt includes specific details the AI should include in its response, such as affected services, error summaries, potential causes, and recommendations. The output is restricted to have a length of a maximum of 1500 characters to match the rich text limit in Notion, which is 2000. This 500-character buffer is intentionally provided to minimize the likelihood that the response will exceed the limit, even if the AI occasionally generates slightly longer text. We send this prompt alongside the actual trace data to a chosen AI model (GPT-4o mini by default). The response is used to generate a ticket.

A ticket contains two items, i.e. the response from the AI model and the duration of the ticket generation. It is the time difference between when the error occurred and the ticket creation (now). To identify when the error occurred in the trace, we iterate over the spans within the trace data to find the earliest span end time.

For the ticket creation and publication, we use Notion API [6]. A new Notion client instance is created with authentication by passing the API key. The ticket is created in a Notion database as shown in Appendix-D. For every new ticket, a new page will be created in the database. Notion is chosen arbitrarily and can be replaced by other ticketing platforms such as Trello, Jira, etc.

The Terraform configuration file is used to automate the creation and configuration of the necessary Google Cloud resources, such as a cloud function, service account, and secret manager. Additionally, it injects the cloud function uri in other configuration files, such as ‘kubernetes/opentelemetry-demo.yaml‘, ‘src/otelcollector/otelcol-config.yml‘, ‘docker-compose.yml‘ from their respective ‘.tpl‘ files. Following is the configuration required to deploy and run the AI ticketing service:

- Packages the source code and uploads it to Google Cloud Storage. The cloud function uses it to handle HTTP requests.
- Use node.js runtime to run the JavaScript application.
- Set the maximum instance of the application, its memory allocation, and timeout.
- Manage the OpenAI API key securely using Google Secret Manager, making it accessible only to the cloud function.
- Manage the Notion API key securely using Google Secret Manager, making it accessible only to the cloud function.
- Manage the model type to be used from the OpenAI, making it accessible to the cloud function through environmental variables.
- Manage the Notion database ID to be used for accessing the Notion database, making it accessible to the cloud function through environmental variables.
- Set permissions to allow the cloud function to be invoked publicly, meaning the ticketing function can be triggered by external HTTP requests.

*4) Architectural Changes:* Our AI ticketing approach is a non-invasive method that runs asyncronically. It does not require architectural changes, however, it functions as a system extension.

- The ‘ConfigMap“opentelemetry-demo-otelcol‘ is configured to send traces to the cloud function. It is also configured to group spans by trace ID and filter out spans without any error.
- Our AI ticketing cloud function is added to ‘src/ticketingservice‘ and deployed via Terraform to Google Cloud Platform.

OpenAI API costs per request, therefore we applied small modifications in some services to overcome repeated requests such as the ones from the load generator and recommendation service, however, they do not affect the system architecture. The following modifications are done in respective services.

- Recommendation service located at ‘src/recommendation service‘ is updated so that when the productCatalog-Failure flag is enabled, it will not request that product anymore to prevent repeated ticket generation.
- Product Catalog service located at ‘src/productcatalog service‘ is updated so that it will not log that the flag is on. This would make the case very easy for the AI assistant and very far from a real use-case scenario.
- In the ‘opentelemetry-demo-loadgenerator‘ deployment, the environmental variable ‘LOCUST_AUTOSTART‘ is set to false to prevent other errors from happening to avoid additional costs in ticketing.

*5) Challenges:* In the OpenTelemetry Collector, individual spans can be filtered based on their attributes. However, filtering entire traces based on the attributes of their spans presents a challenge. We successfully implemented a solution to drop spans that do not have an error status code, which allowed us to retain only the error-related spans within a trace while discarding the others, even if they were part of the same trace. To address this limitation, a potential solution would involve developing a filter that operates at the trace level rather than the span level. This filter would assess traces based on their spans and then either retain or discard the entire trace accordingly.

When errors are introduced through the flagd service, it results in the repetitive generation of the same error due to retries, causing the Ticketing Service to create multiple identical error tickets. To address this issue, a preprocessing step for the trace data can be implemented in another service. This process would involve hashing specific attributes of the trace data, such as the error message, and storing the hash in a database. When a new trace is received, its hash can be compared to those already stored in the database to determine if a ticket for the same error has already been created.

To address the previous issue, we also attempted to introduce custom error flags within the flagd service. However, the technical complexities of the project posed significant challenges in this regard. The baseline project often references other packages which makes it harder to control the error flow and design. The system is not resilient enough to introduce custom errors and results in service failure which triggers

connection errors flooding the AI ticketing with unnecessary requests. Assuming successful integration of these custom flags, we could narrow the scope of errors to specific actions. This would ensure that errors only occur following specific actions generated by user interactions with the frontend, allowing for more targeted testing of the Ticketing Service. Due to these challenges, we could not apply custom errors initially declared in the baseline report.

