HM Hochschule für angewandte Wissenschaften München
*Alexander Edelemann*, alexanderedelmann.com
Rohde & Schwarz supervisor: Meik Kottkamp
Examiner: Prof. Braun

Rohde & Schwarz
**Oktober 2025**

# Flask Demo Application to Showcase R&S SMW200A MIMO Fading Features

**Abstract** **This work presents a proof-of-concept application that demonstrates the integration of simulation data into the R&S SMW200A MIMO fader. Alongside the practical implementation, theoretical foundations are outlined, covering essential MIMO principles and the fundamentals of vector signal generators. The core architecture and code concepts of the application are described in detail, providing documentation that enables future extensions and adaptations. The application thus serves both as a demonstrator of advanced fading capabilities and as a foundation for further development.**

## CONTENTS



## 1 INTRODUTION

The evolution of mobile communication has been characterized by transformative architectural paradigm shifts that fundamentally redefined how we connect and communicate. The transition to LTE (Long Term Evolution), commonly known as 4G, represented one of the most significant inflection points in telecommunications history. Unlike previous generations that relied on a hybrid approach—where voice traffic utilized dedicated circuit-switched connections while data was transmitted via packet switching (introduced in later 3G technologies such as HSPA+)—LTE revolutionized the entire network architecture by adopting a unified all-IP, packet-based framework. This architectural transformation encapsulated all communication services—voice, video, messaging, and data—within Internet Protocol packets, eliminating the complexity and cost of maintaining separate circuit-switched infrastructure while delivering a seamless, internet-native mobile experience [1, 2].

Beyond its architectural innovations, LTE marked a turning point for antenna technology by becoming the first cellular standard to integrate MIMO (multiple-input multiple-output) as a core system component rather than an optional enhancement. Standardized under 3GPP Release 8 in 2008-2009, LTE initially deployed 2×2 and 4×4 MIMO configurations, with LTE-Advanced (Release 10) subsequently scaling to 8×8 implementations [3, 4, 5]. This fundamental shift in spatial signal processing enabled dramatic improvements in spectral efficiency and data throughput, directly addressing the exponential growth in mobile broadband demand driven by bandwidth-intensive applications including real-time video streaming platforms (YouTube, Netflix), cloud-based gaming, high-

definition video conferencing, and the emerging Internet of Things ecosystem. In essence, LTE established the high-performance mobile internet infrastructure that underlies our contemporary digital society and created the technological foundation for subsequent generations [6].

The advent of 5G New Radio (NR) has further accelerated this evolution through revolutionary physical layer innovations. Key technological advances include massive MIMO systems deploying 64, 128, or even more antenna elements, sophisticated beamforming algorithms, exploitation of millimeter-wave spectrum bands through advanced carrier aggregation techniques, and flexible OFDM numerology that adapts to diverse service requirements [7, 8]. These enhancements collectively enable multi-gigabit throughput, ultra-reliable low-latency communication (URLLC), and massive machine-type communications (mMTC)—expanding the wireless ecosystem far beyond traditional smartphone applications to encompass autonomous vehicles, Industry 4.0 manufacturing systems, augmented reality platforms, and smart city infrastructure.

Looking ahead to 6G, anticipated to commercialize around 2030 [9], it promises even more transformative capabilities. While standardization efforts are in their early stages (2025-2028), the research community is actively exploring revolutionary technologies including AI-native network orchestration, integrated sensing and communications (ISAC) capabilities, terahertz frequency exploitation, and seamless integration with edge-intelligence frameworks [10, 11, 9]. Although specific performance targets remain under development, 6G envisions orders-of-magnitude improvements in data rates, latency, connection density, and network intelligence compared to 5G systems.

In this context, advanced test and measurement solutions become increasingly critical for bridging theoretical concepts with practical implementation challenges. The project presented in this work demonstrates a comprehensive proof-of-concept application that accurately simulates realistic MIMO fading scenarios using the Rohde & Schwarz SMW200A vector signal generator integrated with sophisticated digital-twin propagation models. By emulating complex real-world channel conditions and multi-antenna system behavior within a controlled laboratory environment, this implementation showcases the SMW200A's essential role in prototyping, validating, and optimizing next-generation wireless systems—particularly as the industry prepares for the demanding requirements of 6G networks.

The following sections provide the necessary theoretical foundation, beginning with fundamental MIMO principles and vector signal generator architectures, before presenting detailed implementation aspects of the demonstration application.

# 2 THEORY MIMO

## 2.1 WHAT IS MIMO ABOUT?

To understand the core concept of multiple-input multiple-output (MIMO) data transmission systems, it is essential to first grasp the ideas of system performance and channel capacity. Adding transmit and/or receive antennas to a communication system can improve both performance and channel capacity, depending on how the antennas are utilized.

Performance typically refers to the reliability of the system, such as the probability of transmission errors. In this context, spatial diversity—leveraging multiple antennas—can help gather additional signal energy, thereby improving the effective signal-to-noise ratio (SNR). On the other hand, spatial multiplexing enables the simultaneous transmission and reception of multiple independent data streams through separate antennas, effectively increasing the peak data rate. This maximum achievable data rate is defined by the channel capacity.

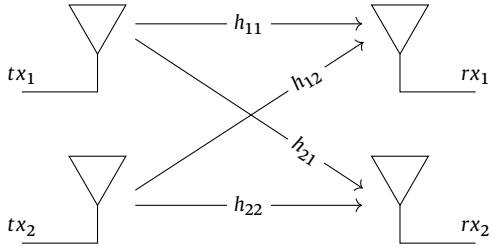



Figure 1 | 2×2 MIMO transmission model

However, the number of spatially multiplexed data streams is limited by the smaller number of antennas on either the transmitter or receiver side. For example, both a 2×2 and a 4×2 MIMO system can transmit at most two independent data streams simultaneously. The 4×2 system, however, benefits from additional spatial diversity, which can lead to improved reliability and overall system performance.

## 2.2 MIMO PERFORMANCE METRICS

**2×2 Narrowband MIMO Channel Matrix** In a narrowband MIMO system, multiple independent data streams are transmitted simultaneously over the same frequency band using multiple transmit antennas and received by multiple receive antennas. Under the so called flat-fading assumption, the channel response remains constant across the signal bandwidth, allowing the noise to be modeled as additive white Gaussian noise (AWGN) that is frequency-independent and identically distributed across all receiver antennas.

The simplest non-trivial case involves 2 transmit and 2 receive antennas. The input-output relationship of this 2×2 MIMO system, illustrated in Fig. 1, is described by the matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} \tag{1}$$

In compact matrix form:

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n}$$

where $\mathbf{y}$ represents the received signal vector, $\mathbf{x}$ is the transmitted signal vector, $\mathbf{H}$ is the $2 \times 2$ complex-valued channel matrix, and $\mathbf{n}$ is the additive noise vector. The channel matrix element $h_{ij}$ represents the complex channel gain from the $j$-th transmit antenna to the $i$-th receive antenna.

**MIMO Channel Evaluation** Spatial multiplexing is viable only when the receiver can successfully solve equation (1) to recover the transmitted signal vector $\mathbf{x}$ from the received signal $\mathbf{y}$. For any rank-deficient channel matrix (rank < 2), the transmitted signals cannot be uniquely determined. However, the feasibility of

spatial multiplexing depends not merely on the matrix rank, but critically on how well-conditioned the channel matrix is.

The singular value decomposition (SVD) provides insight into the matrix conditioning by decomposing the channel matrix into three constituent matrices:

$$\mathbf{H} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H \tag{2}$$

where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices[1] and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values $\sigma_1, \sigma_2$ of $\mathbf{H}$. Geometrically, the SVD decomposes the channel transformation into a sequence of operations: rotation (by $\mathbf{V}^H$), scaling (by $\mathbf{\Sigma}$), and rotation (by $\mathbf{U}$). This decomposition effectively decouples the MIMO channel into independent parallel single-input single-output (SISO) sub-channels, where each singular value represents the effective gain of its corresponding sub-channel.

The condition number of $\mathbf{H}$, defined as $\kappa = \sigma_{\max}/\sigma_{\min}$, quantifies the channel's suitability for spatial multiplexing. A condition number close to unity indicates well-conditioned channels with high spatial multiplexing potential, while large condition numbers suggest poor channel orthogonality and degraded performance due to increased sensitivity to noise and interference.

