2/10/25 Entry

Project Initialization

Halfway through this phase I had to add other key tasks and restructure key tasks to ensure clarity and technical precision. One major change was separating API and backend communication from system architecture. Initially, API design was grouped under system architecture, but I realized that defining the RESTful API structure, endpoint specifications, and backend responsibilities required a narrower focus. This narrow focus ensures clear documentation of how Node.js handles requests, interacts with the oscilloscope via PyVISA, and manages data storage.

With API concerns now isolated, the system architecture task could focus solely on the high-level structure of the project. This includes defining how the frontend (Angular and Chart.js), backend (Node.js), database (JSON), and hardware (oscilloscope via USB-TMC) interact. The technical design document (TDD) now emphasizes the overall system blueprint, while API details remain in their own section. Additionally, I expanded hardware-software interaction planning by documenting USB-TMC communication, SCPI command execution, and oscilloscope response expectations. This ensures the system can reliably retrieve and process measurement data while also being adaptable for future test equipment.

The CI/CD and Kubernetes planning also needed refinement. Initially, Kubernetes was loosely grouped with CI/CD, but I needed to explicitly define how services, pods, namespaces, and replica sets will be managed, along with evaluating AWS, Firebase, or a self-hosted cluster for deployment. By isolating this planning, I can better implement RBAC security, automated testing workflows in GitHub Actions, and proper container management strategies. These changes clarify how updates and deployments will be handled while ensuring scalability.

Security and authentication planning also required additional structure. Instead of a general security section, I created a dedicated OAuth implementation plan, outlining OAuth provider selection (Google, GitHub, etc.), authentication flow, and user roles and permissions. This ensures clear role-based access for hiring managers, developers, and general users, while also securing frontend-backend communication via token verification and expiration strategies. Properly documenting this now will streamline the authentication process during implementation.

These refinements provide a logical breakdown of responsibilities, ensuring each system component is properly planned and documented. API and backend tasks are

now clearly distinguished, hardware integration is more structured, Kubernetes deployment is well-defined, and authentication is securely outlined. This restructuring enhances clarity, maintainability, and scalability, setting a strong foundation for the next phases of development.

Moving forward, I am introducing **Key Focus Areas** for each phase to provide a structured approach to planning and execution. These lists were not included in the Finalized Plan. This will ensure that every stage of development remains organized and that critical components are properly addressed. I will continue this format for all remaining phases of the project.

And with that, we begin…

Below are the Key Focus areas for Phase I.

Key Focus Areas for Phase 1

1. System Architecture and Design
2. API and Backend Communication
3. OAuth-based Authentication
4. Hardware-Software Interaction Planning
5. SCPI Commands and PyVISA Implementation
6. Kubernetes and CI/CD Considerations
7. Formal Documentation and Diagrams

Restructured Key Tasks, What to Document, and Deliverables

Key Task 1: Define System Architecture

Objective: Establish the structure of the backend, frontend, and database for how the system components will interact.

Subtasks:

- Design the backend architecture using Node.js with a RESTful API
- Design the frontend architecture using Angular and Chart.js for data visualization
- Establish the data storage format using JSON
- Plan hardware interaction points, focusing on USB-TMC and PyVISA integration
- Identify technology stack and dependencies, justifying each selection

Details:

This phase ensures a structured system that effectively integrates frontend, backend, and hardware components. By defining technology choices early, I can streamline implementation and avoid compatibility issues. Additionally, focusing on data flow and communication protocols between these components prevents architectural bottlenecks later in development.

What to Document:

- System architecture diagram (backend, frontend, hardware interactions)
- API and framework decisions (Node.js, Angular, JSON, Chart.js)
- Justification for technology choices

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.

<u>Key Task 2: Define API and Backend Communication</u>

Objective: Define the API's structure and how the backend interacts with the oscilloscope and frontend.

Subtasks:

- Design RESTful API endpoints for signal generation and oscilloscope data retrieval
- Define data handling and JSON storage strategy
- Evaluate WebSocket integration for real-time data streaming
- Ensure secure API access using OAuth token verification

Details:

The API is a critical component that links the frontend and backend while interfacing with the oscilloscope. A well-defined API structure ensures smooth data retrieval and command execution. If real-time responsiveness is needed, WebSockets may supplement REST calls. Proper authentication and request validation are crucial to maintaining security.

What to Document:

- RESTful API specifications (endpoints, request/response formats)
- Backend interface with the oscilloscope and Python-generated signals
- Data structure for oscilloscope readings and system logs
- Justification for potential WebSocket integration

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.

Key Task 3: Define Hardware-Software Interaction

Objective: Ensure smooth communication between the software system and oscilloscope.

Subtasks:

- Document USB-TMC communication setup with necessary drivers
- Define SCPI command structure for oscilloscope communication
- Implement Python-based signal generation using SciPy and Sounddevice
- Develop test scripts using PyVISA to verify oscilloscope responses

Details:

Proper hardware integration ensures the oscilloscope receives commands and returns accurate data. The USB-TMC protocol enables communication, while SCPI commands allow precise control over measurements. A structured test plan using PyVISA will confirm that the system can properly communicate with the oscilloscope.

What to Document:

- Hardware-software interaction plan
- USB-TMC setup process and required drivers
- SCPI command list and expected oscilloscope responses
- Alternative signal generation methods if Python cannot reach 300 kHz

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.

2. PyVISA Test Script

The PyVISA Test Script is a Python-based script used to validate communication with the oscilloscope via USB-TMC. It will send basic SCPI commands, retrieve responses,

and confirm data integrity. This ensures that the backend correctly interfaces with the oscilloscope. I will complete this by writing and running test scripts that execute key SCPI commands, ensuring the oscilloscope responds as expected.

3. SCPI Commands List

The SCPI Commands List documents the standardized commands used to control and retrieve data from the oscilloscope. It includes setup, measurement retrieval, and signal processing commands. This ensures that all system interactions with the oscilloscope follow a structured command format. I will complete this by identifying, testing, and documenting the necessary SCPI commands based on oscilloscope specifications.

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

Objective: Secure system access and define authentication mechanisms.

Subtasks:

- Select an OAuth provider (Google, GitHub, etc.)
- Design authentication flow, including token handling and expiration policies
- Define user roles and permissions (admin, hiring managers, general users)

Details:

A robust authentication system is necessary to prevent unautorized access. OAuth ensures that users securely log in and access the system while limiting permissions based on predefined roles. Proper documentation will ensure that the frontend, backend, and API integrate OAuth seamlessly.

What to Document:

- OAuth provider decision (pros/cons of each option)
- Authentication flow diagram
- User roles and permissions structure
- How the frontend and backend handle token verification

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.

Key Task 5: Plan Kubernetes and CI/CD Integration

Objective: Automate deployment and infrastructure management.

Subtasks:

- Design CI/CD pipeline using GitHub Actions for automated testing
- Define Kubernetes deployment architecture (pods, services, namespaces)
- Evaluate hosting options (AWS, Firebase, or self-hosted cluster)
- Implement security best practices (Role-based access control or RBAC, container isolation)

Details:

A scalable and automated deployment strategy ensures seamless updates and efficient testing. Kubernetes provides a flexible infrastructure for managing services and ensures reliability across multiple environments. Security considerations such as RBAC and namespace isolation will prevent unauthorized access.

What to Document:

- 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

Deliverables:

1. 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.

2. 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.

Key Task 6: GitHub Version Control and Documentation

Objective: Set up a structured repository with clear documentation.

Subtasks:

- Establish GitHub repository structure
- Create an initial README file with an overview and setup guide
- Define version control guidelines (branching, strategy, commit conventions)

Details:

A well-structured repository improves collaboration and maintainability. The README serves as an entry point for developers, while the version control strategy ensures smooth contributions.

What to Document:

- Repository structure and organization
- Initial README file with project goals, architecture, and setup instructions
- Guidelines for version control and collaboration

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.

