SpecPipe: Democratizing Access to Spectrum Data

Project 85

Authors: Hao-Ming Hsu, Omair Alam, Will Almy, Alice Lee

Contents

Executive Summary

In the digital landscape, the radio spectrum serves as the backbone for a plethora of technologies, including WiFi and 5G. However, the potential for individual exploration and scientific analysis in this domain is hampered by the lack of accessible, scalable, and customizable systems for users to access, analyze and contribute to spectrum data. Recognizing these challenges and opportunities, this paper introduces SpecPipe: an open-source modern, scalable AI/ML-facilitating data pipeline that empowers users to engage with radio data using minimal hardware and configuration requirements. Through a detailed comparison with related works, we demonstrate SpecPipe's advancements in accessibility, extensibility, and scalability over previous efforts in this field. We then elaborate on the architecture of SpecPipe that meets the aforementioned goals and enables users to implement their applications via Software Development Toolkits (SDKs) aimed at black-boxing the core system. We outline tangible results including building the SpecPipe framework, SpecPipe Python SDKs, Grafana visualizations for load monitoring, and an extensive documentation website. Finally, we propose directions for future work to expand the capabilities of SpecPipe and foster a wider community of contributors and users.

1. Introduction

Radio spectrum powers everything around us, from WiFi, 5G, GPS, and airplane navigation data to the common FM radio found in cars. Having a system that can empower engineers, scientists, and hobbyists to access, analyze, and contribute to this data through their own radios is of paramount importance to scientific growth and individual exploration. The link to our GitHub repositories can be found at: https://github.com/ml4wireless/specpipe, https://github.com/ml4wireless/specpipe-sdk-py, and https://github.com/ml4wireless/specpipe-demo.

A. Problems and Opportunities

Creating a system that can allow users to meet the aforementioned goals is a complex endeavor due to the diversity of signal types in the spectrum as well as the complications in dealing with data of this magnitude.

Radio waves operate within a subset of the electromagnetic wavelengths, typically occupying frequencies between 3 kHz to 300 GHz. Various signal types subdivide across many non-overlapping bands within this range; for example, FM tends to operate between 88 – 108 MHz. These signals not only vary in their signature and the bands they use, but also in the geographic locations they are emitted from. Additionally, they require different algorithms to process and decode them. Besides the complication in signal processing, the sheer volume of spectrum data being produced every second poses a data engineering challenge, up to multiple Gigabytes per minute. Users accessing both raw and processed spectrum data require near real-time updates from receivers across the world, so the system needs to have high bandwidth, low latency, and must guarantee data integrity.

B. Proposed Solution: SpecPipe

To resolve these issues, we have developed SpecPipe: a modern, scalable AI/ML-facilitating data pipeline for spectrum. Specifically, we have focused on the goals of usability, customizability as well as ease of machine learning integrations and monitoring. We have accomplished these goals by building SpecPipe as an open-source project free for people to access and use, with easy-to-follow documentation, and a plethora of startup examples that allow users to understand our framework interactively. This platform's core values of accessibility, extensibility, and scalability ensure that individual users can start to work with radio data with inexpensive hardware, minimal configuration, and easy onboarding docs.

2. Related Works

Several research endeavors have taken place to enable public access to spectrum data. Zheleva et al. developed an end-to-end system, Spectrum Observatory [10], to measure and characterize spectrum data. Building on top of Spectrum Observatory, Roy, S. designed CityScape [12], which is a metro-scale observatory that measures long-duration IQ data. Iyer et al. presented SpecNet [11] which enables collection and measurement of real-time spectrum. The radio frequency community has also developed an open-sourced web tool, IQEngine [13], for analyzing, processing, and sharing radio frequency data.

In the following sections, we perform a literature review with the most related works, including the previous generation of SpecPipe [9], Electrosense [1], Electrosense+ [2]. The overall comparison of the works is shown in Table 1.

A. Previous Generation of SpecPipe

Jiang et al. proposed the first version of SpecPipe, which is an end-to-end data pipeline for spectrum data. They utilized the pipeline to develop a showcase website for interacting with the airplane tracker application. This work builds upon several existing approaches in the areas of data pipeline design and spectrum sensing. The client program receives and processes the raw ADS-B data from airplanes, followed by the annotator module to enrich the data with additional information. With the annotated information, Elasticsearch [7], a distributed search engine, not only stores but also searches and analyzes the data. Finally, the backend web server fetches annotated ADS-B data from Elasticsearch and serves it to the front end. Besides the system and the showcase website, they provided documentation for deploying the system and a monitoring system health dashboard with Prometheus [8].

B. Electrosense

Rajendran et al. proposed a centralized system architecture, Electrosense, which facilitated the accessibility of radio data. Their goal was to create a reliable and efficient environment for the public to utilize spectrum data while addressing potential security and privacy concerns. They accomplished this by crowdsourcing and utilizing low-cost sensors based on software-defined radios (SDRs) [3] and Raspberry Pis, measuring spectrum data in the range of 20

MHz to 6 GHz. The main infrastructure that controls data flow is a Message Queue Telemetry Transport (MQTT). The distributed system backend is composed of an ingestion layer, a speed layer, and a serving layer. The speed layer uses Apache Kafka [6], a message queuing system, as a buffer for the incoming data, preventing data loss. There are two different data pipelines following the ingestion layer which are the speed and serving layers. The batch layer employs the Hadoop Distributed File System (HDFS) [4] and Apache Spark [5] to perform nearly real-time parallel processing. Finally, the serving layer provides an open API for users to easily access the processed data.

By combining all the components, they successfully monitored spectrum data using low-cost sensors with a centralized backend system.

C. Electrosense +

Building on top of Electrosense, Calvo-Palomino et al. developed Electrosense+, which also allows real-time spectrum data decoding. Besides the general decoding purpose, they also added a peer-to-peer communication feature to the architecture. With this direct pipeline between users and sensors, they increased the throughput for scalable data decoding. In addition to the peer-to-peer channel, they provided users incentives by developing a token reward system, where users earn tokens whenever their sensors are online and being used. Security is another important topic mentioned in their work. Since users do not receive raw spectrum data, they added a privacy layer on top of the decoding process to prevent malicious data leakage.

D. Limitations of Previous Works

The aforementioned works addressed the issue of giving public access to spectrum data, still, there are some limitations of the mentioned solutions. For example, the documentation and guidelines provided by the authors are limited, such that they still leave a high barrier of entry for the users. Moreover, the architectures as designed are unable to be customized. In other words, the data format that the system is receiving is not able to be modified in real time. Another limitation of Electrosense and Electrosense+ is that the system design lacks a health check component. Therefore, potential issues or failures within their systems may be overlooked, leading to increased downtime. Although the previous generation of SpecPipe has a health check component, the provided information on the dashboard is still limited. For example, it does not show the geolocation, data rate, and registration time of each device.

Due to these intricacies, the prior generation of SpecPipe, Electrosense, and Electrosense+ lack a myriad of key features including usability and flexibility that have prevented their mass adoption by end users. Created to fulfill specific needs as mentioned above, they lack extensibility and observability. Parsing through the user manuals for these systems is time-consuming and tedious, making development difficult and error-prone. These limitations prevent a general adoption of said technologies to less technical or resourceful audiences, and therefore, limit the ability of users to be able to use and contribute to spectrum data.

