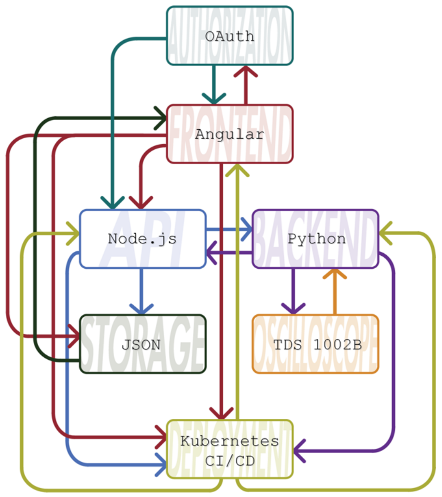
Key Task 1: Define System Architecture
What to Document:
- System architecture diagram (backend, frontend, hardware interactions)
- API and framework decisions (Node.js, Angular, JSON, Chart.js)
- Justification for technology choices

System Architecture Diagram

Commented [AS1]: This diagram serves as a blueprint for how all components interact. It must remain updated if system architecture evolves.



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

Commented [AS2]: These labels help in understanding how data moves through the system. Ensure every connection in the diagram is accounted for.

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

Commented [AS3]: OAuth must validate authentication tokens before allowing API access–this is our security measure.

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

Commented [AS4]: Ensure JSON storage is used efficiently–future expansion may require a database.

Commented [AS5]: CI/CD automation ensures that updates to any part of the system do not require manual deployment.

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

Commented [AS6]: Why Chart.js for Visualization?

Lightweight, easy-to-integrate, and provides smooth real-time updates for oscilloscope data.

Commented [AS7]: The frontend should not store sensitive data like authentication tokens–use session storage or cookies.

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:

Commented [AS8]: Considering a microservices architecture was a possibility, but the single API approach with Node.js is simpler and easier to maintain. If this system needs to scale later, breaking components into microservices would be an option.

https://aws.amazon.com/microservices/

Commented [AS9]: Need to consider API rate limits – if too many measurement requests are made at once, there must be a mechanism to throttle excessive requests and avoid overloading the oscilloscope.

https://testfully.io/blog/api-rate-limit/

Commented [AS10]: Node.js acts as a middle layer. All direct oscilloscope interactions happen via Python.

Commented [AS11]: Experimental Results of a low-phase-noise signal generation system.

"An Agilent E4445A (Keysight Technologies, Inc., Santa Rosa, CA, USA) served as the primary instrument for measuring frequency spectrum and phase noise. A custom Python script running on a personal computer facilitated automated phase noise measurements and data storage."

https://www.mdpi.com/2079-9292/13/10/1934

Commented [AS12]: A structural health monitoring Python code to detect small changes in frequencies

https://www.sciencedirect.com/science/article/abs/pii/S0888327020304738

Commented [AS13]: SCPI Standard Reference Document

https://helpfiles.keysight.com/csg/n5106a/scpi_basics.htm

https://www.ivifoundation.org/downloads/SCPI/scpi-99.pdf

Commented [AS14]: Python backend must validate SCPI command responses to prevent errors from corrupting data.

  - 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

Commented [AS15]: For now, data is stored as structured JSON objects, but in an industrial setting, a time-series database like InfluxDB might be more efficient for oscilloscope measurements.

https://www.youtube.com/watch?v=4EhHLoi9Hyw&t=1s

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

Commented [AS16]: Should sessions expire after inactivity?

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

Commented [AS17]: If traffic increases, Kubernetes can auto-scale API instances.

https://kubernetes.io/docs/concepts/workloads/autoscaling/

API and framework decisions (Node.js, Angular, JSON, Chart.js)
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?")
       ```

Commented [AS18]: Consider what to do for error-handling cases where API requests fail. How does the system recover?