The SVD components are computed through eigendecomposition. The right singular vectors are obtained from:

$$\mathbf{H}^H\mathbf{H} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H \tag{3}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with eigenvalues $\lambda_i = \sigma_i^2$. The left singular vectors are then computed as:

$$\mathbf{u}_i = \frac{1}{\sigma_i}\mathbf{H}\,\mathbf{v}_i \tag{4}$$

**MIMO Channel Capacity** Channel capacity represents the theoretical maximum data rate (in bits/s/Hz) that can be achieved with arbitrarily small error probability under ideal coding and modulation.

For a MIMO channel defined by equation (1), assuming an input signal $\mathbf{x}$ of zero mean with covariance matrix[2] $\mathbf{C_x} = E\{\mathbf{x}\mathbf{x}^H\}$ and AWGN $\mathbf{n}$ with covariance $\mathbf{C_n} = \sigma^2\mathbf{I}$, the channel capacity is given by:

$$C(\mathbf{H}) = B\log_2 \det\left(\mathbf{I} + \mathbf{C_n}^{-1}\mathbf{H}\mathbf{C_x}\mathbf{H}^H\right) \quad \left[\frac{\text{bits}}{\text{s}\cdot\text{Hz}}\right] \tag{5}$$

This result, as established by Shannon's coding theorem assumes that the channel matrix is known at the receiver and that $\mathbf{x}$ is Gaussian distributed.

In the practical case where the transmitter does not have channel state information (CSI), the total power $P$ should be equally distributed across all $n_t$ transmit antennas, that is, $\mathbf{C_x} = \frac{P}{n_t}\mathbf{I}$. Defining the signal-to-noise ratio (SNR) as $\rho = \frac{P}{\sigma^2}$, equation (5) simplifies to:

$$C(\mathbf{H}) = B\log_2 \det\left(\mathbf{I} + \frac{\rho}{n_t}\mathbf{H}\mathbf{H}^H\right) \quad \left[\frac{\text{bits}}{\text{s}\cdot\text{Hz}}\right] \tag{6}$$

This expression reveals that MIMO capacity scales logarithmically with SNR and can potentially achieve a linear increase with the minimum of the number of transmit and receive antennas under favourable channel conditions. A comprehensive derivation of these results can be found in standard references such as [12].

**Water-Filling Power Allocation** Equation (5) demonstrates that maximizing channel capacity requires optimal selection of the input covariance matrix $\mathbf{C_x}$. When channel state information is available at the transmitter—typically obtained through feedback mechanisms—power can be strategically allocated across the spatial eigenmodes of the channel to maximize capacity.

This optimization leads to the water-filling algorithm, which allocates power proportional to the channel strength: stronger eigenmodes (corresponding to larger singular values) receive more power, while weaker eigenmodes may receive reduced power or be turned off entirely. The intuitive "water-filling" analogy envisions pouring a fixed amount of water over a surface with varying depths representing inverted channel gains—water naturally flows to fill the deepest areas first, corresponding to allocating more power to channels with better signal-to-noise characteristics.

The optimal input covariance matrix for a $n_t \times n_r$ MIMO system with $m = \min(n_t, n_r)$ spatial degrees of freedom is:

$$\mathbf{C_x} = \mathbf{V}\text{diag}\,(p_1, p_2, ..., p_m)\,\mathbf{V}^H \tag{7}$$

Here, $\mathbf{V}$ is a unitary matrix obtained from eigendecomposition $\mathbf{H}^H\mathbf{C_n}^{-1}\mathbf{H} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H$. The power allocation coefficients $c_i$ can be determined using the method of Lagrange multipliers subject to the total power constraint $\sum_p^m p_i = P$, yielding:

$$p_i = \max\left(\mu - \frac{1}{\lambda_i}, 0\right) \tag{8}$$

where $\mu$ is the water level (Lagrange multiplier) chosen to satisfy the power constraint, and $\lambda_i$ are the eigenvalues contained in $\mathbf{\Lambda}$.

The optimized transmitted signal vector carrying $m$ independent data streams $s_1, s_2, \ldots, s_m$ (assumed zero-mean with unit variance) is then constructed as:

$$\mathbf{x} = \mathbf{V}\text{diag}\left(\sqrt{p_1}, \sqrt{p_2}, ..., \sqrt{p_m}\right)\begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \end{bmatrix} \tag{9}$$

This precoding strategy effectively performs a change of basis to the channel's eigenmodes, enabling independent transmission over parallel sub-channels with gains proportional to the singular values. A detailed derivation of the water-filling algorithm can be found in [12].

[13, 14, 15]

[1] The superscript $H$ denotes the Hermitian (conjugate transpose).

[2] A covariance matrix is defined as: $\mathbf{C_x} = E\{(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^H\}$

### 2.3 CHANNEL ESTIMATION AND SIMULATION

**Fading** Fading represents one of the most significant challenges in wireless communication systems, particularly in MIMO configurations where multiple antenna elements must maintain reliable signal transmission across diverse propagation paths.

In wireless channels, fading occurs due to multipath propagation, where transmitted signals reach the receiver via multiple paths of varying lengths, resulting in constructive and destructive interference that causes signal amplitude and phase to fluctuate randomly over time and frequency. This phenomenon is particularly pronounced in MIMO systems, where the spatial separation of antenna elements creates additional complexity in the fading characteristics experienced by each transmission path.

To ensure reliable device performance under such conditions, complex fading scenarios have been standardized based on real-world measurements. These scenarios form part of mandatory conformance tests used to verify MIMO receiver performance (see, for example, LTE specifications [16]). As specified by 3GPP (3rd Generation Partnership Project), devices must pass such conducted fading tests to achieve certification. The purpose of these tests is to evaluate the user equipment (UE) receiver under dynamically varying channel conditions that emulate realistic wireless environments.

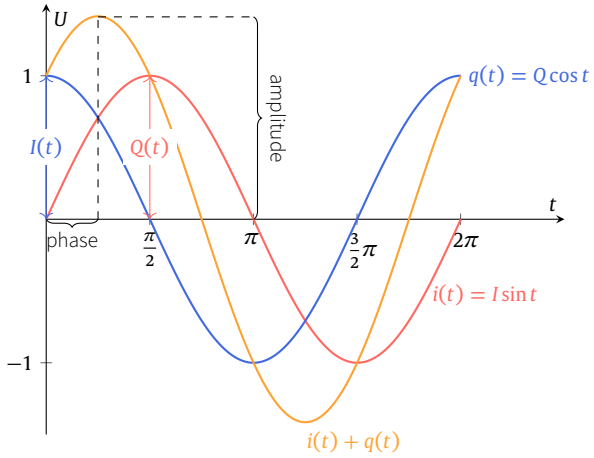
**Empirical Modelling** The 3GPP standards provide comprehensive channel models specifically designed for MIMO system evaluation. Examples include the Spatial Channel Model (SCM), the SCM-Extended (SCME), and more recent frameworks in 3GPP TR 36.873 and TR 38.901 [17, 18]. These models are derived from extensive field measurements and specify realistic fading scenarios across a wide range of deployment types: Urban Macro (UMa), Urban Micro (UMi), Rural Macro (RMa), and Indoor Hotspot (InH).

Key parameters such as angular spreads, delay spreads, path loss, and spatial correlation are carefully defined, ensuring that test conditions reflect real-world propagation. Although these models are statistical rather than geometry-based, they are validated against large-scale measurement campaigns [14], making them highly suitable for standardized system simulations. By serving as a common foundation across the industry, they enable fair comparison of MIMO performance under realistic yet reproducible conditions.

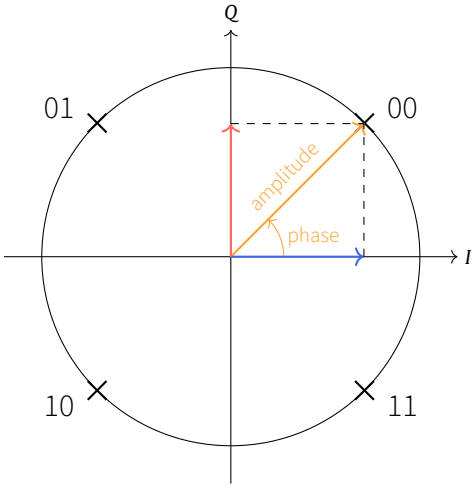
**Ray Tracing** In contrast to empirical models, ray tracing offers a deterministic and geometry-based approach to simulating wireless propagation. Ray tracing techniques model the physical interactions between electromagnetic waves and the environment, including reflection, diffraction, and scattering, based on detailed three-dimensional representations of the surroundings [19].