*6) Evaluation:* Some examples of generated tickets are publicly accessible.[1]

To test the AI ticketing service, we utilized the feature flags in the OpenTelemetry Demo. Before we modified the Product Catalog service, the response from the GPT-4o mini was able to accurately identify the error, pinpoint its possible cause, and provide a recommended solution. Specifically, it correctly detected that a feature flag was enabled, leading to an internal error, and advised disabling the feature flag as a corrective measure as expected. Incorporating custom errors with specific causes, such as a "NullPointerException" or a type error, would enhance the realism of error tickets and resemble a real-world scenario. However, exceptions may be too big to fit in context, therefore logging or trimming is needed. After replacing the "flag enabled" log with the "product not found" log, AI could detect the object ID that caused the error. The generated tickets vary in terms of the generated tags even when generated due to repetition of the same error.

For effective debugging, it is crucial to have a comprehensive view of all spans included in the trace associated with an error. This makes it essential to include this information in the error ticket. However, a current limitation we face is the ability to filter only spans, not entire traces, as mentioned in the Challenges section. As a result, when spans without an error status code are excluded, some portions of the trace data are also lost. This truncation prevents us from having a complete view of all services involved in the trace.

Another limitation mentioned in the Challenges section is the creation of identical errors. OpenAI Assistant API can be leveraged to keep a history of the error tickets created and avoid creating the same ticket multiple times. Although this approach requires more effort, we believe it can enhance the quality of the ticketing service. However, due to the increased costs associated with this enhancement, we opted not to implement it in our project.

Despite this limitation, we argue that services within a trace that do not have any spans with an error status code are not directly "affected" by the error, as they continue to function correctly. The error state may only propagate through these services, potentially leading to further errors downstream. Identifying the root cause of the error, which can be effectively accomplished by capturing spans with an error status code, should generally suffice for most debugging purposes. However, to enhance the ticketing service, capturing a complete view of the trace and including all services involved would provide a more comprehensive understanding and could be beneficial for resolving complex issues.

The selected metric for evaluating the ticket service is the mean time to detect (MTTD) and the mean time to recover (MTTR) the error tickets. Since MTTR depends on the development team of the corresponding software project, we cannot measure it in our work and will focus only on MTTD. Utilizing the .js script we iterate through all generated tickets in Notion and calculate the mean of *duration of ticket generation* values. Currently, our MTTD equals 22,77 seconds.

*7) Feedback and Applicability:* The AI ticketing service demonstrated has broad applicability in cloud-native applications consisting of multiple micro-services, where effective error detection and resolution are critical. One requirement is that the microservices are instrumented to collect, process, and export telemetry data. Thanks to OpenTelemetry this can be achieved for a wide range of programming languages. By leveraging the feature flag mechanism within the OpenTelemetry Demo, the AI was able to accurately diagnose and provide actionable recommendations for resolving errors, highlighting its potential to be integrated into diverse software systems. Even though the test cases were not completely realistic because of the challenges posed by error creation, we interpret the results of the AI ticketing service to be promising.

In practice, this AI-driven approach to creating error tickets can significantly help streamline the debugging process, particularly in complex systems where multiple micro-services interact and tracing data is difficult to investigate manually. The ability to identify the root cause of an error through the analysis of spans with error status codes would help development teams save time in detecting error causes through manual debugging. Hence they can focus their time more on solving the errors instead of on detecting them. However, the current limitation of filtering only spans, not entire traces, suggests that while this method is effective in many cases, it may not always provide a full picture, especially in more intricate error scenarios. Further improvements to the AI ticketing service would also increase its applicability.

As a future work, the OpenAI's Assistant API with a vector store for tickets can be utilized with an adequate budget together with a self-managed vector store for traces. An application can map trace data to vector space allowing semantic comparison of traces. Traces exceeding a similarity threshold can be sent to the AI assistant which already knows existing tickets. It could modify the tickets or generate new ones with respect to their relatedness. The ticketing service (e.g. Jira, etc.) can trigger a "delete" action on the vector store when a ticket is resolved and deployed.

### B. Automated API Documentation

*1) Background:* gRPC stands for Google Remote Procedure Calls, developed by Google in 2015. It is a client-server communication model built on HTTP2. gRPC uses Protocol Buffers to describe service interfaces and facilitate communication between gRPC servers and clients. gRPC service

[1]https://www.notion.so/179f83708c30426d92afddaa157d3503?v=b729e3d04ceb47b58cec09f8d95a052a&pvs=4

interfaces are defined in proto files, specifying remote methods, method parameters, and return types as protocol buffer *messages*. Each message is a logical record of information containing a number of name-value pairs called fields.[7]

On the other hand, REST APIs use JSON over HTTP/1.1 and are typically documented using the OpenAPI Specification, a structured standard documentation that describes HTTP APIs with YAML or JSON. This standard enhances design, development, infrastructure configuration, developer experience, and testing, supporting efficient API management and communication.[8]

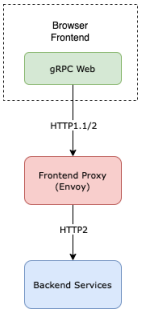


Fig. 2. gRPC Web Architecture with Frontend Proxy

When the frontend uses HTTP and the backend uses gRPC, direct communication requires the frontend to handle gRPC, which browsers do not natively support. This is because browsers typically use HTTP/1.1 or HTTP/2, while gRPC operates over HTTP/2 with binary Protocol Buffers. Figure 2 illustrates how gRPC Web acts as a bridge, allowing the browser to interact with gRPC services. The Envoy proxy between gRPC Web and the backend translates HTTP requests from the browser into gRPC calls for microservices.[9] This allows the frontend to use HTTP while the backend operates with gRPC.