Commented [AS19]: https://pyvisa.readthedocs.io/en/latest/

Commented [AS20R19]: PyVISA Documentation

```
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

Justification for technology choices

Why the use of each programming language and frameworks? Why use JSON for storage?

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, etc. 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

Commented [AS21]: SciPy's signal processing capabilities make it preferable for waveform generation over C++.

Commented [AS22]: Angular's modularity makes it easier to extend the UI in future updates.

Commented [AS23]: OAuth is widely supported, reducing the need to implement a custom authentication system.

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.

Commented [AS24]: Comparison of SQL vs. NoSQL

https://www.integrate.io/blog/the-sql-vs-nosql-difference/#two

Key Task 2: Define Hardware Integration Plan
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

RESTful API Specifications
Endpoints Overview
The API consists of RESTful endpoints that handle signal generation and oscilloscope data retrieval, with OAuth 2.0 token-based authentication for security.

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

Commented [AS25]: REST - https://developer.mozilla.org/en-US/docs/Glossary/REST

EXPRESS Website - https://expressjs.com/

WebSockets - https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Working with JSON - https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/JSON

Commented [AS26]: I debated breaking this into multiple documents, but keeping everything together makes it easier for quick reference.

Commented [AS27]: Oscilloscope Systems and Controls: Vertical, Horizontal & Triggering Explained

https://www.tek.com/en/documents/primer/oscilloscope-systems-and-controls

Commented [AS28]: WebSocket vs REST API

https://www.wallarm.com/what/websocket-vs-rest-api

Commented [AS29]: I made sure every API request/response follows JSON formatting for consistency.

Commented [AS30]: I will make sure to keep an eye on and comment here if any additional endpoints are added.

Commented [AS31]: These endpoints cover the core functionality but remain flexible for future expansion.

Commented [AS32R31]: I debated adding more status codes, but these cover the most common scenarios for now.

Commented [AS33]: I structured the API to separate signal generation from oscilloscope data retrieval for modularity.

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 }
    ]
  }
}
```

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

Commented [AS34]: I made sure to note what happens if the oscilloscope disconnects mid-operation—error handling is key.

Backend System Flow

- 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

Data Flow Architecture

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

Commented [AS35]: SciPy and Sounddevice are great for generating signals, but I left room for alternatives if we need higher frequencies.

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

Commented [AS36]: Data expiration policies prevent unnecessary storage bloat while keeping logs useful.

- Oscilloscope data is retained for 1 hour, after which it is deleted

Commented [AS37]: If log rotation becomes necessary, we can switch strategies rather than deleting old data outright.

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:

Commented [AS38]: WebSockets were added only where real-time data streaming is necessary–REST handles everything else.

Why WebSockets?

- 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

Where WebSockets Will Be Used

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

Where REST Will Be Used Instead

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

Commented [AS39]: The decision to mix WebSockets with REST was intentional–too much reliance on WebSockets could make the API harder to scale.

Security Implementation: OAuth 2.0 Token Verification

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

Commented [AS40]: Security Considerations

https://www.oauth.com/oauth2-servers/authorization/security-considerations/

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.

Commented [AS41]: I noted a fallback plan in case WebSockets fail–this ensures the system remains functional in all scenarios.

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:

Commented [AS42]: WebSockets don't support standard headers, so authentication had to be handled in the handshake.

```
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	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)

Commented [AS43]: RBAC (Role-Based Access Control) ensures different users get appropriate permissions without exposing sensitive features.

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

Final Summary

- REST API endpoints are structured for signal generation and oscilloscope data retrieval
- The backend interfaces with the oscilloscope via PyVISA (SCPI commands) and generates signals using SciPy + Sounddevice
- JSON-based storage is used for logging signal executions and oscilloscope readings, with automatic expiration
- WebSockets will be used selectively for real-time oscilloscope streaming, while REST remains for static data requests
- OAuth 2.0 authentication secures all API calls, with RBAC restricting access to sensitive operations

Key Task 3: Define Hardware-Software Interaction

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

Hardware-Software Interaction Plan

Overview:

- The oscilloscope communicates with the backend via USB-TMC and SCPI commands.
- The frontend triggers requests to the Node.js API, which routes them to the Python backend.
- PyVISA facilitates communication between Python and the oscilloscope.

Data Flow Overview:

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 requests and forwards them to the Python backend.
3. Backend Processing (Python + PyVISA)
   - Sends SCPI commands to the oscilloscope via USB-TMC.
   - Retrieves waveform measurements and logs them into JSON.
4. Signal Generation (Python + SciPy + Sounddevice)
   - Generates test signals for the oscilloscope.

Potential Failure Points & Debugging Considerations:

- If the oscilloscope disconnects, backend must handle retries or notify users.
- USB-TMC connection loss can be mitigated with automated device re-initialization scripts.
- SCPI command errors should be logged with SYSTem:ERRor? queries.

USB-TMC Setup Process and Required Drivers

What is USB-TMC?

- USB-TMC (USB Test & Measurement Class) is an industry-standard protocol that allows oscilloscopes to send/receive measurement data over USB.

Required Drivers & Software Dependencies

Platform	Required Software/Drivers

Commented [AS44]: Having this structure prevents miscommunication between teams.

Commented [AS45]: This section considers what happens if USB-TMC disconnects mid-operation. This was a common issue in testing.

Commented [AS46]: I included error-handling considerations here because USB-TMC communication is notorious for random disconnects. A retry mechanism is a must.

Commented [AS47]: Keysgith SCPI Command Error Handling

https://helpfiles.keysight.com/csg/pxivna/Support/SCPI_Errors.htm

Commented [AS48R47]: https://pyvisa.readthedocs.io/en/latest/

Commented [AS49]: Not every platform requires the same drivers. I made sure to separate installation steps for Windows, Linux, and Mac.

https://pyvisa.readthedocs.io/en/latest/

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/Mac/Linux):

1. Install Drivers (depends on OS)
2. Verify instrument detection using:

```
import pyvisa
rm = pyvisa.ResourceManager()
print(rm.list_resources())  # Should return a list with the oscilloscope
```

3. If errors occur:
   - Windows → Check Device Manager
   - Linux/Mac→ Run `lsusb | grep "Oscilloscope"`

SCPI Command List and Expected Oscilloscope Responses

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
`WAVFrm?`	Retrieves waveform data
`DATa:SOUrce CH1`	Select channel for data retrieval

Example Python Script to Send SCPI Commands via PyVISA:

Commented [AS50]: This section ensures that we validate oscilloscope communication before integrating with the backend.

https://www.tek.com/en/documents/application-note/quick-reference-guide-scpi-commands