Task Execution Process

Key Task 1: Define System Architecture

Subtasks:

- Design the backend architecture using Node.js with a RESTful API

Design a Node.js-based RESTful API that acts as an intermediary between the frontend (Angular + Chart.js), the backend (Python for signal processing), and the oscilloscope (via PyVISA and USB-TMC). The API will be responsible for handling requests from the frontend, processing data in Python, and managing authentication via OAuth. This subtask focuses on structuring the Node.js API layer and its integration with the core Python backend ensuring secure, efficient, and scalable communication across all system components.

→ Backend Architecture Design Overview (Components Involved)

The backend architecture consists of two primary components:

1. Node.js (Express.js) API Layer → Handles RESTful API requests, manages authentication (OAuth), and acts as a bridge between the frontend and backend.
2. Python Backend → Processes signal generation, interacts with the oscilloscope via PyVISA, and executes SCPI commands.

This structure ensures modularity and scalability by separating API handling from hardware-specific processing.

→ Core Responsibilities of the API (What Does the API Do?)

- Act as a gateway between the frontend and backend
- Receive requests from the frontend (e.g., signal generation parameters, oscilloscope commands)
- Forward requests to the Python backend for processing
- Handle responses and send structured JSON data back to the frontend
- Implements OAuth authentication and user role management
- Logs API activity for debugging and monitoring purposes

→ Core Responsibilities of the Backend (What Does the Backend Do?)

- Process requests from the API
- Communicate with the oscilloscope
- Generate and process signals
- Format data for API response
- Implement error handling and logging

- Ensure efficient execution

→ High-Level Architecture Flow Diagram (How the Components Interact)

Frontend (Angular) → REST API Request → Node.js API → Process Request → Python Backend → Send Data → Oscilloscope (PyVISA)

- Frontend Requests → Users interact with the Angular UI to request data (e.g., generate a signal, retrieve oscilloscope readings)
- API Routes and Controllers → The request is sent to the Node.js RESTful API via predefined endpoints (e.g., `/api/signal/generate`)
- Backend Processing (Python) → The Node.js API calls the Python backend using an internal process (e.g., `child_process.spawn` or a WebSocket connection)
- Communication with the Oscilloscope → The Python backend uses PyVISA and SCPI commands to interact with the oscilloscope and retrieve data
- Response Handling → The Python backend sends data back to Node.js, which then formats and returns it as JSON to the frontend
- Authentication and Security → OAuth authentication verifies user access before requests are processed

→ Backend Architecture Components (Breaks Down API Structure, Python Communication, Security, And Logging)

- API Structure (Node.js and Express.js)
    - The API will be built using Express.js for efficient request handling
    - All requests will follow RESTful API conventions
    - API endpoints will be structured as follows:

```
/api
├── /auth                → OAuth authentication
├── /users               → User management (roles, permissions)
├── /signal/generate     → Sends signal generation requests to backend
├── /oscilloscope/data   → Retrieves oscilloscope measurement data
├── /logs                → API logging (requests, errors)
├── /config              → Stores user-defined oscilloscope and signal
│                          settings
```

- Backend Signal Processing and Hardware Communication (Python)
    - The Python backend processes oscilloscope communication and signal generation
    - It interacts with the oscilloscope using PyVISA and SCPI commands

    - It receives requests from the API, executes commands, and returns processed data

```
/backend
├── /scripts               # Python scripts for signal processing &
│                            oscilloscope control
│   ├── signal_gen.py      # Generates signals for testing
│   ├── oscilloscope.py    # Communicates with oscilloscope via PyVISA
│   ├── scpi_commands.py   # SCPI command definitions
│   ├── data_processing.py # Parses oscilloscope data
├── /tests                 # Unit tests for backend functionality
│   ├── test_signal_gen.py
│   ├── test_oscilloscope.py
├── main.py                # Entry point for executing backend processes
├── requirements.txt       # Python dependencies (PyVISA, NumPy, Flask, etc.)
```

- Communication Between API and Python Backend
    - The Node.js API will call the Python backend via:
        - Child Process (child_process.spawn) for executing Python scripts
        - WebSockets or HTTP requests (Flask API) if real-time streaming is needed
- Data Handling & JSON Response
    - API will format responses into structured JSON

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

- Authentication and Security (OAuth)
    - All requests will be authenticated using OAuth tokens before being processed
    - Middleware will verify user roles and permissions before executing requests
- Logging and Monitoring
    - API requests, errors, and oscilloscope interactions will be logged for debugging
    - Node.js will include an error-handling middleware to catch and log issues

→ Backend Workflow (Detailed Step-by-step Execution of API Operations)

- The frontend (Angular + Chart.js) sends a request to the Node.js API to generate a signal or retrieve oscilloscope data
- The Node.js API routes the request to the Python backend via:
    - Child Process (`child_process.spawn`) for executing Python scripts
    - WebSockets or HTTP requests (Flask API) if real-time streaming is needed
- The Python backend processes the request, interacts with the oscilloscope via PyVISA, and retrieves measurement data
- The Python backend returns the results to the Node.js API, which formats them as JSON and sends them to the frontend
- OAuth authentication ensures only authorized users interact with the API

→ Backend Workflow (Detailed Step-by-step Execution of Python Backend Operations)

- Receiving requests from the Node.js API
    - The Python backend listens for requests from the API
    - API sends a POST request for signal generation or a GET request for oscilloscope data
    - Request parameters (e.g., frequency, amplitude, waveform type) are extracted from the API payload
- Validating request data
    - The backend checks for missing or invalid parameters
    - If the request is invalid, the backend returns a structured error response to the API:

```
{
  "status": "error",
  "message": "Invalid frequency range"
}
```

    - If the request is valid, the backend proceeds with execution
- Handling signal generation requests
    - The backend uses NumPy/SciPy to generate the requested waveform
    - The generated signal is converted into a format compatible with the oscilloscope
    - If an external USB-based function generator is used, the backend sends SCPI commands to configure the device accordingly
- Communicating with the oscilloscope via PyVISA
    - The backend establishes a connection with the oscilloscope using PyVISA and USB-TMC

    - A SCPI command is sent to the oscilloscope to retrieve measurement data
    - The oscilloscope returns a raw measurement value, which the backend parses and processes
- Processing and formatting data for API response
    - The retrieved measurement data is formatted into a JSON-compatible structure
    - Example:

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

    - The formatted response is sent back to the API, which forwards it to the frontend
- Error handling and logging
    - The backend logs all commands sent to the oscilloscope for debugging purposes
    - If a hardware failure or SCPI error occurs, it returns a structured error response and logs the issue

→ Backend Implementation Strategy

- Step 1: Set Up Node.js and Express.js
    - Initialize a Node.js project and install Express.js
    - Configure middleware for CORS, logging, and OAuth authentication
    - Define API route handlers for structured request processing
- Step 2: Define API Routes and Backend Communication
    - Implement RESTful routes for authentication, signal generation, and oscilloscope data retrieval
    - Use child process or WebSockets to execute Python scripts from Node.js
    - Format Python responses as structured JSON for frontend consumption
- Step 3: Implement OAuth Authentication
    - Integrate OAuth authentication middleware to validate API requests
    - Define user roles and permissions for controlled access
- Step 4: Establish Error Handling & Logging
    - Implement structured API logging for debugging requests and errors

    - Define a unified error-handling mechanism for API failures
- Step 5: Test API Functionality
    - Use Postman or cURL to test endpoints
    - Validate communication between Node.js, Python, and the oscilloscope

2/11/25 Entry

- Design the frontend architecture using Angular and Chart.js for data visualization

This subtask focuses on designing the frontend architecture using Angular for the UI framework and Chart.js for real-time data visualization. The frontend will interact with the Node.js RESTful API, allowing users to configure signal parameters, retrieve oscilloscope data, and visualize measurement results dynamically. The goal is to ensure a responsive, efficient, and intuitive user interface that integrates seamlessly with the backend.