Advanced simulation tools such as Ansys HFSS or Remcom Wireless InSite utilize ray tracing to generate high-fidelity channel responses. Emerging platforms such as Ansys Perceive EM (API) enable the creation of digital twins—virtual replicas of real-world environments—by combining data from LiDAR scans (for precise geometry) and photogrammetry (for material classification). This allows for highly accurate modeling of propagation phenomena, particularly in complex environments like dense urban or indoor settings.

Ray tracing is particularly valuable for evaluating site-specific performance, beamforming strategies, and propagation behaviors at millimeter-wave and sub-THz frequencies, where environmental




(a) Superposition of two sinusoidal signals that are in quadrature. The resulting signal is sinusoidal and its amplitude and phase can be controlled by adjusting the *I* and *Q* values.




(b) Phasor diagram illustrating QPSK modulation. By restricting the *I* and *Q* values to ±1, four symbols i.e. two bits can be mapped to a modulation state.

Figure 2 | Illustration of vector signals.

effects dominate signal behavior [20]. While computationally demanding, it complements empirical models by providing fine-grained spatial insights and enabling scenario-specific performance evaluations that statistical models cannot capture.

## 3 VECTOR RF SIGNAL GENERATORS

### 3.1 ANALOG VS VECTOR SIGNAL GENERATORS

Two fundamental types of signal generators exist: analog and vector signal generators. Analog signal generators produce continuous wave (CW) signals and can apply analog modulations including amplitude modulation (AM), phase modulation (PM), and frequency modulation (FM). Vector signal generators, in contrast, can generate arbitrary digital waveforms by representing signals as time-dependent vectors with both magnitude and phase components, rather than single scalar values. While this capability makes vector signal generators more versatile, analog signal generators remain widely used due to their excellent signal stability and significantly lower cost.

## 3.2 VECTOR SIGNALS AND MODULATIONS

**Vector Signal Representation** Vector signals are characterized by their magnitude and phase at each point in time. Any such signal can be synthesized by combining two sinusoidal waves in quadrature—that is, with identical frequencies but phase-shifted by $\pi/2$ radians (90°). This is illustrated in Fig. 2a. The amplitudes of these orthogonal components are designated as $I(t)$ (in-phase) and $Q(t)$ (quadrature), respectively. The resulting composite signal remains sinusoidal, but its instantaneous amplitude and phase (orange in the figure) can be precisely controlled by adjusting the $I$ and $Q$ values.

**I/Q Modulation Schemes** Modulation in the context of telecommunication conveys information by changing some aspect of an RF carrier such as amplitude, phase, and/or frequency. So-called state modulation is used to transmit digital data. A state is represented by a distinct combination of amplitude and phase, i.e., $I$ and $Q$ values of an RF carrier (see Fig. 2b).

With $m$ distinct states, each symbol can represent $\log_2(m)$ bits of information. The arrangement and number of these states define the modulation scheme. For example:

- QPSK (Quadrature Phase Shift Keying): $I$ and $Q$ can take on values $\pm 1$, mapping 2 bits to one symbol (Fig. 2b). This modulation is commonly used in satellite transmission, cable modems, and cellular phone systems.
- Quadrature Amplitude Modulation (QAM) increases data rate by adding amplitude variation to phase shifts. In 64-QAM, for instance, 64 distinct I/Q combinations can encode 6 bits per symbol.

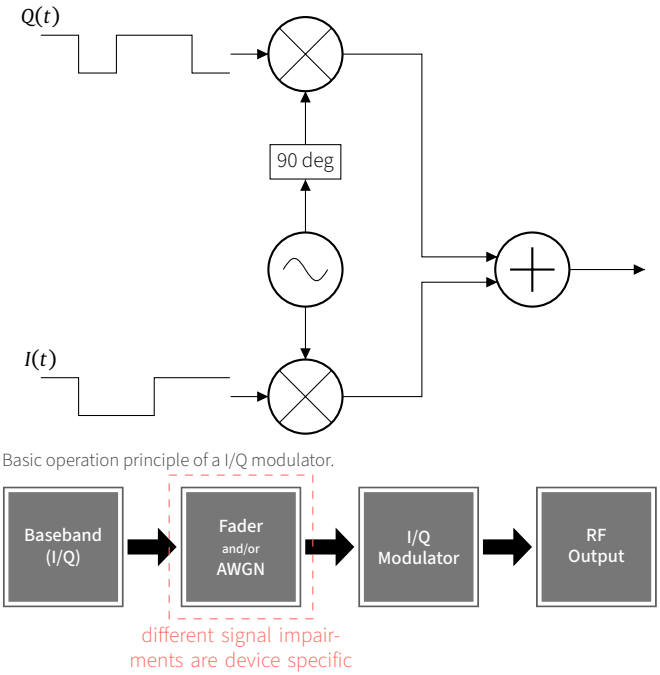
In over-the-air (OTA) systems, fewer bits per symbol are typically used compared to wired systems due to susceptibility to noise and interference. Modern communication systems often employ adaptive modulation, dynamically changing the modulation scheme based on channel conditions to balance robustness and throughput.

## 3.3 VECTOR RF SIGNAL GENERATION

The signal processing chain of vector signal generators is usually broken down into modular blocks, which allow for sophisticated signal generation and manipulation. This chain of blocks is illustrated in Fig. 3b, and the components are elaborated in the following:

**Baseband Processing** The baseband section serves as the first processing stage within a vector signal generator. Here, the desired $I$ and $Q$ components are typically generated digitally using a digital signal processor (DSP) at very low frequencies or near DC. These digital I/Q signals are subsequently converted to analog waveforms using high-resolution digital-to-analog converters (DACs). The output consists of analog voltage waveforms representing the amplitude variations of the $I$ and $Q$ components over time.

**I/Q Modulator** The I/Q modulator (see Fig. 3a) forms the core of the vector signal generator's upconversion process. A local oscillator (LO) generates a sinusoidal carrier signal at an intermediate frequency (IF), which is then split into two paths: one direct path and one phase-shifted by $\pi/2$. These LO signals are individually mixed (multiplied) with the baseband $I$ and $Q$ components,




(a) Basic operation principle of a I/Q modulator.




(b) Signal flow inside a VSG.

Figure 3 | Illustration of VSG building blocks.

respectively. The resulting modulated signals are then summed to produce the desired waveform with the correct amplitude and phase characteristics at the IF.

**RF Processing** The RF processing stage provides the final frequency translation from IF to RF. It mixes the IF signal with a high-frequency local oscillator to reach the desired RF output carrier frequency and adjusts output power as specified by the user.

**Signal Impairments** Many vector signal generators offer the capability to introduce signal impairments such as fading, various types of noise, interferers, and I/Q imbalances. These impairments simulate real-world effects commonly observed in RF systems and applications. Consequently, the ability to define and control these impairments is crucial for the design, testing, and debugging of RF devices.

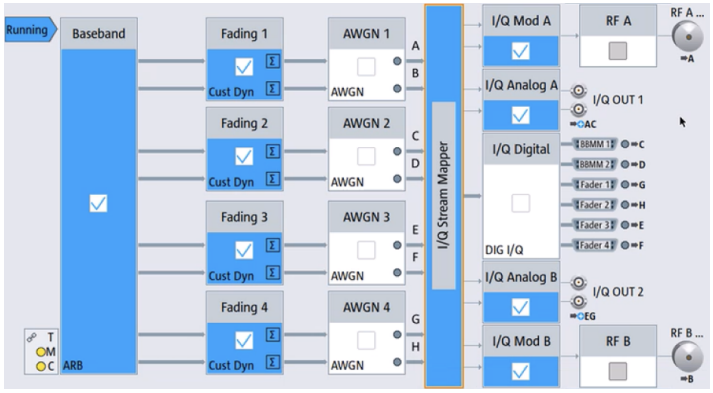
**Trigger Output** The trigger output provides a marker signal indicating the start of waveform generation or specific timing events within the signal sequence. This synchronized trigger can be used to coordinate measurements with external instruments such as oscilloscopes or spectrum analyzers. The trigger signal is typically available through a BNC connector for easy connection to test equipment.