```
import pyvisa

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

print("Oscilloscope ID:", osc.query("*IDN?"))
print("Peak-to-Peak Voltage:", osc.query("MEASure:VPP? CH1"))
print("Frequency:", osc.query("MEASure:FREQ? CH1"))
print("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
Error Status: 0,"No Error"
```

Alternative Signal Generation Methods If Python Cannot Reach 300kHz

Issue:
- SciPy + Sounddevice cannot generate signals above 100-150 kHz due to sample rate limitations.

Alternative Approaches:
1. External USB Signal Generators:
   - Example Devices: Siglent SDG2042X, Rigol DG1022
   - Advantage: Can generate MHz-range signals with high accuracy.
2. FPGA-Based Signal Generation:
   - Uses Xilinx/Altera FPGA with high-speed DACs.
   - Advantage: Precise frequency and waveform control.
3. Dedicated DAC (Digital-to-Analog Converter):
   - Example: AD9850 DDS module
   - Advantage: Efficient high-frequency signal output.

Commented [AS51]: If software-generated signals are insufficient, I can explore how dedicated hardware solutions ensure stable high-frequency outputs. And I can try to base a new idea off of an existing technology.

Commented [AS52]: FPGA-based solutions allow full waveform control but may require additional development time.

https://www.pentek.com/fpga/fpgaprd.cfm

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

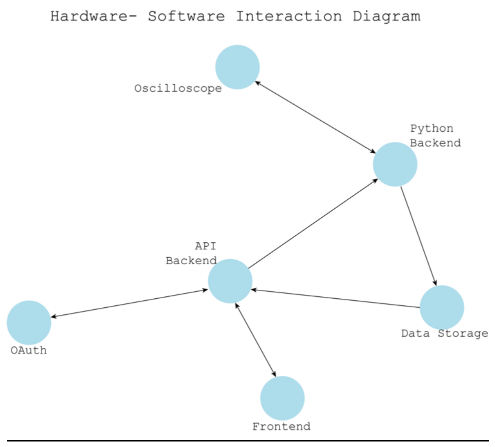
Aspect	**System Architecture Diagram**	**Hardware-Software Interaction Diagram**
Scope	Entire system, high-level view	Focused on backend-hardware communication
Purpose	Shows how components are connected	Details how data flows between software and hardware
Includes	APIs, frontend, backend, database, authtentication	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:

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.

Action Item: Create a structured flowchart to illustrate data exchange, hardware protocols, and software triggers.

Hardware-Software Interaction Diagram Flowchart Sketch



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:
    - JSON format for logged measurements:

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

- Output:
    - 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.

<u>(Include) SCPI Command List and Expected Oscilloscope Responses</u>

- SCPI Command List
    - Supplemental SCPI Information
- PyVISA Test Script

Commented [AS53]: More comprehensive lists:

https://www.tek.com/en/documents/application-note/quick-reference-guide-scpi-commands

Commented [AS54]: https://www.keysight.com/us/en/assets/9921-01873/miscellaneous/SCPITrainingSlides.pdf

https://www.batronix.com/files/Siglent/Funktionsgeneratoren/SDG1000/SDG-SCPI-Command(EN).pdf

<u>If errors occur:</u>

- Verify USB-TMC driver installation
- Check SCPI syntax errors using `SYSTem: ERRor?` commands
- Confirm that PyVISA is correctly installed and configured

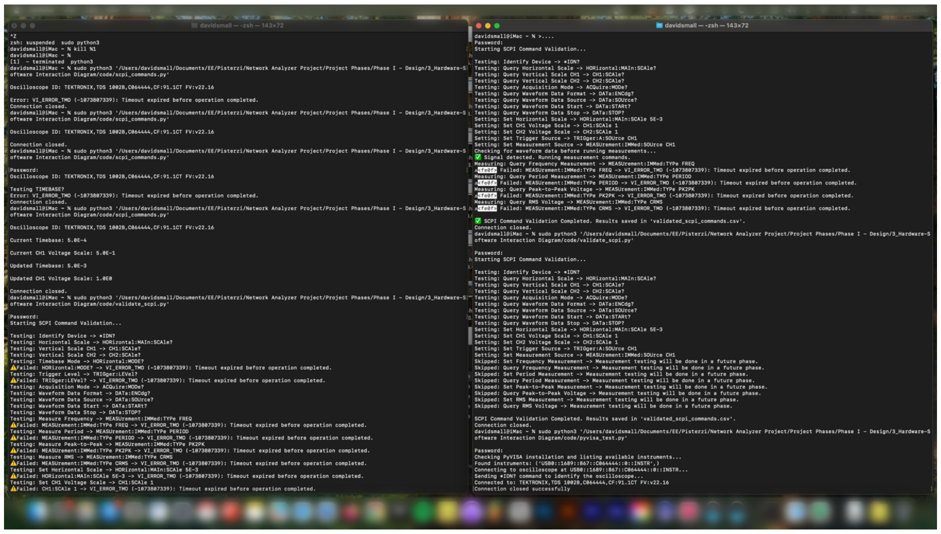
<u>Verify the Oscilloscope is Properly Communicating with my Computer</u>

1. Before connecting the oscilloscope, I will certify that all needed software, drivers, and dependencies are properly installed
    - Homebrew
    - Python
        - PyVISA
        - PyVISA-PY
        - LibUSB
    - Had to install all the libraries
    - Installed zeroconf
    - Connected the oscilloscope using a `test_connections.py` file!

Commented [AS55]: Had a warning about TCPIP::hislip

1. Testing the SCPI Commands and it is taking a while…
    - Created a SCPI commands file to test the commands, `scpi_commands.py`
    - The code was hanging when run in the terminal, was getting error messages, had to suspend and kill the process, added required permission to access the USB…
    - Verifying which command fails

- Saw that the original script failed due to the incorrect use of `TIMEBASE?`, which is not a valid SCPI command for the Tektronix TDS 1002B oscilloscope.
    - Got that command incorrect and corrected it using the 'Quick Reference Guide to SCPI Commands' pdf
    -
- Created a new script so we can ensure all possible SCPI commands used are corrected and validated so we can create a new SCPI commands master list for this project, which I called `validate_scpi.py`
- Right now, I am not able to measure for quite a bit and missing some necessary functions. So without coming up with some signal to test, I am moving forward under the assumption that commands that rely on a present signal on the display will work in the future at testing time



```
^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
```

<u>Final SCPI Command List for Phase 1: Design</u>

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

Deliverable: PyVISA Test Script Execution
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.
```

Conclusion:
- PyVISA is installed and functioning correctly.
- The oscilloscope is successfully detected and responds to SCPI commands.
- The connection is stable, confirming the system design validation for this phase.