→ Frontend Architecture Overview (Components Involved)
The frontend architecture consists of the following components:

1. Angular Framework → Manages the frontend UI, component structure, and API interactions
2. Chart.js → Handles data visualization for amplitude and phase measurements
3. REST API Communication → Uses HTTP requests to interact with the Node.js API
4. State Management → Manages application state (e.g., selected signals, oscilloscope settings, and retrieved data)
5. User Authentication (OAuth) → Ensures secure access to system features based on user roles

This structure ensures modularity, scalability, and ease of maintenance while providing real-time visualization of oscilloscope data.

→ Core Responsibilities of the Frontend (What Does the Frontend Do?)

- Provide a UI for signal generation controls (frequency, amplitude, waveform selection, etc.)
- Display real-time oscilloscope data (amplitude, phase, time-domain plots) using Chart.js
- Send API requests to control signal generation and retrieve oscilloscope data
- Parse and display oscilloscope data received from the backend
- Manage user authentication via OAuth (Google, GitHub, or another provider)

- Ensure responsive and dynamic UI interactions

→ High-Level Frontend Workflow (How the Components Interact)

User Interaction → UI Component → API Request (Angular HTTP Client) → Node.js API → Python Backend → Oscilloscope → Response Sent Back to UI → Chart.js Updates Visualization

- User selects signal parameters (e.g., frequency, amplitude, waveform) from the UI
- Angular sends an API request to Node.js to generate the signal
- Node.js forwards the request to the Python backend, which interacts with the oscilloscope via PyVISA
- Python backend retrieves oscilloscope measurements and sends them to Node.js
- Node.js API formats the data into JSON and sends it back to Angular
- Chart.js updates the UI with the real-time measurement data

→ Frontend Architecture Components (Breaks Down UI Structure, API Communication, and Visualization)

- Angular Project Structure

```
/frontend
 ├── /src
 │    ├── /app
 │    │    ├── /components   # UI Components (signal controls, charts, etc.)
 │    │    ├── /services     # API Communication (REST requests)
 │    │    ├── /pages        # Main pages (Dashboard, Settings, Authentication)
 │    │    ├── /models       # Interfaces for handling data types
 │    │    ├── app.module.ts # Main Angular module
 │    │    ├── app.component.ts # Root component
 ├── angular.json           # Angular project configuration
 ├── package.json           # Dependencies (Angular, Chart.js, OAuth)
 ├── README.md              # Frontend documentation
```

- API Communication (Angular HTTP Client)
    - Uses Angular's HTTP Client module to interact with the Node.js REST API
    - Handles POST requests for signal generation and GET requests for oscilloscope data
- Data Visualization with Chart.js
    - Displays real-time oscilloscope data (waveforms, amplitude, phase response)

    - Uses line charts for continuous signal plotting
- Authentication and Security (OAuth Integration)
    - Manages user login and authentication using OAuth
    - Controls access permissions based on user roles
- State Management
    - Stores selected signal parameters and retrieved oscilloscope data for a seamless user experience

→ Frontend Workflow (Detailed Step-by-step Execution of Frontend Operations)

- User Interaction with the UI
    - The user selects signal parameters from the UI (e.g., frequency, amplitude, waveform type)
    - If the user modifies existing settings, the UI updates the state in real time
    - The user submits the request to generate a signal or retrieve oscilloscope data
- Angular Processes the User Input
    - The selected parameters are stored in Angular's component state
    - Before sending the request, the frontend validates the inputs to prevent errors (e.g., frequency out of range)
- Sending Requests to the Node.js API
    - Angular's HTTPClientModule sends a POST request to the API for signal generation or GET request for oscilloscope data
    - Example POST request to generate a signal:

```
this.http.post('/api/signal/generate', {
  frequency: 1000,
  amplitude: 5.0,
  waveform: "sine"
}).subscribe(response => {
  console.log(response);
});
```

    - The request includes authentication tokens (OAuth) to ensure secure access
- Waiting for the API response
    - While waiting for the backend to process the request, the UI displays a loading indicator
    - If an error occurs (e.g., invalid parameters, API timeout), the UI shows an error message
- Receiving and Processing API Data

    - The Node.js API sends a JSON response containing oscilloscope measurement data
    - Angular receives the response and parses the JSON into structured data
    - Example JSON response from API

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

    - If the response contains measurement data, the frontend prepares the data for visualization
- Updating the UI and Chart.js Visualization
    - Chart.js updates the graph with the new measurement data
    - The UI dynamically adjusts to reflect the received oscilloscope values in real time
    - The user can now view signal characteristics (e.g., waveform shape, amplitude, phase response)
- Error Handling and User Feedback
    - If authentication fails, the user is redirected to the OAuth login page
    - If there is no response from the backend, the UI displays an error message
    - All errors are logged in the browser console and can be reported via an optional logging service

→ Frontend Implementation Strategy

- Step 1: Set up Angular project and install dependencies
- Step 2: Define UI components and page structure
    - Create dashboard components for user interaction (signal settings, live data visualization
    - Implement responsive UI using Angular material (e.g., MatButtonModule for buttons, MatFormFieldModule for input fields, MatProgressSpinner for loading states)
- Step 3: Configure API communication
    - Use Angular HTTP Client Module for GET and POST requests to interact with the backend API
    - Implement services to fetch oscilloscope data and send control parameters

- Step 4: Integrate Chart.js for data visualization
    - Define real-time graph updates based on oscilloscope data
    - Implement configurable chart options for amplitude, phase, and frequency plots
- Step 5: Implement OAuth authentication
    - Integrate OAuth login flow using a provider like Google or GitHub
    - Ensure tokens are stored and validated before making API requests
- Step 6: Test Frontend Functionality
    - Use mock API responses to verify UI behavior
    - Test real-time visualization using mock data, followed by live API calls to verify oscilloscope integration

2/12/25 Entry

- Establish the data storage format using JSON

This subtask focuses on defining the data storage format using JSON for managing oscilloscope measurement data, user configurations, and system logs. JSON (JavaScript Object Notation) is chosen for its lightweight structure, human-readability, and compatibility with the frontend (Angular), backend (Node.js and Python), and external storage systems if needed in future expansions.

The goal is to standardize JSON formats for data exchange between system components, ensuring efficiency, consistency, and scalability across the project.

→ JSON Architecture Overview

The JSON architecture consists of primary storage and processing points:

1. Frontend (Angular + Chart.js)
    - Stores temporary oscilloscope data for real-time visualization
    - Saves user-defined signal configurations for quick retrieval
    - Parses and processes JSON responses from API before displaying measurement data
2. Backend (Node.js + Python)
    - Formats measurement data into JSON before sending it to the frontend
    - Receives JSON-encoded user input (e.g., signal generation parameters)
    - Stores log entries in structured JSON format
3. External JSON Storage (Logs and Future Expansions)

- System logs (errors, API requests) are stored in JSON-based log files
- JSON may later be exported to a database or cloud storage if needed

→ Data Storage Needs and Use Cases

The system will use JSON for three primary types of data storage:

1. Oscilloscope Measurement → Stores signal readings, timestamps, and metadata
2. User Configurations → Stores signal generation settings (e.g., frequency, amplitude, waveform type)
3. System Logs → Tracks API requests, error logs, and backend processing details

This structure ensures that data remains accessible, organized, and easy to parse across different system layers.