The OpenTelemetry Demo has a similar structure, as shown in figure 3. The application's microservices communicate using gRPC and HTTP. To facilitate communication between the browser frontend and those backend services using gRPC, gRPC Web transports HTTP requests from the frontend to the proxy, which then forwards them to the appropriate backend services.

In this approach, API calls will be documented by capturing traffic and sending it to OpenAI's LLM. Since gRPC uses Protocol Buffers, which serialize data into a compact binary format, HTTP traffic will be captured from gRPC Web before being translated into gRPC. Consequently, captured API traffic will be documented into the OpenAPI Specification by AI. This documentation maps gRPC calls to HTTP requests and responses, leveraging OpenAPI's robust ecosystem for consistent and efficient API management.[8] This allows developers to benefit from a wide range of tools available for API development and testing.

*2) Preparation for Refactoring:* The refactoring focuses on generating automated API documentation using LLM based on traffic observation. Baseline API documentation from the

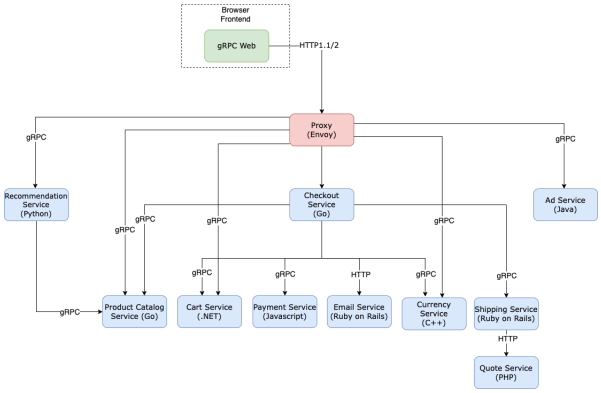


Fig. 3. System Architecture Diagram for Astronomy Store

source code is essential before starting the refactoring. In `demo.proto` file, gRPC services are defined using Protocol Buffers. Based on this proto definition, gRPC uses the protocol buffer compiler `protoc` to generate code in different languages. To create OpenAPI documentation from gRPC service definitions, two methods are suggested: using gRPC Gateway and Manual Documenatation.

*a) Using gRPC Gateway:* The gRPC Gateway creates a reverse proxy server that translates RESTful HTTP APIs to gRPC.[10] This tool can generate an OpenAPI specification from the gRPC service described in the proto file. First, the proto file is updated with HTTP annotations for each service to specify HTTP methods and paths. Google APIs repository which contains `annotations.proto` must be referenced to work correctly with these HTTP annotations in proto file. After updating the proto file and cloning GoogleAPIs, running the `protoc` command will generate Go bindings and a gRPC Gateway stub. The gRPC Gateway uses these Go bindings to interpret gRPC service definitions and generate the necessary proxy code, which handles the conversion from gRPC to HTTP.[10] The OpenAPI generator of protoc then creates an OpenAPI specification in JSON format that represents the gRPC service as a RESTful API. This generated OpenAPI specification can be found in the file named `demo.swagger.json` located in the `docs` directory. It contains 20 APIs, including 5 for `FeatureFlagService`, which are not actual APIs and one for an HTTP service, the `EmailService`, which cannot be implemented by the proto file.

*b) Manual Documentation:* Manual documentation involves reviewing the proto files and the source code to understand both the gRPC service definitions and the remaining two HTTP services, `EmailService` and `QuoteService`. Even though it is more time-consuming, this method provides flexibility and allows integrating both gRPC and HTTP in one documentation that automated tools may not achieve. As a final baseline API documentation, there are 14 valid gRPC APIs written in the proto file and two more HTTP APIs from the source code, totaling 16 documented APIs. The comprehensive documentation is saved in the `open_api.yaml` file

located in the `docs` directory.

*3) Refactoring Procedure:* The Automated API Documentation approach aims to generate API documentation by analyzing network traffic from a Kubernetes cluster, leveraging a generative AI model. This process is structured into three steps:

**1. Traffic Interceptor**

Several approaches were considered for extracting network traffic:

- **OpenTelemetry Integration**. OpenTelemetry microservices provide span data via Jaeger but lack request and response payloads, which are critical for API documentation.
- **Service Mesh Integration**. Using a service mesh with Istio and Envoy can extract payloads, but this method is complex, resource-intensive, and limited to single endpoint invocations, so it was discarded.
- **eBPF Technology**. eBPF allows kernel-level traffic interception, but its configuration is complex, dependent on specific kernel versions, and lacks easy integration for complete payload extraction.
- **Kubeshark (Selected Approach)**. Kubeshark, a Kubernetes-specific traffic analyzer, was chosen for its simplicity, ease of use, and effective traffic visualization. It supports eBPF and packet capture with KFL filtering, though the community edition is limited to two nodes.

