Technical Documentation: Implementing and Interfacing with Graphical User Interfaces in Cloud-Based Jupyter Environments

Introduction

Purpose and Scope

This document provides a definitive technical guide for implementing and interacting with Graphical User Interfaces (GUIs) within remote Jupyter notebook environments. The scope encompasses major cloud GPU platforms, including Google Colab, Vast.ai, and Lightning.ai. The primary objective is to detail every viable methodology for creating interactive UIs that can be controlled and utilized by end-users without requiring direct access to the localhost of the cloud-based virtual machine. The methodologies are categorized into three distinct strategies: native in-notebook GUI frameworks, frameworks that transform notebooks into standalone web applications, and advanced cloud-to-local architectures that treat the notebook kernel as a powerful, remotely accessible backend.

The Localhost Barrier

A fundamental technical challenge in cloud-based notebook environments is the "localhost barrier." When a web server or service is initiated within a cloud virtual machine—be it a Jupyter server, a Flask application, or a custom service—it typically binds to a port on the machine's local network interface, such as 127.0.0.1:8000 or localhost:8866.[1] In a local development setup, this address is directly accessible from a web browser on the same machine. However, cloud VMs are sandboxed environments; their network interfaces are not exposed to the public internet by default, and firewalls typically block all incoming traffic.[3] Consequently, an end-user cannot simply type the cloud VM's
localhost address into their browser and connect. This document outlines the canonical

solutions to circumvent this barrier, enabling the creation of rich, interactive applications on powerful cloud hardware that are accessible from anywhere.

Methodological Overview

The solutions to the localhost barrier can be broadly classified into three strategic approaches, each with distinct use cases and levels of complexity:

1. **In-Notebook GUI Frameworks:** These libraries render interactive HTML elements directly within the output cells of a Jupyter notebook. They are ideal for data exploration, interactive parameter tuning, and building simple tools that coexist with the code and its narrative. Key frameworks in this category include Ipywidgets, Gradio, and Panel.
2. **Notebook-as-Application Frameworks:** These tools transform an entire Jupyter notebook into a standalone, interactive web application, often hiding the underlying code to present a polished user interface. This approach requires a mechanism to expose the application's port to the public internet. Frameworks like Voila and Streamlit exemplify this strategy.
3. **Advanced Cloud-to-Local Architectures:** This advanced approach decouples the GUI from the notebook entirely. The notebook, running on a cloud GPU, is converted into a powerful backend microservice, exposing its functionality through an API or a real-time communication channel. A separate client application, which can be a local desktop program or another web service, then interacts with this backend. This architecture is enabled by technologies such as FastAPI and WebSockets.

Target Audience and Prerequisites

This document is intended for an audience of professional AI/ML developers, data scientists, and MLOps engineers. It assumes a strong proficiency in Python, extensive experience with the Jupyter notebook ecosystem, and a working familiarity with fundamental cloud computing concepts. The code examples and architectural patterns presented are designed for practical implementation in real-world projects.

Cloud Platform Considerations

While the techniques described are broadly applicable across Linux-based cloud environments, platform-specific nuances can affect implementation.

- **Google Colab:** As a highly customized and somewhat restrictive Jupyter environment, Colab often requires specific workarounds. For instance, its cell-based rendering isolation can necessitate special initialization calls for certain libraries, a detail that will be highlighted where relevant.[6]

- **Vast.ai & Lightning.ai:** These platforms typically provide more standard, robust JupyterLab environments running in Docker containers.[9] This often leads to a more seamless experience, particularly for tools that are deeply integrated with the standard Jupyter ecosystem. For example, Vast.ai explicitly recommends running Jupyter directly over attempting to connect a Colab frontend to a "local runtime" on their service, citing better stability.[10] Lightning AI Studios also provide a fully integrated environment for development and deployment, with native support for frameworks like Gradio and Streamlit.[13]

Part 1: In-Notebook GUI Frameworks

This section details libraries that enable the creation of interactive GUIs directly within the output cells of a Jupyter notebook. This approach is powerful for augmenting the traditional, static nature of notebooks, allowing for dynamic data exploration and interactive control over code execution. The frameworks discussed represent a spectrum of abstraction. Ipywidgets offers a low-level, deeply integrated toolkit that feels native to the notebook's cell-based structure. In contrast, Gradio and Panel provide higher-level abstractions, enabling the construction of self-contained applications that are rendered within a single output cell, treating the notebook more as a development and hosting environment than a document to be augmented. The choice between these tools often depends on whether the goal is to add a simple control to a document (ipywidgets) or to embed a complete application within it (Gradio or Panel).

1.1 Ipywidgets: The Native Interactive Toolkit

Ipywidgets are the foundational library for bringing interactivity to Jupyter notebooks. They provide a rich set of UI controls that bridge the gap between the Python kernel where code is executed and the user's browser where the notebook is displayed.[15]

1.1.1 Core Concepts and Architecture

The functionality of ipywidgets is rooted in a synchronized, dual-component architecture. For every widget, there is a Python object (a Widget instance) residing in the Jupyter kernel and a corresponding JavaScript object (a WidgetModel instance) in the browser front-end.[17] These two components maintain a synchronized state. Any change to a widget's property in the browser (e.g., a user moving a slider) is automatically communicated to the Python object in the kernel, and conversely, any programmatic change to the widget's state in Python is

reflected in the browser's UI.[17]
This synchronization is facilitated by a communication layer known as "Comms," an asynchronous messaging API built into the Jupyter protocol. Comms allow for the exchange of JSON-able data blobs between the kernel and the front-end, abstracting away the complexities of WebSockets and the underlying web server.[17]
Dependencies and Installation

- **Installation:** The library can be installed via standard package managers:
  - pip install ipywidgets [15]
  - conda install -c conda-forge ipywidgets [16]
- **Environment:** In modern Jupyter environments like JupyterLab 3.0+ and the classic Notebook, the necessary front-end extensions are typically enabled automatically upon installation of the Python package.[16] In Google Colab,
  ipywidgets is pre-installed and functions natively without any additional configuration required for basic usage.[19]

1.1.2 The interact Function: The Simplest Path to Interactivity

The easiest way to begin using ipywidgets is through the interact function. This high-level utility automatically generates UI controls by introspecting the signature of a Python function and creating widgets for its arguments.[19]
Basic Example: Function-driven UI
The interact function maps argument types to appropriate widgets, creating a simple and intuitive way to explore code.

