Quick Summary of Phase 1

Key Task	What to Document	Deliverable
Define System Architecture	System architecture diagram, API and framework decisions	System Architecture Diagram, Technical Design Document
Define API and Backend Communication	RESTful API specifications	API Specifications Document
Define Hardware-Software Interaction	Document USB-TMC communication, Define SCPI command structure, Develop test scripts	Hardware-software Interaction Diagram, PyVISA Test Script, SCPI Commands List
Plan Authentication (OAuth-based Security)	OAuth provider decision (pros/cons of each option), User roles and permissions structure, How the frontend and backend handle token verification	Authentication Flow Diagram, Technical Document on OAuth Implementation, User Roles and Permissions Document
Plan Kubernetes and CI/CD Integration	CI/CD pipeline design (triggers, build steps, testing, deployment), Kubernetes architecture (pods, services, namespaces), Hosting environment selection (comparison and justification), Security measures for Kubernetes deployment	CI/CD Workflow, Kubernetes Deployment Plan
GitHub Version Control and Documentation	Repository structure and organization, Initial README file with project goals, architecture, and setup instructions, Guidelines for version control and collaboration	GitHub Repository Setup, Initial ReadMe file

Key Task 1: Define System Architecture

Deliverables:

1. System Architecture Diagram

The System Architecture Diagram provides a visual representation of how the major components of the Network Analyzer Project interact. It outlines the frontend, backend, database, hardware components (oscilloscope), APIs, and authentication layers. This diagram is crucial for understanding:

- How the frontend (Angular + Chart.js) communicates with the API (Node.js + RESTful API)
- How the backend interacts with the oscilloscope using PyVISA and USB-TMC
- How data flows between system components, including storage in JSON format
- How OAuth authentication secures user access
- How the CI/CD pipeline and Kubernetes infrastructure support deployment

This diagram will serve as a high-level blueprint, ensuring that all components are well-integrated and function cohesively.

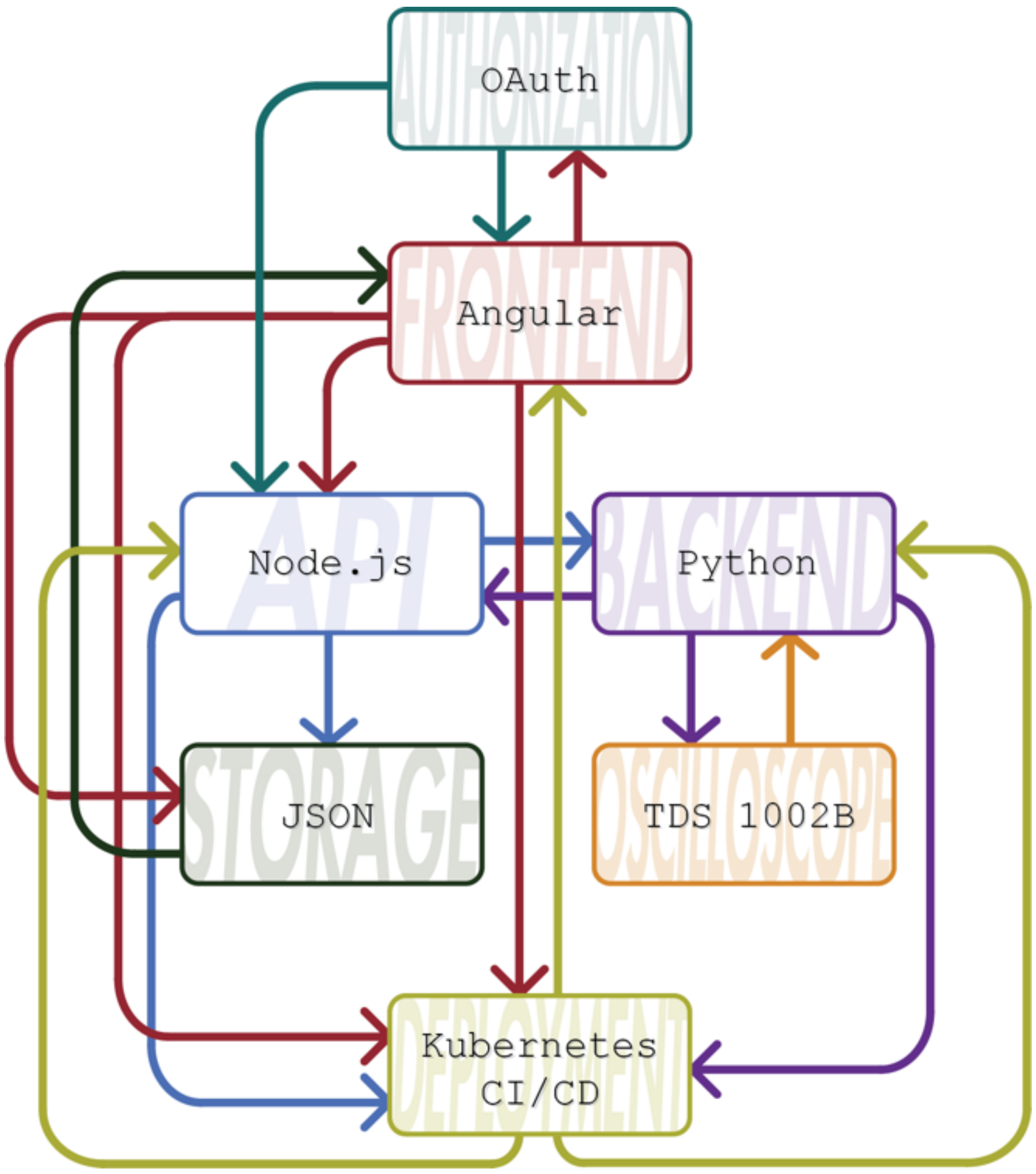
2. Technical Design Document

The Technical Design Document is a detailed written document that expands on the System Architecture Diagram by providing in-depth explanations of each component's implementation. It includes:

1. Overview of the system – Purpose, scope, and key technologies used
2. System architecture breakdown – Explanation of how frontend, backend, and hardware interact, along with justification for each technology
3. API design – RESTful API specifications, including endpoints, request/response formats, and security measures.
4. Hardware integration plan – USB-TMC setup, SCPI command list, and oscilloscope communication.
5. Authentication and security – OAuth implementation details, user roles, and access control.
6. Deployment strategy – Kubernetes deployment, CI/CD workflow, and hosting environment.
7. Scalability and future considerations – How the system can be expanded for additional functionality.

This document serves as technical documentation for both development and future enhancements, ensuring consistency, maintainability, and clarity in implementation.

System Architecture Diagram



Data Flow Labels (From the System Architecture Diagram Image Above)

From	To	Data Flow Label
Angular (Frontend)	OAuth (Authentication)	User Login Request & Token Verification
OAuth (Authentication)	Angular (Frontend)	Provide User Authentication Token
OAuth (Authentication)	Node.js (API)	Validate Token & Assign User Role
Angular (Frontend)	Node.js (API)	User Request: Signal Parameters & UI Settings
Node.js (API)	Python (Backend)	Command Execution Request: Signal Processing or Oscilloscope Query
Python (Backend)	TDS 1002B (Oscilloscope)	SCPI Commands for Signal Measurement
TDS 1002B (Oscilloscope)	Python (Backend)	Measurement Data: Amplitude, Phase, Frequency
Python (Backend)	Node.js (API)	Formatted Data Response: Processed Oscilloscope Measurements
Angular (Frontend)	JSON (Storage)	Store User Preferences (Signal Configurations)
Node.js (API)	JSON (Storage)	Store Data: Measure Logs, User Configurations
JSON (Storage)	Angular (Frontend)	Retrieve Stored Data for UI Display
Kubernetes (Deployment)	Angular (Frontend)	Deploy UI Updates
Kubernetes (Deployment)	Python (Backend)	Deploy Backend Processing Services
Kubernetes (Deployment)	Node.js (API)	Automated API Deployment
Angular (Frontend)	Kubernetes (Deployment)	Trigger Frontend Deployment Pipeline
Python (Backend)	Kubernetes (Deployment)	Trigger Backend Deployment Pipeline
Node.js (API)	Kubernetes (Deployment)	Trigger API Deployment Pipeline

Technical Design Document

System Overview

Purpose

The Network Analyzer Project is designed to facilitate real-time signal analysis by integrating a frontend user interface, a backend processing system, and an oscilloscope for hardware measurement. It enables users to generate signals, send them to an oscilloscope, and retrieve amplitude, phase, and frequency response measurements. This allows for effective Bode plot visualization, aiding in system characterization and testing.

This system is designed to be scalable, modular, and extendable, allowing for future enhancements, such as database integration, AI-based signal analysis, and additional measurement hardware.

Scope

- *Frontend (Angular + Chart.js)* → Provides a UI for signal generation, measurement retrieval, and real-time data visualization.
- *Backend (Node.js + Python)* → Acts as the communication bridge, handling API requests, executing Python scripts for signal processing, and interfacing with the oscilloscope via PyVISA and SCPI commands.
- *Oscilloscope (TDS 1002B)* → Captures signal data, measuring amplitude, phase shift, and frequency response.
- *Data Storage (JSON)* → Logs oscilloscope readings and user configurations for easy retrieval.
- *Authentication (OAuth)* → Implements user authentication and role-based access control.
- *Deployment (Kubernetes + CI/CD)* → Automates scalable, continuous deployment for the entire system.