	Electrosense [1]	Electrosense+ [2]	First generation of SpecPipe [9]	SpecPipe (This work)
Message broker	MQTT	MQTT	NATS	NATS
Peer-to-peer communication	No	Yes	No	Yes
System health monitoring	No	No	Grafana Dashboard	Grafana Dashboard
Extensibility	Low	Low	Low	High
Barrier to entry	High	High	Medium	Low
Real-time data support	Yes	Yes	No	Yes

Table 1. Comparison of the works

3. Design Goals

In this section, we focus on mainly three aspects of the system, accessibility, extensibility, and scalability.

A. Accessibility

SpecPipe is designed for simple, rapid deployment on any platform. It provides containerized solutions using Docker that allow for one-click deployment on various infrastructure environments and platforms. Comprehensive documentation and hands-on examples guide users through a wide range of use cases ranging from basic to advanced. The documentation covers topics such as connecting data sources, configuring pipelines, inferencing with ML models, and generating visual results. This enables users across skill levels to quickly get up and running with their desired spectrum analysis tasks.

B. Extensibility

SpecPipe is designed as a highly modular and flexible framework so users can easily develop custom plugins with any technology stack. It is deeply integrated with the open-source software ecosystem and provides out-of-the-box support for leading open source technologies. For example, it uses NATS JetStream for scalable streaming data ingestion. Apache Spark integrates natively for distributed big data processing and analytics. gRPC powers high-performance communication between components. SpecPipe also offers deep integration with popular machine learning frameworks like TensorFlow and PyTorch for custom machine learning (ML) model development and deployment.

The modular architecture makes it simple to extend SpecPipe by developing connections to new data sources, adding additional processing nodes, or implementing custom ML algorithms. Developers can leverage their preferred languages and frameworks to extend functionality through language-agnostic APIs. SpecPipe handles the underlying orchestration and deployment so users can focus on writing their own custom logic to suit their specific needs. The extensive open-source integrations also allow users to tap into rich ecosystems of analytics, data science, statistics, and ML tools.

C. Scalability

SpecPipe is optimized for scalability from the start. It can easily scale to thousands of sensor nodes with minimal effort as a result of its distributed architecture. Horizontal scaling is supported out of the box by adding new pipeline instances, while auto-scaling capabilities dynamically allocate resources based on load. SpecPipe maintains a small memory footprint even at a massive scale to minimize computing and energy costs. One way it achieves this is by leveraging Apache Spark for distributed processing, meaning petabyte-scale datasets can be analyzed across clusters while individual nodes operate efficiently.

SpecPipe also offers fine-grained tuning of scalability by allowing different pipeline stages to scale independently. Additional ingestion capacity can be added to handle high data volumes from sensors, while the distributed processing layer can scale analysis and ML workloads independently of the ingestion rates. SpecPipe automatically handles the routing of data across the scale-out architecture. This flexible scaling unlocks new spectrum analysis scenarios involving vast numbers of IoT sensors or complex analytics.

4. Architecture

SpecPipe is built to transport, store, and organize large amounts of data. The architecture supports many ML and AI workloads related to Spectrum Data, but does not implement these ML/AI workloads itself. End users do not modify the code of SpecPipe directly in most cases, but rather implement their own applications using SpecPipe's flexible APIs.

Term	**Description**
NATS (Neural Autonomic Transport System)	NATS is the key pillar that SpecPipe was built upon, handling communication, data, and configuration requests across nodes. NATS is a lightweight and high-performance messaging system designed for distributed systems, offering simplicity, reliability, and scalability for cloud-native applications.
Edge Node	An edge node is a device (such as a laptop or a Raspberry Pi) that has a software-defined radio attached to it via USB and is connected to the system. The

Term	Description
	radio is listening at a particular frequency that is initially set when the edge device registers with the system but can be changed dynamically later.
Applications	Applications are software that run on devices connected to the system. An application can receive spectrum from an edge node, get metadata of edge nodes (such as sampling rate, geolocation), and update the configuration of an edge node.
Health Check Server	The Health Check Server is an example of an Application that can run on devices connected to the system. The purpose of the health check system is to check the health of the nodes.
Controller API	A controller API serves as the interface for managing and orchestrating resources within SpecPipe, enabling Applications to programmatically interact with and manipulate the configuration and behavior of the Edge Nodes. SpecPipe currently supports 3 APIs for both FM devices and IQ devices, including read device configurations, update devices, and get all devices. The Open API is listed in Appendix A.

Table 2. Key Terms for Architecture

The SpecPipe architecture can be understood using the key terms of the architecture (Table 2), the architecture diagram (Figure 1) and the information paths supported (Table 3).

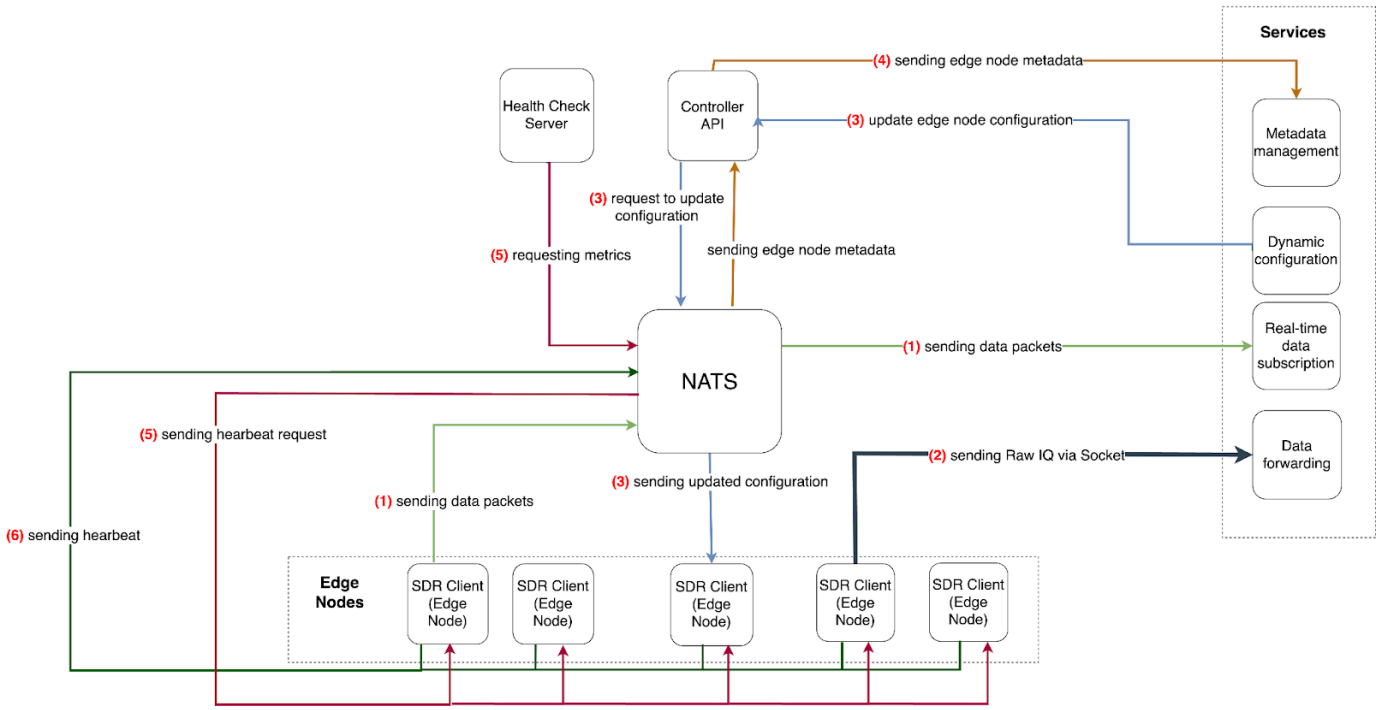
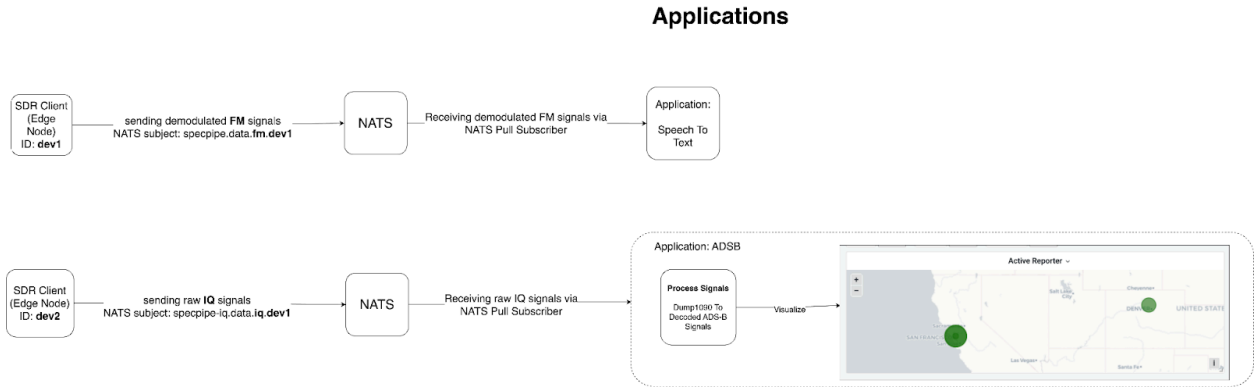