Python

Show codeCopy
```
# Full Code Entry
# This demonstrates the automatic widget generation of the interact function.
# As seen in [19, 21]
from ipywidgets import interact
import ipywidgets as widgets

def f(x):
  return x

# An IntSlider is generated for an integer keyword argument.
print("Integer Slider:")
interact(f, x=10);
```

```
# A Checkbox is generated for a boolean.
print("\nCheckbox:")
interact(f, x=True);

# A Text widget is generated for a string.
print("\nText Box:")
interact(f, x='Hello World!');
```

Widget Abbreviations

interact supports a shorthand syntax for specifying widget ranges and options, which is highly effective for rapid prototyping. A tuple of numbers like (min, max, step) creates a slider, while a list of strings or tuples creates a dropdown menu.19

Type Annotations

Modern Python code benefits from type hints, and interact can leverage them to infer the correct widget type. Annotating a function argument with bool will generate a checkbox, str will generate a text box, and so on. This practice leads to cleaner and more self-documenting code.19

1.1.3 A Comprehensive Catalog of Core Widgets

Beyond interact, the ipywidgets library provides a wide array of explicit widget classes for more granular control over the UI.

- **Numeric Widgets:** For handling integer and float values.
    - IntSlider, FloatSlider: Standard sliders for selecting a value within a range.[22]
    - IntRangeSlider, FloatRangeSlider: Allow for the selection of a min/max range.[23]
    - FloatLogSlider: A slider with a logarithmic scale, useful for parameters that span several orders of magnitude.[22]
    - BoundedIntText, BoundedFloatText: Text boxes that accept numeric input but are constrained to a specific range.[20]
- **Boolean Widgets:** For true/false states.
    - Checkbox: A standard checkbox.[22]
    - ToggleButton: A button that can be toggled on or off.[16]
    - Valid: A read-only indicator, useful for displaying status (e.g., success/failure).[22]
- **Selection Widgets:** For choosing from a list of options.
    - Dropdown: A classic dropdown menu.[22]
    - RadioButtons: A set of radio buttons for mutually exclusive choices.[22]
    - SelectMultiple: A list box that allows for multiple selections.[22]
- **String Widgets:** For text input and display.
    - Text, Textarea: Single-line and multi-line text input fields.[22]
    - Password: A text input that obscures its content. Note that this is not a secure method for collecting sensitive information, as the value is transmitted unencrypted and may be stored in plain text if the notebook is saved.[22]

    - Label: Displays static text and can render LaTeX equations.[24]
- **Button and Output Widgets:**
    - Button: A standard clickable button used to trigger actions.[23]
    - Output: A widget that can capture and display output from other code, including text, images, and other widgets.[20]
- **File and Display Widgets:**
    - Image: Displays an image from a file or byte string.[23]

1.1.4 Advanced Usage: Layout, Styling, and Event Handling

For creating complex user interfaces, ipywidgets provides powerful tools for arranging widgets and handling user interactions.
Layout and Containers
Individual widgets can be organized into sophisticated layouts using container widgets. These containers use a flexbox model to control the alignment, orientation, and distribution of their children.24

- HBox: Arranges child widgets horizontally.
- VBox: Arranges child widgets vertically.
- Accordion: Stacks children in a vertically collapsible group.
- Tabs: Displays children in separate, selectable tabs.
- GridBox: Arranges children in a two-dimensional grid.[23]

The following example demonstrates how to combine several widgets into a functional application layout and connect them with event handlers.

Python

Show codeCopy

```
# Full Code Entry
# This example builds a simple interactive application using layout containers and event
handling.
# Based on concepts from [20, 24]
from ipywidgets import HBox, VBox, Button, IntSlider, Label, Output
from IPython.display import display
import matplotlib.pyplot as plt
import numpy as np

# Create widgets
slider = IntSlider(description='Frequency:', min=1, max=10, value=2)
button = Button(description='Generate Plot')
output_area = Output()
```

```
# Define event handler for the button
def on_button_click(b):
    with output_area:
        output_area.clear_output(wait=True)
        freq = slider.value
        x = np.linspace(0, 2 * np.pi, 200)
        y = np.sin(x * freq)

        fig, ax = plt.subplots()
        ax.plot(x, y)
        ax.set_title(f'Sine Wave with Frequency = {freq}')
        plt.show()

# Link the event handler to the button's on_click event
button.on_click(on_button_click)

# Arrange widgets in containers
controls = HBox([slider, button])
app_layout = VBox([controls, output_area])

# Display the final layout
display(app_layout)
```

The interactive_output Function
For more decoupled application logic, interactive_output allows a widget's value to be connected to a function's input without creating a direct UI for the function itself. This is useful when a single control needs to update multiple, separate outputs.21
Low-Level Widget API
For developers needing to create entirely new types of widgets, ipywidgets provides a low-level API. By subclassing DOMWidget in Python and creating a corresponding JavaScript DOMWidgetView, one can build custom widgets that integrate seamlessly into the Jupyter ecosystem. This extensibility is a key feature of the framework, with cookiecutter templates available to streamline the development of custom widget libraries.16

1.2 Gradio: Rapid Prototyping and Effortless Sharing

Gradio is an open-source Python library designed to create clean, user-friendly web interfaces for machine learning models, APIs, or any Python function with minimal code. It is particularly well-suited for creating demos and prototypes directly from a notebook environment.[26]

1.2.1 Core Concepts: Functions as Apps

Gradio's core philosophy is to abstract the process of UI creation into a simple mapping. A developer provides three key components:

1. A Python function to execute (fn).
2. A description of the expected user inputs (inputs).
3. A description of the function's outputs (outputs).