Key Technologies Used

Layer	**Technology**	**Purpose**
Frontend (UI and Visualization)	Angular, Chart.js, Angular Material	Provides real-time graphing, user input management, and a modular UI framework
Backend (Processing and Hardware Communication)	Python, PyVISA, SciPy, NumPy	Processes signal generation, measurement retrieval, and communicates with the oscilloscope

API Layer (Backend Interface)	Node.js, Express.js	Handles API requests, authentication, and data logging
Hardware (Signal Acquisition)	Oscilloscope (TDS 1002B), USB-TMC, SCPI Commands	Captures real-world signals and returns measurement data
Data Storage	JSON	Lightweight storage of oscilloscope data before database expansion
Authentication	OAuth (Google/GitHub), JWT	Implements secure user authentication
Deployment & CI/CD	Kubernetes, GitHub Actions, Docker	Automates deployment, scaling, and version control

Breakdown of the System Architecture Diagram

Components and their interactions in a structured format:

1. Frontend (Angular + Chart.js)
   - What it does:
     - Provides a UI for user input (frequency, amplitude, waveform selection)
     - Sends API requests to the backend for signal generation or oscilloscope measurements
     - Receives measurement data and displays the Bode plot using Chart.js
   - How it connects:
     - Sends API requests to Node.js for processing
     - Retrieves JSON data for stored user configurations
     - Authenticates users via OAuth, passing tokens to Node.js for validation
     - Triggers frontend deployments in Kubernetes CI/CD when new updates are pushed

2. API Layer (Node.js + Express.js)
   - What it does:
     - Serves as the central API for the entire system
     - Handles requests from the frontend for signal generation and oscilloscope measurements

        - Calls Python scripts to generate signals and retrieve oscilloscope measurements
        - Stores measurement data in JSON format for logging and retrieval
    - How it connects:
        - Receives HTTP requests from the Angular frontend
        - Forwards execution commands to the Python backend for signal processing
        - Stores results in JSON for future retrieval
        - Validates user authentication tokens received from OAuth
        - Triggers API deployments in Kubernetes CI/CD when new updates are pushed

3. Backend Python (SciPy, NumPy, Sounddevice, PyVISA)
    - What it does:
        - Generates test signals (sine, square, triangle waves) using SciPy and NumPy
        - Outputs signals via USB using Sounddevice
        - Uses PyVISA to send SCPI commands to the TDS 1002B oscilloscope
        - Reads and processes oscilloscope data (phase shift & amplitude response), formats the data as JSON responses, and sends it to the Node.js API
    - How it connects:
        - Receives commands from Node.js to adjust signal parameters
        - Sends SCPI commands over USB-TMC to communicate with the oscilloscope
        - Returns measured data to Node.js for storage and frontend display

4. Oscilloscope (TDS 1002B via USB-TMC)
    - What it does:
        - Captures real-world signal measurements for phase shift and amplitude analysis
        - Measures signal response at different frequencies using SCPI commands
    - How it connects:
        - Connected to the Python backend via USB-TMC
        - Receives SCPI commands from the Python backend

        - Returns measurement data to Python for processing and API response

5. Data Storage (JSON)
    - What it does:
        - Stores oscilloscope readings in JSON for real-time access
        - Serves as an intermediate data storage before expanding to a relational (SQL) or NoSQL database
    - How it connects:
        - Node.js logs oscilloscope readings into JSON
        - The frontend retrieves JSON data to display results

6. Authentication (OAuth)
    - What it does:
        - Secures user authentication and assigns access roles based on permissions
        - Ensures only authenticated users can send requests
    - How it connects:
        - Handles login requests from the frontend
        - Issues tokens to the Node.js API for user verification
        - Confirms authentication status with Angular

7. Deployment (Kubernetes + CI/CD)
    - What it does:
        - Automates deployment of frontend, backend, and API services
        - Manages scalability and continuous updates for the system
    - How it connects:
        - GitHub commits trigger CI/CD workflows
        - Kubernetes deploys new versions automatically
        - Ensures all system components run in isolated, containerized environments

API-to-Backend Workflow

1. User Interaction and API Request (Frontend → API)
    - User selects signal parameters in the Angular Frontend
    - Angular sends a POST request to the Node.js API via the HTTP client module
    - The request includes:

```
{
  "frequency": 1000,
  "amplitude": 5.0,
  "waveform": "sine"
}
```

- Node.js validates the request (e.g., checks if the frequency is within the valid range)

2. API to Backend Communication (Node.js → Python Backend)
    - If valid, Node.js forwards the request to the Python backend for processing
    - Communication between Node.js and Python can happen in two possible ways
        - Option 1: Child Process Execution (Default)
            - Node.js spawns a child process using `child_process.spawn()` to run the Python script

              ```
              const { spawn } = require('child_process');
              const pythonProcess = spawn('python',
              ['backend/scripts/signal_gen.py', '1000', '5.0',
              'sine']);
              ```

        - Option 2: WebSockets (For Real-Time-Streaming)
            - If real-time continuous data is needed, WebSockets can be used instead of a one-time HTTP request
3. Backend Processing
    - The Python backend receives the request parameters
    - It determines whether to:
        - Generate a synthetic signal using SciPy/NumPy
        - Communicate with the oscilloscope via PyVISA
    - If interacting with the oscilloscope:
        - The backend sends a SCPI command over USB-TMC:

          ```
          oscilloscope.write("MEASure:VPP?")
          vpp_value = oscilloscope.read()
          ```

        - The oscilloscope responds with measured data (e.g., amplitude, phase, frequency)
4. Backend to API Response (Python → Node.js API)
    - The Python backend processes the measurement data and formats it as JSON:

```
{
```

```
  "status": "success",
  "data": {
    "frequency": 1000,
    "amplitude": 5.0,
    "phase": 0.5
    }
}
```

  - It then sends the formatted JSON response back to the Node.js API
  - The API performs a final validation before returning the response to the frontend

5. API Response to Frontend (API → Frontend)
   - The Node.js API returns the JSON response to the Angular frontend
   - The frontend parses the JSON data and updates Chart.js for real-time visualization
6. Logging and Storage (API → JSON Storage)
   - The Node.js API logs the oscilloscope readings and user actions in JSON format for debugging and analysis
   - Example log entry:

```
{
  "log_id": "log_20250210_001",
  "timestamp": "2025-02-10T14:30:10Z",
  "level": "info",
  "component": "backend",
  "message": "Signal generated successfully",
  "details": {
    "frequency": 1000,
    "amplitude": 5.0,
    "waveform": "sine"
  }
}
```

   - If historical data retrieval is needed, the frontend can fetch stored configurations via the API

SCPI Command List

Identification and Status Commands

Command	**Description**	**Example Response**
`*IDN?`	Queries the oscilloscope for identification details	`"TEKTRONIX, TDS1002B, C102220, CF:91.1CT FV:v22.11"`
`*RST`	Resets the oscilloscope to default settings	No Response

`*CLS`	Clears the oscilloscope's status registers	No response
`*OPC?`	Checks if the last operation is complete	`"1"`

Measurement Commands (Amplitude, Frequency, Phase, Time)

Command	**Description**	**Example Response**
`MEASure:VPP?`	Returns voltage peak-to-peak (Vpp) measurement	`"3.5"` (Volts)
`MEASure:VAVG?`	Returns the average voltage measurement	`"1.2"` (Volts)
`MEASure:FREQuency?`	Measures the frequency of the input signal	`"1000"` (Hz)
`MEASure:PHASe?`	Measures the phase shift between two signals	`"0.5"` (Degrees)
`MEASure:PERiod?`	Measures the signal period	`"0.001"` (Seconds)
`MEASure:RISetime?`	Measures rise time from 10% to 90% of signal amplitude	`"0.0005"` (Seconds)
`MEASure:FALLtime?`	Measures fall time from 90% to 10% of signal amplitude	`"0.0006"` (Seconds)

Channel Configuration Commands

Command	**Description**	**Example Response**
`CHANnel1:DISPlay ON`	Enables Channel 1 display	No Response
`CHANnel1:SCALe 2.0`	Sets Channel 1 vertical scale to 2.0V per division	No Response
`CHANnel1:POSition 0`	Sets Channel 1 offset position to center	No Response
`CHANnel2:DISPlay OFF`	Disables Channel 2 display	No Response

Trigger Configuration Commands

Command	**Description**	**Example Response**
`TRIGger:MODE AUTO`	Sets trigger mode to auto	No Response
`TRIGger:MODE NORMal`	Sets trigger mode to normal	No Response
`TRIGger:EDGE:SOURce CHANnel1`	Sets trigger source to Channel 1	No Response
`TRIGger:LEVel 1.5`	Sets the trigger level to 1.5V	No Response

Acquisition and Data Retrieval Commands

Command	**Description**	**Example Response**
`ACQuire:STATE ON`	Starts data acquisition	No Response
`ACQuire:STATE OFF`	Stops data acquisition	No Response
`ACQuire:MODE AVERage`	Sets acquisition mode to average	No Response
`DATa:SOURce CHANnel1`	Sets the oscilloscope data source to Channel 1	No Response
`WAVFrm:DATA?`	Retrieves waveform data from the oscilloscope	Binary response (Waveform Data)

Utility and Display Commands

Command	**Description**	**Example Response**
`DISplay:CLEar`	Clears the oscilloscope display	No Response
`HARDCopy START`	Captures and prints a screenshot from the oscilloscope	No Response

Example SCPI Script Using PyVISA

```
import pyvisa

# Open communication with the oscilloscope
rm = pyvisa.ResourceManager()
oscilloscope = rm.open_resource("USB::0x0699::0x0363::C102220::INSTR")

# Query device identification
print(oscilloscope.query("*IDN?"))

# Retrieve peak-to-peak voltage measurement
oscilloscope.write("MEASure:VPP?")
vpp_value = oscilloscope.read()
print(f"Voltage Peak-to-Peak: {vpp_value} V")

# Set trigger level
oscilloscope.write("TRIGger:LEVel 1.5")

# Retrieve waveform data
oscilloscope.write("WAVFrm:DATA?")
waveform_data = oscilloscope.read()
print(f"Waveform Data: {waveform_data}")

# Close the connection
oscilloscope.close()
```

Justification for Technology Choices
This explains reasoning behind the languages and frameworks I chose. The reason why I chose these languages and frameworks is because I want to be compliant with not only industry standards but showcase tools that will assist in proving my understanding of key concepts applicable to a software job position at Keysight and assist in bolstering my job application.