Figure 1. SpecPipe Architecture

Information path	**Explanation**
Data Flow (Line '1' in Figure 1)	Raw IQ and Demodulated radio data is sent from Edge Nodes via NATS to Applications.
Peer to Peer Flow (Line '2' in Figure 1)	Raw IQ radio data is sent from Edge Nodes to Applications via a socket. This is useful for sending high bit rate (> 5 mbps) Raw IQ Data since if this data would be sent via NATS, the system would get bogged down.
Control Flow 1 (Line '3' in Figure 1)	Applications update the configuration of an edge node (such as changing its sampling rate or frequency) via the Controller API. The Controller API then publishes a message on the appropriate NATS subject to change that setting for an Edge Node.
Control Flow 2 (Line '4' in Figure 1)	Applications can get edge node metadata (such as their location and sampling rate) via the Controller API.
Monitoring Flow 2 (Line '5', '6' in Figure 1)	Applications can monitor the health of the Edge Nodes by running the health command. When this command is issued, NATS sends heartbeat requests to all the Edge Nodes (line 5). Then, the edge nodes respond with a heartbeat (line 6) to the server via NATS.

Table 3. SpecPipe Information Paths

Many examples have been created for using the SpecPipe framework (https://ml4wireless.github.io/specpipe/examples), two of which (FM and ADS-B) are shown below:



A. Cloud Architecture

To support running in a Cloud environment, the entire application stack has been containerized into various Docker images, bundled together with a Docker Compose manifest for easy deployment of key services. One such image contains the Controller service or the backend API server that supports interaction with SpecPipe and any number of edge nodes. Additional images are built to be run on the edge nodes themselves which set up a standardized interface for data collection across different host environments. A majority of SpecPipe's application code is written in Go, a highly performant compiled language, allowing for resource-efficient deployments to Cloud environments or hosts with low computing power. Lastly, NATS represents the key pillar of SpecPipe's functionality, handling communication, data transfer, and forwarding of configuration requests across nodes.

B. Edge Node Architecture

SpecPipe's scalable infrastructure allows any number of edge nodes to connect and begin collecting data, either for immediate consumption or recording to long-term storage. The edge container contains multiple utilities for handling data, including various common decoding methods such as FM and AM radio. Direct streaming of IQ data is also supported but may be bandwidth intensive. Additional functionality includes the ability of edge nodes to directly forward data in a peer-to-peer environment without needing to interface directly with the server; this supports the separation of concerns at the edge and can support customized decoding or data processing methods before data is sent over the network.

C. Software Development Kit

With a flexible system to support many workflows on both the client and the server, additional functionality includes an auto-generated API and client SDK for interacting with SpecPipe's various systems. These APIs help with configuration, data consumption, and monitoring from external applications without needing to modify any internal

SpecPipe source code. These APIs and SDKs can be used by a wide variety of programming languages and workflows, ensuring future compatibility with multiple products and technology stacks.

5. Deliverables

Publicly accessible artifacts for SpecPipe include three source code repositories. *specpipe* itself contains the components to run the backend infrastructure in the cloud and also includes customizable software for distribution to edge nodes. There are also six different independent examples shown under the *_example* directory of this repository. An additional repository, *specpipe-sdk-py*, contains SpecPipe's Python SDK which allows developers to interact with and develop tooling around SpecPipe with their preferred language of choice. The complete software bill of materials can be found in Appendix B

Documentation has been a key priority in SpecPipe's development, aligning with our project goals of creating an accessible solution to users from all backgrounds. The system architecture, as well as onboarding steps, are included both in our project README files as well as a standalone documentation site hosted within GitHub Pages, as shown in Figure 2 and Figure 3. The steps to get SpecPipe up and running are presented in a quick and minimal fashion while also providing additional resources for advanced users looking to customize and extend the system. The documentation website is developed using Docusaurus.

For many users, the quickest way to understand a software system is to see real world examples of how it can be used. Therefore, *specpipe-demo*, covering a variety of different use cases that SpecPipe can help with is presented. The website is developed using TypeScript using the framework React, specifically Next.js framework. We use TailwindCSS as the CSS framework so as to streamline and expedite the development process. The details of the demos are discussed in the following sections A to D.