Gradio then automatically generates a complete web interface based on these components.[26]

Dependencies and Installation

- **Installation:** Gradio is installed via pip and requires Python 3.10 or higher.
    - pip install --upgrade gradio [26]

1.2.2 The Interface Class

The gr.Interface class is the high-level, quickest way to build a Gradio app. It is ideal for straightforward models where inputs map directly to outputs.

Basic Example: Image-to-Text Model

This example demonstrates a simple image classification interface. A dummy function simulates a model's prediction.

Python

Show codeCopy

```
# Full Code Entry
# This conceptual example demonstrates the simplicity of the gr.Interface class.
# Based on Gradio's principles from [26, 27]
import gradio as gr
import numpy as np
from PIL import Image

# Dummy function simulating a model prediction
def image_classifier(image: Image.Image) -> dict:
  # In a real application, this would involve a trained model.
  # For demonstration, we return fixed probabilities.
  return {"cat": 0.7, "dog": 0.3}

demo = gr.Interface(
  fn=image_classifier,
  inputs=gr.Image(type="pil", label="Upload an Image"),
  outputs=gr.Label(num_top_classes=2, label="Classification Result"),
    title="Simple Image Classifier",
    description="Upload an image of a cat or a dog to see the model's prediction."
)

demo.launch()
```

1.2.3 The Magic of share=True

A standout feature of Gradio is its integrated sharing capability, which directly addresses the "localhost barrier." The developers of Gradio recognized that deployment and networking are often the biggest hurdles to sharing an ML demo.[26] By integrating a tunneling service directly into the
launch() method, they made sharing trivial. This design choice is a primary reason for Gradio's popularity, especially for creating quick demos and for hosting on platforms like Hugging Face Spaces.[30]
When demo.launch(share=True) is called, Gradio generates a temporary, publicly accessible URL (e.g., https://12345.gradio.app). This URL forwards all traffic through a secure tunnel to the Gradio server running within the sandboxed cloud VM. This allows anyone with the link to interact with the model remotely, while all computation remains on the cloud instance. These public links typically expire after 72 hours, making them ideal for temporary sharing and collaboration.[26]

1.2.4 Advanced Layouts with gr.Blocks

For applications that require more than a simple input-output mapping, gr.Blocks provides a flexible, low-level API for designing custom layouts and complex, multi-step data flows.[26]
With Blocks, components are instantiated within a with gr.Blocks() as demo: context manager. Layout elements like gr.Row(), gr.Column(), and gr.Tab() allow for precise control over the UI's appearance. The application's interactivity is defined by event listeners (e.g., .click(), .change(), .submit()) that connect components to functions, specifying the flow of data.[28]
Advanced Example: Multi-step Image Processing Chain
This example demonstrates a more complex UI with multiple tabs and chained operations, showcasing the power of gr.Blocks.

Python

Show codeCopy

```
# Full Code Entry
# This example illustrates custom layouts and chained event listeners using gr.Blocks.
# Based on concepts from [32, 33, 59]
import gradio as gr
import numpy as np
from PIL import Image, ImageOps

def grayscale(image):
    if image is None:
        return None
    return image.convert("L")

def invert_colors(image):
    if image is None:
        return None
    return ImageOps.invert(image)

with gr.Blocks(theme=gr.themes.Soft()) as demo:
    gr.Markdown("## Multi-Step Image Processing Pipeline")

    with gr.Tab("Image Operations"):
        with gr.Row():
            input_img = gr.Image(type="pil", label="Input Image")
            processed_img = gr.Image(label="Processed Image")
        with gr.Row():
            grayscale_btn = gr.Button("Convert to Grayscale")
            invert_btn = gr.Button("Invert Colors")

    with gr.Tab("Text Operations"):
        text_input = gr.Textbox(label="Input Text")
        text_output = gr.Textbox(label="Reversed Text")
        reverse_btn = gr.Button("Reverse Text")

    # Define the data flows with event listeners
    grayscale_btn.click(fn=grayscale, inputs=input_img, outputs=processed_img)
    invert_btn.click(fn=invert_colors, inputs=input_img, outputs=processed_img)
    reverse_btn.click(fn=lambda s: s[::-1], inputs=text_input, outputs=text_output)

demo.launch(share=True)
```

1.2.5 Extensions and Ecosystem

The Gradio ecosystem extends beyond basic UI creation. It includes specialized components like gr.ChatInterface for rapidly building chatbots, as well as Python and JavaScript client libraries (gradio_client, @gradio/client) that allow any Gradio app to be used programmatically as an API.[26] Furthermore, its seamless integration with Hugging Face Spaces provides a free and robust platform for permanently hosting Gradio applications.[30]

1.3 Panel: High-Performance Interactive Dashboards

Panel, a core component of the HoloViz ecosystem, is a powerful Python library for building complex, interactive dashboards and data exploration tools. It is designed to integrate smoothly with a wide range of plotting libraries and data science tools, making it an excellent choice for data-intensive applications.[35]

1.3.1 Core Concepts: Reactive Programming and the HoloViz Ecosystem

Panel's design is centered around a reactive programming model. This means that relationships are defined between widgets and functions, and the UI automatically updates when a dependency changes. It has a "batteries-included" philosophy, offering first-class support for popular visualization libraries like Bokeh, Plotly, and Matplotlib, as well as deep integration with other HoloViz tools such as hvPlot (for simple plotting APIs) and Datashader (for visualizing massive datasets).[35]

Dependencies and Installation

- **Installation:** Panel can be installed via standard package managers:
  - pip install panel [35]
  - conda install panel [35]

1.3.2 Building Dashboards in Notebooks

A Panel application is constructed from two primary building blocks: Panes and Layouts.

- **Panes:** A Pane is a wrapper around a single object that makes it displayable. Panel has panes for nearly any type of Python object, including plots, dataframes, images, and markdown.[6]
- **Layouts:** Layouts (also called Panels) like pn.Column, pn.Row, and pn.Tabs are containers used to arrange panes and widgets into a structured dashboard.