## 3.4 THE R&S SMW200A VECTOR SIGNAL GENERATOR

The Rohde & Schwarz SMW200A vector signal generator is considered to be best commercially available of its kind. The objective of this proof-of-concept application is to showcase its standout features, emphasizing its suitability for testing and measurement tasks aligned with current communication standards such as LTE and 5G, as well as for the development of future 6G technologies particularly involving digital twin cityscape approaches.

**Technical Capabilities** Key capabilities include its support for advanced MIMO system simulation and sophisticated dynamic fading modeling. For MIMO applications, the SMW200A supports

(a) Snapshot of the Ansys Perceive EM simulation scenario.




(b) SMW200A configuration for the Ansys Perceive EM scenario.

Figure 4 | Ansys Perceive scenario currently implemented the demonstration application.

configurations of up to eight transmit and receive antennas, or alternatively, two independent 2×2 MIMO systems. In terms of fading simulation, up to four fading blocks can be configured, each supporting up to 12 dynamic paths. These paths can replicate standardized 3GPP channel models or be freely customized directly on the device. As discussed earlier, channels can also be modeled using external simulation tools. By converting the simulation output into a format compatible with the SMW200A, the device can accurately reproduce the simulated channel conditions—enabling user equipment to be tested under virtually any conceivable scenario.

**Ansys Perceive EM Demonstration Scenario** One of the scenarios included in the demo application, was simulated using Ansys Perceive EM. The scenario involves two mobile units: a ground vehicle and an airborne drone navigating through an urban environment. Both receive signals from a base station positioned atop a tall building. Fig. 4a depicts a snapshot of the scenaro visualization. In this setup, the SMW200A is configured as a 4×2×2 MIMO system, utilizing all available fading paths—24 per user in total, equally divided between the two users (vehicle and drone) across two fading blocks. This configuration is illustrated in Fig. 4b. Since the simulation files already include noise, the SMW's AWGN (Additive White Gaussian Noise) blocks are disabled.

# 4 DEMONSTRATION APPLICATION DOCUMENTATION

## 4.1 OVERALL OBJECTIVE

The goal of the demonstration application is to provide a graphical user interface (GUI) that allows a user to remotely connect to the Rohde & Schwarz SMW200A[3] vector signal generator and select from a set of predefined scenarios generated by a simulation tool. Based on the selected scenario, the SMW should automatically configure itself using appropriate parameters. Once configuration is complete, the user can initiate the playback of the scenario, during which the SMW generates the corresponding RF signals in real time. These signals can be observed on an oscilloscope connected to the generator. Simultaneously, the GUI plays back a video visualization of the scenario—generated by the simulation tool—synchronized with the signal output of the SMW.

## 4.2 CHOOSING THE FLASK FRAMEWORK

The SMW200A supports remote control via SCPI (Standard Commands for Programmable Instruments) and provides an official Python API, which facilitates streamlined device communication. As a result, Python is the natural choice for the core of this application.

Although it is possible to create graphical user interfaces in Python using libraries like Tkinter or PyQt6, such desktop-based GUIs can become limiting in terms of scalability, extensibility, and design customization—especially as the application evolves to support more complex features. To avoid these constraints, this project adopts a browser-based architecture, where the GUI is rendered in the browser using HTML, CSS, and JavaScript, offering greater flexibility and user experience possibilities.

To integrate Python-based device communication with a web-based front end, a web framework is needed. Popular Python web frameworks include Django, Flask, and FastAPI. Django is generally suited for large-scale, full-stack applications with built-in ORM and administrative tooling, while Flask and FastAPI are more lightweight and modular — ideal for custom applications with specific backend needs.

For this project, Flask was chosen due to its simplicity and minimal setup overhead, making it well-suited for rapid prototyping under time constraints. Its relatively gentle learning curve also allows experienced web developers to quickly integrate Python back-end functionality with front-end components. However, in hindsight, FastAPI would have been a more suitable choice, particularly due to its native support for asynchronous request handling, which could offer improved performance and responsiveness in future iterations of the application.

## 4.3 FLASK BASICS

Flask is a lightweight and minimalist web framework for Python that reduces web request handling to its most fundamental building block: the function. A typical Flask application starts with a Python file (e.g., app.py) where the Flask class is instantiated to create the application object. Routes are defined using Python decorators, such as `@app.route('/')`, which map URL paths to Python functions (known as view functions).

[3]Henceforth also referred to as just SMW

To render HTML content in response to client requests, Flask uses the `render_template()` function. This function loads HTML templates stored in a directory named templates. Flask integrates the Jinja2 templating engine, which allows embedding Python variables and control structures into HTML. For instance:

```
render_template('control_smw.html', ip=session['ip'])
```

would render `control_smw.html` with `ip` available for use within the template.

The IP address previously provided by the user is stored in the Flask session, which enables data persistence across requests. Flask sessions are stored client-side in signed cookies, ensuring integrity via the server's secret key. Session data persists for the duration of the session — typically until the user closes their browser, logs out, or after a configured timeout. The session can be modified during any request context using the `session` object.

Static assets such as JavaScript, CSS, and image files are placed in a separate directory named `static`. These assets can be referenced in templates using the Jinja `url_for()` function, for example:

```
<script src="{{ url_for('static', filename='js/
    connect_smw.js') }}"></script>
```

**Application Structure** The demonstration application follows a standard Flask project layout:

```
flask_demo_app/
├── smw_controller/
│   ├── smw_manager.py
│   └── ...
├── threading_helper/
│   └── thread_configurer.py
├── ...
├── static/
│   ├── js
│   │   ├── connect_smw.js
│   │   └── ...
│   ├── css
│   └── ...
├── templates/
│   ├── index.html
│   ├── control_smw.html
│   └── ...
└── app.py
```

The application currently includes two main pages:

- Index Page (`index.html`): Allows the user to enter the static IP address of the SMW200A to establish a connection.
- Scenario Control Page (`control_smw.html`): Becomes accessible once the device connection is established and allows users to configure and run simulation scenarios.

**Basic Routing Logic** The application logic ensures that the user cannot access the scenario page (`control_smw.html`) unless a valid connection to an SMW is present. If an IP address exists in the session and the connection succeeds, the user is redirected automatically to the scenario page. Otherwise, they are sent back to the index page.

The following code excerpt demonstrates this core routing behavior:

```
# app.py
from flask import Flask, render_template, session,
    redirect, url_for
import smw_manager

app = Flask(__name__)
app.secret_key = 'your_secret_key'  # Required for
    session handling

@app.route('/')
def index():
  ip = session.get('ip')
  if ip:
    try:
      smw_manager.get_smw(ip)
      return render_template('control_smw.html', ip=ip
    )
    except Exception as e:
      session.pop('ip', None)
      return render_template('index.html', error=f"SMW
     not found: {e}")
  return render_template('index.html')

@app.route('/control_smw')
def control_smw_page():
  if 'ip' not in session:
    return redirect(url_for('index'))
  return render_template('control_smw.html', ip=
    session['ip'])

@app.route('/disconnect_smw', methods=['GET', 'POST'])
def disconnect_smw():
  session.pop('ip', None)
  smw_manager.disconnect()
  return redirect(url_for('index'))

if __name__ == '__main__':
  app.run(debug=True, host='0.0.0.0', port=5001)
```

→ Note: The secret key must be set in a real deployment for secure session handling. In production, this should be managed via environment variables or configuration files rather than hardcoded.

## 4.4 USING SCPI AND PYTHON WITH R&S DEVICES

Modern Rohde & Schwarz (R&S) instruments support remote control via SCPI (Standard Commands for Programmable Instruments), a standardized ASCII-based command protocol used across test and measurement equipment. For Python integration, R&S provides a common API library called RsInstrument, which supports a wide range of devices. For specific instruments like the SMW200A Vector Signal Generator, an extended device-specific library—`RsSmw`—is available. This library offers a higher-level, Pythonic interface built on top of the SCPI protocol, tailored to the unique capabilities of the SMW200A.

SCPI commands are typically divided into write and query types:

- Write Command example:

  ```
  'SOURce1:FREQuency 3 GHz'
  ```

- Query Command example:

  ```
  'SOUR1:FREQ?'
  ```

  The `?` suffix turns it into a query, returning the current frequency setting of the source.