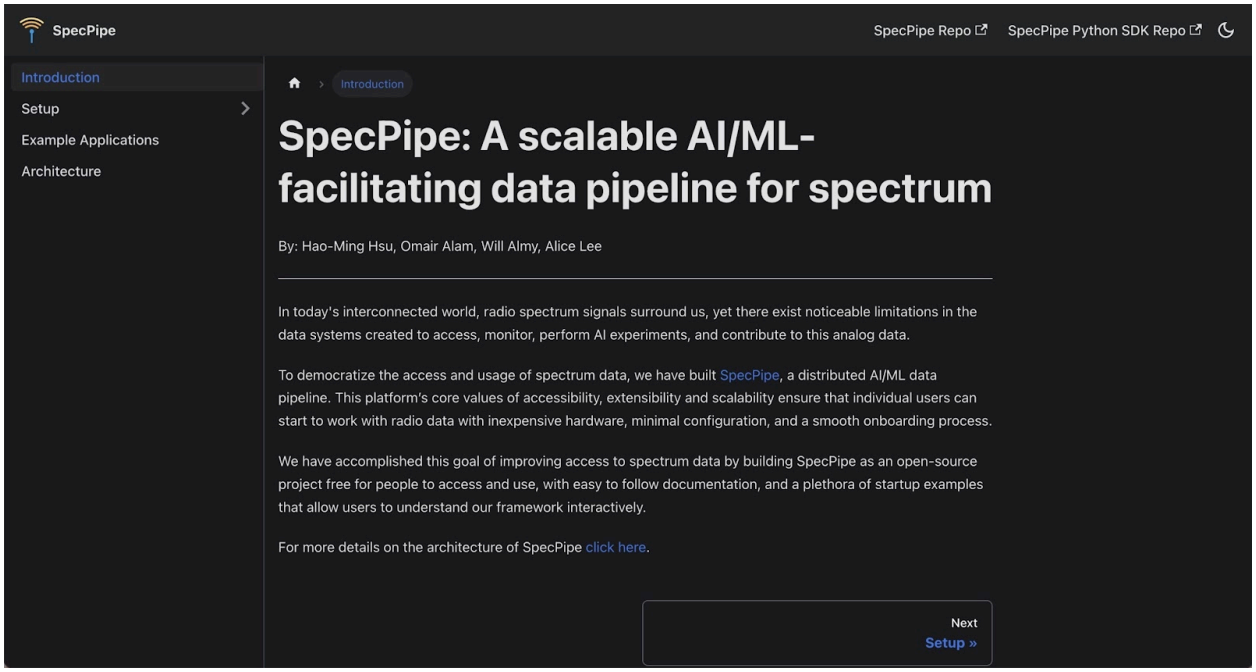


Figure 2. SpecPipe documentation website hosted on GitHub pages using Docusauraus framework.

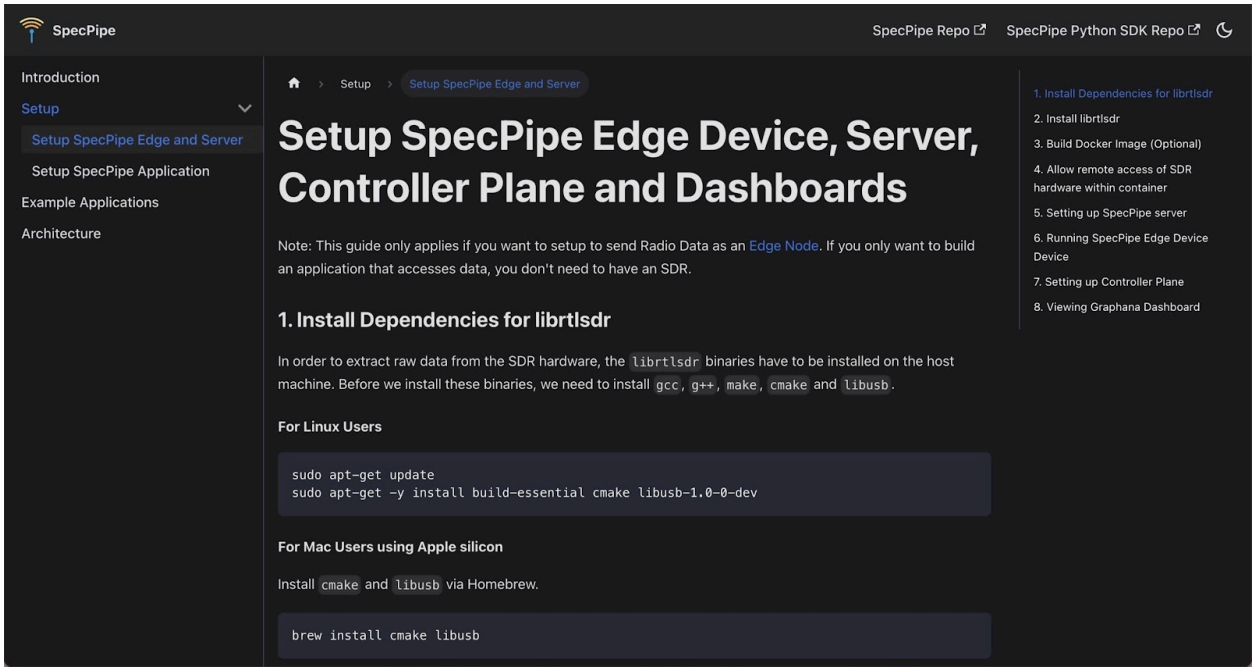


Figure 3. SpecPipe documentation website illustrating commands to setup SpecPipe.

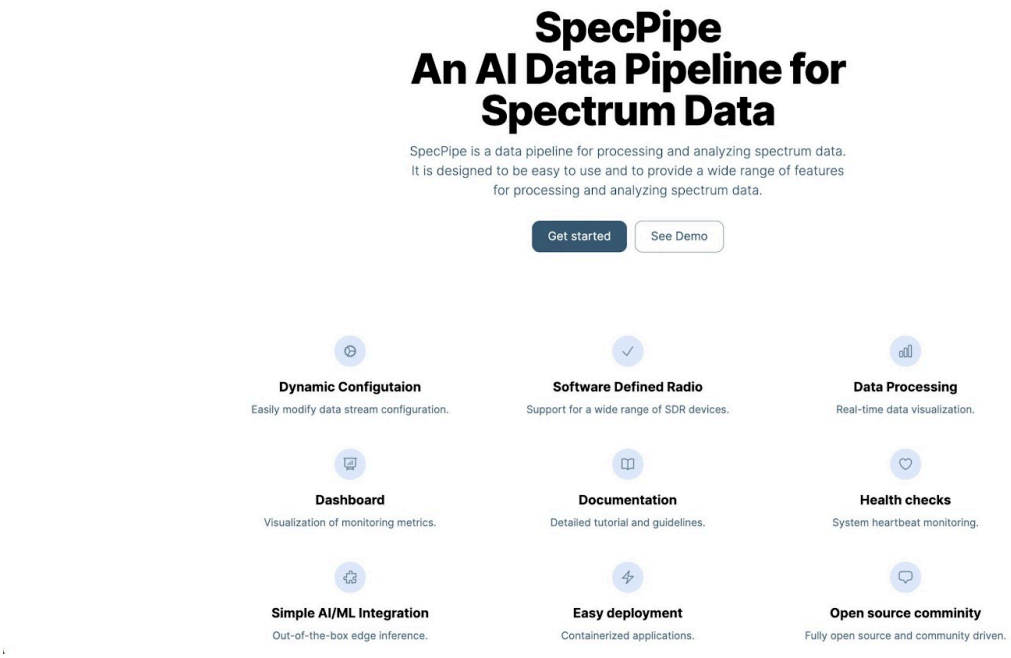


Figure 4. SpecPipe demo website landing page.

A. Monitoring Dashboard

In addition to the system itself, we have also developed tooling and workflows for observability in a production environment, as shown in Figure 5 and Figure 6. A Grafana dashboard template is included with the SpecPipe source code with various panels to monitor resource usage, data flow, and other key metrics around system health and performance. Metrics are exposed to Grafana through Prometheus, a time series database that is built with user-friendly extensibility in mind. After registering an edge device, its information will automatically appear on the Grafana dashboard, including charts of device geo-locations and online status, all ready to use in our system.