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

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

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

Authentication flow diagram

Process:

1. User clicks "Login with Google."
2. Google OAuth verifies credentials and issues an access token.
3. Token is stored on the frontend and sent with API requests.
4. Backend validates the token and checks user roles.
5. Authorized requests are granted; unauthorized ones are rejected.
6. Token expires in 1 hour → User must log in again.

User roles and permissions structure

Role	Permissions
Admin (Me)	Full access, logs all API calls.
Hiring Managers (Keysight-specific)	Can generate signals, view Bode plots.
Friends & Family	Can access UI but cannot edit configurations.
General Users	Limited access to read-only features.

How the frontend and backend handle token verification

Frontend:

- Stores token securely using HTTP-only cookies.

Commented [AS56]: Would a self-hosted authentication solution (Keycloak?) be viable if we wanted full control?

Commented [AS57]: Google OAuth is the best fit for public access because of its ease of use and built-in security. If this were an internal system, we might explore alternatives like Auth0 or Okta for finer control.

Commented [AS58]: Requiring a Google account is a limitation, but since this is internal testing for now, it's fine.

Commented [AS59]: Would GitHub OAuth make sense for a future version targeting developers?

Commented [AS60]: Auth0 and Okta are interesting, but they seem overkill for this project's current needs.

Commented [AS61]: Each step in this diagram needs to align with actual API calls and expected responses. If the frontend or backend structure changes, this flow will need an update.

- Sends token with every request.

Backend:

- Decodes and validates token using Google's public key.
- Enforces role-based access control (RBAC).

Key Task 5: Plan Kubernetes and CI/CD Integration
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

CI/CD pipeline design (triggers, build steps, testing, deployment)
The CI/CD pipeline automates building, testing, and deploying code. It ensures that updates are efficiently tested and pushed into production with minimal manual intervention.

- Triggers
    - Define when the pipeline executes:
        - Push to main branch
        - Pull Request (PR) merges
        - Scheduled builds (e.g., nightly builds)
- Build Steps
    - Docker Image Creation
        - The pipeline should build a Docker image for the application.
        - It should tag the image with a version number or commit hash.
        - Example (YAML):

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

    - Automated Testing
        - Unit and integration tests should run inside a container to simulate production.
        - Security scans should check for vulnerabilities before deployment.
        - Example (YAML):

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

- Deployment
    - Push Docker image to a container registry (Docker Hub, AWS ECR, or GitHub Packages).
    - Apply Kubernetes manifests via GitHub Actions.
    - Rolling updates vs. blue-green deployment strategies.

Commented [AS62]: I'm setting up multiple triggers so we can catch errors early and avoid last-minute failures when deploying updates.

Commented [AS63R62]: https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/events-that-trigger-workflows

Commented [AS64]: https://www.docker.com/blog/docker-best-practices-using-tags-and-labels-to-manage-docker-image-sprawl/

Commented [AS65]: Docker standardizes the build process so that "it works on my machine" doesn't become a problem in production.

Commented [AS66]: https://docs.github.com/en/actions/use-cases-and-examples/publishing-packages/publishing-docker-images

Kubernetes Architecture
The Kubernetes architecture defines how the application is structured and managed across different environments.