While SCPI keywords are case-insensitive, the convention is to capitalize required characters and use lowercase for optional parts for readability.

Using Python, these commands can be transmitted through objects instantiated from either `RsInstrument` or the device-specific `RsSmw`. Queries and write commands of `RsSmw`-objects can be sent

using the `.utilities.query()` and `.utilities.write()` methods respectively. Alternatively, `RsSmw` provides high-level wrapper methods that abstract away the raw SCPI calls.

The following code shows a basic example of initializing connections to an SMW200A and a compatible R&S oscilloscope:

```
from RsInstrument import RsInstrument
from RsSmw import RsSmw

ip_mxo = '10.102.80.10'
ip_smw = '10.102.80.11'

# Connection to R&S MXO4 Oscilloscope using generic
    RsInstrument
mxo = RsInstrument(f"TCPIP::{ip_mxo}::HISLIP")

# Connection to SMW200A using device-specific RsSmw
    API
smw = RsSmw(f"TCPIP::{ip_smw}::HISLIP")

# Query device identification
idn_smw = smw.utilities.query('*IDN?')
idn_mxo = mxo.query('*IDN?')

# Set communication timeout to 4 seconds
smw.timeout = 4000
```

## 4.5 INTEGRATION GOALS FOR THE DEMO APPLICATION

The primary objective of the demo application is to allow the user to configure and control the SMW200A for playback of simulation-driven scenarios. As mentioned earlier, these scenarios are generated externally (e.g., by a digital twin simulation tool). While the baseband waveform can be arbitrary, the appropriate MIMO configuration with user dynamic fading has to be set. Moreover, the fading blocks have to be provided with the files containing the simulation data. Specifically, these files contain the time dependent gains (`.fad_udyn` files) and phases (`.txt` files) for each fading-path (12 for each fading block). These files have to be located somewhere on the SMW. Once configured, the chosen scenario should be emulated in real-time by the signal generator.

The process is divided into two main stages:

1. Configuration Phase: The SMW is configured based on the selected scenario parameters. This stage may take up to 2 minutes, depending on scenario complexity.
2. Playback Phase: Once configured, the SMW enters a standby state, awaiting a user-triggered ”play” command. The signal output should loop endlessly until the user issues a stop command.

To ensure a responsive user experience, the app should:

- Provide live feedback on the configuration status.
- Allow user control over starting and stopping playback.
- Prevent unintentional triggering during configuration.

These requirements will be addressed in the backend using Python threading to handle long-running tasks and Flask-SocketIO for real-time communication between the server and the browser-based frontend. Both components will be detailed in the following sections.

## 4.6 PYTHON THREADING

Threading in Python enables concurrent execution of code, which is essential for tasks that would otherwise block the main program flow. In this demonstration application, threading is used to offload long-running hardware configuration processes—such as setting up the SMW or the connected MXO oscilloscope—to background threads. This prevents the web server from becoming unresponsive during configuration operations.

**Configuration Threading Design** To separate threading logic from the actual configuration code, a reusable class called `ThreadConfigurer` was implemented. This class provides a clean interface to run arbitrary functions in background threads, avoiding duplication of threading logic across the application. The key benefits of this design include:

- Modularity: Any configuration function can be passed to the `ThreadConfigurer` instance without modifying its internal logic
- Testability: The threading mechanism remains generic and easy to test in isolation
- Extensibility: New configuration functions can be added without changing the threading infrastructure
- Thread Safety: Proper synchronization prevents race conditions when accessing shared hardware resources

### ThreadConfigurer Class Breakdown

```
import threading
from typing import Optional, Callable, Any, Dict

class ThreadConfigurer:

  def __init___(self, device_lock):
    self._thread: Optional[threading.Thread] =
    None
    self._thread_lock = device_lock
```

→ `_thread` stores the currently running thread.

→ `_thread_lock` is a `threading.Lock()`-instance used to prevent race conditions, which can occur when multiple threads attempt to access shared resources (e.g., the SMW object) concurrently. This is especially important because many hardware APIs are not thread-safe.

```
def is_running(self) -> bool:
  with self._thread_lock:
    return self._thread is not None and self.
    _thread.is_alive()
```

→ This method checks whether a background thread is currently active.

```
def run_function(self, func: Callable, *args, **
    kwargs) -> Any:
    if self.is_running():
      return None

    with self._thread_lock:
      def thread_wrapper():
        try:
          func_result = func(*args, **kwargs)
        except Exception as e:
          error_msg = f"Error in ThreadConfigurer.
    run_function: {str(e)}"
```

```
      self._thread = threading.Thread(target=
    thread_wrapper)
      self._thread.daemon = True
      self._thread.start()
      return None
```

→ This is the core utility of the `ThreadConfigurer`.

→ It wraps the provided `func` callback in a new thread, preventing blocking operations from halting the main program (or Flask app).

→ The thread is marked as daemon, ensuring it doesn't prevent program termination.

```
def stop_thread(self, timeout: float =
    DEFAULT_TIMEOUT) -> bool:
  if not self.is_running():
    return True
  with self._thread_lock:
    if self._thread and self._thread.is_alive():
      self._thread.join(timeout=timeout)

    if self._thread.is_alive():
        return False
    return True
  return True
```

→ Gracefully attempts to stop the running thread using `join()`, which blocks until the thread terminates or the timeout is reached.

→ Returns `True` if successful, `False` otherwise.

```
def cleanup(self) -> None:
  if self.is_running():
    print("Cleaning up running thread")
    if not self.stop_thread():
      print("Could not cleanly stop thread during
    cleanup")
```

→ This method can be called during shutdown or when reconfiguring the device to ensure no threads are left running.

**Usage Example** The scenario configuration implementation demonstrates proper usage of the `ThreadConfigurer`:

```
# smw_controller/configure_smw.py
import atexit
from  threading_helper.thread_configurer import
    smwConfigurer

# Register cleanup on exit
atexit.register(smwScenarioConfigurer.cleanup)

def configure_smw_for_scenario(*args, **kwargs)
  # configuration logic here...

def configure_scenario(args, kwargs):
  """Wrapper function to run the configuration in a
    thread-safe manner."""
  return smwConfigurer.run_function(
    configure_smw_for_scenario, *args, **kwargs)
```

→ It is recommended to use the same `ThreadConfigurer` instance (singleton pattern) created with a `smw_lock` (`threading.Lock()`-instance) at the module level as `smwConfigurer` for all SMW-related configuration to ensure thread safety and prevent race conditions.

→ It is important to always call `cleanup()` during application shutdown to prevent resource leaks.

→ The `*args` and `**kwargs` pattern provides flexible function argument handling, allowing any configuration function to be wrapped without modifying the `ThreadConfigurer` implementation. This makes the system highly extensible for different types of hardware configuration tasks.

**Scenario Playback Threading Design** The scenario playback system requires different logic from the configuration threading outlined previously. The primary distinction is that the playback function must run as a potentially endless loop that can be terminated at any time by the user. To address this requirement, a specialized class called `ScenarioRunner` was created to handle all backend logic related to scenario playback.

This class implements a state machine architecture using a helper enum class `ScenarioStatus` to track and manage the various phases of scenario execution.

```
from enum import Enum
from typing import Optional, Dict, Any
import threading
import time
import atexit

class ScenarioStatus(Enum):
  IDLE = "idle"
  RELOADING = "reloading"
  RUNNING = "running"
  ERROR = "error"
  STOPPING = "stopping"
```

The state transitions follow this logical flow:

- `IDLE` → `RELOADING` → `RUNNING` → `IDLE`
- Any state can transition to `ERROR` or `STOPPING` based on conditions
- `STOPPING` always transitions back to `IDLE`

**ScenarioRunner Class Breakdown:**

```
class ScenarioRunner:
  def __init__(self,smw_lock):
    self._loop_thread: Optional[threading.Thread]
    = None
    self._stop_event = threading.Event()
    self._status = ScenarioStatus.IDLE
    self._thread_lock = smw_lock
    self._error_message: Optional[str] = None

    # Register cleanup on application exit
    atexit.register(self.cleanup)
```

→ `_loop_thread`: Thread object responsible for running the scenario loop.

→ `_stop_event`: A `threading.Event()` used to signal graceful termination to the running thread. Unlike locks, events can be checked from within loops without blocking.