Languages, Frameworks, et al. Decisions

Languages / Framework / Technology	**Role**	**Justification**
Node.js (Backend API)	Handles HHTP requests, interacts with Python, and logs oscilloscope data	Efficient for handling asynchronous API requests
Express.js (Node.js Framework)	Lightweight framework for RESTful API	Simplifies API development and middleware integration
Python (SciPy + Sounddevice, PyVISA)	Generates signals, communicates with the oscilloscope via SCPI commands	SciPy is optimized for signal processing, and PyVISA enables oscilloscope communication
Angular (Frontend)	UI framework for managing user input and real-time updates	Modular, scalable, and widely used in industry
Chart.js (Data Visualization)	Real-time graphing of frequency response	Lightweight, interactive charting for oscilloscope data
JSON (Data Storage)	Stores oscilloscope logs, user settings, and API response history	Lightweight and compatible across all system layers
OAuth (Authentication)	Secure user authentication via third-party OAuth providers	Uses industry-standard protocols for login and access control
Kubernetes (CI/CD & Deployment)	Manages scalable deployment	Automates scaling and backend, frontend, and API updates

Pertinent Addenda:

- RESTful API Explanation: The API follows REST principles for stateless, structured communication. Each endpoint can point to a resource or an action, I can produce stateless requests, and I will use standard HTTP methods.
- C++ Feasibility Notes: (Optional) C++ could possibly be integrated into Node.js for performance-critical operations, but Python remains the core backend. For, I will not be able to use C++ in most areas, specifically for the signal generation. Because SciPy and Sounddevice are built with Python in mind.

Justification for JSON Storage over SQL
I am using JSON storage to showcase knowledge using this data storage technique. JSON also serves as a great data storage utility as opposed to a database because I believe it will be easier to implement. Plus, the development will be simpler if I do not have to set up a database overhead.

Factor	**JSON Storage**	**SQL Database**
Ease of Use	Simple file-based storage, no database setup required	Requires schema design, SQL queries, and database management
Performance for Small Data	Optimized for storing transient or real-time measurement data with SQL query overhead	SQL queries introduce overhead for simple data storage
Flexibility	Schema-less structure allows easy modification	Requires schema changes for new data fields
Integration with Frontend/Backend	Easily parsed and used by both Angular (frontend) and Node.js (API)/Python (backend)	SQL data must be queried, structured, and converted into JSON
Scalability	Suitable for small to medium-sized data storage needs	Better for large-scale applications with frequent queries

Conclusion: Since the system only needs to store oscilloscope readings, user configurations, and logs, JSON provides a lightweight, flexible, and easy-to-manage storage solution without the overhead of an SQL database.

Future Database Expansion (When SQL Could Be Utilized)

While JSON works well for this project, a relational SQL database (PostgreSQL, MySQL) or NoSQL (MongoDB, Firebase) may be required if:

- Data size increases significantly (e.g., storing millions of readings)
- Complex queries and analytics (e.g., retrieving historical trends) are needed
- Multiple users need concurrent access and modifications

In such a case, the system can migrate from JSON to a database while maintaining API compatibility.

Key Task 2: Define Hardware Integration Plan
Deliverables

1. API Specifications Document

The API Specifications Document defines the structure, endpoints, request/response formats, and authentication for the Node.js-powered RESTful API. It details how the frontend interacts with the backend (Python) and the oscilloscope. This document ensures consistency, security, and efficiency in data exchange. To complete this, I will define RESTful endpoints, specify JSON response formats, and document authentication mechanisms.

API Specifications Document

Overview

The API Specifications Document defines the structure, endpoints, request/response formats, and authentication mechanisms for the Node.js-powered RESTful API used in the Network Analyzer project. It outlines how the frontend (Angular app) interacts with the backend (Node.js & Python) and how the backend interfaces with the oscilloscope to retrieve and process measurement data. It ensures secure data exchange through OAuth 2.0 authentication, role-based access control (RBAC), and encrypted communications.

This document ensures consistency, security, and efficiency in data exchange between system components.

The Hardware-software Integration Plan

The system integrates a USB-TMC-compliant oscilloscope with a software-based signal generator. The backend API enables remote instrument control, while the frontend visualizes measurement data.

How SciPy & Sounddevice Generate Signals

- SciPy generates a sine wave array
- Sounddevice plays the array through USB (or sound card alternative)

Alternatives if Python Can't Reach 300 kHz

- Use an external USB signal generator instead of Python
- Consider FPGA-based signal generation for precision

Backend Interface with Oscilloscope and Python-Generated Signals

USB-TMC/PyVISA Setup Process

- The oscilloscope communicates using PyVISA over USB-TMC. SCPI commands are issued via the Python API to retrieve phase and amplitude data
- PyVISA acts as a bridge between Python and USB-TMC devices
- The TDS 1002B uses USB-TMC, which follows the IEEE 488.2 standard for instrument communication

Breakdown of Hardware-Software Interaction

Each component in the diagram plays a crucial role in how data flows from signal generation to oscilloscope measurement and visualization. Here's a structured explanation:

1. Computer (Central Processing Unit)
    - Role: The main processing unit that controls signal generation, data collection, and visualization.
    - Interacts With:
        - o Python Processing Unit (Runs signal generation scripts)
        - o Backend API (Handles communication between software layers)
        - o Oscilloscope (Retrieves real-time measurement data)
2. Oscilloscope (TDS 1002B via USB-TMC)
    - Role: Captures input signals, measures phase and amplitude, and sends measurement data back.
    - Interacts With:
        - o Waveform Generator (Receives input signals for measurement)
        - o Python Processing Unit (Receives SCPI commands, responds with phase/amplitude data)
        - o Backend API (Provides measurement data for visualization)
3. Waveform Generator
    - Role: Generates test signals at varying frequencies for the oscilloscope to measure.
    - Interacts With:
        - o Oscilloscope (Provides test signals)
        - o Python Processing Unit (Can be software-controlled for automated sweeps)
4. Frontend UI (Angular + Chart.js)
    - Role: Provides a graphical user interface for input selection (frequency, amplitude, waveform) and real-time visualization.
    - Interacts With:
        - o Backend API (Fetches oscilloscope readings)
        - o Data Storage (Retrieves historical measurement data)
5. Backend API (Node.js + Express.js)
    - Role:

        - Handles API requests from the frontend
        - Routes data between Python and the UI
        - Logs oscilloscope measurement data into JSON storage
    - Interacts With:
        - Python Processing Unit (Receives measurement data)
        - Frontend UI (Sends oscilloscope readings for visualization)
        - Data Storage (Logs processed readings)
6. Python Processing Unit (SciPy + PyVISA)
    - Role:
        - Uses SciPy to generate signals
        - Uses PyVISA to communicate with the oscilloscope via SCPI commands
        - Processes received data before passing it to the backend
    - Interacts With:
        - Oscilloscope (Sends SCPI commands, receives measurement data)
        - Backend API (Sends processed measurement data)
        - Waveform Generator (Controls signal generation in automated sweeps)
7. Data Storage (JSON + Future Database Expansion)
    - Role: Temporarily stores oscilloscope measurement data for real-time access.
    - Interacts With:
        - Backend API (Logs and retrieves measurement data)
        - Frontend UI (Fetches stored readings)
8. Deployment & Automation (Kubernetes + CI/CD)
    - Role: Automates builds, testing, and deployment of both backend and frontend services.
    - Interacts With:
        - Backend API & Frontend UI (Ensures seamless updates)
        - GitHub (CI/CD Pipeline) (Triggers builds and deployment)

<u>Backend System Flow</u>

- The frontend sends API requests to the Node.js backend, which handles authentication and routes commands to the Python backend
- The Python backend communicates with the oscilloscope (via PyVISA and SCPI commands) to retrieve waveform measurements
- The backend can also generate test signals using SciPy + Sounddevice, outputting them via USB to the oscilloscope

<u>Data Flow Architecture</u>

1. Frontend (Angular + Chart.js UI)
    - Sends API requests for signal generation and oscilloscope data retrieval
2. Backend (Node.js + Express + OAuth 2.0)
    - Authenticates API requests and forwards them to the Python backend
3. Backend Processing (Python + PyVISA)
    - Sends SCPI commands to the oscilloscope via USB-TMC
    - Logs oscilloscope readings in JSON storage for frontend access
4. Signal Generation (Python + SciPy + Sounddevice)
    - Generates waveforms that are output to the oscilloscope for testing

RESTful API Specifications

API Endpoints Definition

A. Signal Generation Endpoints

Method	Endpoint	Description
POST	`/api/signal/generate`	Initiates signal generation with specified parameters
GET	`/api/signal/status`	Retrieves the current status of signal generation
DELETE	`/api/signal/stop`	Stops signal generation

Example Request / Response

POST `/api/signal/generate`

- Request Body (JSON)

```
{
  "frequency": 1000,
  "amplitude": 1.5,
  "waveform": "sine"
}
```

- Response (JSON)

```
{
  "status": "success",
  "message": "Signal generation started",
  "signal_id": "abc123"
}
```

B. Oscilloscope Data Retrieval Endpoints

Method	Endpoint	Description
GET	`/api/oscilloscope/data`	Retrieves amplitude and phase data from the oscilloscope

GET	`/api/oscilloscope/settings`	Retrieves oscilloscope configuration settings
POST	`/api/oscilloscope/set`	Adjusts oscilloscope settings

Example Request / Response

GET `/api/oscilloscope/data`

- Response (JSON)

```
{
  "status": "success",
  "data": {
    "timebase": "5ms/div",
    "voltage": "2V/div",
    "waveform": [
      { "time": 0.0, "voltage": 0.0 },
      { "time": 0.1, "voltage": 0.5 },
      { "time": 0.2, "voltage": 1.0 }
    ]
  }
}
```

<u>Authentication and Security Considerations</u>

→ OAuth 2.0 Token Verification

All API and WebSocket interactions require OAuth 2.0 authentication to prevent unauthorized access.

→ REST API Authentication

Each API request must include an Authorization header with a Bearer Token.

```
Authorization: Bearer xyz-12345
```

If the token is invalid, the request is rejected with a 401 Unauthorized error.

→ WebSocket Authentication

Since WebSockets do not use headers for authentication, the OAuth token must be included in the connection request.