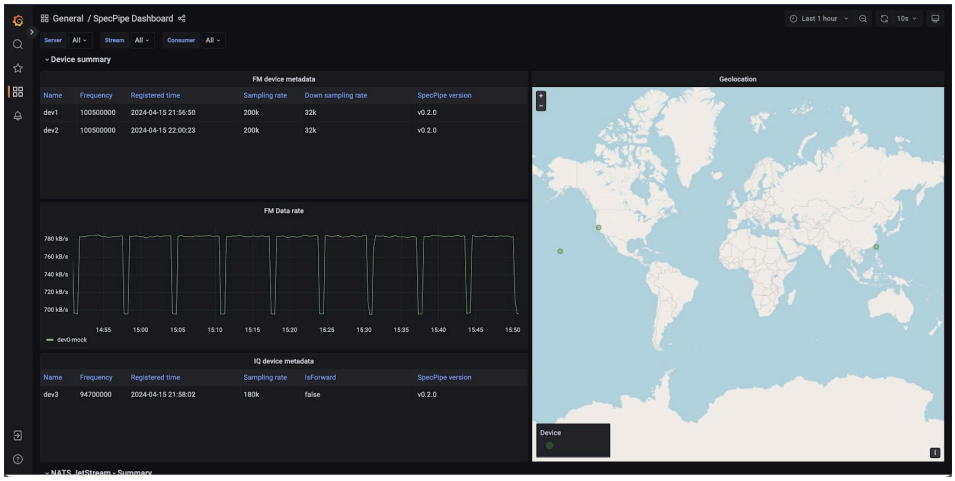


Figure 5. Grafana Dashboard of SpecPipe showing the geolocation, metadata, data rate of FM devices as well as the metadata of IQ devices.

Figure 6. Grafana Dashboard showing the system health status of NATS JetStream, including storage used and consumer metrics.

B. Speech to text

In the speech to text demo, a basic FastAPI web server is employed to retrieve FM data from NATS. This data is then decoded into .wav chunks, optimized for compatibility with the SpeechRecognition model. The resulting text is streamed to listeners via websocket communication. Additionally, a demonstration frontend application accompanies the setup, utilizing websocket reception to display live text in a scrollable text box. We streamed the audio from our mock server, *dev-0-mock*, to ensure that the audio contains words. As shown in Figure 7, the corresponding caption was automatically generated.

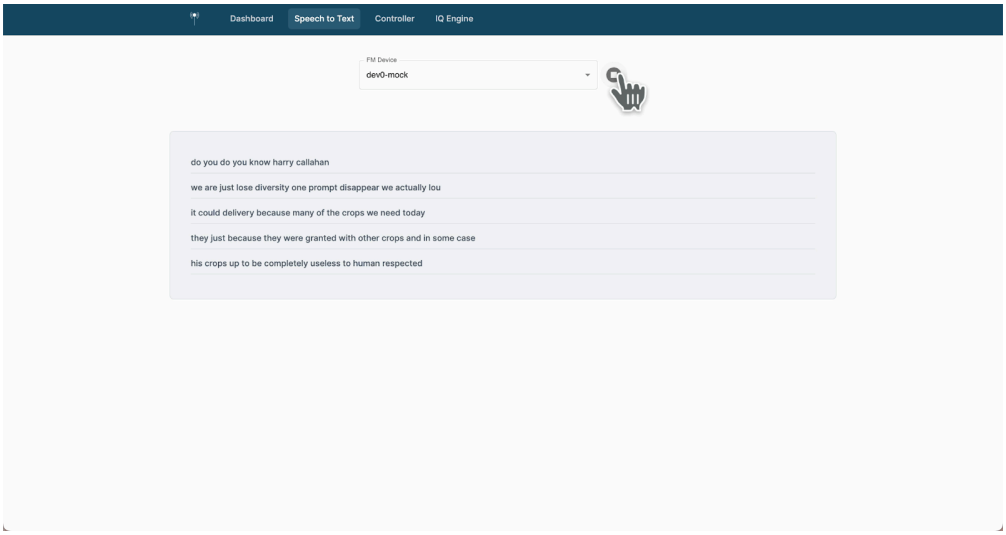


Figure 7. Speech to text demo of the website. The captions are generated using the speech to text model.

C. Controller plane

The controller plane comprises a device selector at the top, defaulting to the mock device, *dev-0-mock*. Clicking the volume button adjacent to it initiates audio playback. Below, three sliders adjust frequency, sample rate, and resample rate. To alter device configuration, adjust the sliders and click the modify button below.

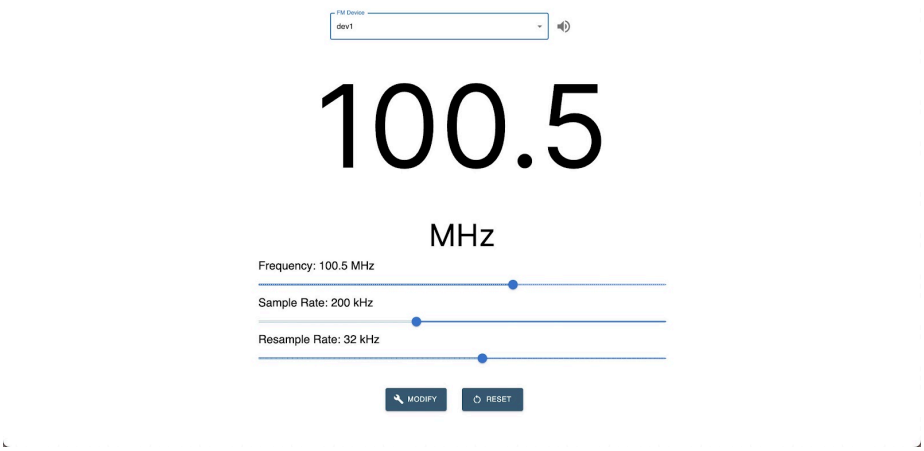


Figure 8. SpecPipe demo website that enables users to modify configuration of registered devices.

D. IQEngine integration

IQEngine is a web-based tool designed for analyzing radio frequency (RF) data, facilitating the processing as well as sharing of RF data and displaying spectrograms directly within web browsers. We have integrated the IQ Engine function into our demo website. Users can initiate analysis by clicking on *Local File Pair* on the left-hand side and uploading example data files, namely *iq_example.sigmf-data* and *iq_example.sigmf-meta*, located in the _examples/plot_iq folder of *specpipe* repository. Once the files are uploaded, the spectrogram is generated for further analysis and processing, as illustrated in Figure 9.