→ `_status`: Tracks the current state using the `ScenarioStatus` enum

→ `_thread_lock`: Thread safety is ensured by using the same `threading.Lock()`-instance passed to the `smwConfigurer`-object.

→ `_error_message`: Stores error details if a failure occurs

```
@property
def status(self) -> Dict[str, Any]:
  status_dict = {"status": self._status.value}
  if self._error_message and self._status ==
    ScenarioStatus.ERROR:
    status_dict["message"] = self._error_message
  return status_dict
```

→ Exposes scenario runner state and error messages in a dictionary format.

→ The `@property` decorator allows access to status information using `runner.status` instead of `runner.status()` providing a clean, attribute-like interface for read-only data.

```
@property
def is_running(self) -> bool:
  return (self._loop_thread is not None and self.
    _loop_thread.is_alive() and not self.
    _stop_event.is_set())
```

→ Indicates whether the scenario loop is actively running.

```
def _set_status(self, status: ScenarioStatus,
    error_message: Optional[str] = None) -> None:
    self._status = status
    self._error_message = error_message
    if status == ScenarioStatus.ERROR:
      print(f"Error: {error_message}")
    else:
      print(f"Status changed to: {status.value}")
```

→ Updates internal state and optionally logs error messages when applicable.

```
def run_scenario_smw(self, smw, duration: int) ->
    None:
    try:
      # Reload phase
      self._set_status(ScenarioStatus.RELOADING)
      smw.utilities.write_str(':ENTity1:SOURce1:BB
    :ARBitrary:TRIGger:ARM:EXECute')

      # Check for early stop signal
      if self._stop_event.is_set():
        self._set_status(ScenarioStatus.IDLE)
        return

      # Wait for reload with cancelation checks
      reload_remaining = RELOAD_TIME
      while reload_remaining > 0 and not self.
    _stop_event.is_set():
        sleep_time = min(0.5, reload_remaining)  #
     Check every 0.5 seconds
        time.sleep(sleep_time)
        reload_remaining -= sleep_time

      # Check for stop signal again before
    continuing
      if self._stop_event.is_set():
        self._set_status(ScenarioStatus.IDLE)
        return

      # Verify operation complete
      response = smw.utilities.query('*OPC?')
      print(f"OPC response: {response}")

      # Execution phase
      self._set_status(ScenarioStatus.RUNNING)
      smw.utilities.write_str(':ENTity1:SOURce1:BB
    :ARBitrary:TRIGger:EXECute')

      # Wait for execution to complete with
    periodic checks
      execution_remaining = duration
      while execution_remaining > 0 and not self.
    _stop_event.is_set():
        sleep_time = min(0.5, execution_remaining)
      # Check every 0.5 seconds
        time.sleep(sleep_time)
        execution_remaining -= sleep_time

      # Execution completed or was interrupted
      if self._stop_event.is_set():
        self._set_status(ScenarioStatus.STOPPING)

    except Exception as e:
      self._set_status(ScenarioStatus.ERROR, str(e
    ))
      raise
```

→ Execute a single iteration of the SMW scenario with proper state management.

→ This method contains the core SMW hardware control logic for scenario playback. It implements a two-phase execution model: reload/preparation followed by execution.

→ Phase 1: `smw.utilities.write_str(':ENTity1:SOURce1:BB:ARBitrary:TRIGger:ARM:EXECute')` restarts the faders and prepares the "arbitrary" baseband entity for the trigger. The SMW is given a predefined time inside the variable `RELOAD_TIME` by the execution of `time.sleep(RELOAD_TIME)`.

→ Phase 2: `smw.utilities.write_str(':ENTity1:SOURce1:BB:ARBitrary:TRIGger:EXECute')` executes the trigger by which the SMW plays outputs the scenario specific RF signals. Using `time.sleep(duration)` the backend can determine when the playback has finished.

→ By breaking the sleep period into smaller intervals, the thread can regularly check for a stop signal, enabling responsive user-driven cancellation during any phase of playback.

```
def loop(self, smw, duration: int) -> None:
    print("Starting scenario loop")
    try:
      while not self._stop_event.is_set():
      try:
        self.run_scenario_smw(smw, duration)
        if not self._stop_event.is_set():
          print("Scenario iteration completed
    successfully")
      except Exception as e:
        if not self._stop_event.is_set():  # Only
    log as error if not intentionally stopping
          self._set_status(ScenarioStatus.ERROR,
    str(e))
          print(f"Error in scenario execution: {
    str(e)}")
          break

      print("Scenario loop stopped normally")
    finally:
    # Ensure we're in a clean state when the loop
    ends for any reason
      if self._status != ScenarioStatus.ERROR:
        self._set_status(ScenarioStatus.IDLE)
```

→ The `loop()` method runs continuously until an external stop is triggered.

→ Ensures clean exit to `IDLE` unless an error occurred.

```
def start_loop(self, **kwargs) -> bool:
    ip = kwargs.get('ip', None)
    duration = kwargs.get('duration', None)
    if duration is None:
      raise ValueError("duration is required")

    with self._thread_lock:
      # Clear any previous error state
      self._error_message = None

      # Check if a thread is already running
      if self.is_running:
        print("Loop is already running")
        return False

      # Handle potentially zombie thread
      if self._loop_thread and self._loop_thread.
    is_alive():
        print("Thread exists but in an unexpected
    state - forcing cleanup")
        self._stop_event.set()
```

```
      try:
        self._loop_thread.join(timeout=2)
      except:
        pass
      self._loop_thread = None

      # Setup and start new thread
      try:
        smw = smw_manager.get_smw(ip)
        self._stop_event.clear()
        self._set_status(ScenarioStatus.IDLE)

        self._loop_thread = threading.Thread(
        target=self.loop,
        args=(smw, max(1, duration)),
        name="ScenarioLoop",
        daemon=True  # Make thread daemon so it
    won't prevent program exit
        )
        self._loop_thread.start()
        print(f"Loop thread started with duration
    {duration}s")
        return True

      except Exception as e:
        self._set_status(ScenarioStatus.ERROR, str
    (e))
        print(f"Failed to start loop: {str(e)}")
        raise
```

→ Starts a new scenario loop in a background thread.

→ Ensures stale or zombie threads are cleaned up before starting.

```
def stop_loop(self) -> bool:
    with self._thread_lock:
      if not self._loop_thread or not self.
    _loop_thread.is_alive():
        print("No active thread to stop")
        self._loop_thread = None  # Ensure
    reference is cleared
        self._set_status(ScenarioStatus.IDLE)
        return False

      try:
        # Signal the thread to stop
        self._stop_event.set()
        self._set_status(ScenarioStatus.STOPPING)
        print("Stop signal sent to thread")

        # Try to send stop command to device
        try:
        smw = smw_manager.get_smw()
        smw.utilities.write_str(':ENTity1:SOURce1:
    BB:ARBitrary:TRIGger:ARM:EXECute')
        print("Stop signal sent to SMW device")
      except Exception as e:
        print(f"Error sending stop signal to SMW:
    {str(e)}")

      # Wait for thread to finish with timeout
      print("Waiting for thread to terminate...")
      self._loop_thread.join(timeout=10)

      # Check if thread actually terminated
      if self._loop_thread.is_alive():
        print("Thread did not stop within timeout
    - marking as zombie")
        # We'll clear the reference anyway to
    allow new threads

      # Reset thread reference and status
      self._loop_thread = None
      self._set_status(ScenarioStatus.IDLE)
      return True

      except Exception as e:
        print(f"Error stopping loop: {str(e)}")
        # Ensure thread reference is cleared even
    on error
        self._loop_thread = None
        self._set_status(ScenarioStatus.IDLE)
        raise
```

→ Sends an interrupt to the running loop and performs cleanup.

→ Attempts to send a soft stop command to the SMW hardware.

```
def cleanup(self) -> None:
    if self._loop_thread and self._loop_thread.
    is_alive():
      print("Cleaning up running thread during
    shutdown")
      self._stop_event.set()
      try:
        smw = smw_manager.get_smw()
        smw.utilities.write_str(':ENTity1:SOURce1:
    BB:ARBitrary:TRIGger:ARM:EXECute')
      except Exception as e:
        print(f"Error sending stop signal during
    cleanup: {str(e)}")

      try:
        self._loop_thread.join(timeout=5)
      except Exception:
      pass

    self._loop_thread = None
```

→ Automatically called on application shutdown to ensure clean termination of any active thread.