Example Client Connection Request:

```
const socket = new WebSocket("wss://example.com", ["Bearer xyz-12345"]);
```

If the browser and WebSocket library will not support passing authentication tokens via subprotocols, I will pass token in URL Query parameter:

```
const socket = new WebSocket("wss://example.com?token=xyz-12345");
```

→ Role-Based Access Control (RBAC)

Role	**Permissions**
Admin	Full access (generate signals, retrieve oscilloscope data, modify settings)
Hiring Managers	Generate signals and retrieve oscilloscope data but cannot modify hardware settings
Friends/Family	Read-only access to previously logged oscilloscope data
General Users	Public Access Role (view past oscilloscope logs)

Example: Protecting API Routes with RBAC:

```
function authorizeRole(role) {
    return (req, res, next) => {
        if (req.user.role !== role) {
            return res.status(403).json({ error: "Forbidden: Insufficient
permissions" });
        }
        next();
    };
}

// Restrict oscilloscope settings updates to Admins only
app.post('/api/oscilloscope/set', authenticateToken, authorizeRole("Admin"),
(req, res) => {
    res.json({ message: "Oscilloscope settings updated" });
});
```

→ Security Best Practices

- Token Expiry: OAuth 2.0 access tokens expire after 1 hour, requiring refresh
- HTTPS Enforcement: TLS encryption for all API and WebSocket connections
- Rate Limiting: Prevent brute-force attacks by limiting API requests per user

Data Structure for Oscilloscope Readings and System Logs

A. `signals.json` (Logs for Generated Signals)
   Stores signal parameters and execution status for monitoring and debugging.

```
{
  "signals": [
    {
      "signal_id": "abc123",
      "timestamp": "2025-02-19T14:30:00Z",
      "user": "dev_01",
      "parameters": {
        "frequency": 1000,
        "amplitude": 1.5,
        "waveform": "sine",
        "duration": 30
      },
      "status": "active"
    }
  ]
}
```

B. `oscilloscope_data.json` (Captured Measurement Data)
   Stores oscilloscope waveform readings for real-time plotting and post-analysis

```
{
  "oscilloscope_readings": [
    {
      "capture_id": "xyz789",
      "timestamp": "2025-02-19T15:00:00Z",
      "user": "dev_01",
      "channel": "CH1",
      "timebase": "5ms/div",
      "voltage_scale": "2V/div",
      "data_points": [
        { "time": 0.0, "voltage": 0.0 },
        { "time": 0.1, "voltage": 0.5 },
        { "time": 0.2, "voltage": 1.0 }
      ]
    }
  ]
}
```

Data Expiration Policy

- Signal logs are stored for 30 minutes after execution
- Oscilloscope data is retained for 1 hour, after which it is deleted

Justification for Potential WebSocket Integration

WebSockets were evaluated as a potential addition for real-time oscilloscope measurements and signal monitoring. The decision to partially integrate WebSockets is based on the following considerations:

<u>Why WebSockets?</u>

- Reduced Latency: Unlike REST polling, WebSockets push data instantly
- Efficiency: No need for repeated API calls, reducing backend load
- Live Visualization: Ideal for real-time oscilloscope readings displayed on the frontend

<u>Where WebSockets Will Be Used</u>

- Real-time oscilloscope waveform streaming
- Live signal monitoring updates

<u>Where REST Will Be Used Instead</u>

- Retrieving oscilloscope settings (static data)
- Sending signal configuration commands

Key Task 3: Define Hardware-Software Interaction

Deliverables:

1. Hardware-software Interaction Diagram

The Hardware-Software Interaction Diagram visually represents how the frontend, API, backend, and oscilloscope communicate. It shows the flow of commands from the frontend to the oscilloscope and the return of measured data. This diagram ensures clarity in system integration and debugging. I will create this using a structured flowchart that illustrates data exchange, hardware protocols, and software triggers.

Hardware-Software Interaction Diagram

Overview:

- The Hardware-Software Interaction Diagram visually represents how different components interact.
- It shows the flow of API requests, SCPI commands, and measurement responses between software and hardware.

The Difference Between the Hardware-Software Interaction Diagram and the System Architecture Diagram

→ What is the System Architecture Diagram?

- The System Architecture Diagram provides a high-level overview of the system's major components and their relationships (e.g., frontend, backend, database, authentication, external APIs).
- It defines the overall system structure without detailing specific communication flows or protocols.

→ What is the Hardware-Software Interaction Diagram?

- The Hardware-Software Interaction Diagram is more detailed and focused on the communication flow between software components and hardware (e.g., how the backend sends commands to the oscilloscope, how data is retrieved, logged, and displayed).
- It illustrates real-time data exchange and interactions between specific system parts rather than providing just a conceptual overview.

→ Key Distinctions

Aspect	**System Architecture Diagram**	**Hardware-Software Interaction Diagram**
Scope	Entire system, high-level view	Focused on backend-hardware communication

Purpose	Shows how components are connected	Details how data flows between software and hardware
Includes	APIs, frontend, backend, database, authtentication	SCPI commands, USB-TMC, PyVISA, signal generation
Level of Detail	Abstract	Technical and implementation-focused
Why It’s Important	Helps define the system structure for development	Ensures accurate implementation of hardware control

Key Components of the Hardware-Software Interaction Diagram:

1. Frontend (Angular + Chart.js)
   - Sends requests to the backend API for signal generation and oscilloscope data retrieval.
2. Backend (Node.js + Express)
   - Handles OAuth authentication.
   - Routes API requests to Python processing unit.
3. Python Processing Unit (PyVISA, SciPy, Sounddevice)
   - Sends SCPI commands to the oscilloscope via USB-TMC.
   - Retrieves waveform data and stores it in JSON logs.
   - Generates test signals using SciPy + Sounddevice.
4. Oscilloscope (TDS 1002B via USB-TMC)
   - Executes SCPI commands and returns measurement data.
5. Data Storage (JSON Logs)
   - Stores oscilloscope readings and generated signals for later retrieval.

Step-by-step Flowchart of Hardware-Software Communication

→ Step 1: Frontend Requests Signal Generation or Measurement Data

Component: Frontend (Angular + Chart.js UI)

Action: User selects frequency, amplitude, and waveform type in UI.

Data Flow:

- User inputs are sent via an API request to the Node.js backend.
- HTTP Request:

```
POST /api/signal/generate
Content-Type: application/json
Authorization: Bearer <ACCESS_TOKEN>

{
  "frequency": 1000,
  "amplitude": 1.5,
  "waveform": "sine"
}
```

Output: API forwards request to the backend for processing.

→ Step 2: Backend (Node.js) Processes the Request and Authenticates It
Component: Backend (Node.js + Express API + OAuth 2.0)
Action:

- Backend validates the request (checks API token and request format).
- If valid, backend forwards data to the Python processing unit via IPC (Inter-Process Communication).
  Data Flow:
- Verification process:

```
app.post('/api/signal/generate', authenticateToken, (req, res) => {
    sendToPython(req.body);
    res.json({ message: "Signal generation request received" });
});
```

Output:

- Request is now sent to the Python processing unit for signal generation or oscilloscope measurement.

→ Step 3: Python Processing Unit Sends SCPI Commands via PyVISA
Component: Python Processing Unit (PyVISA, SciPy, Sounddevice)
Action:

- If the request is for signal generation, Python creates a waveform and outputs it via USB.
- If the request is for oscilloscope measurement, Python sends an SCPI command via PyVISA.

SCPI Command Execution Flow:

```
import pyvisa

rm = pyvisa.ResourceManager()
osc = rm.open_resource("USB0::0x0699::0x0363::C010191::INSTR")

# Send SCPI command to retrieve peak-to-peak voltage
osc.write("MEASure:VPP? CH1")
voltage = osc.read()

print("Peak-to-Peak Voltage:", voltage)
```

Output:

- If signal generation, the oscilloscope receives a generated signal via USB.
- If measurement, the oscilloscope responds with SCPI data.

→ Step 4: Oscilloscope Executes the SCPI Command and Sends Back Data
Component: Oscilloscope (TDS 1002B via USB-TMC)
Action:

- The oscilloscope processes the SCPI command and returns measured values.
- Example SCPI output:

```
4.97E+00
```

Output:

- The oscilloscope sends this value back through USB-TMC to the Python processing unit.

→ Step 5: Python Processing Unit Stores the Data in JSON Logs
Component: Data Storage (JSON Logs)
Action:

- The Python backend parses the oscilloscope response and stores it in JSON for retrieval.
- Data Flow:
    - o JSON format for logged measurements:

```
{
  "oscilloscope_readings": [
    {
      "timestamp": "2025-02-19T15:00:00Z",
      "channel": "CH1",
      "voltage_pp": 4.97
    }
  ]
}
```

- Output:
    - o This JSON data is now available for retrieval by the backend API.

→ Step 6: Backend Retrieves the Data and Sends It to the Frontend
Component: Backend (Node.js + Express API)
Action:

- Backend fetches oscilloscope data from JSON storage and returns it to the frontend.

Data Flow:

- HTTP Response:

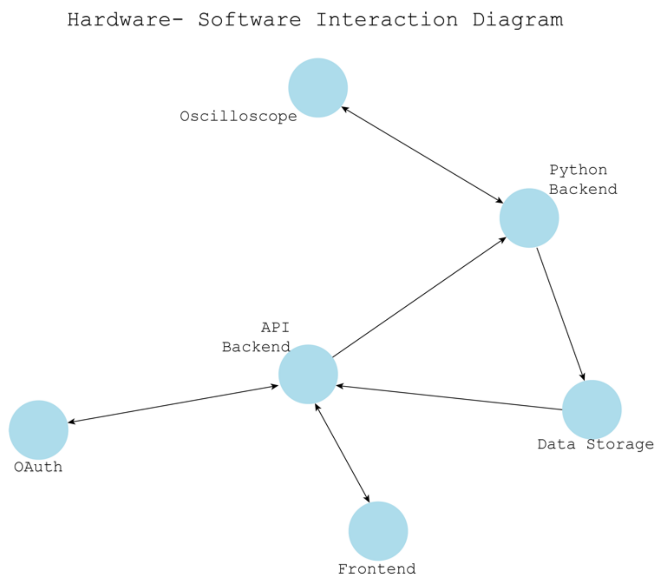
```
{
  "status": "success",
  "data": {
    "voltage_pp": 4.97
  }
}
```

Output:
- The frontend receives the measured oscilloscope data and updates the chart in real-time.

→ Step 7: Frontend Displays Measured Data to the User
Component: Frontend (Angular + Chart.js UI)
Action:
- The frontend visualizes the oscilloscope data in Chart.js for user review.

Output:
- The user sees the oscilloscope's measured values displayed dynamically.