**2. Refactor OpenTelemetry Demo**

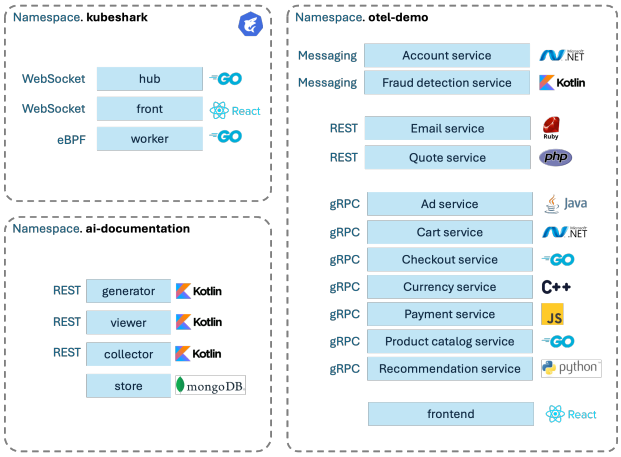


Fig. 4. API Documentation namespaces and microservices

The OpenTelemetry Demo project primarily exposes its microservices using various communication protocols:

- **gRPC protocol:** Eight microservices, including the ad service, cart service, currency service, checkout service, payment service, product catalog service, recommendation service, and shipping service.
- **REST API:** Two microservices—email service and quote service.
- **Messaging system:** Two microservices—account service and fraud detection service.
- Additionally, there is one frontend service.

All services are deployed within a Kubernetes cluster under the same namespace, ”otel-demo.” This configuration is illustrated in Figure 4.

To enforce the restriction of a maximum of two nodes in the cluster, the decision was made to retain only the target project for the API, excluding the entire OpenTelemetry stack (Prometheus, Grafana, Jaeger, Open Search, and OpenFeature with feature flagd) from the Kustomize version.

Additionally, the Terraform configuration for the cloud providers was modified to limit the maximum number of nodes and to bypass the default rule of one node per availability zone. For Google Cloud Platform (GCP), larger machines such as the e2-standard-4 (with 4 vCPUs and 16 GB of memory) were used, while for AWS, the t3.large instance (with 2 vCPUs and 8 GB of memory) was selected.

**3. Traffic Collection**

As shown in Figure 5, Kubeshark collects traffic from two nodes of the Kubernetes cluster. The KFL expression is configured to filter traffic specifically to the otel-demo namespace and restrict it to HTTP REST APIs. Once traffic capture is enabled in Kubeshark, the collector microservice establishes a WebSocket connection and sends the KFL expression as a message.

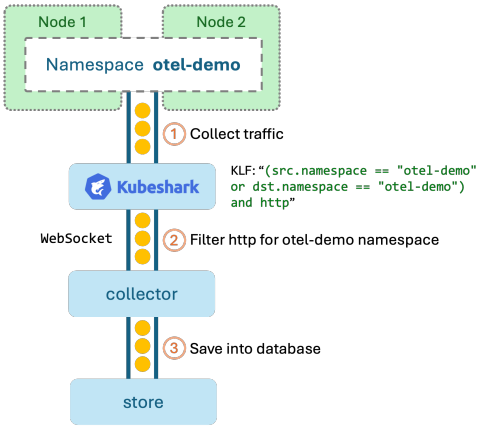


Fig. 5. Traffic collection from otel-demo to API documentation store

Kubeshark then begins streaming messages through the WebSocket to the collector. For each complete message, the collector filters out unnecessary details, such as Kubernetes metadata and status information, focusing only on the metadata necessary for generating API documentation—specifically, the request, response, method, and path. This data is then stored in MongoDB within the ”traffic” collection.

**4. Generation: From Traffic to Open API Specification**

Once the traffic is stored in MongoDB, the API documentation generation process, powered by Generative AI (via OpenAI supported models gpt-4o, gpt-4o-mini, gpt-3.5-turbo), can begin. Due to billing constraints, live API generation was not implemented. Instead, the user must manually trigger the API generation process.

Before starting the generation, the user needs to configure the **OpenAI API key** within the generator process.

This is achieved by invoking a REST endpoint (**POST /api/config/openai-key**), as the generator microservice uses lazy initialization. The first invocation must include the API key setup. If a subsequent request tries to set the key again, it will trigger an error, and the only way to recover is by restarting the service.