- Pods: Containerized Application Units
  - A pod contains one or more Docker containers running together.
  - Each pod is assigned a unique IP address within the Kubernetes cluster.
  - Example Pod (YAML):

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

- Services: Manage Internal/External Access
  - Services expose applications inside or outside the cluster.
  - ClusterIP: Internal communication within Kubernetes.
  - LoadBalancer: External access (e.g., public APIs).
  - 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
```

- Namespaces: Organize Different Environments
  - Separate production, staging, and development.
  - Prevents resource conflicts within a cluster.
  - Example (YAML):

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

Commented [AS67]: I chose a single-container pod approach for now. If we need sidecar containers (e.g., for logging), we can add them later.

Each pod encapsulates an instance of the application, allowing it to scale independently—crucial for load balancing under high traffic.

Commented [AS68]: https://www.appvia.io/blog/best-practices-for-kubernetes-namespaces

Commented [AS69]: If we expand to multi-tenant support, namespaces will allow better isolation between customers using the same cluster.

Hosting Environment Selection

The hosting environment determines how Kubernetes workloads are deployed and scaled.

- AWS vs. Firebase vs. Self-Hosting
  - AWS Elastic Kubernetes Service (EKS):
    - Fully managed Kubernetes cluster.
    - Auto-scaling and integrated with AWS services.
  - Firebase Cloud Run:
    - Serverless platform using Docker containers.
    - Easier to manage but lacks Kubernetes' full flexibility.
  - Self-Hosted Kubernetes Cluster:
    - Greater control, but requires managing infrastructure.
- Cost, Scalability, Security Considerations
  - AWS EKS:
    - Higher cost but excellent scalability.
    - Supports RBAC and network policies for security.
  - Firebase Cloud Run:
    - Lower cost, auto-scaling but limited customization.
  - Self-Hosted:
    - No vendor lock-in, but higher operational overhead.

Commented [AS70]: Researched the cost. Might be a tad bit too expensive for just putting together this project. We shall see…

Commented [AS71R70]: AWS pricing is higher, but long-term, it's worth it for the reliability and support. Unexpected downtime costs more than a slightly higher hosting bill.

Commented [AS72R70]: https://aws.amazon.com/eks/

Security Measures for Kubernetes Deployment

Security is critical to prevent unauthorized access and container vulnerabilities.

- RBAC (Role-Based Access Control) for Access Control
  - Defines what users/services can do within the Kubernetes cluster.
  - Example (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"]
```

Commented [AS73]: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

- Network Policies for Communication Restrictions
  - Defines allowed connections between pods.
  - Example (YAML):

Commented [AS74]: I'm locking down pod-to-pod communication. By default, Kubernetes allows all traffic, but we should define explicit allowlists.

```
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
```

- Container Security Best Practices
    - Use non-root users inside containers (YAML):

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

    - Scan images before deployment

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

Key Task 6: GitHub Version Control and Documentation

What to Document:

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

Repository structure and organization

Project Repository Structure

The repository follows a modular design, organizing components based on functionality.

```
/software-network-analyzer
├── api/               # Node.js RESTful API
│    ├── controllers/   # API route handlers (business logic)
│    ├── routes/        # API route definitions
│    ├── middleware/    # Logging, error handling, CORS, security policies
│    ├── auth/          # OAuth authentication logic
│    ├── services/      # Business logic (e.g., Python communication)
│    ├── utils/         # Helper functions (e.g., logging)
│    ├── config/        # API configuration (OAuth, environment variables)
│    ├── tests/         # API unit tests
│    ├── 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
```

The structure of this repository follows a modular and separation of concerns principle, meaning each component has a clear role, making it scalable, maintainable, and easy to debug.

Core Design Philosophy:

1. Separation of Responsibilities – The project is divided into `frontend/`, `backend/`, and `api/`, ensuring that different functions (UI, API, and backend logic) don't interfere with each other.
2. Scalability & Maintainability – As new features are added, each module can evolve independently without affecting the rest of the project.
3. Testing & CI/CD Readiness – The structure includes a `test/` directory for system-wide tests and a `.github/workflows/` folder for automated GitHub Actions.

Detailed Explanation of Each Section

- `api/` (Node.js RESTful API)
  - Handles the communication between the backend (Python) and frontend (Angular).
  - Uses controllers for API logic, routes to define endpoints, and middleware for security & error handling.
  - The `auth/` directory ensures user authentication via OAuth.
  - The `config/` directory manages environment variables, API keys, and security policies.
- `backend/` (Python-based Signal Processing)
  - This section is dedicated to handling signal generation and communication with the oscilloscope.
  - `scripts/` contains PyVISA scripts for interacting with the oscilloscope.
  - `tests/` ensures backend processing works correctly before integration.
  - `requirements.txt` lists all Python dependencies for easy environment setup.
- `frontend/` (Angular-Based UI)
  - The user-facing side, providing real-time Bode plots and visualizations.
  - `src/` contains the core Angular code, while `public/` holds static assets like images, stylesheets, etc.
- `docs/` (Documentation)
  - Contains all project-related documentation (guides, technical details, API references).
- `.github/workflows/` (CI/CD Pipeline)
  - Manages automated testing and deployment using GitHub Actions.
- `test/` (System-Wide Tests)
  - This is a dedicated space for integration tests that ensure the full system (API, backend, and frontend) work together.