Hardware-Software Interaction Diagram Flowchart Sketch:



→ The Hardware-Software Interaction Diagram visually represents how different components interact within the system. It outlines the flow of API requests, SCPI commands, and measurement responses between the frontend, backend, Python processing unit, and oscilloscope.
→ Why This Diagram is Important

- Ensures correct hardware integration by illustrating real-time data exchange.
- Clarifies API, backend, and oscilloscope interactions to avoid communication failures.
- Aids debugging and troubleshooting by showing the expected flow of commands and responses.

PyVISA Test Script Execution

→ The PyVISA Test Script is designed to validate the communication between the Python backend and the oscilloscope via USB-TMC and SCPI commands. This test ensures that the oscilloscope is detected, properly receives commands, and returns expected responses.
→ Test Script Execution
See Folder in the pathway Phase I - Design/3_Hardware-Software Interaction Diagram/ named, 'Code'
→ Objective: Validate the connection between the computer and the oscilloscope using PyVISA.
→ Test Script: Executed the pyvisa_test.py script to confirm communication with the oscilloscope.
→ Response from Terminal after Running Script:

```
Checking PyVISA installation and listing available instruments...
Found instruments: ('USB0::1689::867::C064444::0::INSTR',)
Connecting to oscilloscope at USB0::1689::867::C064444::0::INSTR...
Sending *IDN? command to identify the oscilloscope...
Connected to: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16
Connection closed successfully.
```

SCPI Command List

→ The SCPI (Standard Commands for Programmable Instruments) Command List documents the commands used to interact with the oscilloscope. These commands allow automated control over measurement settings and data retrieval.
→ Validated SCPI Commands:

Category	Command	Description	Status
Identification & General Queries	`*IDN?`	Identify the connected oscilloscope.	Validated
	`HORizontal:MAIn:SCAle?`	Query the horizontal scale (timebase).	Validated
	`CH1:SCAle?`	Query CH1 voltage scale.	Validated

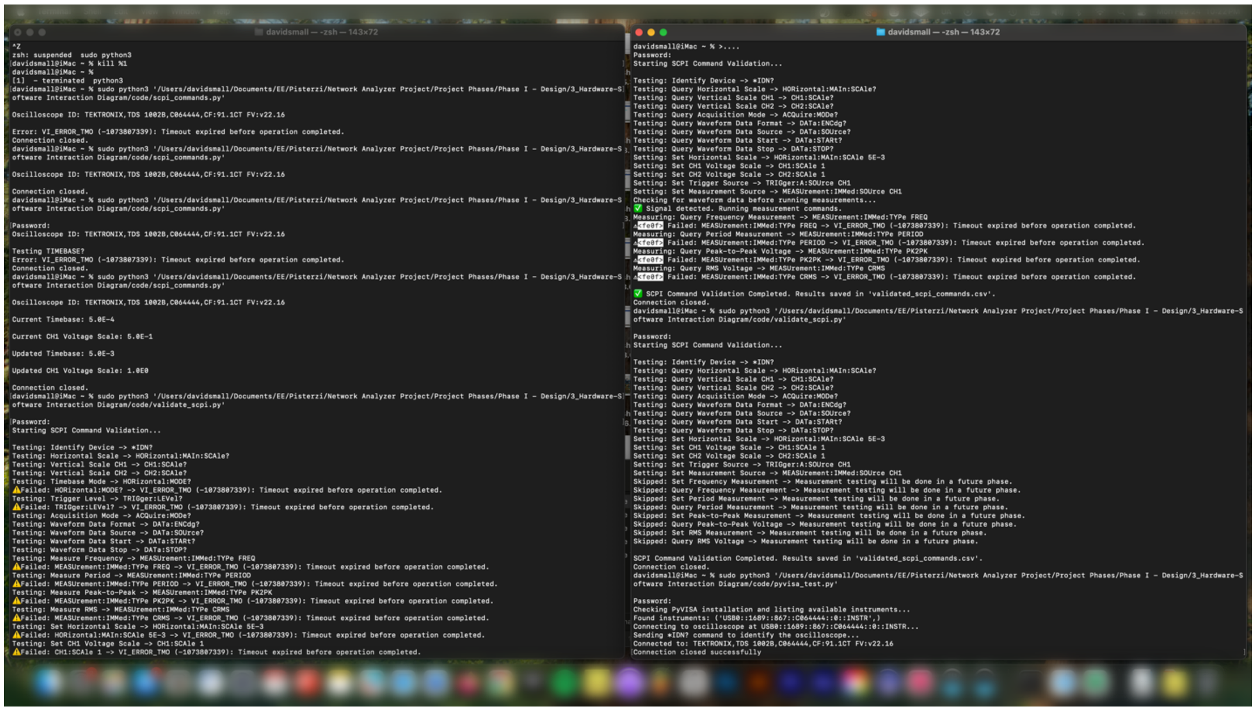
	`CH2:SCAle?`	Query CH2 voltage scale.	Validated
	`ACQuire:MODe?`	Query the current acquisition mode.	Validated
	`DATa:ENCdg?`	Query the waveform data encoding format.	Validated
	`DATa:SOUrce?`	Query the current waveform data source.	Validated
	`DATa:STARt?`	Query the waveform data start point.	Validated
	`DATa:STOP?`	Query the waveform data stop point.	Validated
Configuration Commands (.write())	`HORizontal:MAIn:SCAle 5E-3`	Set horizontal scale to 5ms/div.	Validated
	`CH1:SCAle 1`	Set CH1 voltage scale to 1V/div.	Validated
	`CH2:SCAle 1`	Set CH2 voltage scale to 1V/div.	Validated
	`TRIGger:A:SOUrce CH1`	Set CH1 as the trigger source.	Validated
	`MEASUrement:IMMed:SOUrce CH1`	Set CH1 as the measurement source.	Validated
Measurement Commands (Deferred for Testing Phase)	`MEASUrement:IMMed:TYPe FREQ`	Set frequency measurement type.	To Be Tested
	`MEASUrement:IMMed:VALue?`	Query frequency measurement.	To Be Tested
	`MEASUrement:IMMed:TYPe PERIOD`	Set period measurement type.	To Be Tested
	`MEASUrement:IMMed:VALue?`	Query period measurement.	To Be Tested
	`MEASUrement:IMMed:TYPe PK2PK`	Set peak-to-peak voltage measurement.	To Be Tested
	`MEASUrement:IMMed:VALue?`	Query peak-to-peak voltage.	To Be Tested

	`MEASUrement:IMMed:TYPe CRMS`	Set RMS voltage measurement.	To Be Tested
	`MEASUrement:IMMed:VALue?`	Query RMS voltage.	To Be Tested

Findings from SCPI & PyVISA Testing

- The oscilloscope was successfully detected and controlled using PyVISA.
- SCPI commands executed correctly, retrieving voltage, frequency, and waveform data.
- Errors encountered (timeouts, incorrect syntax) were resolved by correcting SCPI syntax based on the oscilloscope manual.
- PyVISA confirmed proper USB-TMC communication, ensuring backend stability.
- Final verification of JSON logs showed properly structured waveform data, confirming the system functions as intended.