**Usage** At the module level, a singleton instance of `ScenarioRunner` is created and exposed via convenience functions:

```
from threading_helper.device_locks import smw_lock
# Create singleton instance
runner = ScenarioRunner(smw_lock)

# Module-level convenience functions
def start_loop(data: dict) -> bool:
  print(data)
  return runner.start_loop(**data)

def stop_loop() -> bool:
  return runner.stop_loop()

def get_status() -> Dict[str, Any]:
  return runner.status

def is_running() -> bool:
  return runner.is_running
```

These functions are easily callable from routes or services within `app.py`.

### 4.7 FLASK-SOCKETIO

WebSocket is a communication protocol that enables continuous, bidirectional data exchange between client and server, making it well-suited for real-time applications such as this demonstration system. Flask provides support for WebSocket-like functionality through the `Flask-SocketIO` extension, which builds on the Socket.IO protocol. On the server side, its usage begins with the initialization of a `SocketIO` instance. This instance exposes the necessary methods for sending and receiving real-time data to and from the frontend:

```python
# app.py
from flask_socketio import SocketIO

socketio = SocketIO(app)
```

While the configuration and scenario playback code can already run in background threads, the frontend currently has no mechanism to receive updates about their progress or status. For example, the frontend must know when SMW configuration has completed before enabling playback. Similarly, during playback, the backend should be able to trigger a restart of the frontend video to ensure proper synchronization with the generated signal.

To meet these requirements, a lightweight helper function can be introduced for status communication:

```python
# socket_helper.py
from flask_socketio import SocketIO

def update_socket_status(socketio: SocketIO,
    event_type: str, status_data: dict) -> None:
  socketio.emit(event_type, status_data)
```

→ The `socketio.emit` method allows sending custom events, ensuring that the frontend only needs to listen for relevant event types (e.g., `'configuring_scenario_status'`).

**Usage Example** By refactoring the configuration logic into smaller subtasks, each step can notify the frontend of its current progress via the helper function. A simplified version of the refactored code looks as follows:

```python
# configuration code (excerpt)

import socket_helper

def configure_smw_for_scenario(socketio: SocketIO, *
    args, **kwargs) -> Dict[str, Any]:
  scenario = kwargs.get('scenario', None)

  try:
    status_data = {
      'status': 'configuring',
      'scenario': scenario,
      'step': 'connecting'
    }
    socket_helper.update_socket_status(socketio,
    'configuring_scenario_status', status_data)

    smw = smw_manager.get_smw(ip)

    status_data = {
      'status': 'configuring',
      'scenario': scenario,
      'step': 'configuring baseband'
    }
    socket_helper.update_socket_status(socketio,
    'configuring_scenario_status', status_data)

    configure_baseband(smw, output_configuration)

    # ...additional configuration steps...

    return {"status": "OK"}

  except Exception as e:
    error_msg = str(e)
    status_data = {
      'status': 'error',
      'scenario': scenario,
      'message': error_msg
    }
    socket_helper.update_socket_status(socketio,
    'configuring_scenario_status', status_data)
    return {"status": f"Error: {error_msg}"}


def configure_scenario(socketio: SocketIO, data: dict)
    -> Dict[str, Any]:
  """Wrapper function to run the configuration in a
    thread-safe manner."""
  return smwScenarioConfigurer.run_function(
  configure_smw_for_scenario, socketio, **data
  )
```

## 4.8 ASYNCHRONOUS FRONTEND INTEGRATION: AJAX AND SOCKET.IO

**AJAX Integration** The HTTP protocol follows a request–response model, where each interaction is initiated by the client. The two predominant request types are `GET` and `POST`. A `GET` request appends parameters to the URL as query strings, making it unsuitable for sensitive or large amounts of data. In contrast, a `POST` request sends data in the body of the request, which is both more secure and capable of handling larger payloads.

Traditional web applications rely on synchronous form submissions that trigger complete page reloads, even when updates are required for only small interface components. This approach introduces several drawbacks: unnecessary computational overhead, loss of user context (including scroll position and form state), and a degraded user experience characterized by visual flickering and perceived latency.

AJAX (Asynchronous JavaScript and XML) addresses this issue by enabling asynchronous communication between client and server. With AJAX, the browser can send requests and process responses in the background, updating only the relevant parts of the page without forcing a reload. Although its name references XML, modern AJAX implementations primarily use JSON due to its lightweight structure and native compatibility with JavaScript.

**SocketIO Integration** The frontend implementation utilizes Socket.IO's JavaScript client library, which executes directly within the browser environment. The library can be included through an appropriate CDN, however to ensure offline functionality and reduce external dependencies, the Socket.IO library is hosted locally after being downloaded from the official Socket.IO distribution.

```html
<script src="{{ url_for('static', filename='js_plugins
    /socket.io.min.js')}}"></script>
```

Initializing an `io()`-object[4] establishes a persistent bidirectional connection with the backend server, enabling the client to both listen for and respond to custom events emitted by the server:

```javascript
const socket = io();
// receive messages from the server
socket.on('message', (data) => {
  console.log('Received from server:', data);
});

// send messages to the server
socket.emit('sendMessage', 'Hello from client!');
```

**Architectural Integration** The frontend of this demonstration application combines the two asynchronous communication techniques to provide both responsiveness and real-time feedback:

- **AJAX (Fetch API)** is used to initiate server-side actions such as configuring the SMW. These requests are asynchronous,

[4] Note: Earlier versions used `io.connect()`.

preventing the UI from freezing and avoiding full page reloads.

- **Socket.IO (WebSockets)** is used to receive continuous status updates from the server during long-running tasks, such as scenario configuration, enabling fine-grained user feedback in real time.

The application's JavaScript implementation adopts a class-based architecture. The scenario page (`control_smw.html`) implements two primary functionalities: SMW configuration and scenario playback, managed by the `ConfigureScenarioController` and `Scenario` classes respectively.

The SMW configuration frontend logic centers on the `applyScenario` and `handleStatusUpdate()` methods within the ConfigureScenarioController class:

ConfigureScenarioConroller–AJAX and SocketIO Integration Breakdown

```
class ConfigureScenarioController {
  constructor() {
    this.currentController = null;
    this.elements = {};
    this.socket = io();
    this.initializeElements();
    this.bindEvents();
  }


  initializeElements() {
      const elements = {
      form: document.getElementById('scenarioForm'
    ),
      configBtn: document.getElementById('
    configBtn'),
      // Additional required DOM elements
  }

  bindEvents() {
    // Socket event handlers
    this.socket.on('configuring_scenario_status',
    (data) => this.handleStatusUpdate(data));
    this.socket.on('configuring_scenario_error', (
    error) => {
      console.error('Socket connection error:',
    error);
      this.updateUI('Connection error', false);
    });
    // DOM event listeners are bound here
  }
```

→ The `io()` function establishes a persistent connection to the backend.

→ DOM element verification ensures reliable interface manipulation

→ Event binding centralizes both Socket.IO and DOM event management

```
async applyScenarioConfiguration() {
  const formData = this.getFormData();

  if (!formData.scenario) {
    this.updateUI('Invalid scenario selected',
    false);
    return;
  }

  this.elements.configBtn.disabled = true;
  this.elements.dropdownBlocker.classList.add('
    active');
  this.updateUI('<div class="loader"></div><span>
    Starting configuration...</span>');
  this.disableButtons();

  try {
    const response = await fetch('/configure-smw-
    for-scenario', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json
    ' },
      body: JSON.stringify(formData),
    });

    const data = await response.json();
    if (!response.ok) throw new Error(data.error
    || 'Unknown error');
  } catch (err) {
    this.updateUI(`${err.message}`, false);
    this.elements.configBtn.disabled = false;
    this.elements.dropdownBlocker.classList.remove
    ('active');
  }
}
```

→ **AJAX trigger:** When a user selects a configuration scenario, the frontend dispatches a **POST** request to initiate the backend configuration process.

→ This request serves exclusively to initiate the configuration process. Given that configuration operations may require several minutes to complete, actual progress monitoring is handled asynchronously through the Socket.IO channel.

→ The Fetch API is used to send form data asynchronously to the backend, avoiding page reloads and maintaining a smooth user experience.

→ A loading state informs the user that configuration is in progress, while error handling ensures clear feedback for both client- and server-side issues.