Interactivity is often achieved using pn.bind, a function that links the values of widgets to the arguments of a Python function. When a widget's value changes, the function is automatically re-run, and the output pane is updated.

Basic Example: Interactive Plot

This example creates a dashboard with two sliders that control the frequency and amplitude of a sine wave plot.

Python

Show codeCopy

```
# Full Code Entry
# This example demonstrates the use of pn.bind for creating a reactive dashboard.
# Adapted from [35]
import panel as pn
import numpy as np
import matplotlib.pyplot as plt

# Load the extension in the notebook
pn.extension()

def create_plot(freq=1.0, ampl=1.0):
    fig = plt.figure(figsize=(6, 4))
    ax = fig.add_subplot(111)
    x = np.linspace(0, 2 * np.pi, 200)
    y = np.sin(x * freq) * ampl
    ax.plot(x, y)
    ax.set_ylim(-2.5, 2.5)
    ax.grid(True)
    plt.close(fig) # Prevents a static version of the plot from displaying
    return fig

# Create widgets
freq_slider = pn.widgets.FloatSlider(name='Frequency', start=0.1, end=5.0, step=0.1,
value=1.0)
ampl_slider = pn.widgets.FloatSlider(name='Amplitude', start=0.1, end=2.0, step=0.1,
value=1.0)

# Bind the function to the widgets
interactive_plot_pane = pn.bind(create_plot, freq=freq_slider, ampl=ampl_slider)

# Arrange components into a dashboard layout
dashboard = pn.Column(
    '## Interactive Sine Wave Plot',
    pn.Row(freq_slider, ampl_slider),
    interactive_plot_pane
)

# Display the dashboard in the notebook cell
dashboard
```

1.3.3 Special Considerations for Google Colab

Panel's functionality in Google Colab is a clear illustration of how non-standard notebook environments can require specific workarounds. Standard Jupyter platforms maintain a persistent communication channel for the entire notebook session. Google Colab, however, appears to isolate the JavaScript execution context for each individual output cell.[7] This isolation prevents the Panel JavaScript libraries, which are loaded by pn.extension(), from persisting between cells.
As a result, developers must adopt one of two strategies:

1. Call pn.extension() at the beginning of *every cell* that is intended to render a Panel object.
2. Call pn.extension(comms='colab') once at the beginning of the notebook. This special configuration automatically injects the necessary JavaScript into every subsequent cell output, ensuring functionality at the cost of increasing the notebook's file size.[8]

Required Setup for Google Colab

Python

Show codeCopy

```
# Full Code Entry
# This cell should be run at the start of any Google Colab notebook using Panel.
# Based on instructions from [7, 60]

# Install/update the necessary packages
!pip install panel hvplot -U

import panel as pn

# Load the Panel extension. In Colab, this must be done in every cell
# that will display a Panel object, or use the comms='colab' argument once.
pn.extension()
```

Furthermore, there is a known incompatibility: the IpyWidgets pane, which allows for the embedding of ipywidgets within a Panel layout, does not function correctly in Google Colab. This means developers must use Panel's native widgets (pn.widgets) when working in this environment.[7]

Part 2: Transforming Notebooks into Standalone Web

Applications

This section explores frameworks that elevate a Jupyter notebook from a development document into a fully-fledged, standalone web application. This approach is ideal for delivering polished, interactive dashboards and tools to end-users, particularly those who are not interested in the underlying code.
A common challenge with this approach in a cloud environment is network accessibility. When a framework like Voila or Streamlit starts a web server, it runs on a port within the isolated cloud VM, making it unreachable from the public internet. The universal enabler for overcoming this is **tunneling**. A tunneling service, such as ngrok, creates a secure reverse proxy. A process inside the VM initiates an outbound connection to the service's public servers. This connection is then used to relay inbound traffic from a public URL back to the specified port on the VM, effectively and securely bypassing the firewall and the localhost barrier.[36] Mastering the pattern of launching a web application and exposing it with a tunneling service is a cornerstone of deploying interactive apps from cloud notebook environments.

2.1 Voila: From Notebook to Production Dashboard

Voila is a tool designed specifically to render Jupyter notebooks as interactive web applications. It executes the notebook from top to bottom, captures all outputs (plots, widgets, markdown), and serves them as a clean HTML page. By default, it hides all code cells (strip_sources=True), presenting a professional dashboard to the user.[39] Critically, Voila maintains a dedicated, live Jupyter kernel for each user session, which is what allows ipywidgets to remain interactive and drive the application's logic.[40]

2.1.1 Dependencies and Installation

- **Installation:** Voila and its dependencies can be installed via pip. For remote deployment from a notebook, a tunneling library like pyngrok is also required.
    - pip install voila ipywidgets pyngrok [37]
- **Core Dependencies:** Voila builds upon nbconvert for notebook conversion and jupyter_server for its web server capabilities.[40] Any libraries used within the notebook itself must also be installed in the environment.

2.1.2 End-to-End Tutorial: Deploying a Voila Dashboard from Google Colab

This tutorial provides a complete, step-by-step guide for creating and deploying a Voila

dashboard from a Google Colab notebook.
Step 1: Install Dependencies in Colab
This cell installs Voila, the necessary ipywidgets, and pyngrok for tunneling.

Python

Show codeCopy

```
# Full Code Entry
!pip install voila ipywidgets pyngrok
```

Step 2: Create the Notebook Content (dashboard.ipynb)
The %%writefile magic command is used to save the JSON content of a simple notebook to a file in the Colab environment. This notebook contains an IntSlider and a Label that updates to show the square of the slider's value.

Python

Show codeCopy

```
# Full Code Entry
# This cell uses %%writefile to create a.ipynb file on the Colab file system.
%%writefile dashboard.ipynb
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs":,
   "source":
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 4
}
```

Step 3: Set Up and Run ngrok

This cell configures pyngrok with an authentication token and starts a public tunnel to port 8866, which is the default port for Voila.