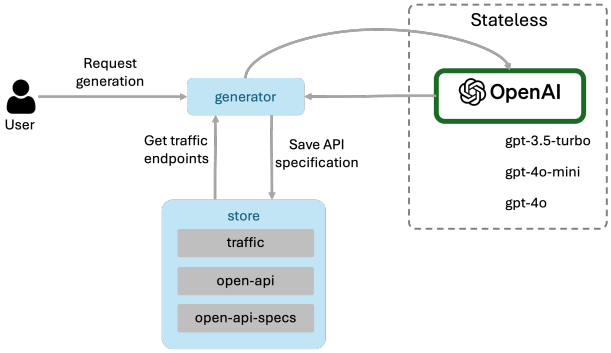


Fig. 6. Manual API generation requested by user

To start the generation process, the user calls the endpoint (**POST /api/documentation/full**), which triggers multiple calls to the OpenAI API, as illustrated in Figure 6. The process begins by extracting unique URLs from the traffic data, excluding query parameters. These URLs are then mapped to "Paths Objects" in the Open API Specification. The AI's objective is to identify "Path Templating" cases, such as mapping /api/cars/1234 and /api/cars/33333 to the templated path /api/cars/id.

For the Open API Specification document, JSON format was chosen. As shown in Figure 7, the specification allows references to other files. This means there is a main container or skeleton that includes all endpoints (paths and HTTP methods), with references to separate files. The process involves generating the main container with the identified endpoints and creating an additional generation for each endpoint to specify the "Operation" section.

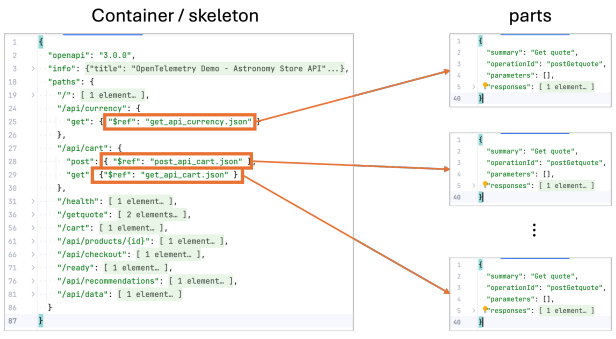


Fig. 7. Open API specification skeleton and referenced parts

The JSON schema for the Open API Specification is provided as part of the AI prompt, but there are instances where the generated output is not valid JSON. In these cases, an additional AI invocation is required to correct the format to ensure it is valid.

## 5. Visualization

After each manual full generation, the collections "open-api-specs" (containing individual parts and the skeleton per version) and "open-api" (which stores the parent document with the global version) are updated. The project viewer uses Swagger UI, which fetches the latest version of the documentation from the endpoint **GET /version/version/api-docs**, where **version** can be either "latest" or a specific version number (e.g., 1, 2, 3, ...).

*4) Architectural Changes:* The initial proposal focused solely on analyzing traffic from the frontend to the other services. However, Kubeshark enables us to target additional namespaces and capture all communications, which can be filtered using KFL expressions.

Additionally, although the common practice is to express an OpenAPI Specification file in YAML, the format was changed to JSON. This change allows us to leverage the capabilities of document databases, while still maintaining compatibility with the OpenAPI Specification.

Lastly, a significant change involved the approach to live documentation. While it was technically feasible to generate documentation live for each incoming traffic request, the costs associated with using the OpenAI models made this option impractical.

*5) Challenges:* One challenge encountered was that the Large Language Model (LLM) tended to focus more on content than on adhering to the specified format. Even when an expected format was explicitly stated, the outputs sometimes failed to comply. Long prompts, which produced lengthy outputs, often led to unexpected or invalid JSON files. To mitigate this issue, the prompts were simplified, and an additional step was introduced to fix the JSON format.

Another limitation of the current scope is that it processes a single traffic instance at a time. However, combining multiple traffic results could significantly improve the quality of the generated documentation. This enhancement would enable the generation of multiple response codes (e.g., success and failure cases) and provide diverse values for individual properties.

*6) Evaluation:* A total of 14 versions of API documentation were created for system evaluation, each adjusted by varying the prompt and language model. Initially, versions 1 to 4 used OpenAPI Specification 3.1.0, but from version 5 onwards, 3.0.0 was adopted to utilize the reference function. Versions 7 and 8, which contained only invalid endpoints, were excluded from the analysis. The evaluation focuses on versions 5 through 14, comparing them against a set of 16 standard endpoints defined by manual documentation.

Table I provides an overview of the endpoints included in each version, out of a total of 16 standard endpoints defined by the manual documentation. Because the generated API documentation captures real traffic, it can reveal discrepancies between expected and actual API behavior. The variation in endpoints across versions is due to differences in the observed traffic. Certain endpoints, like "/products/search," exist in the source code but seem to be rarely used or possibly unusable in practice because no traffic was observed for them.