→ The asynchronous flow ensures that the GUI remains responsive while the SMW is being configured in the background.

```
handleStatusUpdate(data) {
  switch (data.status) {
    case SCENARIOCONFIGCONSTANTS.STATES.
    CONFIGURING:
    const stepMessage = data.step || 'Applying
    settings...';
    this.updateUI(`<div class="loader"></div><span
    >${stepMessage}</span>`);
    break;

    case SCENARIOCONFIGCONSTANTS.STATES.SUCCESS:
    // update session after successful
    configuration
    fetch('/update-session', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json
    ' },
      body: JSON.stringify(data),
    });
    this.updateUI('<span>SMW successfully
    configured.</span>');
    this.elements.configBtn.disabled = false;
    this.elements.dropdownBlocker.classList.remove
    ('active');
    this.enableButtons();
    break;

    case SCENARIOCONFIGCONSTANTS.STATES.ERROR:
    this.updateUI(`<span>Error: ${data.message ||
    'Unknown error'}</span>`);
    this.elements.configBtn.disabled = false;
    this.elements.dropdownBlocker.classList.remove
    ('active');
    break;

    default:
```

```
    console.warn('Unknown status:', data.status);
    this.elements.configBtn.disabled = false;
    this.elements.dropdownBlocker.classList.remove
    ('active');
  }
}

updateUI(content, isHTML = true) {
  if (isHTML) {
    this.elements.statusDiv.innerHTML = content;
  } else {
    this.elements.statusDiv.textContent = content;
  }
}
```

- **Socket.IO Status Updates:** This frontend code listens for real-time updates from the backend and updates the UI accordingly.
- The client monitors `"configuring_scenario_status"` events emitted by the server, with event types organized in the `SCENARIOCONFIGCONSTANTS` helper class for maintainability
- Each incoming update dynamically modifies the `statusDiv` DOM element, keeping the user informed about progress.
- Button enabling occurs only after successful configuration, ensuring safe frontend-backend synchronization.
- Following successful configuration, state parameters are persisted in Flask's `session` object via AJAX, as session modifications require request context.

This combination ensures that user actions are immediately acknowledged, while longer-running processes are monitored transparently with real-time updates. To guarantee that all DOM elements are available before initialization, the controller is instantiated within a `'DOMContentLoaded'` event handler:

```
// Initialize controller when DOM is ready
document.addEventListener('DOMContentLoaded', () => {
  try {
    window.configureScenarioController = new
    ConfigureScenarioController();
    // Initialize with default scenario
    configureScenarioController.updateScenarioInfo(
    SCENARIOCONFIGCONSTANTS.VIDEO.DEFAULT_SCENARIO);
  } catch (error) {
    console.error('Failed to initialize settings
    controller:', error);
  }
});
```

### 4.9 ADDITIONAL NOTES AND CONCLUDING REMARKS

The demonstration application features fully custom styling, including dropdown menus, video pop-ups, and interactive buttons. While this traditional approach required additional coding effort, it was initially preferred due to time constraints that limited opportunities for a learning-by-doing approach with more modern technologies. Alternatively, leveraging a frontend library such as React with a component library like MUI, or adopting a utility-first framework such as Tailwind CSS, could have simplified and accelerated the styling process. Nevertheless, the current design effectively fulfills its intended purpose.

Beyond the user interface, the application also integrates functionality to convert simulation data into SMW-compatible formats, such as `.udyn_fad` and phase-list files. This feature is partially implemented: the current pipeline produces intermediate results, but finalization still relies on a MATLAB script. A future improvement will involve rewriting this MATLAB processing step in Python to achieve full integration within the app.

Overall, the application successfully demonstrates the intended proof of concept and highlights the advanced fading capabilities of the R&S SMW200A. It has already been showcased at several events, including the 6G Summit in Dresden, where it effectively communicated both the technical potential and the user experience of this state-of-the-art platform.

## REFERENCES


[1] Mehmet Ulema. "Long Term Evolution (LTE)". In: *Fundamentals of Public Safety Networks and Critical Communications Systems: Technologies, Deployment, and Management*. 2019, pp. 121–144. DOI: 10.1002/9781119369554.ch8.

[2] Alexander Kukushkin. "4G-Long Term Evolution (LTE) System". In: *Introduction to Mobile Network Engineering: GSM, 3G-WCDMA, LTE and the Road to 5G*. 2018, pp. 205–291. DOI: 10.1002/9781119484196.ch11.

[3] 3rd Generation Partnership (3GPP). *3GPP Releases*. URL: https://portal.3gpp.org/#/55934-releases (visited on 07/30/2025).

[4] Powertec Telecommunications. *LTE Evolutions | Powertec Information Portal*. URL: https://portal.powertec.com.au/technical-library/wireless/wireless-technologies/cellular/4g-lte/lte-evolutions (visited on 07/30/2025).

[5] Wireless Excellence Limited. *LTE 3GPP releases Overview*. URL: https://www.cablefree.net/wirelesstechnology/4glte/overview-of-lte-3gpp-releases (visited on 07/30/2025).

[6] Tej Raj et al. "Advances in MIMO Antenna Design for 5G: A Comprehensive Review". In: *Sensors* 23.14 (2023). ISSN: 1424-8220. DOI: 10.3390/s23146329. URL: https://www.mdpi.com/1424-8220/23/14/6329.

[7] Qualcomm Technologies. *Making 5G NR a reality*. 2016. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/making_5g_nr_a_reality.pdf (visited on 07/30/2025).

[8] Yusuf Olayinka Imam-Fulani et al. "5G Frequency Standardization, Technologies, Channel Models, and Network Deployment: Advances, Challenges, and Future Directions". In: *Sustainability* 15.6 (2023). ISSN: 2071-1050. DOI: 10.3390/su15065173. URL: https://www.mdpi.com/2071-1050/15/6/5173.

[9] Francesco Pica and Dr. Lola Awoniyi-Oteri. *Path to 6G: Envisioning next-gen use cases for 2030 and beyond*. URL: https://www.qualcomm.com/news/onq/2024/06/path-to-6g-envisioning-next-gen-use-cases-for-2030-and-beyond (visited on 07/30/2025).

[10] Danny Kai Pin Tan et al. "Integrated Sensing and Communication in 6G: Motivations, Use Cases, Requirements, Challenges and Future Directions". In: *2021 1st IEEE International Online Symposium on Joint Communications & Sensing (JC&S)*. 2021, pp. 1–6. DOI: 10.1109/JCS52304.2021.9376324.

[11] Telefonaktiebolaget LM Ericsson. *6G networks*. URL: https://www.ericsson.com/en/6g (visited on 07/30/2025).

[12] B.P. Lathi and Zhi Ding. *Modern Digital and Analog Communication Systems*. Fifth Edition. Oxford series in electrical and computer engineering. New York, NY: Oxford University Press, 2019. ISBN: 9780190686840. URL: https://lccn.loc.gov/2017034966.

[13] Andrea Goldsmith. "Capacity of Wireless Channels". In: *Wireless Communications*. Cambridge University Press, 2005, pp. 99–125.

[14] Andreas F. Molisch. *Wireless Communications*. 2nd ed. Wiley, 2012.

[15] Heinz Mellein and Manuel Mielke. "Assessing a MIMO Channel". In: Rohde & Schwarz. Aug. 2017. URL: https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_application/application_notes/8nt1/8NT1_0e_Assessing_MIMO_Ch.pdf.

[16] *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; E-UTRA; User Equipment (UE) conformance specification; Radio transmission and reception; Part 1: Conformance testing (Release 16)*. Tech. rep. TS 36.521-1. 3GPP, 2020.

[17] *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Study on 3D channel model for LTE (Release 12)*. Tech. rep. TR 36.873. 3GPP, 2017.

[18] *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Study on channel model for frequencies from 0.5 to 100 GHz (Release 14)*. Tech. rep. TR 38.901. 3GPP, 2018.

[19] Steffen Jaeckel et al. "QuaDRiGa: A 3-D multi-cell channel model with time evolution for enabling virtual field trials". In: *2014 IEEE Vehicular Technology Conference (VTC Spring)*. IEEE. 2014, pp. 1–5.

[20] Theodore S. Rappaport et al. "Wireless Communications and Applications Above 100 GHz: Opportunities and Challenges for 6G and Beyond". In: *IEEE Access* 7 (2019), pp. 78729–78757.