Python

Show codeCopy

```
# Full Code Entry
# This cell configures and starts the ngrok tunnel.
# Based on patterns from [5, 38, 61]
from pyngrok import ngrok
import os

# Terminate any existing ngrok tunnels
ngrok.kill()

# Get your authtoken from https://dashboard.ngrok.com/get-started/your-authtoken
NGROK_AUTH_TOKEN = "YOUR_NGROK_AUTH_TOKEN"  # <-- PASTE YOUR NGROK AUTH TOKEN HERE
ngrok.set_auth_token(NGROK_AUTH_TOKEN)

# Open a tunnel to the Voila port (default is 8866)
public_url = ngrok.connect(8866)
print(f"✅ Voila dashboard is now publicly available at: {public_url}")
```

Step 4: Launch Voila in the Background

This final command starts the Voila server, pointing it to the dashboard.ipynb file. The nohup command and the & symbol ensure the process runs in the background and continues to run even after the Colab cell has finished executing.

Python

Show codeCopy

```
# Full Code Entry
# This command launches the Voila server as a background process.
# Based on [42, 58]
!nohup voila --no-browser --port=8866 dashboard.ipynb &
```

The user can now navigate to the public URL provided by ngrok to view and interact with their live dashboard.

2.1.3 Customization and Extensions

Voila's appearance can be significantly altered using templates. For example, the voila-material template provides a Material Design look and feel.[41] For more permanent deployments, platforms like Binder, Heroku, and Google App Engine are common choices that offer more stability and features than a temporary
ngrok tunnel.[42]

2.2 Streamlit: Building Data-Centric Applications

Streamlit is an open-source framework that turns Python scripts into shareable web applications with a remarkably simple API. Its core execution model differs from Voila's; on each user interaction (e.g., moving a slider), Streamlit re-runs the entire script from top to bottom to update the UI.[43] This straightforward, script-based approach makes it incredibly fast for building data-focused applications.

2.2.1 Core Concepts and Installation

- **Scripting to Apps:** The central idea is to write a standard Python script. Calls to st. functions (e.g., st.slider, st.write) create interactive widgets and display content in the resulting web app.[43]
- **Dependencies and Installation:**
  - pip install streamlit pyngrok [37]

2.2.2 End-to-End Tutorial: Deploying a Streamlit App from Google Colab

Since Streamlit operates on .py files, a common workflow in a notebook environment is the "script-in-a-cell" pattern. The %%writefile magic command is used to save the content of a cell as a Python script, which can then be executed by the Streamlit server.
Step 1: Install Dependencies

Python

Show codeCopy
```
# Full Code Entry
!pip install streamlit pyngrok
```
Step 2: Write the Streamlit App to a File (app.py)
This cell defines a simple Streamlit app and saves it to app.py.

Python

Show codeCopy

```
# Full Code Entry
# This cell uses %%writefile to create the app.py script.
# Based on [43, 57, 62]
%%writefile app.py
import streamlit as st
import pandas as pd
import numpy as np

st.set_page_config(layout="wide")

st.title('Interactive Data App in Colab')

st.sidebar.header('User Input Parameters')
num_points = st.sidebar.slider('Number of points', 10, 1000, 100)

st.write(f"Generating a chart with {num_points} data points.")

# Generate some random data
chart_data = pd.DataFrame(
    np.random.randn(num_points, 2),
    columns=['a', 'b'])

st.line_chart(chart_data)
```

Step 3: Set Up and Run ngrok

This cell starts a tunnel to port 8501, the default port for Streamlit.

Python

Show codeCopy

```
# Full Code Entry
from pyngrok import ngrok

# Terminate any existing tunnels
ngrok.kill()

# Authenticate ngrok
NGROK_AUTH_TOKEN = "YOUR_NGROK_AUTH_TOKEN" # <-- PASTE YOUR NGROK AUTH
TOKEN HERE
ngrok.set_auth_token(NGROK_AUTH_TOKEN)
```

```
# Open a tunnel to the Streamlit port
public_url = ngrok.connect(8501)
print(f"✅ Streamlit app is now publicly available at: {public_url}")
```

Step 4: Launch Streamlit in the Background
This command executes the app.py script using the Streamlit server.

Python

Show codeCopy

```
# Full Code Entry
# This command launches the Streamlit server as a background process.
# Based on [51, 57, 62]
!nohup streamlit run app.py --server.port 8501 &
```

As an alternative to ngrok, localtunnel can also be used. However, it may require the user to enter their public IP address on a security landing page before accessing the app.[45]

2.2.3 Deployment and Ecosystem

While ngrok is excellent for temporary sharing, the standard and recommended path for deploying public Streamlit apps is the Streamlit Community Cloud. It is a free hosting platform that integrates directly with GitHub repositories, automatically updating the deployed app whenever new code is pushed.[46]

Part 3: Advanced Cloud-to-Local GUI Architectures

This section details the most flexible and powerful paradigm for creating interactive applications with cloud backends: decoupling the GUI entirely from the Jupyter notebook. In this model, the notebook, with its access to powerful GPUs and a pre-configured environment, transitions into a dedicated microservice. It exposes its computational capabilities through a well-defined interface, such as a REST API or a WebSocket server. The GUI, which can be a native desktop application, a web front-end, or even another script, runs on the user's local machine and communicates with the cloud backend over the network. This architecture is necessitated by requirements for custom, high-performance, or non-web UIs. To facilitate this, a lightweight yet powerful web framework like FastAPI is ideal for creating the API layer due to its performance and automatic documentation features.[48] For applications requiring real-time, low-latency interaction, such as streaming inference results or controlling a simulation, the standard HTTP request-response cycle introduces too much overhead. This latency issue drives the adoption of the WebSocket protocol, which establishes a persistent, bidirectional communication channel for instantaneous data exchange.[48]

3.1 The Tunneling Foundation: A Deeper Look at ngrok

For these advanced architectures, ngrok remains the foundational technology that makes the cloud-based backend service accessible to the local client.