| Service | Baseline API Endpoints | V5 | V6 | V9 | V10 | V11 | V12 | V13 | V14 |
|---|---|---|---|---|---|---|---|---|---|
| CartService | POST /cart/addItem | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| | GET /cart/{userId} | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | DELETE /cart/{userId}/empty | | | | | | | | |
| Recommendation Service | GET /recommendations/{userId} | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| ProductCatalog Service | GET /products/list | | ✓ | ✓ | | | | | |
| | POST /products/search | | | | | | | | |
| | GET /products/{id} | ✓ | ✓ | | | | | | ✓ |
| ShippingService | POST /shipping/getQuote | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | POST /shipping/shipOrder | | | | | | | | |
| CurrencyService | POST /currency/convert | ✓ | | | | | | | |
| | GET /currency/supported | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| PaymentService | POST /payment/charge | | | | | | | | |
| EmailService | POST /email/sendOrderConfirmation | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| CheckoutService | POST /checkout/placeOrder | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| AdService | POST /ads/get | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| QuoteService | POST /quote/calculateQuote | | ✓ | | ✓ | | | | |
| Valid API Endpoints | | 9 | 10 | 6 | 5 | 8 | 8 | 7 | 8 |
| Invalid Endpoints | | 0 | 0 | 6 | 5 | 2 | 2 | 2 | 1 |
| None API Endpoint | | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 2 |
| Legacy API Endpoint | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| Total Detected Endpoints | | 12 | 14 | 14 | 13 | 13 | 13 | 12 | 12 |

TABLE I
DOCUMENTED API ENDPOINTS ACROSS VERSIONS

Several key observations can be made regarding this AI-generated API documentation. Firstly, there are differences in the notation of some mappings compared to the manual documentation. For instance, the "/api/data" endpoint in the automated documentation likely corresponds to the "/api/ads" endpoint in the manual documentation. This discrepancy may arise from differences in how the API was monitored or named during traffic analysis versus manual documentation. Additionally, similar endpoints such as "/cart" and "/api/cart" are documented. While both serve the purpose of retrieving cart information, the "/cart" endpoint may be a legacy endpoint, possibly used in older systems or internally for specific use cases, as it returns an unusual API response.

The documentation also includes endpoints captured from actual traffic that are not typically considered API endpoints, reflecting a broader coverage of the system. For example, a health check endpoint for monitoring the health and availability of the service is included. However, the documentation process shows some errors caused by the large language model used. Some endpoints have an invalid format, incorrect reference file names, or lack descriptions. The number of such endpoints varies significantly across versions, largely influenced by the prompt used, which in turn affects the overall quality of the API documentation.

The selected metric is "Documentation Coverage," which measures the proportion of existing endpoints that are documented. Documentation coverage is expressed as the ratio of documented endpoints to total exposed endpoints. It is calculated using the following formula:

$$\text{documentation coverage} = \frac{\text{documented endpoints}}{\text{total endpoints}} \quad (1)$$

Based on the baseline API documentation from III-B2 Preparation for Refactoring, the number of endpoints explicitly exposed through static analysis of the source code is 16. Before the refactoring, 14 gRPC APIs are defined in the proto file. Therefore, the baseline documentation coverage is $\frac{14}{16} = 0.875$.

Second, through a dynamic analysis of the network traffic of the running system. This will help identify implicit endpoints, such as health checks and monitoring provided by third-party libraries. Nevertheless, the network traffic analysis will only detect active endpoints and relies on the coverage of the load generator tool, making it possible to miss some existing but unused endpoints.

According to Table 1, the documented endpoints ranged from 12 to 14. For instance, in version 5, there are 12 documented endpoints out of the 16 expected endpoints, resulting in a documentation coverage of $\frac{12}{16} = 0.75$. Similarly, versions 6 and 9 each cover 14 endpoints, achieving a documentation coverage of $\frac{14}{16} = 0.875$, which is the highest observed coverage among the analyzed versions.

The expected coverage rate was set at 100%, meaning all 16 standard endpoints should ideally be documented. However, in practice, the highest observed coverage across the evaluated versions was 87.5%, which matches the baseline coverage but falls short of full coverage. This is because the documentation is generated based on real traffic captured during the evaluation, rather than a static analysis of the source code. For example, certain endpoints may exist in the source code but are rarely or never triggered during the period of traffic capture, making them difficult to document automatically. This variability makes achieving 100% coverage difficult because it depends on how much and what type of traffic is observed.

*7) Feedback and Applicability:* While the implementation of the automatic API documentation generation system has provided valuable insights, the service currently operates under certain limitations, including the use of only two nodes due to the cost constraints. Additionally, it focuses on capturing and documenting HTTP JSON format traffic, as gRPC calls are binary and serialized. For gRPC services, it uses gRPC Web to capture HTTP requests before they are translated into gRPC. This allows the LLM to produce accurate and readable documentation for the entire API traffic.

Several areas for future improvement have been identified. One of the key issues is handling large traffic volumes, which can delay the process due to token limitations. Developing a filtering method to handle large traffic volumes or optimizing the data transfer process could mitigate this problem. Additionally, another important issue is to combine and reuse captured traffic data efficiently to continuously update the documentation. Implementing a delta analysis between API versions could identify changes effectively, facilitating targeted updates and enhancing documentation efficiency. Further refinement of the system involves critical decisions such as choosing the appropriate GPT model (e.g., GPT-4, GPT-4o, or a custom LLM) to improve the quality and reliability of the generated documentation. Moreover, the quality of the generated documentation varies with the prompts provided to the LLM. Therefore, ongoing refinement of standardized

templates based on feedback and output quality is needed.