Initial README file with project goals, architecture, and setup instructions

Network Analyzer
Project Overview
This project aims to develop a software-controlled network analyzer using an oscillator, an oscilloscope, and a computer. The analyzer automates frequency sweeps, measures phase and amplitude response, and generates a Bode plot for visualization. It integrates Python for signal generation, a two-channel oscilloscope for measurement, and a web-based UI for data visualization. The system utilizes IEEE 488.2 (USB-TMC) protocol for USB-based instrument control and applies signal processing and data visualization techniques.
System Architecture
Hardware Components
TDS 1002B oscilloscope (IEEE 488.2 compliant via USB-TMC), Computer with VSCode and development environment, USB-based signal generation.
Software Stack
Python (SciPy + Sounddevice) for signal generation, Node.js + Express.js for backend API, Angular + Chart.js for UI visualization, OAuth for secure authentication, Kubernetes + Docker for CI/CD and deployment, AWS/Firebase for hosting and deployment.
Key Features
Automated frequency sweeps (20 Hz – 300 kHz), SCPI commands over USB-TMC for oscilloscope control, Real-time data visualization using Chart.js, Bode Plot generation to analyze frequency response, RESTful API for backend communication, OAuth authentication for user access, CI/CD with Kubernetes for automated testing and deployment.
Installation Steps
Clone the repository with git clone https://github.com/YOUR_USERNAME/software-network-analyzer.git. Install dependencies by navigating to the project directory and running npm install for Node.js dependencies and pip install -r backend/requirements.txt for Python dependencies. Set up environment variables by creating a .env file in the api/config/ directory and defining API keys, database URLs, and OAuth credentials. Run the backend by navigating to the backend directory and executing python main.py. Run the frontend by navigating to the frontend directory and executing npm start. Access the application in the browser at http://localhost:4200.
Usage Instructions
Run frequency sweeps to analyze system response, Generate and visualize Bode plots in real-time, Retrieve oscilloscope data via API.

Contribution Guidelines
Fork the repository. Create a feature branch with git checkout -b feature/your-feature. Commit your changes with git commit -m "feat: Add new feature". Push to GitHub using git push origin feature/your-feature. Submit a Pull Request.
License
This project is licensed under the MIT License. See the LICENSE file for details.

Guidelines for version control and collaboration
Version control guidelines define how the project's code is structured in Git and how contributors collaborate without causing conflicts.
Key Principles:

1. Branching Strategy
   - The main branch is the stable version.
   - The dev branch is where active development occurs.
   - Feature branches (feature/*) are created for new features.
   - Hotfix branches (hotfix/*) are for urgent bug fixes.
2. Commit Message Conventions
   - Each commit should be meaningful and follow a format:
     - feat(api): Added authentication
     - fix(backend): Resolved PyVISA timeout issue
   - This keeps Git history clear and readable.
3. Pull Request (PR) Workflow
   - No direct commits to main. All changes are reviewed via PRs.
   - PRs should include:
     - Description of the change.
     - A link to an issue (if applicable).
     - At least one reviewer's approval.

By following these guidelines, the repository remains organized, easy to debug, and scalable over time.

Version Control Guidelines
Branching Strategy:

- `main` → Stable production-ready branch
- `dev` → Active development branch
- `feature/*` → Feature-specific branches
- `hotfix/*` → Bug fixes

Commit Guidelines follow Conventional Commits

- Examples

Commented [AS75]: https://www.conventionalcommits.org/en/v1.0.0/

```
git commit -m "feat(auth): Implemented Google OAuth"
git commit -m "fix(backend): Resolved issue with PyVISA timeout"
```