3.1.1 How Tunneling Works

The ngrok service operates as a reverse proxy. The pyngrok client, running inside the cloud VM, establishes a secure, outbound connection to ngrok's globally distributed servers. This connection is kept alive and used to create a public-facing URL. When an external client sends a request to this public URL, ngrok's servers relay the traffic through the established tunnel back to the specific port on the localhost of the cloud VM where the backend service (e.g., a FastAPI or WebSocket server) is listening. This entire process securely bypasses the VM's restrictive firewall rules.[36]

3.1.2 Programmatic Control with pyngrok

The pyngrok library provides a convenient Python API to manage the tunneling process from within the notebook.

- ngrok.set_auth_token("YOUR_TOKEN"): Authenticates the client with the ngrok service. This is a required step.[5]
- ngrok.connect(port, protocol): Opens a new tunnel. The port specifies the local port to expose, and protocol can be 'http' (for web traffic) or 'tcp' (for raw TCP traffic, often used for WebSockets or SSH).[37] The function returns a tunnel object containing the public URL.
- ngrok.disconnect(public_url) and ngrok.kill(): Used to programmatically close specific tunnels or all active tunnels, respectively.[51]

3.2 API-Driven Interaction with FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python. Its key features—including exceptional speed, automatic data validation via Pydantic, and self-generating interactive documentation (Swagger UI)—make it an outstanding choice for creating a robust API backend within a Jupyter notebook.[49]

3.2.1 End-to-End Tutorial: Cloud GPU as an API Backend

This tutorial demonstrates how to run a FastAPI server in a Colab notebook to expose a machine learning function and how to call it from a local Python client.
Step 1: The Cloud Notebook Server (api_server.ipynb)
This notebook sets up and runs a FastAPI server in a background thread to keep the main notebook cell available.

Python

Show codeCopy

```
# Full Code Entry
# This notebook cell installs dependencies, defines, and launches the FastAPI server.
# Based on concepts from [63, 64, 65]

# 1. Install dependencies
!pip install fastapi "uvicorn[standard]" pyngrok nest_asyncio

# 2. Import libraries
from fastapi import FastAPI, HTTPException
import uvicorn
from pyngrok import ngrok
import nest_asyncio
import threading
import time

# 3. Define a dummy function simulating a heavy computation on a GPU
def run_gpu_model(prompt: str) -> dict:
    # In a real scenario, this would load a model and run inference.
    print(f"Processing prompt: '{prompt}' on the cloud GPU...")
    time.sleep(2) # Simulate work
    return {"prompt": prompt, "generated_text": f"This is the generated result for '{prompt}'"}

# 4. Create the FastAPI app
app = FastAPI()

@app.post("/predict/")
def predict(payload: dict):
    prompt = payload.get("prompt")
    if not prompt:
        raise HTTPException(status_code=400, detail="Prompt not provided")
    result = run_gpu_model(prompt)
    return result

# 5. Define the function to run Uvicorn server
def run_app():
    nest_asyncio.apply()
    uvicorn.run(app, host="0.0.0.0", port=8000)

# 6. Run the app in a separate thread
thread = threading.Thread(target=run_app)
thread.start()

# 7. Expose the server with ngrok
NGROK_AUTH_TOKEN = "YOUR_NGROK_AUTH_TOKEN" # <-- PASTE YOUR NGROK AUTH
TOKEN HERE
ngrok.set_auth_token(NGROK_AUTH_TOKEN)
public_url = ngrok.connect(8000)
print(f"✅ FastAPI server is live at: {public_url}")
print(f"📚 API documentation (Swagger UI) is available at: {public_url}/docs")
```

Step 2: The Local PC Client (local_client.py)

This script, run on a user's local machine, uses the requests library to send data to the cloud API endpoint and receive the results.

Python

Show codeCopy

```
# Full Code Entry
# This local client script interacts with the remote FastAPI server.
# It uses the requests library as detailed in [66, 67, 68]
import requests
import json

# The public URL from the Colab notebook output (without the /docs part)
api_base_url = "YOUR_NGROK_PUBLIC_URL_HERE" # e.g.,
"https://1a2b-34-56-78-90.ngrok.io"
predict_endpoint = f"{api_base_url}/predict/"

# The data to send in the POST request body
payload = {"prompt": "A detailed description of the solar system"}

print(f"Sending request to: {predict_endpoint}")
```

```
try:
    # Make the POST request with a JSON payload
    response = requests.post(predict_endpoint, json=payload, timeout=10)

    # Raise an exception for HTTP error codes (4xx or 5xx)
    response.raise_for_status()

    # Parse the JSON response
    data = response.json()

    print("\n✅ Successfully received response from cloud GPU:")
    print(json.dumps(data, indent=2))

except requests.exceptions.RequestException as e:
    print(f"\n❌ An error occurred: {e}")
```

This architecture effectively turns the cloud notebook into a callable, on-demand compute resource, controlled entirely by the local client application.

3.3 Real-Time Bidirectional Communication with WebSockets

For applications requiring continuous, low-latency interaction, WebSockets provide a superior alternative to the standard HTTP request-response model. A WebSocket connection is a full-duplex communication channel established over a single, long-lived TCP connection, allowing both the client and server to send messages at any time.[48]
ngrok can seamlessly tunnel WebSocket traffic through its standard HTTP tunnels, or more robustly through a dedicated TCP tunnel.[55]

3.3.1 End-to-End Tutorial: Interactive Control via WebSocket

This tutorial demonstrates setting up a WebSocket server in a Colab notebook that echoes back capitalized messages, and a local command-line client that interacts with it in real-time.
Step 1: The Cloud Notebook WebSocket Server (websocket_server.ipynb)
This notebook uses the websockets library to create and run a simple echo server in a background thread.