The new service can be applied to other Kubernetes microservice applications to capture API traffic in existing services and generate OpenAPI documentation. Automating the documentation process with an LLM significantly speeds up the generation of API documentation, saving time and resources. Ultimately, this service is expected to enhance both the maintainability and usability of applications. By providing integrated API documentation that encompasses both HTTP and gRPC services in OpenAPI specification, it can provide developers API usage guidelines and detailed descriptions of the API. Furthermore, the service analyzes real API traffic, dynamically adapting the generated documentation to changes in API usage. This provides valuable insights into which APIs are actively used and helps developers in making informed decisions about service modifications and enhancements.

## IV. Conclusion

This research explored two AI-integrated approaches to enhance microservices within a Kubernetes environment. These approaches address challenges in error handling and API documentation, improving system resilience and efficiency. Integrating AI services, such as those from OpenAI, significantly enhances the quality and functionality of microservice applications. AI-Driven Ticket Generation automates error detection and resolution, reducing downtime and improving stability. Automated API Documentation uses AI to analyze network traffic, generating comprehensive documentation that bridges different communication protocols, ensuring consistent API management. However, these enhancements come with costs, particularly in API usage fees and infrastructure demands. Effective cost management can be achieved through selective AI invocation and resource optimization.

Testing and resilience are critical when deploying autonomous services. AI-driven processes must be rigorously tested to ensure reliability, and resilience measures, such as fallback mechanisms, are necessary to mitigate potential AI failures. In Automated API Documentation, outputs must be validated against standards to ensure accuracy and reliability.

Ethical and sustainability considerations are also crucial. Ensuring transparency in AI decisions and optimizing resource use are key to maintaining trust and reducing environmental impact. Privacy must be safeguarded, particularly when handling sensitive data in API documentation.

In conclusion, integrating AI through AI-Driven Ticket Generation and Automated API Documentation offers significant improvements in microservice resilience and efficiency. However, this integration requires careful attention to costs, testing, resilience, and ethical considerations. By managing these factors, organizations can leverage AI to build more robust and sustainable systems, aligning with modern software development demands.

Appendix-A2

## References


[1] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.

[2] R. Qamili, S. Shabani, and J. Schneider, "An intelligent framework for issue ticketing system based on machine learning," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)*, 2018, pp. 79–86.

[3] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," *Journal of Systems and Software*, p. 112110, 2024.

[4] O. Authors. (2024) Opentelemetry demo documentation. Accessed on: Jun. 21, 2024. [Online]. Available: https://opentelemetry.io/docs/demo/

[5] ——. (2024) Opentelemetry collector. Accessed on: Aug. 9, 2024. [Online]. Available: https://opentelemetry.io/docs/collector/

[6] N. A. Authors. (2024) Notion api. Accessed on: Aug. 14, 2024. [Online]. Available: https://developers.notion.com/

[7] Google, "Introduction to grpc," 2021. [Online]. Available: https://grpc.io/docs/what-is-grpc/

[8] OpenAPI, "The openapi specification," Dec 2023. [Online]. Available: https://www.openapis.org/what-is-openapi

[9] Google, "grpc on web," 2021. [Online]. Available: https://grpc.io/docs/platforms/web/

[10] I. Gengo, "grpc-gateway," 2015. [Online]. Available: https://github.com/grpc-ecosystem/grpc-gateway.git

## APPENDIX

### A. Generative AI - Creation Log

In this section, we provide a detailed log of how generative AI tools were utilized in the creation of this paper. The specific parts of the text generated or assisted by AI are marked in blue within the main body of the document. The following are the prompts used to obtain these AI-generated responses:

*1) Generation for Abstract:*

- **Prompt:** "You are a researcher of a novel AI application for cloud native applications. You wrote this paper about infusing AI to a set of microservices from the OpenTelemetry Demo Project called Astronomy Store. You have made two different approaches to refactor the Astronomy Store. The first one is AI-Driven Ticket Generation, using AI to automatically generate support ticket. The second one is Automated API Documentation, documenting API based on the network traffic. Based on the following excerpt of the paper, please make an abstract of this paper just like a normal scientific paper. It must be short and concise."

*2) Generation for Conclusion:*

- **Prompt:** "You are a researcher of a novel AI application for cloud native applications. You wrote this paper about infusing AI to a set of microservices from the OpenTelemetry Demo Project called Astronomy Store. You have made two different approaches to refactor the Astronomy Store. The first one is AI-Driven Ticket Generation, using AI to automatically generate support ticket. The second one is Automated API Documentation, documenting API based on the network traffic. Based on the following excerpt of the paper, please make a conclusion for the first approach (AI-Driven Ticket Generation) in about 700 words. Please highlight more on the part of challenges, evaluation, and feedback and applicability. Here is the excerpt of the AI-Driven Ticket Generation approach."
- **Prompt:** "Now I give you the excerpt of the second approach (Automated API Documentation). Based on the following excerpt of the paper, please make a conclusion for the second approach (Automated API Documentation) in about 700 words. Please highlight more on the part of challenges, evaluation, and feedback and applicability. Here is the excerpt of the Automated API Documentation approach."
- **Prompt:** "Based on both approaches and your responses before, please answer the following questions
  1. Which types of AI services can you integrate and how does it change the qualities and cost of your system?
  2. How do you implement testing and resilience when using such autonomous services?
  3. How can you ensure and attest the ethical and sustainability implication of this new integration?
  Please combine the answer of these question and all your previous answer of the conclusion of all approaches to about 600 words. Think of this as the conclusion part of your research paper."
- **Prompt:** "please make it without title for each section, and make reading each section flows better to the next section, making it one conclusion, as we saw normally in a research paper"
- **Prompt:** "Now please make it shorter, about 300 words while maintaining the important points. For the first approach, please don't say Autonomous Error Handling, instead say AI-Driven Ticket Generation"