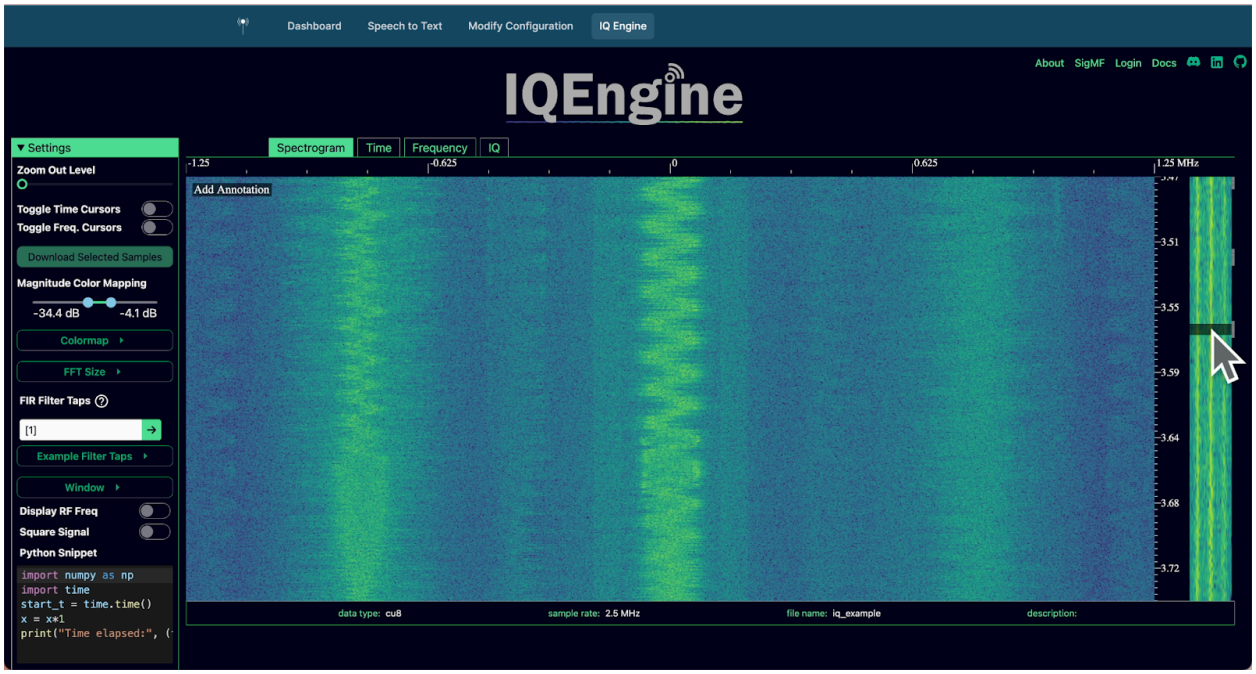


Figure 9. IQEngine is a web-based application to process, visualize, and analyze spectrum data.

6. Conclusion

In this section we will cover SpecPipe's accomplishments as well as future areas of improvement for this project.

A. Results

Holistically, SpecPipe has achieved many of our initial design goals which center on making radio spectrum data accessible for collection and analysis in many different environments. Accessibility has been a primary driver of multiple key pillars in our project, with our open-source repository, extensive documentation, and diverse set of examples allowing beginner level developers, students, class projects, and other organizations to easily stand up a comprehensive system for handling spectrum data. It takes 4 commands to start a SpecPipe edge device, 1 command to start a SpecPipe server, and 1 import statement in a Python script to access SpecPipe's Python SDK. In our initial user testing, we have seen users able to start up a new SpecPipe edge device within 2 minutes.

SpecPipe's work on extensibility and customization has also increased the number of users that can benefit from the development of the system, while a highly performance and resource efficient solution allows consumers without comprehensive infrastructure support to still utilize complex data pipelines.

B. Future Works

Despite all of the progress that has been made, there is additional room for improvements in future years. Examples of additional work areas may include additional support for third-party integrations, including direct streaming to open-source spectrum data platforms such as IQEngine. Additional SDKs for other languages than Python such as

C++, Rust, and Go Programming Language may also lower barriers to entry for other development ecosystems. These are features that we identified as valuable additions, however were not able to include due to time constraints.

Additionally, while the SpecPipe team has done load-testing, there hasn't been much usability testing for this project. Given that a key goal of this project is to democratize Spectrum data, it is critical to conduct rigorous usability studies to evaluate how SpecPipe meets the needs of diverse user groups. For these studies, the research question could focus on identifying the challenges users encounter during installation, extension, and use of SpecPipe, as well as determining the learnability of the system. The intended participants for this study would include hobbyists, academics, principal investigators, and small research teams who do not have a large engineering team at their disposal. Qualitative methods such as interviews and direct observations, along with quantitative methods like task completion time and error rates can be used to gather comprehensive insights into user experiences across these diverse groups. By understanding these challenges, we can tailor future enhancements to better serve our users' needs. While there is more to be accomplished in the future, this work lays the foundation for future contributions, innovations, and open source development

7. References

[1] Rajendran, Sreeraj, et al. "Electrosense: Open and big spectrum data." IEEE Communications Magazine 56.1 (2017): 210-217.

[2] Calvo-Palomino, Roberto, et al. "Electrosense+: Crowdsourcing radio spectrum decoding using IoT receivers." Computer Networks 174 (2020): 107231.

[3] Sadiku, Mathew NO, and Cajetan M. Akujuobi. "Software-defined radio: a brief overview." Ieee Potentials 23.4 (2004): 14-15.

[4] Shvachko, Konstantin, et al. "The hadoop distributed file system." 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). Ieee, 2010.

[5] Zaharia, Matei, et al. "Apache spark: a unified engine for big data processing." Communications of the ACM 59.11 (2016): 56-65.

[6] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." Proceedings of the NetDB. Vol. 11. No. 2011. 2011.

[7] Elasticsearch GitHub: elastic/elasticsearch-labs: Notebooks & Example Apps for Search & AI Applications with Elasticsearch (github.com)

[8] Rabenstein, B., & Volz, J. (2015). Prometheus: A Next-Generation Monitoring System (Talk). Dublin: USENIX Association.

[9] Z. Jiang, Victor Li et al. "SpecPipe: A scalable cloud-based AI/ML-facilitating data pipeline for spectrum" May 2023. https://ml4wireless.github.io/adsb-nats/assets/files/paper-3a780e70dcd5731c5c213f8b1a20a505.pdf

[10] Zheleva, Mariya Zhivkova, et al. "Enabling a nationwide radio frequency inventory using the spectrum observatory." IEEE Transactions on Mobile Computing 17.2 (2017): 362-375.

[11] Iyer, Anand, et al. "Specnet: Spectrum sensing sans frontieres." Proc. NSDI. 2011.

[12] Roy, S., et al. "Cityscape: A metro-area spectrum observatory." 2017 26th International Conference on Computer Communication and Networks (ICCCN). IEEE, 2017.