Python

Show codeCopy
# Full Code Entry

```
# This notebook cell sets up and launches the WebSocket server.
# It uses the 'websockets' library as shown in [69, 70, 71]

# 1. Install dependencies
!pip install websockets pyngrok

# 2. Import libraries
import asyncio
import websockets
from pyngrok import ngrok
import threading

# 3. Define the WebSocket handler
async def handler(websocket):
    print(f"Client connected: {websocket.remote_address}")
    try:
        async for message in websocket:
            print(f"<<< Received from client: {message}")
            # Process the message and send a response
            response = f"Server echoes: {message.upper()}"
            await websocket.send(response)
            print(f">>> Sent to client: {response}")
    except websockets.ConnectionClosedError:
        print(f"Client disconnected: {websocket.remote_address}")

# 4. Define the server startup coroutine
async def start_server():
    async with websockets.serve(handler, "localhost", 8765):
        await asyncio.Future()  # Run forever

# 5. Run the server in a background thread
def run_async_server():
    asyncio.run(start_server())

thread = threading.Thread(target=run_async_server)
thread.start()
print("WebSocket server started in background thread on localhost:8765")

# 6. Expose the server with an ngrok TCP tunnel
NGROK_AUTH_TOKEN = "YOUR_NGROK_AUTH_TOKEN" # <-- PASTE YOUR NGROK AUTH
TOKEN HERE
ngrok.set_auth_token(NGROK_AUTH_TOKEN)
tcp_tunnel = ngrok.connect(8765, "tcp")
```

```
print(f"✅ WebSocket server is publicly available at: {tcp_tunnel.public_url}")
```

Step 2: The Local PC WebSocket Client (local_ws_client.py)
This local client script establishes a persistent connection to the server via the ngrok TCP endpoint and allows for interactive, real-time messaging.

Python

Show codeCopy

```
# Full Code Entry
# This local client script uses the 'websockets' library to connect to the remote server.
# Based on client examples from [54, 72]
import asyncio
import websockets

# The public URL from the Colab output, e.g., "tcp://0.tcp.ngrok.io:12345"
# It must be parsed and converted to the "ws://hostname:port" format.
ngrok_tcp_url = "0.tcp.ngrok.io:12345"  # <-- REPLACE WITH YOUR NGROK TCP ADDRESS
uri = f"ws://{ngrok_tcp_url}"

async def interact_with_server():
    print(f"Connecting to {uri}...")
    try:
        async with websockets.connect(uri) as websocket:
            print("✅ Connection established. Type a message and press Enter.")
            print("Type 'exit' to close the connection.")

            while True:
                command = await asyncio.to_thread(input, "Message to send: ")
                if command.lower() == 'exit':
                    break

                await websocket.send(command)
                print(f">>> Sent: {command}")

                response = await websocket.recv()
                print(f"<<< Received: {response}")

    except (websockets.exceptions.ConnectionClosed, ConnectionRefusedError) as e:
        print(f"❌ Connection failed: {e}")
    finally:
        print("Connection closed.")
```

```
if __name__ == "__main__":
    asyncio.run(interact_with_server())
```

This advanced architecture provides the lowest latency and most responsive user experience, making it suitable for highly interactive applications where the cloud notebook serves as a live computational engine.

Conclusion and Decision Framework

Comparative Analysis of Methodologies

The methodologies presented in this document offer a spectrum of solutions for building and deploying interactive GUIs from cloud-based Jupyter environments. Each approach involves a distinct set of trade-offs between ease of implementation, user experience, layout flexibility, and architectural complexity.

- **In-Notebook GUIs (ipywidgets, Gradio, Panel):** These are the fastest to implement and are ideal for development-time interactivity and simple tool creation. They live within the notebook, tightly coupling the UI with the code. However, they are limited by the notebook's cell-based layout and can offer a less polished experience for non-technical end-users. Gradio stands out in this category with its effortless share=True feature, making it the premier choice for rapid, temporary demo sharing.
- **Notebook-as-Application Frameworks (Voila, Streamlit):** These frameworks provide a significantly better end-user experience by transforming the notebook into a clean, standalone web application. They require the additional step of setting up a tunneling service like ngrok but offer a much more professional result. They are excellent for sharing polished dashboards and data-driven stories.
- **Advanced Cloud-to-Local Architectures (FastAPI, WebSockets):** This approach offers the ultimate in power and flexibility. By decoupling the UI from the backend, it allows for the creation of any type of client application (web, desktop, mobile) that can leverage the cloud GPU. This comes at the cost of increased architectural complexity, requiring the developer to manage two separate applications (client and server) and the communication between them. This is the preferred method for production-grade services and highly custom interactive systems.

Recommendations for Use-Case Scenarios

The optimal choice of framework depends directly on the project's specific goals:

- **For interactive data exploration, debugging, and parameter tuning during development:** Use **Ipywidgets**. Its native integration makes it the most seamless and lightweight option for augmenting the development workflow.

- **For quickly creating and sharing a machine learning model demo with colleagues or clients for feedback:** Use **Gradio** with share=True. Its combination of simplicity and built-in sharing is unmatched for this purpose.
- **For building and sharing a polished, code-hidden dashboard for presentation to non-technical stakeholders:** Use **Voila** or **Streamlit** in combination with ngrok. Voila is ideal if the dashboard is already built with ipywidgets, while Streamlit offers a faster path if starting from a Python script.
- **For creating a production-like service with a custom local GUI or integrating cloud GPU computation into an existing application:** Use **FastAPI** (for request-response interactions) and/or **WebSockets** (for real-time, bidirectional communication), exposed via ngrok.

Table: GUI Framework Feature & Compatibility Matrix

The following table provides a high-level comparison of the primary frameworks discussed in this document, serving as a quick-reference guide for selecting the appropriate tool.