Terminal Shell Screenshot



```
^Z
zsh: suspended  sudo python3
davidsmall@iMac ~ % kill %1
davidsmall@iMac ~ %
[1]  - terminated  python3
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/scpi_commands.py'

Oscilloscope ID: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16

Error: VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/scpi_commands.py'

Oscilloscope ID: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16

Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/scpi_commands.py'

Password:
Oscilloscope ID: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16

Testing TIMEBASE?
Error: VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/scpi_commands.py'

Oscilloscope ID: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16

Current Timebase: 5.0E-4

Current CH1 Voltage Scale: 5.0E-1

Updated Timebase: 5.0E-3

Updated CH1 Voltage Scale: 1.0E0

Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/validate_scpi.py'

Password:
Starting SCPI Command Validation...

Testing: Identify Device -> *IDN?
Testing: Horizontal Scale -> HORizontal:MAIn:SCAle?
Testing: Vertical Scale CH1 -> CH1:SCAle?
Testing: Vertical Scale CH2 -> CH2:SCAle?
Testing: Timebase Mode -> HORizontal:MODE?
⚠️Failed: HORizontal:MODE? -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Trigger Level -> TRIGger:LEVel?
⚠️Failed: TRIGger:LEVel? -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Acquisition Mode -> ACQuire:MODe?
Testing: Waveform Data Format -> DATa:ENCdg?
Testing: Waveform Data Source -> DATa:SOUrce?
Testing: Waveform Data Start -> DATa:STARt?
Testing: Waveform Data Stop -> DATa:STOP?
Testing: Measure Frequency -> MEASUrement:IMMed:TYPe FREQ
⚠️Failed: MEASUrement:IMMed:TYPe FREQ -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Measure Period -> MEASUrement:IMMed:TYPe PERIOD
⚠️Failed: MEASUrement:IMMed:TYPe PERIOD -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Measure Peak-to-Peak -> MEASUrement:IMMed:TYPe PK2PK
⚠️Failed: MEASUrement:IMMed:TYPe PK2PK -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Measure RMS -> MEASUrement:IMMed:TYPe CRMS
⚠️Failed: MEASUrement:IMMed:TYPe CRMS -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Set Horizontal Scale -> HORizontal:MAIn:SCAle 5E-3
⚠️Failed: HORizontal:MAIn:SCAle 5E-3 -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Testing: Set CH1 Voltage Scale -> CH1:SCAle 1
⚠️Failed: CH1:SCAle 1 -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
```

```
davidsmall@iMac ~ % >....
Password:
Starting SCPI Command Validation...

Testing: Identify Device -> *IDN?
Testing: Query Horizontal Scale -> HORizontal:MAIn:SCAle?
Testing: Query Vertical Scale CH1 -> CH1:SCAle?
Testing: Query Vertical Scale CH2 -> CH2:SCAle?
Testing: Query Acquisition Mode -> ACQuire:MODe?
Testing: Query Waveform Data Format -> DATa:ENCdg?
Testing: Query Waveform Data Source -> DATa:SOUrce?
Testing: Query Waveform Data Start -> DATa:STARt?
Testing: Query Waveform Data Stop -> DATa:STOP?
Setting: Set Horizontal Scale -> HORizontal:MAIn:SCAle 5E-3
Setting: Set CH1 Voltage Scale -> CH1:SCAle 1
Setting: Set CH2 Voltage Scale -> CH2:SCAle 1
Setting: Set Trigger Source -> TRIGger:A:SOUrce CH1
Setting: Set Measurement Source -> MEASUrement:IMMed:SOUrce CH1
Checking for waveform data before running measurements...
✅ Signal detected. Running measurement commands.
Measuring: Query Frequency Measurement -> MEASUrement:IMMed:TYPe FREQ
⚠<fe0f> Failed: MEASUrement:IMMed:TYPe FREQ -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Measuring: Query Period Measurement -> MEASUrement:IMMed:TYPe PERIOD
⚠<fe0f> Failed: MEASUrement:IMMed:TYPe PERIOD -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Measuring: Query Peak-to-Peak Voltage -> MEASUrement:IMMed:TYPe PK2PK
⚠<fe0f> Failed: MEASUrement:IMMed:TYPe PK2PK -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
Measuring: Query RMS Voltage -> MEASUrement:IMMed:TYPe CRMS
⚠<fe0f> Failed: MEASUrement:IMMed:TYPe CRMS -> VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.

✅ SCPI Command Validation Completed. Results saved in 'validated_scpi_commands.csv'.
Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/validate_scpi.py'

Password:
Starting SCPI Command Validation...

Testing: Identify Device -> *IDN?
Testing: Query Horizontal Scale -> HORizontal:MAIn:SCAle?
Testing: Query Vertical Scale CH1 -> CH1:SCAle?
Testing: Query Vertical Scale CH2 -> CH2:SCAle?
Testing: Query Acquisition Mode -> ACQuire:MODe?
Testing: Query Waveform Data Format -> DATa:ENCdg?
Testing: Query Waveform Data Source -> DATa:SOUrce?
Testing: Query Waveform Data Start -> DATa:STARt?
Testing: Query Waveform Data Stop -> DATa:STOP?
Setting: Set Horizontal Scale -> HORizontal:MAIn:SCAle 5E-3
Setting: Set CH1 Voltage Scale -> CH1:SCAle 1
Setting: Set CH2 Voltage Scale -> CH2:SCAle 1
Setting: Set Trigger Source -> TRIGger:A:SOUrce CH1
Setting: Set Measurement Source -> MEASUrement:IMMed:SOUrce CH1
Skipped: Set Frequency Measurement -> Measurement testing will be done in a future phase.
Skipped: Query Frequency Measurement -> Measurement testing will be done in a future phase.
Skipped: Set Period Measurement -> Measurement testing will be done in a future phase.
Skipped: Query Period Measurement -> Measurement testing will be done in a future phase.
Skipped: Set Peak-to-Peak Measurement -> Measurement testing will be done in a future phase.
Skipped: Query Peak-to-Peak Voltage -> Measurement testing will be done in a future phase.
Skipped: Set RMS Measurement -> Measurement testing will be done in a future phase.
Skipped: Query RMS Voltage -> Measurement testing will be done in a future phase.

SCPI Command Validation Completed. Results saved in 'validated_scpi_commands.csv'.
Connection closed.
davidsmall@iMac ~ % sudo python3 '/Users/davidsmall/Documents/EE/Pisterzi/Network Analyzer Project/Project Phases/Phase I - Design/3_Hardware-S
oftware Interaction Diagram/code/pyvisa_test.py'

Password:
Checking PyVISA installation and listing available instruments...
Found instruments: ('USB0::1689::867::C064444::0::INSTR',)
Connecting to oscilloscope at USB0::1689::867::C064444::0::INSTR...
Sending *IDN? command to identify the oscilloscope...
Connected to: TEKTRONIX,TDS 1002B,C064444,CF:91.1CT FV:v22.16
Connection closed successfully
```

Key Task 4: Plan Authentication (OAuth-based Security)

Deliverables:

1. Authentication Flow Diagram

The Authentication Flow Diagram illustrates how OAuth-based authentication secures access to the system. It details the user login process, token issuance, validation, and role-based access control. This ensures that only authorized users interact with the system. I will create this by mapping out the authentication process from login request to token verification using a flowchart.

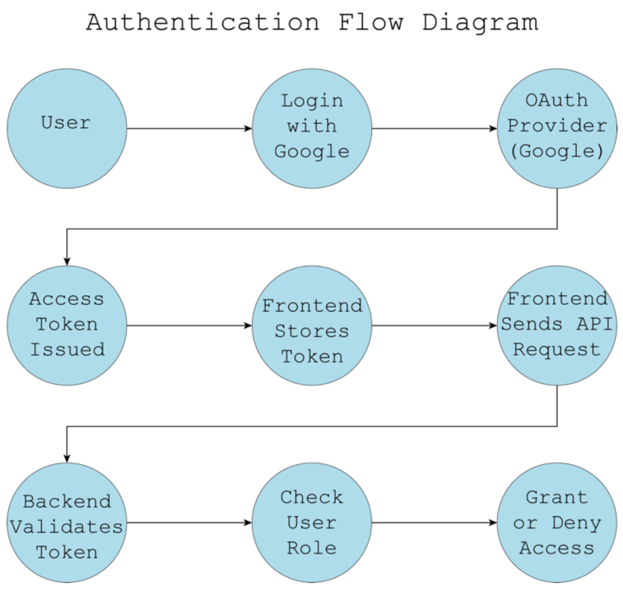
2. Technical Document on OAuth Implementation

The Technical Document on OAuth Implementation provides a detailed guide on integrating OAuth authentication into the system. It includes provider selection, token handling, security considerations, and implementation steps. This ensures secure, scalable authentication across the frontend, API, and backend. I will complete this by documenting the authentication setup, token lifecycle management, and access control policies.

3. User Roles and Permissions Document

The User Roles and Permissions Document outlines different access levels for users (e.g., hiring managers, developers, general users). It defines what each user type can and cannot do within the system. This ensures secure and structured access control. I will complete this by defining role-based access levels and implementing permission logic within OAuth.

Authentication Flow Diagram



Explanation of the Authentication Process

→ Overview:

- The Authentication Flow Diagram illustrates how OAuth authentication secures access to the system.
- It details the user login process, token issuance, validation, and role-based access control.

→ Authentication Steps:

Steps	**Description**
Step 1: User initiates login	The user clicks "Login with Google" in the frontend UI.
Step 2: OAuth provider (Google) handles authentication	Google presents a login screen and verifies user credentials.
Step 3: Google issues an access token	If authentication is successful, an access token is granted to the frontend.
Step 4: Frontend stores the access token	The frontend temporarily holds the token in HTTP-only cookies (for security).
Step 5: Frontend sends API requests with the token	Every API request includes the Authorization: Bearer <TOKEN> header.
Step 6: Backend validates the token	The backend verifies the token against Google's public key to confirm its legitimacy.
Step 7: Backend checks user role	Based on the decoded token, the backend determines if the user is an Admin, Hiring Manager, Friends/Family, or General User.
Step 8: Backend grants or denies access	If the token is valid and the user has appropriate permissions, the request is processed. If not, a 403 Forbidden response is returned.

Token Expiration and Refresh Logic

- Access Token Expiration:
  - The OAuth access token expires after 1 hour
  - After expiration, the user must either log in again or use a refresh token
- Refresh Token Mechanism
  - If refresh tokens are enabled, the frontend can request a new access token without logging in again
  - The backend verifies the refresh token and issues a new access token
- Security Considerations

   - Short-lived tokens (1-hour expiration) enhance security
   - Refresh tokens should only be stored securely (e.g., HTTP-only cookies)
   - Tokens should never be exposed in the frontend JavaScript

Error Handling and Token Validation Failures

→ Common Authentication Errors & How to Handle Them:

Error Type	Cause	Solution
Invalid Token	Token is malformed or tampered with.	Reject request, log error, and prompt re-authentication.
Expired Token	Access token has passed the 1-hour expiration.	Require user to log in again or use refresh token.
Missing Token	API request is missing an Authorization header.	Return 401 Unauthorized and inform the frontend.
Unauthorized Access	User lacks permissions for the requested action.	Return 403 Forbidden and log access attempt.

Final Summary of Authentication Flow

→ Key Takeaways:

- OAuth authentication ensures secure and controlled system access
- Access tokens expire in 1 hour, and refresh tokens allow re-authentication
- Backend verifies tokens and applies user role-based restrictions
- Error handling and security best practices prevent unauthorized access

Technical Document on OAuth Implementation

OAuth Provider Decision (pros/cons of each option)

→ **Decision:** Google OAuth is selected for its ease of integration, security, and support for public testing.

Provider	Pros	Cons
Google OAuth	Easy setup, built-in security, official documentation	Requires a Google account
GitHub OAuth	Developer-friendly, used in CI/CD workflows	Not ideal for public users
Auth0 / Okta	Highly customizable, enterprise-level security	More complex setup

Flow of Authentication:

1. User logs in via OAuth provider (Google).
2. Google authenticates the user and issues an access token.
3. Frontend stores the token temporarily (e.g., localStorage, sessionStorage).
4. Frontend sends the token with API requests.
5. Backend validates the token using Google's public key.
6. Backend checks user role and grants appropriate access.
7. Token expires after 1 hour → User must refresh authentication.

Token Handling & Expiration Policies:

- Access tokens expire after 1 hour.
- Refresh tokens allow users to request new access tokens.
- Token storage:
    - Frontend: Secure HTTP-only cookies (better than localStorage).
    - Backend: No long-term storage of access tokens (only session tracking).
- Revoking Access: Users can revoke OAuth permissions via Google Security.

User Roles and Permissions:

→ How OAuth Roles Are Applied

OAuth-based authentication assigns roles to users based on their credentials and limits access to system resources accordingly. After a user successfully logs in, their role is decoded from the OAuth token and used to enforce Role-Based Access Control (RBAC).

- Role-Based Access Control (RBAC) Mechanism
    - After token validation, the backend checks the user's role before processing API requests
    - The user’s role determines what endpoints they can or cannot access
    - Unauthorized actions result in a 403 Forbidden error.

→ Example: Role-Based API Access Control Logic (Backend)

In the following example, only Admins and Hiring Managers are allowed to generate signals:

```
app.post('/api/signal/generate', authenticateToken, (req, res) => {
    if (req.user.role !== "Admin" && req.user.role !== "HiringManager") {
        return res.status(403).json({ error: "Access Denied" });
    }
    res.json({ message: "Signal generation allowed." });
});
```

Breakdown of Access Control: Admins & Hiring Managers → Can generate signals. Friends/Family & General Users → Cannot generate signals, will receive `403 Forbidden`.