→ Core Responsibilities of JSON Data Storage

- Store oscilloscope readings in a structured format for visualization
- Maintain user-defined signal generation parameters for persistence
- Ensure JSON data is structured consistently across frontend and backend
- Support logging of API interactions, backend processing, and errors
- Facilitate easy data retrieval and parsing by the frontend and backend

→ JSON Workflow

1. JSON Creation (User Input and Signal Configurations)
   - The user configures signal settings (e.g., frequency, amplitude, waveform type)
   - Angular converts user input into a JSON request object
   - Example JSON request from the frontend:

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

2. JSON Processing in the Backend
   - The Node.js API receives JSON data and validates it

- If valid, the API forwards the JSON request to the Python backend
- The Python backend processes the request, generates the signal, and sends a response

3. JSON Response Handling
   - The backend formats measurement data into JSON before returning it
   - Example JSON response from backend to frontend:

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

   - The Angular frontend parses the JSON, updating the UI and visualization graphs
4. JSON Logging and Storage
   - API and backend store logs as structured JSON for debugging
   - Example JSON log entry

```
{
  "log_id": "log_20250210_001",
  "timestamp": "2025-02-10T14:30:10Z",
  "level": "error",
  "component": "backend",
  "message": "SCPI command timeout",
  "details": {
    "command_sent": "MEASure:VPP?",
    "error_code": 408
  }
}
```

   - Logs help track errors and API interactions

→ JSON Schema for Each Data Type

1. Oscilloscope Measurement Data (Real-Time and Stored Data)
   - This JSON object represents a single oscilloscope reading retrieved in real-time or stored for later use

```
{
  "status": "success",
  "timestamp": "2025-02-10T14:23:45Z",
  "data": {
```

```
    "frequency": 1000,
    "amplitude": 5.0,
    "phase": 0.5,
    "waveform": "sine",
    "unit": "V"
  }
}
```

2. User Configuration Data (Signal Generation Settings)
    - Stores user-defined signal presets for quick reuse

```
{
  "user_id": "abc123",
  "saved_presets": [
    {
      "preset_name": "Test Signal 1",
      "parameters": {
        "frequency": 500,
        "amplitude": 3.0,
        "waveform": "square",
        "duty_cycle": 50
      }
    }
  ]
}
```

3. System Logs (API Requests, Errors, Backend Processing)
    - Captures logs for debugging API and backend operations

```
{
  "log_id": "log_20250210_001",
  "timestamp": "2025-02-10T14:30:10Z",
  "level": "error",
  "component": "backend",
  "message": "SCPI command timeout",
  "details": {
    "command_sent": "MEASure:VPP?",
    "error_code": 408
  }
}
```

→ Where JSON is Used in System Architecture

- Frontend Usage (Angular)
    - Stores temporary oscilloscope readings for visualization in Chart.js
    - Saves user-defined signal configurations locally for quick access
- Backend Usage (Node.js API and Python)
    - Formats measurement data into JSON before sending it to the frontend

    - Logs API interactions and backend processing errors in structured JSON files
- Future Scalability
    - JSON can be stored in a database or a file-based logging system for long-term storage

→ JSON Data Flow Across System Components

User Interaction → Frontend (Angular) → API Request (JSON Format) → Backend (Node.js → Python) → Oscilloscope → Data Response (JSON Format) → Chart.js Updates UI

→ JSON Implementation Strategy
- Step 1: Define JSON Data Structures
- Step 2: Implement JSON Handling in Backend
- Step 3: Store JSON Data Efficiently
- Step 4: Validate JSON Exchange Between Frontend and Backend
- Step 5: Test JSON Data Flow in the System

- Plan hardware interaction points, focusing on USB-TMC and PyVISA integration

This subtask focuses on establishing the hardware interaction points for the network analyzer system, specifically how the backend will communicate with the oscilloscope using USB-TMC (USB Test & Measurement Class) and PyVISA. The oscilloscope will be controlled via SCPI (Standard Commands for Programmable Instruments) commands, which allow the system to send queries and retrieve measurement data.

The goal is to define a structured approach for hardware communication, ensuring reliable data retrieval, minimal latency, and seamless integration between software and hardware components.

→ Hardware Interaction Overview

The hardware interaction consists of two key components:
1. USB-TMC Protocol (Hardware Communication Layer)
    - USB-TMC (USB Test and Measurement Class) is a standard communication protocol that allows the oscilloscope to be controlled over USB

    - This protocol enables direct communication with the oscilloscope via PyVISA in the Python backend
    - Ensures efficient data transfer with minimal latency
2. PyVISA (Software Control Layer)
    - PyVISA is a Python library that provides an interface to control test instruments via USB, GPIB, Ethernet, or serial connections
    - It enables the backend to send SCPI commands to the oscilloscope and retrieve measurement data
    - PyVISA will be used in the Python backend to manage instrument connections

→ Hardware Communication Flow:

Node.js API → Python Backend → PyVISA (VISA Library) → USB-TMC Driver → Oscilloscope → Measurement Data Response

→ Core Responsibilities of Hardware Communication

- Establish a stable connection between the Python backend and the oscilloscope via USB-TMC
- Use PyVISA to send SCPI commands and retrieve oscilloscope measurements
- Ensure error handling for instrument connection failures and invalid SCPI commands
- Implement structured logging to track command execution and hardware responses
- Optimize data retrieval speed to support real-time signal processing

→ High-Level Hardware Workflow (Step-by-step Execution of Hardware Communication)

1. Establishing Connection with the Oscilloscope
    - The Python backend detects the connected oscilloscope via USB-TMC
    - PyVISA initializes a session with the instrument
    - The system sends an identification query to verify the connection:

```
import pyvisa
rm = pyvisa.ResourceManager()
oscilloscope =
rm.open_resource("USB::0x0699::0x0363::C102220::INSTR")
print(oscilloscope.query("*IDN?"))  # Query the device
identity
```

2. Sending SCPI Commands for Data Retrieval
    - The backend sends SCPI commands to request measurement data
    - Example: Retrieving voltage peak-to-peak measurement:

```
oscilloscope.write("MEASure:VPP?")
vpp_value = oscilloscope.read()
print(f"Voltage Peak-to-Peak: {vpp_value} V")
```

3. Processing and Formatting Data
    - The retrieved data is parsed and formatted into JSON for API response
    - Example structured response

```
{
  "status": "success",
  "timestamp": "2025-02-10T14:23:45Z",
  "data": {
    "measurement": "voltage_peak_to_peak",
    "value": 3.5,
    "unit": "V"
  }
}
```

4. Error Handling and Logging
    - If an instrument connection fails, the system logs an error and returns an appropriate message
    - Example error handling for an invalid SCPI command:

```
{
  "status": "success",
  "timestamp": "2025-02-10T14:23:45Z",
  "data": {
    "measurement": "voltage_peak_to_peak",
    "value": 3.5,
    "unit": "V"
  }
}
```

5. Returning Data to the API
    - The formatted JSON response is sent to the Node.js API for frontend retrieval
    - The frontend receives the data and updates Chart.js with real-time oscilloscope values

→ SCPI Commands List for Oscilloscope Control

Command	Description	Example Response
`*IDN?`	Query oscilloscope identity	`"TEKTRONIX, TDS1002B, C102220, CF:91.1CT FV:v22.11"`
`MEASure:VPP?`	Measure voltage peak-to-peak	`"3.5"` (Volts)
`MEASure:FREQuency?`	Measure signal frequency	`"1000"` (Hz)
`MEASure:PHASe?`	Measure signal phase shift	`"0.5"` (Degrees)
`CHANnel1:SCALe 2.0`	Set channel 1 vertical scale to 2.0V per division	No response
`ACQuire:STATE ON`	Start data acquisition	No response
`ACQuire:STATE OFF`	Stop data acquisition	No response

→ Where Hardware Integration Fits in System Architecture

- Python Backend
  - Uses PyVISA to send SCPI commands over USB-TMC
  - Retrieves measurement data and formats it into JSON for the API
- Node.js API
  - Sends request for oscilloscope data to the Python backend
  - Receives formatted JSON responses and forwards them to the frontend
- Frontend (Angular + Chart.js)
  - Displays real-time oscilloscope readings based on API responses
- Future Scalability
  - PyVISA van be extended to support other instruments (e.g., function generators, power meters)
  - Additional data logging features can be implemented for long-tmerm measurement tracking

→ Hardware Data Flow Across System Components

Frontend (User Input) → API Request (Node.js) → Python Backend (Signal Processing & PyVISA) → USB-TMC (Instrument Communication) → Oscilloscope (Measurement Retrieval) → Backend Formats Data (JSON) → API Response Sent to Frontend → Chart.js Updates UI

→ Hardware Integration Implementation Strategy

- Step 1: Setup PyVISA and Connect to Oscilloscope

- Install PyVISA:

```
pip install pyvisa
```

- Detect connected instruments:

```
import pyvisa
rm = pyvisa.ResourceManager()
print(rm.list_resources())  # List all connected devices
```

- Step 2: Implement SCPI Command Execution in Python Backend
  - Write Python functions for sending SCPI commands and retrieving data
  - Ensure valid responses and error handling
- Step 3: Format Measurement Data in JSON
  - Convert retrieved oscilloscope data into structured JSON responses
  - Ensure data follows consistent formatting for frontend display
- Step 4: Implement API Endpoint in Node.js
  - Create backend API routes to send requests to the Python backend
- Step 5: Test End-to-End Hardware Communication
  - Verify that PyVISA detects the oscilloscope and executes SCPI commands
  - Ensure API successfully retrieves and returns oscilloscope data

- Identify technology stack and dependencies, justifying each selection

This subtask focuses on selecting and justifying the technology stack and dependencies used in the Network Analyzer project. Each component—frontend, backend, API, database (JSON storage), authentication, CI/CD, and hardware integration—requires a carefully chosen set of technologies to ensure scalability, efficiency, and maintainability.