[13] IQEngine website. https://iqengine.org/

8. Appendix

A. API Specification

Method	Request URL	Sample Request Body	Response Schema (Status code 200)	Description

GET	/fm/devices/{devicename}	N/A	{
			"device": {
			"register_ts": 0,
			"specpipe_version": "string",
			"name": "string",
			"sample_rate": "string",
			"resample_rate": "string",
			"freq": "string",
			"longitude": 0,
			"latitude": 0
			}
			}
				Read FM device configuration

PUT	/fm/devices/{devicename}	{
		"freq": "string",
		"sample_rate": "string",
		"resample_rate": "string"
		}
			{
			"device": {
			"register_ts": 0,
			"specpipe_version": "string",
			"name": "string",
			"sample_rate": "string",
			"resample_rate": "string",
			"freq": "string",
			"longitude": 0,
			"latitude": 0
			}
			}
				Update FM device

GET	/fm/devices	N/A	{
			"devices": [
			{
			"register_ts": 0,
			"specpipe_version": "string",
			"name": "string",
			"sample_rate": "string",
			"resample_rate": "string",
			"freq": "string",
			"longitude": 0,
			"latitude": 0
			}
			]
				List FM devices

			} 	

GET	/iq/devices/{devicename}	N/A	{
			  "device": {
			  "register_ts": 0,
			  "specpipe_version": "string",
			  "name": "string",
			  "sample_rate": "string",
			  "freq": "string",
			  "longitude": 0,
			  "latitude": 0,
			  "forward": true
			  }
			}
				Read IQ device configuration

PUT	/iq/devices/{devicename}	{
		  "freq": "string",
		  "sample_rate": "string"
		}
			{
			  "device": {
			  "register_ts": 0,
			  "specpipe_version": "string",
			  "name": "string",
			  "sample_rate": "string",
			  "freq": "string",
			  "longitude": 0,
			  "latitude": 0,
			  "forward": true
			  }
			}
				Update IQ device

GET	/iq/devices	N/A	{
			  "devices": [
			  {
			  "register_ts": 0,
			  "specpipe_version": "string",
			  "name": "string",
			  "sample_rate": "string",
			  "freq": "string",
			  "longitude": 0,
			  "latitude": 0,
			  "forward": true
			  }
			  ]
			}
				List IQ devices

B. Software Bill of Materials

Versions enumerated in following appendix sections:

Software Material	**Description**
Go	Primary language for SpecPipe edge node and server controller software
Python	Primary language for SpecPipe auto-generated API utilities and example projects
Docker	Utility to containerize SpecPipe edge and server nodes across platforms
Grafana	Dashboard customization software to support SpecPipe's monitoring tools
Prometheus	Time-series database used to support reporting, storage, and monitoring of quantitative metrics
Docusaurus	Framework for deploying SpecPipe's documentation site
NATS	Messaging system for handling large-scale data applications
dump1090	Utility to process and decode aircraft ADS-B data
librtlsdr	Interface for interacting with software defined radios
FastAPI	Web server to support interfacing with examples and demo applications
SpeechRecognition	Python library for running local speech-to-text models
Next.js	React framework for server side rendering of demo applications
React	Library for generating HTML outputs based on client application needs
Tailwind CSS	Utility for stylization of React pages in our demo applications
TypeScript	Utility for type checking and static analysis of our frontend demo applications

C. Exact Software Requirements

SpecPipe Go Packages

Go 1.2.0
github.com/ThreeDotsLabs/watermill v1.2.0
github.com/ThreeDotsLabs/watermill-nats/v2 v2.0.2
github.com/getkin/kin-openapi v0.120.0
github.com/gin-contrib/cors v1.4.0
github.com/gin-gonic/gin v1.9.1
github.com/nats-io/nats.go v1.31.0
github.com/oapi-codegen/runtime v1.0.0
github.com/sirupsen/logrus v1.9.3
github.com/spf13/cobra v1.8.0
github.com/spf13/viper v1.17.0
google.golang.org/grpc v1.61.1
google.golang.org/protobuf v1.32.0
github.com/apapsch/go-jsonmerge/v2 v2.0.0 // indirect
github.com/bytedance/sonic v1.10.0-rc3 // indirect
github.com/chenzhuoyu/base64x v0.0.0-20230717121745-296ad89f973d // indirect
github.com/chenzhuoyu/iasm v0.9.0 // indirect
github.com/fsnotify/fsnotify v1.6.0 // indirect
github.com/gabriel-vasile/mimetype v1.4.2 // indirect
github.com/gin-contrib/sse v0.1.0 // indirect
github.com/go-openapi/jsonpointer v0.19.6 // indirect
github.com/go-openapi/swag v0.22.4 // indirect
github.com/go-playground/locales v0.14.1 // indirect
github.com/go-playground/universal-translator v0.18.1 // indirect
github.com/go-playground/validator/v10 v10.14.1 // indirect
github.com/goccy/go-json v0.10.2 // indirect
github.com/golang/protobuf v1.5.3 // indirect
github.com/google/uuid v1.4.0 // indirect
github.com/hashicorp/hcl v1.0.0 // indirect
github.com/inconshreveable/mousetrap v1.1.0 // indirect
github.com/invopop/yaml v0.2.0 // indirect
github.com/josharian/intern v1.0.0 // indirect
github.com/json-iterator/go v1.1.12 // indirect
github.com/klauspost/compress v1.17.1 // indirect
github.com/klauspost/cpuid/v2 v2.2.5 // indirect
github.com/leodido/go-urn v1.2.4 // indirect
github.com/lithammer/shortuuid/v3 v3.0.7 // indirect
github.com/magiconair/properties v1.8.7 // indirect
github.com/mailru/easyjson v0.7.7 // indirect
github.com/mattn/go-isatty v0.0.19 // indirect

github.com/mitchellh/mapstructure v1.5.0 // indirect
github.com/modern-go/concurrent v0.0.0-20180306012644-bacd9c7ef1dd // indirect
github.com/modern-go/reflect2 v1.0.2 // indirect
github.com/mohae/deepcopy v0.0.0-20170929034955-c48cc78d4826 // indirect
github.com/nats-io/nkeys v0.4.6 // indirect
github.com/nats-io/nuid v1.0.1 // indirect
github.com/oklog/ulid v1.3.1 // indirect
github.com/pelletier/go-toml/v2 v2.1.0 // indirect
github.com/perimeterx/marshmallow v1.1.5 // indirect
github.com/pkg/errors v0.9.1 // indirect
github.com/sagikazarmark/locafero v0.3.0 // indirect
github.com/sagikazarmark/slog-shim v0.1.0 // indirect
github.com/sourcegraph/conc v0.3.0 // indirect
github.com/spf13/afero v1.10.0 // indirect
github.com/spf13/cast v1.5.1 // indirect
github.com/spf13/pflag v1.0.5 // indirect
github.com/subosito/gotenv v1.6.0 // indirect
github.com/twitchyliquid64/golang-asm v0.15.1 // indirect
github.com/ugorji/go/codec v1.2.11 // indirect
go.uber.org/atomic v1.9.0 // indirect
go.uber.org/multierr v1.9.0 // indirect
golang.org/x/arch v0.4.0 // indirect
golang.org/x/crypto v0.18.0 // indirect
golang.org/x/exp v0.0.0-20230905200255-921286631fa9 // indirect
golang.org/x/net v0.20.0 // indirect
golang.org/x/sys v0.16.0 // indirect
golang.org/x/text v0.14.0 // indirect
google.golang.org/genproto/googleapis/rpc v0.0.0-20240205150955-31a09d347014
gopkg.in/ini.v1 v1.67.0 // indirect
gopkg.in/yaml.v3 v3.0.1 // indirect
github.com/ThreeDotsLabs/watermill v1.2.0
github.com/ThreeDotsLabs/watermill-nats/v2 v2.0.2
github.com/nats-io/nats.go v1.31.0
github.com/google/uuid v1.3.1 // indirect
github.com/klauspost/compress v1.17.1 // indirect
github.com/lithammer/shortuuid/v3 v3.0.7 // indirect
github.com/nats-io/nkeys v0.4.6 // indirect
github.com/nats-io/nuid v1.0.1 // indirect
github.com/oklog/ulid v1.3.1 // indirect
github.com/pkg/errors v0.9.1 // indirect
golang.org/x/crypto v0.14.0 // indirect
golang.org/x/sys v0.13.0 // indirect
github.com/gordonklaus/portaudio v0.0.0-20230709114228-aafa478834f5