→ How Roles Are Assigned in OAuth
When a user logs in using OAuth, their token includes a custom claim specifying their role. The backend extracts and interprets this role when processing API requests.

```
function authenticateToken(req, res, next) {
    const token = req.headers['authorization']?.split(' ')[1];

    if (!token) return res.status(401).json({ error: "Unauthorized: No token 
provided" });

    jwt.verify(token, process.env.OAUTH_SECRET, (err, user) => {
        if (err) return res.status(403).json({ error: "Forbidden: Invalid 
token" });

        req.user = user;
        next();
    });
}
```

→ Error Handling for Unauthorized Users

Error Type	**Cause**	**Solution**
Invalid Token	Token is malformed or tampered with.	Reject request, log error, and prompt re-authentication.
Expired Token	Access token has passed the 1-hour expiration.	Require user to log in again or use refresh token.
Missing Token	API request is missing an `Authorization` header.	Return `401 Unauthorized` and inform the frontend.
Unauthorized Access	User lacks permissions for the requested action.	Return `403 Forbidden` and log access attempt.

→ Summary: OAuth Role-Based Security
- OAuth ensures only authorized users can access system functionalities.
- RBAC enforces role-specific API access restrictions.
- Unauthorized users receive a 403 Forbidden response.
- Backend extracts roles from the OAuth token and applies access logic dynamically.

Implementation Steps (Frontend and Backend OAuth Setup)

→ Steps to Register Google OAuth:

1. Go to Google Cloud Console.
2. Create a new project.
3. Navigate to APIs & Services → Credentials.
4. Click "Create Credentials" → "OAuth 2.0 Client ID".
5. Set Application Type to "Web Application".
6. Add Authorized Redirect URIs (e.g., https://myapp.com/auth/callback).
7. Save and copy the generated Client ID and Client Secret.

→ Implement OAuth Login in Frontend

- The frontend is responsible for:
    - Redirecting users to Google for login
    - Storing and sending the OAuth token with API requests
- Steps to Implement Google OAuth in Frontend:
    1. Install the required library:

       ```
       npm install react-google-login
       ```

    2. Use the OAuth login button in React:

       ```
       import { GoogleLogin } from 'react-google-login';

       const onSuccess = (response) => {
           console.log("OAuth Token:", response.tokenId);
           localStorage.setItem("access_token", response.tokenId);
       };

       const onFailure = (error) => {
           console.log("Login Failed:", error);
       };

       <GoogleLogin
           clientId="YOUR_GOOGLE_CLIENT_ID"
           buttonText="Login with Google"
           onSuccess={onSuccess}
           onFailure={onFailure}
           cookiePolicy={'single_host_origin'}
       />;
       ```

    3. Store the token in HTTP-only cookies to prevent JavaScript access:

       ```
       document.cookie = `access_token=${response.tokenId}; HttpOnly;
       Secure; SameSite=Strict`;
       ```

→ Validate OAuth Token in Backend

The backend must verify the OAuth token before processing API requests.

- Steps to Validate Token in Backend (Node.js/Express):
    1. Install dependencies:

       ```
       npm install express jsonwebtoken axios
       ```

    2. Create an authentication middleware:

       ```
       const axios = require('axios');
       const jwt = require('jsonwebtoken');

       async function authenticateToken(req, res, next) {
           const authHeader = req.headers['authorization'];
           const token = authHeader && authHeader.split(' ')[1];

           if (!token) return res.status(401).json({ error:
       "Unauthorized: Token required" });

           try {
               // Verify token using Google's public key
               const googlePublicKeys = await
       axios.get("https://www.googleapis.com/oauth2/v3/certs");
               const decoded = jwt.verify(token,
       googlePublicKeys.data.keys[0].n, { algorithms: ["RS256"] });

               req.user = decoded;  // Attach user details
               next();
           } catch (error) {
               return res.status(403).json({ error: "Invalid token" });
           }
       }

       module.exports = authenticateToken;
       ```

    3. Apply authentication to secure API routes:

       ```
       const express = require('express');
       const authenticateToken = require('./authMiddleware');

       const app = express();

       app.get('/api/protected', authenticateToken, (req, res) => {
           res.json({ message: "Secure API data", user: req.user });
       });

       app.listen(3000, () => console.log("Server running on port
       3000"));
       ```

→ Handling Token Expiration and Refresh

OAuth access tokens expire after 1 hour. The system must implement refresh tokens or require the user to log in again.

- Steps to Handle Token Expiration:
    1. On frontend, check token expiration before sending requests:

```
function isTokenExpired(token) {
    const decoded = JSON.parse(atob(token.split('.')[1]));
    return decoded.exp * 1000 < Date.now();
}
```

    2. If token is expired, request a new token (if refresh tokens are enabled):

```
fetch("/api/auth/refresh", {
    method: "POST",
    credentials: "include"
}).then(res => res.json()).then(data => {
    document.cookie = `access_token=${data.token}; HttpOnly;
Secure; SameSite=Strict`;
});
```

    3. Backend route for refresh token handling:

```
app.post('/api/auth/refresh', (req, res) => {
    const refreshToken = req.cookies.refresh_token;
    if (!refreshToken) return res.status(401).json({ error:
"Refresh token required" });

    // Verify and issue new access token
    jwt.verify(refreshToken, process.env.JWT_SECRET, (err, user)
=> {
        if (err) return res.status(403).json({ error: "Invalid
refresh token" });

        const newAccessToken = jwt.sign({ userId: user.id },
process.env.JWT_SECRET, { expiresIn: '1h' });
        res.json({ token: newAccessToken });
    });
});
```

→ Summary: Implementation Steps

Step	Description
1. Register App	Register OAuth app in Google Cloud and obtain credentials.
2. Implement Frontend Login	Use `react-google-login` to integrate Google OAuth.
3. Store Tokens Securely	Store access tokens in HTTP-only cookies.

4. Validate Token in Backend	Verify OAuth token with Google's public key before granting access.
5. Handle Expiration	Refresh tokens expire in 30 days, access tokens expire in 1 hour.
6. Secure API Endpoints	Use `authenticateToken` middleware to protect routes.

Security Considerations

→ Token Storage & Handling

Tokens must be stored securely to prevent exposure to JavaScript and protect against XSS attacks.

- Use HTTP-Only Cookies
    - Access tokens should NEVER be stored in localStorage or sessionStorage
    - HTTP-only cookies prevent JavaScript from accessing tokens, mitigating XSS risks
    - Example of setting an HTTP-only, secure cookie:

```
        document.cookie = `access_token=${token}; HttpOnly; Secure;
SameSite=Strict`;
```

- Set a Short Expiration Time for Tokens
    - Access tokens expire within 1 hour to limit the risk of token theft
    - Refresh tokens expire in 30 days but must only be stored server-side
- Restrict Token Scope & Permissions
    - Assign minimum required permissions to tokens
    - Example: A general user should not have access to admin-level API endpoints
- Ensure Tokens Are Revocable
    - Implement a token blacklist to revoke compromised tokens
    - Example: Store invalidated tokens in a database and deny API access if a token is found in the blacklist

→ OAuth Authentication Security Best Practices

The OAuth flow must be secure against unauthorized access. The following best practices ensure that only legitimate users can authenticate.

- Use PKCE (Proof Key for Code Exchange) for Public Clients
    - Why? PKCE prevents authorization code interception attacks
    - How? It adds a challenge-response mechanism to the OAuth flow
- Enforce Rate Limiting on Login Attempts
    - Why? Prevents brute-force attacks on login endpoints

- How? Limit login attempts to 5 per minute per IP using middleware

```
const rateLimit = require("express-rate-limit");
const loginLimiter = rateLimit({
    windowMs: 1 * 60 * 1000, // 1 minute
    max: 5, // limit each IP to 5 requests per minute
    message: "Too many login attempts. Try again later.",
});

app.post("/api/auth/login", loginLimiter, (req, res) => {
    // Handle login
});
```

- Secure Redirect URIs
  - Ensure OAuth providers ONLY redirect to trusted domains (e.g., https://myapp.com/auth/callback)
  - Never allow wildcard domains (e.g., http://*)
- Enforce HTTPS for All OAuth Requests
  - OAuth MUST NOT be used over HTTP to prevent Man-in-the-Middle (MITM) attacks
  - Enforce HTTPS using Express.js security middleware:

```
const helmet = require("helmet");
app.use(helmet());
```

- Implement Multi-Factor Authentication (MFA) for Admins
  - Admin users MUST be required to authenticate with MFA before performing sensitive actions

→ Preventing Token Leakage & Unauthorized API Access

Once a token is issued, it must only be used by the authorized client. The following security measures prevent token abuse.

- Restrict API Access by User Role
  - Use Role-Based Access Control (RBAC) to limit which users can access which endpoints.
  - Example:

```
app.get('/api/admin/data', authenticateToken, (req, res) => {
    if (req.user.role !== "Admin") {
        return res.status(403).json({ error: "Access Denied" });
    }
    res.json({ message: "Admin data retrieved." });
});
```

- Use Token Revocation for Logouts

    - Upon logout, invalidate the refresh token to prevent reuse
    - Example: Store refresh tokens in a database and remove them on logout
- Log and Monitor API Requests
    - Keep a record of all API authentication attempts for auditing
    - Store logs in a secure logging system (e.g., AWS CloudWatch, ELK Stack)

→ Summary: Security Considerations

Security Measure	**Implementation**
Secure Token Storage	Use HTTP-only cookies for access tokens, never localStorage.
Short Expiry for Tokens	Access tokens expire in 1 hour, refresh tokens in 30 days.
Rate-Limit Logins	Limit 5 login attempts per minute per IP.
Role-Based Access Control	Restrict API access based on user role.
Multi-Factor Authentication	Admins must enable MFA for sensitive operations.
Token Revocation	Remove refresh tokens on logout.
API Logging & Monitoring	Store authentication logs for security audits.

User Roles and Permissions Document

How OAuth Roles Are Applied

OAuth authentication assigns a predefined role to each user after login. The backend validates the role from the OAuth token and enforces access permissions accordingly.