The goal is to document why each technology was selected, ensuring alignment with project requirements, performance goals, and future expansion possibilities.

→ Technology Stack Overview

The Network Analyzer system consists of seven primary layers, each with its own technology stack:

Layer	Technology Used	Justification

Frontend (UI and Visualization)	Angular, Chart.js, Angular Material	Provides a scalable, component-based UI, real-time data visualization, and responsive design
API Layer (Backend Interface)	Node.js, Express.js	Ensures non-backing I/O, scalable API handling, and seamless communication between frontend and backend
Backend (Signal Processing and Hardware Communication	Python, PyVISA, SciPy, NumPy	Python is optimal for scientific computing, signal generation, and oscilloscope control via SCPI
Hardware (Signal Measurement and Acquisition)	Oscilloscope (Tektronix TDS 1002B), USB-TMC, SCPI Commands	Captures real-world signals, enabling real-time data acquisition and measurement
Data Storage (JSON-Based Storage	JSON, File-based logging	Lightweight, human-readable, and compatible across frontend, backend, and API
Authentication and Security	OAuth (Google/GitHub), JWT	Secure user authentication using industry-standard OAuth and token-based access control
Deployment and CI/CD	Kubernetes, GitHub Actions, Docker	Automates containerized deployments, version control, and scaling using Kubernetes and CI/CD workflows

→ Core Responsibilities of the Technology Stack

- Ensure seamless interaction between frontend, backend, and hardware components
- Support real-time data processing and visualization
- Provide a scalable, modular architecture that allows for future expansions

- Implement robust authentication and security measures
- Enable automated deployment, testing, and version control
- Establish direct communication with the oscilloscope for data acquisition

→ Justification for Each Technology in Detail

1. Hardware: Oscilloscope (Tektronix TDS 1002) + USB-TMC + SCPI
    - Oscilloscope: Captures real-world signals, enabling real-time data acquisition
    - USB-TMC (USB Test and Measurement Class): Standard protocol for USB-based instrument control
    - SCPI (Standard Commands for Programmable Instruments): Industry-standard command set for interfacing with test equipment
    - **Justification**: The oscilloscope is the core measurement device, and USB-TMC and SCPI ensures precise data retrieval via PyVISA
2. Frontend: Angular + Chart.js
    - Angular: Provides a component-based structure, built-in state management, and strong frontend-backend interaction support
    - Chart.js: Enables real-time oscilloscope data visualization using interactive, scalable graphs
    - Angular Material: Ensures a responsive and modern UI for better user experience
    - **Justification**: Angular's modular design allows scalability, while Chart.js offers real-time data plotting with minimal performance overhead
3. API Layer: Node.js + Express.js
    - Node.js: Efficient non-blocking event-driven architecture for handling API requests
    - Express.js: Simplifies RESTful API development with middleware support
    - **Justification**: Node.js efficiently bridges the frontend and backend, allowing fast API communication while supporting high-concurrency applications
4. Backend: Python + PyVISA + SciPy
    - Python: Optimized for scientific computing and hardware communication
    - PyVISA: Enables oscilloscope control via USB-TMC and SCPI commands
    - SciPy and NumPy: Used for signal generation, processing, and mathematical computations
    - **Justification**: Python's extensive libraries simplify hardware control, while NumPy/SciPy enable efficient signal processing

5. Data Storage: JSON
    - JSON-based data storage: Lightweight and compatible with both frontend and backend
    - File-based JSON logging: Maintains structured logs for API and backend debugging
    - **Justification**: JSON provides efficient and human-readable data exchange, ensuring easy integration with all system components
6. Authentication and Security: OAuth + JWT
    - OAuth (Google/GitHub Authentication): Secure third-party authentication for users
    - JWT (JSON Web Token): Provides token-based session management for API security
    - **Justification**: OAuth offers secure, scalable authentication, while JWT ensures tokenized stateless security
7. Deployment and CI/CD: Kubernetes + GitHub Actions + Docker
    - Kubernetes: Manages scalable deployment and container orchestration
    - GitHub Actions: Automates CI/CD pipelines for testing, version control, and deployment
    - Docker: Creates containerized environments for reproducible builds
    - **Justification**: Kubernetes enables scalable deployment, while CI/CD automation ensures reliable and continuous integration

→ Where Each Technology Fits in System Architecture

Frontend (Angular + Chart.js) → API Layer (Node.js + Express.js) → Backend (Python + PyVISA + NumPy) → Oscilloscope (Tektronix TDS 1002B + USB-TMC) → Data Storage (JSON Logging) → Authentication (OAuth + JWT) → Deployment & CI/CD (Kubernetes + GitHub Actions)

→ Technology Implementation Strategy

- Step 1: Set Up Development Environment
    - Install required dependencies for each technology stack component
- Step 2: Implement API and Backend Communication
    - Define Node.js API endpoints and Python backend processing
- Step 3: Integrate Frontend with API
    - Implement Angular services for API communication
    - Configure Chart.js for real-time oscilloscope data visualization
- Step 4: Implement Authentication and Security
    - Set up OAuth provider integration and JWT-based session management

- Step 5: Configure CI/CD and Deployment Workflow
    - Automate GitHub Actions workflows for testing and deployment
    - Deploy using Docker + Kubernetes for scalability

2/17/25 Entry
2/18/25 Entry

What to Document:
- System architecture diagram (backend, frontend, hardware interactions)
- API and framework decisions (Node.js, Angular, JSON, Chart.js)
- Justification for technology choices

2/19/25 Entry

Deliverables:
1. System Architecture Diagram
2. Technical Design Document

Key Task 2: Define API and Backend Communication

Subtasks:

- Design RESTful API endpoints for signal generation and oscilloscope data retrieval

→ API Overview

- The API will facilitate communication between the frontend (Angular app), backend (Node.js/Express server), and hardware components (Python-controlled oscilloscope and signal generator)
- All endpoints will follow RESTful principles and use JSON for data exchange
- OAuth 2.0 will be implemented for secure access

→ API Endpoints Definition

A. **Signal Generation Endpoints**

These endpoints allow the frontend to trigger signal generation via the Python backend.