### B. GitLab Repository

A group space "**cnae_ss_2024**" was created in GitLab and is located https://git.tu-berlin.de/cnae_ss_2024/. This group contains the projects associated with the exploration and approaches. Currently, there are four projects: **Opentelemetry Demo**, **gitlab-profile**, **Api Documentation** and**AI Ticketing**.

The REPRODUCTION.md file is located here https://git.tu-berlin.de/cnae_ss_2024/gitlab-profile/-/blob/main/REPRODUCTION.md.

### C. ChatGPT Prompt

The following prompt is used for ticket generation through Chat GPT:

**Prompt:** You are a Support Engineer at a software company that provides cloud solutions. Do not exceed 1500 characters! Use your knowledge base to analyse the given trace data from OpenTelemetry Collector and generate a support ticket containing the following information:

- Affected services
- A summary of logs for services with error
- Possible cause of error
- Possible solution recommendation
- Additional notice if it is just due to a service temporarily unavailable
- 3-5 tags to categorize the issue

### D. Error Ticket in Notion

Figure 8 displays the generated error ticket as a page in Notion. Further processing of the text to match the '.md' format is left outside of the scope of this work to meet time restrictions.

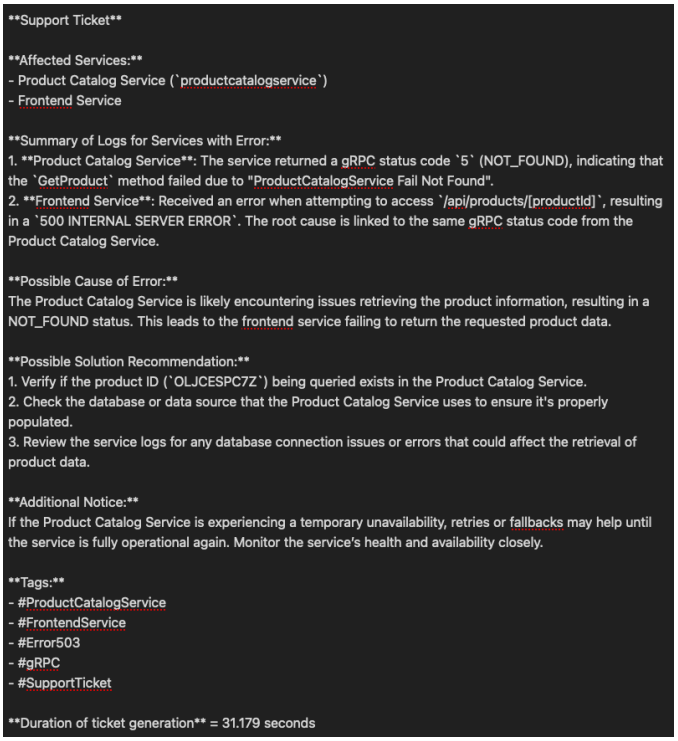


Fig. 8. Error ticket created in Notion as a page

## *E. Report Contributions*

| Responsible | Section |
|---|---|
| Juan Manuel Goyes Coral | III-B3 Refactoring Procedure |
| Juan Manuel Goyes Coral | III-B4 Architectural Changes |
| Juan Manuel Goyes Coral | III-B5 Challenges |
| Juan Manuel Goyes Coral | II System Overview |
| Seungmi Lee | I Introduction |
| Seungmi Lee | III-B1 Background |
| Seungmi Lee | III-B6 Evaluation |
| Seungmi Lee | III-B7 Feedback and Applicability |
| Johanes Albert Simohartono | Abstract |
| Johanes Albert Simohartono | III-A1 Background |
| Johanes Albert Simohartono | III-A2 Preparation for Refactoring |
| Johanes Albert Simohartono | III-A3 Refactoring Procedure |
| Johanes Albert Simohartono | IV Conclusion |
| Deniz Mert Tecimer | III-A3 Refactoring Procedure, |
| Deniz Mert Tecimer | III-A4 Architectural Changes, |
| Deniz Mert Tecimer | III-A5 Challenges |
| Deniz Mert Tecimer | III-A6 Evaluation |
| Deniz Mert Tecimer | III-A7 Feedback and Applicability |
| Kerem Yavuz Erkal | III-A3 Refactoring Procedure |
| Kerem Yavuz Erkal | III-A5 Challenges |
| Kerem Yavuz Erkal | III-A6 Evaluation |
| Kerem Yavuz Erkal | III-A7 Feedback and Applicability |