→ Role-Based Access Control (RBAC) Mechanism

- After token validation, the backend checks the user's role before processing API requests.
- The user's role determines what endpoints they can or cannot access.
- Unauthorized actions return a 403 Forbidden response.

→ Key Permissions by Role

- Admin & Hiring Managers → Can generate signals and retrieve oscilloscope data.
- Friends/Family & General Users → Cannot generate signals but can view UI data.

User Roles and Their Access Permissions

User Role	**Description**	**Permissions**	**Restrictions**

Admin (Me)	Full system access, development, and security control	- Can view and edit all system settings. - Can generate and manage signals. - Can access authentication logs. - Can add or remove users.	None
Hiring Managers	Keysight-specific users for testing and recruitment.	Can generate signals and view the Bode plot.	- Cannot modify system settings. - Cannot add/remove users.
Friends & Family	Personal users who can view the interface.	Can access the UI and view data.	Cannot edit configurations or generate signals.
General Users	Public-facing users with the lowest access level.	Can view limited oscilloscope logs.	- Cannot interact with system settings. - Cannot generate signals or modify configurations.

How the Backend Enforces Role-Based Access

The backend verifies the user role from the OAuth token and applies access control rules before allowing API interactions.

→ Example of API Role Enforcement

Below is a conceptual representation of how the system restricts access based on roles:

```
app.get('/api/admin/logs', authenticateToken, (req, res) => {
    if (req.user.role !== "Admin") {
        return res.status(403).json({ error: "Access Denied: Admins only" });
    }
    res.json({ message: "Log data retrieved." });
});
```

Only I (the admin) can access API logs.

How Users Are Assigned Roles

Each user is assigned a role at account creation. The backend retrieves role information when validating an OAuth token.
→ Example OAuth Token Payload:

```
{
    "sub": "1234567890",
    "name": "John Doe",
    "email": "johndoe@example.com",
    "role": "Admin",
    "exp": 1717356522
}
```

The "role" field determines user permissions.
If the role is missing, the system defaults to General User (lowest access level).

Error Handling for Unauthorized Users
If a user attempts to access an unauthorized resource, the backend enforces security by returning proper HTTP responses.

Unauthorized access is strictly enforced through OAuth-based role validation. If a user lacks the necessary role permissions, the backend responds with a `403 Forbidden` error, preventing privilege escalation. Additionally, authentication failures return `401 Unauthorized` if the token is missing or invalid. All failed access attempts are logged for security monitoring.

Key Task 5: Plan Kubernetes and CI/CD Integration

Deliverables:

0. CI/CD Workflow Diagram

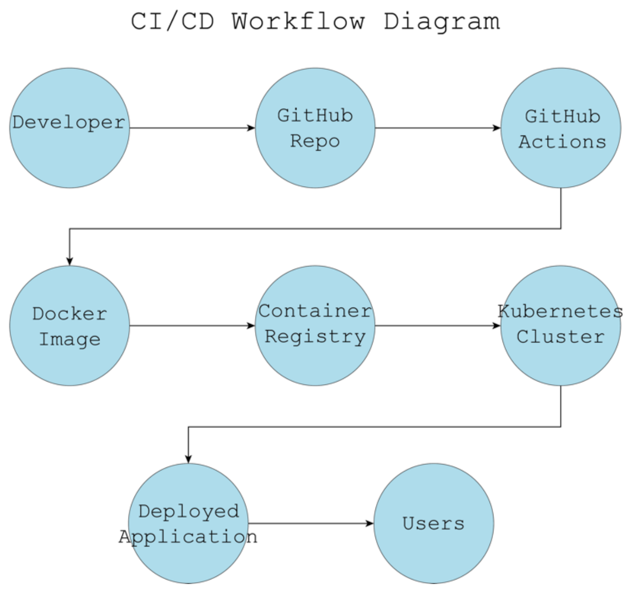
The CI/CD Workflow Diagram visually represents the automated build, testing, and deployment pipeline for the project. It details how code changes are tested, merged, and deployed using GitHub Actions and Kubernetes. This ensures continuous integration, reliability, and automated deployments. I will complete this by mapping out each CI/CD stage, including triggers, testing steps, and deployment actions.

1. Kubernetes Deployment Plan

The Kubernetes Deployment Plan details how the system will be deployed using Kubernetes, including pods, services, namespaces, and security policies. It ensures scalability, automated recovery, and efficient resource management. This plan will include cluster architecture, resource allocation, and CI/CD integration. I will complete this by configuring Kubernetes manifests, defining deployment strategies, and securing access.

CI/CD Workflow Diagram

The CI/CD Workflow Diagram visually represents the automated build, testing, and deployment pipeline. It shows how GitHub Actions integrates with Docker and Kubernetes for continuous deployment.



Overview of the CI/CD Pipeline

- Goal: Automate build, testing, and deployment.

- Pipeline Tools: GitHub Actions, Docker, Kubernetes.
- Stages:
    1. Code Commit & Push: Developers push code to GitHub.
    2. Build Stage:
        - GitHub Actions triggers Docker image creation.
        - The image is tagged and pushed to a container registry.
    3. Testing Stage:
        - Unit tests and integration tests run in a container.
        - Security scans check for vulnerabilities.
    4. Deployment Stage:
        - Kubernetes manifests are applied to the cluster for rolling updates.

→ CI/CD Workflow Diagram Labels & Clarifications Table

Component	**Clarification/Label to Add**
GitHub Actions	Build & Test: Runs automated tests before creating the Docker image. *Testing is part of the workflow.*
Docker Image	Image Build: Converts the application into a containerized format. *Clarifying its purpose.*
Container Registry	Stores the built Docker image for deployment. *Reinforcing its role.*
Kubernetes Cluster	Deploy Containers: Pulls images from the container registry and runs them in pods. *Explaining that Kubernetes pulls and runs the application.*
Deployed Application → Users	Application is live and accessible to end users. *Emphasizing final accessibility.*

CI/CD Pipeline Stages & Example Configurations

1. Triggers
    - The pipeline runs when:
        - Code is pushed to main
        - Pull requests are merged
        - A scheduled nightly build executes

Example GitHub Actions trigger:

```
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  schedule:
    - cron: '0 2 * * *'  # Run at 2 AM UTC daily
```

2. Build Stage
    - Builds Docker container
    - Tags image with commit hash
    - Pushes image to container registry

Example Docker Build Step:

```
- name: Build Docker Image
  run: docker build -t my-app:${{ github.sha }} .
```

Example Docker Push to AWS ECR:

```
- name: Push Image to AWS ECR
  run: |
    docker tag my-app:${{ github.sha }} my-ecr-repo/my-app:${{ github.sha }}
    docker push my-ecr-repo/my-app:${{ github.sha }}
```

3. Testing Stage
    - Runs automated tests in container
    - Security scans for vulnerabilities

Example Test Execution in Docker:

```
- name: Run Tests
  run: docker run --rm my-app:${{ github.sha }} pytest
```

Example Security Scan:

```
- name: Scan Image for Vulnerabilities
  run: docker scan my-app:${{ github.sha }}
```

4. Deployment Stage
    - Deploys to Kubernetes
    - Uses kubectl apply to update pods

Example Kubernetes Deployment Step:

```
- name: Deploy to Kubernetes
```

```
  run: kubectl apply -f k8s/deployment.yaml
```

Kubernetes Deployment Plan

The Kubernetes Deployment Plan defines how the system runs inside a Kubernetes cluster. It ensures scalability, automated recovery, and resource efficiency.

Overview of Kubernetes Deployment

- Cluster Architecture:
  - Workloads run in pods.
  - Services expose the application.
  - Namespaces separate environments (production, testing, development).
- Deployment Strategies:
  - Rolling Updates: New pods replace old ones gradually.
  - Blue-Green Deployment: New version runs parallel to minimize downtime.

Kubernetes Architecture Components

1. Pods: Containerized Application Units
   - Each pod contains one or more Docker containers.
   - Each pod gets a unique IP inside the cluster.

Example Pod (YAML):

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: app-container
      image: my-app:latest
```

2. Services: Exposing the Application
   - ClusterIP: Internal service for backend communication.
   - LoadBalancer: Publicly exposes the application.

Example Service YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

3. Namespaces: Environment Isolation
   - **Prevents conflicts** between production and testing.

Example Namespace (YAML):

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

Hosting Environment Selection

Comparison of AWS, Firebase, and Self-Hosting

Option	Pros	Cons
AWS EKS	Scalable, integrated security (IAM, RBAC)	Higher cost
Firebase	Serverless, auto-scaling	Limited Kubernetes support
Self-Hosting	Full control, no vendor lock-in	Higher maintenance

Decision: AWS EKS selected for scalability and security.

Security Measures for Kubernetes Deployment

1. Role-Based Access Control (RBAC)

   - Defines **permissions for users/services**.

Example Role YAML:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: production
  name: developer
rules:
  - apiGroups: [""]
```

```
    resources: ["pods", "services", "deployments"]
    verbs: ["get", "list", "create", "update"]
```

2. Network Policies for Security
    - Restricts which pods can talk to each other.

Example Network Policy (YAML):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-external-traffic
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: my-app
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: backend
```

3. Container Security Best Practices
    - Run containers as non-root users:

```
securityContext:
  runAsUser: 1000
  runAsGroup: 1000
  readOnlyRootFilesystem: true
```

- Scan images for vulnerabilities before deployment:

```
docker scan my-app:latest
```

Key Task 6: GitHub Version Control and Documentation

Deliverables:

1. GitHub Repository Setup

The GitHub Repository Setup establishes the version control structure for the project. It includes directories for backend, frontend, test scripts, and documentation while enforcing branching and commit guidelines. This ensures organization, collaboration, and code integrity. I will complete this by creating the repository, setting up folder structures, and enforcing version control best practices.

2. Initial README file

The Initial README.md serves as the entry point for developers, providing an overview of the project, setup instructions, and contribution guidelines. It ensures that new contributors can quickly understand and get started with the project. I will complete this by writing a structured README that includes project goals, system architecture, installation steps, and usage instructions.

GitHub Repository Setup

Check GitHub:
https://github.com/alexandersmall-ee/network-analyzer

Initial README file

Check GitHub:
https://github.com/alexandersmall-ee/network-analyzer