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

Request / Response Formats

1. 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"
}
```

2. GET `/api/signal/status`
    - Response (JSON)

```
{
  "status": "active",
  "frequency": 1000,
  "amplitude": 1.5,
  "waveform": "sine",
  "duration": 30
}
```

3. DELETE `/api/signal/stop`
    - Response (JSON)

```
{
  "status": "success",
  "message": "Signal generation stopped"
}
```

B. **Oscilloscope Data Retrieval Endpoints**
These endpoints fetch data from the oscilloscope, allowing the frontend to visualize waveform measurements.

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

Request / Response Formats

1. 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 }
    ]
  }
}
```

2. GET `/api/oscilloscope/settings`
    - Response (JSON)

```
{
  "status": "success",
  "settings": {
    "timebase": "5ms/div",
    "voltage_scale": "2V/div",
    "trigger_mode": "auto"
  }
}
```

3. POST `/api/oscilloscope/set`
    - Request Body (JSON)

```
{
  "timebase": "2ms/div",
  "voltage_scale": "1V/div",
  "trigger_mode": "normal"
}
```

    - Response (JSON)

```
{
  "status": "success",
  "message": "Oscilloscope settings updated"
}
```

→ Authentication and Security Considerations

- OAuth 2.0 will be used for token-based authentication
- Each request must include an Authorization header:

```
Authorization: Bearer <ACCESS_TOKEN>
```

- Rate limiting and request validation will be applied to prevent abuse

→ Justification for These Endpoints

- **Modularity**: The API structure keeps signal generation and oscilloscope communication separate, making it easy to maintain
- **Extensibility**: Additional functionalities (e.g., WebSockets for real-time streaming) can be layered on top

- **Security**: OAuth-based authentication ensures only authorized users access the API
- **Consistency**: Follows RESTful principles with clear, predictable URLs and JSON responses

- Define data handling and JSON storage strategy

This section defines how data from signal generation and oscilloscope measurements will be structured, stored, and retrieved within the backend. The goal is to ensure efficient data handling while maintaining scalability, security, and performance.

→ Overview of Data Storage Strategy

I will use JSON-based storage for logging and temporary data retention, ensuring that the backend efficiently organizes and retrieves oscilloscope readings and signal generation parameters.

- **Primary Storage:** JSON files for logs and transient data
- **Future Expansion:** If needed, I can transition to a NoSQL database like **MongoDB** for scalability
- **Security Considerations:** JSON logs will be protected with **access controls**, and sensitive data will be encrypted where necessary

→ Overview of Data Storage Strategy

The backend will maintain two primary JSON files:

A. Logs for Generated Signals `signals.json`
   - Stores parameters of all generated signals
   - Enables tracking of active and past signals
   - Useful for debugging and signal replication

Example Structure:

```
{
  "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"
    },
    {
      "signal_id": "def456",
      "timestamp": "2025-02-19T14:45:00Z",
      "user": "dev_02",
      "parameters": {
        "frequency": 500,
        "amplitude": 2.0,
        "waveform": "square",
        "duration": 60
      },
      "status": "completed"
    }
  ]
}
```

B. Captured Measurement Data `oscilloscope_data.json`
   - Stores time-series oscilloscope readings
   - Facilitates post-processing and visualization in the frontend
   - Allows caching of recent data to reduce direct oscilloscope queries

Example Structure:

```
{
  "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 Handling Workflow

1. Signal Generation
   - When a signal is generated, its parameters are logged in `signals.json`
   - The frontend can query active signals via `/api/signal/status`
2. Oscilloscope Data Retrieval

    - When a measurement is taken, data is temporarily stored in `oscilloscope_data.json`
    - The backend will periodically clear outdated readings to avoid excessive storage usage.
3. Data Expiration Policy
    - Active signals are kept for 30 minutes after generation.
    - Oscilloscope readings older than 1 hour are removed to maintain efficiency.
    - If a database integration is required in the future, time-series data can be indexed for long-term storage.
    - If storage usage grows, we may implement log rotation instead of strict deletion

→ Justification for JSON-Based Storage

- **Lightweight & Human-Readable:** JSON allows easy debugging and direct log review
- **Structured but Flexible:** Can be extended with additional metadata fields if needed
- **Low Overhead:** No database setup required, making it ideal for a self-contained system
- **Future Scalability:** Can transition to MongoDB or another NoSQL solution if persistent storage is required

- Evaluate WebSocket integration for real-time data streaming

→ Overview

The current API primarily uses **RESTful endpoints** for communication, which works well for on-demand data retrieval. However, **real-time oscilloscope measurements** and **live signal monitoring** may benefit from **WebSocket integration**, allowing the frontend to receive continuous updates without repeatedly polling the backend. This evaluation will determine:

- Whether WebSockets provide a meaningful advantage over REST for oscilloscope data
- Which data streams would benefit from WebSocket integration
- Potential implementation challenges and alternatives

→ WebSocket vs. REST: Pros and Cons

Feature	**REST API (Current)**	**WebSockets (Proposed)**

Latency	Higher (Client must poll the server)	Lower (Real-time push updates)
Efficiency	Inefficient for frequent updates	Efficient for continuous data streaming
Scalability	Scales well due to stateless nature	Requires persistent connections, more server load
Use Case Fit	Best for request/response data (e.g., logs, settings)	Best for continuous oscilloscope readings and live signal monitoring
Security	OAuth token in headers	OAuth token must be included in the WebSocket handshake
Implementation Complexity	Simple (Express.js REST API, stateless)	More complex (WebSocket server setup, connection handling, stateful)

→ Recommended Use Cases for WebSockets
WebSockets should only be used where real-time data streaming is necessary. I will use hybrid integration, using REST for standard requests and WebSockets for continuous monitoring.
Best Use Cases for WebSockets:

- Live Oscilloscope Readings: Instead of making repeated REST requests, the frontend subscribes to updates
- Live Signal Generation Monitoring: The frontend receives real-time status updates on active signals

REST-Only Use Cases (No WebSockets Needed):

- Retrieving Oscilloscope Settings: This data is static and does not need constant updates
- Sending Configuration Changes: REST is better for one-time requests like adjusting voltage scales

→ WebSocket Implementation Strategy
If WebSockets are integrated, the backend will:

1. Maintain a WebSocket server alongside the REST API
2. Emit real-time updates for oscilloscope data and signal generation
3. Authenticate WebSocket connections using OAuth tokens

Proposed WebSocket Event Structure

- Client subscribes to data updates
- Backend pushes real-time oscilloscope readings or signal status

Example WebSocket Communication
Client → Server (Subscribe to updates)

```
{
  "event": "subscribe",
  "type": "oscilloscope_data",
  "auth_token": "xyz-123"
}
```

Server → Client (Subscribe to updates)

```
{
  "event": "oscilloscope_update",
  "data": {
    "time": 0.0,
    "voltage": 0.0
  }
}
```

→ Challenges & Considerations

1. Server Load: WebSockets require persistent connections, which can strain the backend. Solution: Use WebSocket connection pooling to optimize performance.
2. Security Risks: WebSockets do not include headers like REST requests. Solution: Authenticate clients via OAuth token in the initial handshake.
3. Frontend Handling: The frontend must efficiently unsubscribe when WebSocket data is no longer needed.

→ Conclusion & Recommendation
WebSockets should be integrated for real-time oscilloscope readings and live signal status updates.

- REST API will still be used for configuration, logs, and request/response interactions
- Hybrid architecture (REST + WebSockets) ensures efficient, secure, and scalable communication

- Ensure secure API access using OAuth token verification

→ Overview of API Security
To protect the REST API and WebSocket communication, we will implement OAuth 2.0 token verification for authentication and role-based access control (RBAC) for authorization.

- Authentication: Ensures that only valid users can access API endpoints
- Authorization: Restricts access to sensitive endpoints based on user roles
- Token Handling: Prevents unauthorized API usage by requiring a valid Bearer Token in every request

→ OAuth 2.0 Authentication Workflow
The authentication process will follow these steps:

1. User logs in via an OAuth provider (Google, GitHub, or custom authentication)
2. OAuth provider issues an access token to the client
3. Client includes the token in every API request via the `Authorization` header
4. The backend validates the token before processing the request
5. If the token is valid, the request proceeds; otherwise, the API returns a 401 Unauthorized response

→ Implementing OAuth in REST API
Every REST request must include a valid Bearer Token in the Authorization header:
Example API Request with OAuth Token:

```
GET /api/oscilloscope/data HTTP/1.1
Host: example.com
Authorization: Bearer xyz-12345
```

Backend Token Validation (Node.js Example)

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

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" });

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

// Apply to protected routes
app.get('/api/oscilloscope/data', authenticateToken, (req, res) => {
    res.json({ message: "Secure oscilloscope data" });
});
```

→ OAuth Token Handling in WebSockets
Unlike REST, WebSockets do not support headers in real-time messages. Instead, the OAuth token must be passed during the initial WebSocket handshake.
Client Sends Authentication Token When Connecting:

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

Backend WebSocket Server Validates Token

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws, req) => {
    const { token } = new URL(req.url,
      `http://${req.headers.host}`).searchParams;
    if (!token || !jwt.verify(token, process.env.OAUTH_SECRET)) {
    ws.close();
    return;
}
    if (!token || !jwt.verify(token, process.env.OAUTH_SECRET)) {
        ws.close();
        return;
    }

    ws.send(JSON.stringify({ message: "WebSocket connection secured" }));
});
```

→ Role-Based Access Control (RBAC)
Certain API Actions should be restricted based on user roles.

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

Example of Role-Based Access Control

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

Commented [AS1]: Was previously…

const token = req.headers.authorization?.split(' ')[1];

Problem: WebSocket handshake does not support standard headers like REST.

Fix: Instead, tokens should be passed in the query parameters or subprotocols.