External Docker Images

grafana/grafana:9.3.6
Nats:2.10
prom/prometheus:v2.45.0

Demo Python Packages

annotated-types==0.6.0
anyio==4.3.0
certifi==2024.2.2
cffi==1.16.0
charset-normalizer==3.3.2
click==8.1.7
fastapi==0.110.1
h11==0.14.0
idna==3.6
nats-py==2.7.2
pocketsphinx==5.0.3
pycparser==2.22
pydantic==2.6.4
pydantic_core==2.16.3
requests==2.31.0
sniffio==1.3.1
sounddevice==0.4.6
SpeechRecognition==3.10.3
starlette==0.37.2
typing_extensions==4.11.0
urllib3==2.2.1
uvicorn==0.29.0
websockets==12.0
contourpy==1.2.0
cycler==0.12.1
fonttools==4.44.3
importlib-resources==6.1.1
kiwisolver==1.4.5
matplotlib==3.8.2
nats-py==2.6.0
numpy==1.26.2
packaging==23.2
Pillow==10.1.0
pyparsing==3.1.1
python-dateutil==2.8.2
six==1.16.0
zipp==3.17.0

annotated-types==0.6.0
anyio==4.3.0
certifi==2024.2.2
charset-normalizer==3.3.2
click==8.1.7
fastapi==0.110.1
h11==0.14.0
idna==3.6
nats-py==2.7.2
pydantic==2.6.4
pydantic_core==2.16.3
requests==2.31.0
sniffio==1.3.1
SpeechRecognition==3.10.3
starlette==0.37.2
typing_extensions==4.11.0
urllib3==2.2.1
uvicorn==0.29.0
websockets==12.0
pytz==2024.1

Frontend Demo Dependencies

"node": ">=18.17.0",
"packageManager": "yarn@1.22.19",
"@emotion/react": "^11.11.4",
"@emotion/styled": "^11.11.0",
"@headlessui/react": "^1.7.18",
"@heroicons/react": "^2.1.1",
"@mui/icons-material": "^5.15.12",
"@mui/material": "^5.15.12",
"@next/bundle-analyzer": "^14.0.3",
"@radix-ui/react-accordion": "^1.1.2",
"@radix-ui/react-checkbox": "^1.0.4",
"@radix-ui/react-dialog": "^1.0.5",
"@radix-ui/react-dropdown-menu": "^2.0.6",
"@radix-ui/react-form": "^0.0.3",
"@radix-ui/react-label": "^2.0.2",
"@radix-ui/react-popover": "^1.0.7",
"@radix-ui/react-radio-group": "^1.1.3",
"@radix-ui/react-scroll-area": "^1.0.5",
"@radix-ui/react-select": "2.0.0",
"@radix-ui/react-slider": "^1.1.2",
"@radix-ui/react-switch": "^1.0.3",
"@radix-ui/react-tabs": "^1.0.4",

"@radix-ui/react-toggle-group": "^1.0.4",
"@radix-ui/react-tooltip": "^1.0.7",
"@semantic-release/changelog": "^6.0.3",
"@semantic-release/commit-analyzer": "^11.1.0",
"@semantic-release/git": "^10.0.1",
"@semantic-release/github": "^9.2.3",
"@semantic-release/npm": "^11.0.1",
"@semantic-release/release-notes-generator": "^12.1.0",
"@t3-oss/env-nextjs": "^0.7.1",
"@trivago/prettier-plugin-sort-imports": "^4.3.0",
"@vercel/otel": "^0.3.0",
"axios": "^1.6.7",
"class-variance-authority": "^0.7.0",
"cors": "^2.8.5",
"dotenv": "^16.4.5",
"express": "^4.18.3",
"lodash": "^4.17.21",
"mongoose": "^8.2.1",
"nats": "^2.21.0",
"nats.ws": "^1.22.0",
"next": "^14.1.4",
"next-compose-plugins": "^2.2.1",
"nodemon": "^3.1.0",
"react": "^18.2.0",
"react-dom": "^18.2.0",
"react-toastify": "^10.0.4",
"stream": "^0.0.2",
"tailwind-merge": "^2.0.0",
"yarn": "^1.22.21",
"zod": "^3.22.4",
"@babel/core": "^7.23.3",
"@babel/plugin-syntax-flow": "^7.23.3",
"@babel/plugin-transform-optional-chaining": "^7.23.4",
"@babel/plugin-transform-react-jsx": "^7.23.4",
"@jest/globals": "^29.7.0",
"@opentelemetry/api": "1.7.0",
"@opentelemetry/resources": "1.18.1",
"@opentelemetry/sdk-node": "0.45.1",
"@opentelemetry/sdk-trace-node": "1.18.1",
"@opentelemetry/semantic-conventions": "1.18.1",
"@playwright/test": "^1.40.0",
"@storybook/addon-essentials": "^7.5.3",
"@storybook/addon-interactions": "^7.5.3",
"@storybook/addon-links": "^7.5.3",

"@storybook/blocks": "^7.5.3",
"@storybook/nextjs": "^7.5.3",
"@storybook/react": "^7.5.3",
"@storybook/test-runner": "^0.15.2",
"@storybook/testing-library": "^0.2.2",
"@testing-library/jest-dom": "^6.1.4",
"@testing-library/react": "^14.1.2",
"@total-typescript/ts-reset": "^0.5.1",
"@types/jest": "^29.5.10",
"@types/node": "^20.10.0",
"@types/react": "^18.2.38",
"@types/react-dom": "^18.2.17",
"@typescript-eslint/eslint-plugin": "^6.12.0",
"@typescript-eslint/parser": "^6.12.0",
"all-contributors-cli": "^6.26.1",
"autoprefixer": "^10.4.18",
"cross-env": "^7.0.3",
"eslint": "8.54.0",
"eslint-config-next": "14.0.3",
"eslint-config-prettier": "^9.0.0",
"eslint-config-react-app": "^7.0.1",
"eslint-plugin-import": "^2.29.0",
"eslint-plugin-react": "7.33.2",
"eslint-plugin-storybook": "^0.6.15",
"eslint-plugin-tailwindcss": "^3.13.0",
"fetch-mock": "^9.11.0",
"gzip-size": "6",
"jest": "^29.7.0",
"jest-environment-jsdom": "^29.7.0",
"mkdirp": "^3.0.1",
"npm-only-allow": "^1.2.6",
"patch-package": "^8.0.0",
"postcss": "^8.4.35",
"postinstall-postinstall": "^2.1.0",
"prettier": "3.0.3",
"prettier-plugin-tailwindcss": "^0.5.7",
"semantic-release": "^22.0.8",
"storybook": "^7.5.3",
"tailwindcss": "^3.4.1",
"ts-jest": "^29.1.1",
"tsc": "^2.0.4",
"typescript": "5.3.2",
"webpack": "5.89.0"