```
        }
        next();
    };
}

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

→ Security Best Practices

- Token Expiry: Access tokens will have a short expiration time (e.g., 1 hour), requiring periodic refresh
- HTTPS Enforcement: All API calls and WebSocket connections will be encrypted with TLS
- Rate Limiting: Prevent brute-force attacks by limiting API requests per user

2/20/25 Entry

What to Document:

- RESTful API specifications (endpoints, request/response formats)
- Backend interface with the oscilloscope and Python-generated signals
- Data structure for oscilloscope readings and system logs
- Justification for potential WebSocket integration

Deliverables

1. API Specifications Document

Key Task 3: Define Hardware-Software Interaction
Subtasks:

- Document USB-TMC communication setup with necessary drivers

2/21/25 Entry

What is USB-TMC?

- USB-TMC (USB Test & Measurement Class) is a protocol defined under IEEE 488.2 that allows instruments like oscilloscopes to communicate with a computer over USB.
- It allows direct hardware communication without needing serial adapters.
- SCPI commands are sent over USB-TMC to control and retrieve measurements from the oscilloscope.

Required Drivers & Software Dependencies

Platform	Required Software/Drivers
Windows	National Instruments VISA, Keysight IO Libraries Suite, or Tektronix OpenChoice
Mac	PyVISA, libusb, NI-VISA for Mac (if required)
Linux	PyVISA, libusb

Installation Steps

→ Windows

1. Install National Instruments VISA (or equivalent driver).
2. Open NI Measurement & Automation Explorer (NI MAX) to check if the oscilloscope is detected.
3. Run the following test script to list connected instruments:

```
import pyvisa
rm = pyvisa.ResourceManager()
print(rm.list_resources())  # Should show USBTMC instrument
```

→ Mac

1. Install Homebrew if not installed

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Install PyVISA

```
pip install pyvisa pyvisa-py
```

3. Verify Instrument List

```
import pyvisa
rm = pyvisa.ResourceManager()
print(rm.list_resources())
```

→ Linux (Ubuntu Systems)

1. Install dependencies

```
sudo apt update
sudo apt install python3-pyvisa python3-pyvisa-py libusb-1.0-0
```

2. Verify the USB connection

```
lsusb | grep "Oscilloscope"
```

3. Ensure the user has permissions

```
sudo chmod 666 /dev/usbtmc*
```

Troubleshooting USB-TMC Issues

Issue	Solution
Device not detected	Check drivers, use `rm.list_resources()` to confirm connection
Permission denied (Linux)	Run `sudo chmod 666 /dev/usbtmc*`
PyVISA Import Error	Ensure `pyvisa` and `pyvisa-py` are installed
Windows VISA issues	Restart PC, check in Device Manager

- Define SCPI command structure for oscilloscope communication

What is SCPI?

- SCPI (Standard Commands for Programmable Instruments) is a universal command set for controlling electronic instruments like oscilloscopes.
- It provides text-based commands to configure, control, and retrieve data from the device.

Basic SCPI Commands for Oscilloscope Communication

SCPI Command	Function

*IDN?	Returns oscilloscope model and manufacturer info
MEASure:VPP? CH1	Measures peak-to-peak voltage on Channel 1
MEASure:FREQ? CH1	Measures frequency of waveform on Channel 1
MEASure:PHASe? CH1, CH2	Measures phase difference between CH1 and CH2
SYSTem:ERRor?	Checks if there are command errors

SCPI Command Usage in Python with PyVISA

```
import pyvisa

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

# Send commands
print("Oscilloscope ID:", osc.query("*IDN?"))
print("Peak-to-Peak Voltage:", osc.query("MEASure:VPP? CH1"))
print("Frequency:", osc.query("MEASure:FREQ? CH1"))

# Check for errors
print("Oscilloscope Error Status:", osc.query("SYSTem:ERRor?"))
```

→ Expected Responses
`*IDN?` might return:

```
TEKTRONIX,TDS 1002B,0,CF:91.1CT FV:v22.11
```

`MEASure:VPP?` CH1 might return:

```
4.97E+00  # Voltage peak-to-peak (4.97V)
```

`SYSTem:ERRor?` Might return:

```
0,"No Error"
```

- Implement Python-based signal generation using SciPy and Sounddevice

How does Python generate test signals?

- SciPy generates waveform arrays, which are played through Sounddevice via USB or sound card.

Generating a 1 kHz sine wave in Python

```
import numpy as np
import sounddevice as sd

fs = 44100  # Sampling rate
```

```
t = np.linspace(0, 1, fs, False)  # 1 second
wave = 0.5 * np.sin(2 * np.pi * 1000 * t)  # 1 kHz sine wave
sd.play(wave, fs)
sd.wait()
```

Limitations of SciPy + Sounddevice

- Maximum output frequency is limited by the sound card sample rate.
- USB-based signal transmission may not be stable above 100 kHz.

Alternative Signal Generation Methods (if Python Cannot Reach 300 kHz)

1. Using an external USB signal generator
    - Example: Siglent SDG2042X or Rigol DG1022
2. FPGA-based signal generation
    - Xilinx or Altera FPGA with DAC output
3. Using a dedicated DAC (Digital-to-Analog Converter)
    - Example: AD9850 DDS module

- Develop test scripts using PyVISA to verify oscilloscope responses

What does the test script do?

- Ensures proper USB-TMC communication.
- Verifies oscilloscope responses to SCPI commands.

Basic PyVISA Test Script:

```
import pyvisa

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

# Run tests
print("Oscilloscope ID:", osc.query("*IDN?"))
print("Peak-to-Peak Voltage:", osc.query("MEASure:VPP? CH1"))
print("Frequency:", osc.query("MEASure:FREQ? CH1"))

# Error checking
print("Oscilloscope Error Status:", osc.query("SYSTem:ERRor?"))
```

Expected Output

```
Oscilloscope ID: TEKTRONIX,TDS 1002B,0,CF:91.1CT FV:v22.11
Peak-to-Peak Voltage: 4.97E+00
Frequency: 1.000E+03
Oscilloscope Error Status: 0,"No Error"
```

If errors occur:

- Check USB connection
- Ensure PyVISA is properly installed
- Verify correct SCPI syntax

What to Document:
- Hardware-software interaction plan
- USB-TMC setup process and required drivers
- SCPI command list and expected oscilloscope responses
- Alternative signal generation methods if Python cannot reach 300 kHz

2/23/25 Entry
2/24/25 Entry

Deliverables:
1. Hardware-software Interaction Diagram
2. PyVISA Test Script
3. SCPI Commands List

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

2/25/25 Entry

Subtasks:

- Select an OAuth provider (Google, GitHub, etc.)

Google OAuth is the easiest for public testing because:

- Extensive documentation and official libraries for various frameworks.
- User-friendly setup via the Google Cloud Console.
- Built-in user account security features (2FA, risk-based authentication).

GitHub OAuth is ideal for developer-based applications because:

- Many tech companies prefer GitHub authentication for internal tools.
- Easier integration for developer-based workflows.
- Not ideal for public access (as users must have a GitHub account).

Other options (Auth0, Okta, Firebase Authentication, AWS Cognito):

- More customizable, but introduce additional complexity.
- Often used for enterprise-grade applications with stricter security.

Decision: Google OAuth will be the provider due to ease of integration, support for public testing, and strong security features.

- Design authentication flow, including token handling and expiration policies

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.

- Define user roles and permissions (admin, hiring managers, general users)

Role	Permissions	Restrictions
Admin (Me)	Full access, logs all API calls.	No Restrictions.
Hiring Managers (Keysight-specific)	Can generate signals, view Bode plots.	Cannot modify system settings.
Friends & Family	Can access UI, but cannot edit configurations.	No backend API access.
General Users	Limited access to read-only features.	No access to signal generation or backend logs.

What to Document:

- OAuth provider decision (pros/cons of each option)
- Authentication flow diagram
- User roles and permissions structure
- How the frontend and backend handle token verification

Deliverables:

1. Authentication Flow Diagram
2. Technical Document on OAuth Implementation
3. User Roles and Permissions Document

Key Task 5: Plan Kubernetes and CI/CD Integration

2/26/25 Entry

Subtasks:

- Design CI/CD pipeline using GitHub Actions for automated testing

Design CI/CD Pipeline Using GitHub Actions for Automated Testing

Automate code integration, testing, and deployment to ensure reliable updates and reduce manual errors.

→ What Needs to Be Done?

- Define CI/CD Triggers: Decide when the pipeline runs—on push, pull request, or scheduled builds.
- Set Up Testing Steps: Automate unit, integration, and security testing before deployment.
- Deploy to Kubernetes: Ensure the pipeline builds Docker images and deploys them automatically.

→ Implementation Plan:

- Use GitHub Actions to define a CI/CD pipeline with YAML-based workflows.
- Implement staging and production environments to prevent bad code from reaching users.
- Add automated rollback in case of failed deployments.

→ Example GitHub Actions CI/CD Pipeline (YAML):

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Set up Docker
        uses: docker/setup-buildx-action@v1

      - name: Build and Push Docker Image
        run: |
```

```
          docker build -t my-app:latest .
          docker tag my-app:latest ghcr.io/my-org/my-app:latest
          docker push ghcr.io/my-org/my-app:latest

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Apply Kubernetes Deployment
        run: kubectl apply -f k8s/deployment.yaml
```

- Define Kubernetes deployment architecture (pods, services, namespaces)

Define Kubernetes Deployment Architecture (Pods, Services, Namespaces)
Establish a structured deployment model using Kubernetes to efficiently manage services and ensure scalability.
→ What Needs to Be Done?

- Define Pods: The smallest deployable unit in Kubernetes that runs the application.
- Create Services: Manage internal and external communication between components.
- Set Up Namespaces: Logical isolation for different environments (dev, staging, production).

→ Implementation Plan:

- Define a deployment manifest to specify replicas, CPU/memory limits, and rolling updates.
- Set up a LoadBalancer service for external access.
- Use ConfigMaps & Secrets to manage environment variables and credentials securely.

→ Example Kubernetes Deployment Manifest (YAML):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
```

```
- name: my-app
  image: ghcr.io/my-org/my-app:latest
  ports:
    - containerPort: 80
  envFrom:
    - configMapRef:
        name: app-config
    - secretRef:
        name: app-secrets
```

- Evaluate hosting options (AWS, Firebase, or self-hosted cluster)

Evaluate Hosting Options (AWS, Firebase, or Self-Hosted Cluster)
Choose a hosting platform that balances cost, performance, and ease of use.
→ What Needs to Be Done?
- Compare AWS, Firebase, and Self-Hosting for Kubernetes deployment.
- Evaluate security, pricing, and scaling capabilities.

→ Implementation Plan:
- AWS (EKS) – Best for production-ready, scalable apps but higher cost.
- Firebase Hosting + Cloud Run – Simple for small apps but lacks full Kubernetes control.
- Self-Hosted Kubernetes Cluster – Maximum control but requires DevOps expertise.

→ Hosting Option Comparison Table:

Hosting Option	Pros	Cons
AWS EKS	Fully managed, scalable, integrates with AWS services	Higher cost, complex networking setup
Firebase	Easy deployment, Google-managed infrastructure	Limited Kubernetes support, not great for backend-heavy apps
Self-Hosted	Full control, cost-effective	Requires more DevOps management, higher setup complexity

- Implement security best practices (Role-based access control or RBAC, container isolation)

Implement Security Best Practices (RBAC, Container Isolation)
Prevent unauthorized access and secure Kubernetes infrastructure.
→ What Needs to Be Done?
- Enforce Role-Based Access Control (RBAC) for managing permissions.

- Use container security best practices like image scanning and runtime monitoring.

→ Implementation Plan:

- RBAC Policies to restrict actions based on user roles.
- Network Policies to control how services communicate within Kubernetes.
- Container Isolation using Security Context settings.

→ Example Kubernetes RBAC Policy:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: production
  name: dev-role
rules:
  - apiGroups: [""]
    resources: ["pods", "services"]
    verbs: ["get", "list", "watch"]
```

What to Document:

- 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

Deliverables:

1. CI/CD Workflow Diagram
2. Kubernetes Deployment Plan

Key Task 6: GitHub Version Control and Documentation

Subtasks:

- Establish GitHub repository structure

```
/software-network-analyzer

├── api/                # Node.js RESTful API
│   ├── controllers/    # API route handlers (business logic)
│   ├── routes/         # API route definitions
│   ├── middleware/     # General middleware (logging, error handling, CORS,
│   │                     security policies)
│   ├── auth/           # Authentication logic (OAuth, JWT, user roles, token
│   │                     handling)
│   ├── services/       # Business logic (e.g., Python communication)
│   ├── utils/          # Helper functions (e.g., logging)
│   ├── config/         # API configuration (OAuth, environment variables)
│   ├── tests/          # Unit tests for API endpoints
│   ├── index.js        # Main entry point for the API
│   ├── package.json    # Node.js dependencies
│   ├── .env            # Environment variables
├── backend/            # Python-based signal processing backend
│   ├── scripts/        # PyVISA & signal generation scripts
│   ├── tests/          # Backend unit tests
│   ├── requirements.txt # Python dependencies
│   ├── main.py         # Main backend execution file
├── frontend/           # Angular frontend
│   ├── src/            # Frontend source code
│   ├── public/         # Static assets
│   ├── package.json    # Frontend dependencies
├── docs/               # Documentation
├── test/               # System-wide testing scripts
├── .github/workflows/  # GitHub Actions for CI/CD
├── README.md           # Project overview
├── LICENSE             # Open-source license
├── .gitignore          # Ignore unnecessary files
```

- Create an initial README file with an overview and setup guide

What to Include:

- Project Title: Software Network Analyzer
- Project Overview: A Node.js and Python-based tool for analyzing and visualizing oscilloscope signals.
- System Architecture
- Installation Steps
- Usage Instructions
- Contribution Guidelines
- License

- Define version control guidelines (branching, strategy, commit conventions)

Branching Strategy

```
main → Stable production-ready branch
dev → Active development branch for new features
feature/* → Feature-specific branches
hotfix/* → Bug fix branches
```

Branching Example for a New Feature (SQL)

```
git checkout -b feature/add-authentication
git commit -m "feat(auth): Implemented Google OAuth"
git push origin feature/add-authentication
```

2/27/25 Entry

What to Document:

- Repository structure and organization
- Initial README file with project goals, architecture, and setup instructions
- Guidelines for version control and collaboration

Deliverables:

1. GitHub Repository Setup
2. Initial README file

Future Tasks to Complete

- API folder structure and Node.js setup
- Express.js-based RESTful API with core routes
- Python backend communication module
- OAuth authentication integration
- Structured JSON responses
- API logging and debugging tools
- Postman test cases for validation