Feature	Ipywidgets	Gradio	Panel (HoloViz)	Voila	Streamlit
Primary Use Case	Native notebook interactivity, parameter tuning [15]	Rapid ML model demos and prototyping [26]	Complex, data-intensive dashboards and tools [35]	Convert notebooks to standalone web apps [39]	Build and share data-centric web apps [43]
Ease of Implementation	Very High (especially with interact)	Very High	Medium	High	High
Layout Control	Medium (via HBox, VBox, GridBox) [23]	High (with gr.Blocks API) [26]	Very High (extensive layout options) [6]	Low (follows notebook cell order)	Medium (top-down script flow with layout primitives)
Built-in Sharing	No	Yes (share=True) [26]	No	No	No (requires Streamlit Community Cloud) [46]
Requires Tunneling	No	No (for temporary sharing)	Yes (to serve as an app)	Yes (to expose from cloud VM) [38]	Yes (to expose from cloud VM) [51]
Google Colab	Excellent	Excellent (runs	Good (requires	Good (requires	Good (requires

Support	(natively supported) [19]	out-of-the-box) [26]	workarounds like pn.extension() in each cell) [7]	ngrok for access) [38]	ngrok for access) [45]
State Management	Kernel-based, widget state is persistent [17]	Re-runs function on interaction [26]	Reactive (re-runs bound functions on change) [35]	Kernel-based, one per user session [40]	Re-runs entire script on interaction [43]

Works cited

1. Tutorial - User Guide - FastAPI - Tiangolo, accessed on August 12, 2025, https://fastapi.tiangolo.com/tutorial/
2. Quickstart — Flask Documentation (3.1.x), accessed on August 12, 2025, https://flask.palletsprojects.com/en/stable/quickstart/
3. Google Colab, accessed on August 12, 2025, https://research.google.com/colaboratory/faq.html
4. Running a public Jupyter Server - Read the Docs, accessed on August 12, 2025, https://jupyter-server.readthedocs.io/en/latest/operators/public-server.html
5. How to run Flask App on Google Colab? - GeeksforGeeks, accessed on August 12, 2025, https://www.geeksforgeeks.org/machine-learning/how-to-run-flask-app-on-google-colab/
6. What is panel? | Data Visualisation in Data Science, accessed on August 12, 2025, https://vda-lab.github.io/visualisation-tutorial/holoviz-what-is-panel.html
7. Develop in other notebook environments — Panel v1.7.5 - HoloViz, accessed on August 12, 2025, https://panel.holoviz.org/how_to/notebook/other_nb.html
8. Using Panel in Google Colab - HoloViz Discourse, accessed on August 12, 2025, https://discourse.holoviz.org/t/using-panel-in-google-colab/251
9. Introduction - Guides - Vast AI, accessed on August 12, 2025, https://docs.vast.ai/
10. Jupyter - Vast.ai, accessed on August 12, 2025, https://docs.vast.ai/instances/jupyter
11. FAQ - Guides - Vast.ai, accessed on August 12, 2025, https://docs.vast.ai/faq
12. Lightning AI · GitHub, accessed on August 12, 2025, https://github.com/lightning-ai
13. How to Train, Deploy & Scale AI Models with Lightning AI | Full Tutorial with Studio + AI Hub, accessed on August 12, 2025, https://www.youtube.com/watch?v=1S70kNN2p0w
14. Host web apps Lightning AI - Docs, accessed on August 12, 2025, https://lightning.ai/docs/overview/host-web-apps
15. ipywidgets - PyPI, accessed on August 12, 2025, https://pypi.org/project/ipywidgets/
16. jupyter-widgets/ipywidgets: Interactive Widgets for the ... - GitHub, accessed on August 12, 2025, https://github.com/jupyter-widgets/ipywidgets
17. Low Level Widget Tutorial — Jupyter Widgets 7.7.2 documentation - IPyWidgets,

accessed on August 12, 2025, https://ipywidgets.readthedocs.io/en/7.x/examples/Widget%20Low%20Level.html
18. ipywidgets - Databricks Documentation, accessed on August 12, 2025, https://docs.databricks.com/aws/en/notebooks/ipywidgets
19. Using Interact.ipynb - Colab, accessed on August 12, 2025, https://colab.research.google.com/github/jupyter-widgets/ipywidgets/blob/master/docs/source/examples/Using%20Interact.ipynb
20. Using ipywidgets in Google Colab for Interactive Python Notebooks, accessed on August 12, 2025, https://www.plus2net.com/python/ipywidgets.php
21. Using Interact — Jupyter Widgets 8.1.7 documentation, accessed on August 12, 2025, https://ipywidgets.readthedocs.io/en/latest/examples/Using%20Interact.html
22. Widget List.ipynb - Colab, accessed on August 12, 2025, https://colab.research.google.com/github/jupyter-widgets/ipywidgets/blob/master/docs/source/examples/Widget%20List.ipynb
23. Widget List — Jupyter Widgets 8.1.5 documentation - IPyWidgets, accessed on August 12, 2025, https://ipywidgets.readthedocs.io/en/8.1.5/examples/Widget%20List.html
24. Layout of Jupyter widgets - IPyWidgets, accessed on August 12, 2025, https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Layout.html
25. Low Level Widget Explanation — Jupyter Widgets 8.1.7 documentation - IPyWidgets, accessed on August 12, 2025, https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Low%20Level.html
26. Quickstart - Gradio, accessed on August 12, 2025, https://www.gradio.app/guides/quickstart
27. Tutorial: Build and Host Machine Learning Demos with Gradio & Hugging Face - Colab, accessed on August 12, 2025, https://colab.research.google.com/github/huggingface/education-toolkit/blob/main/02_ml-demos-with-gradio.ipynb
28. huggingface-cn/gradio-docs-cn: Create UIs for your machine learning model in Python in 3 minutes - GitHub, accessed on August 12, 2025, https://github.com/huggingface-cn/gradio-docs-cn
29. gradio-app/gradio: Build and share delightful machine ... - GitHub, accessed on August 12, 2025, https://github.com/gradio-app/gradio
30. Gradio, accessed on August 12, 2025, https://www.gradio.app/
31. 4 Streamlit Alternatives for Building Python Data Apps - Anvil Works, accessed on August 12, 2025, https://anvil.works/articles/4-alternatives-streamlit
32. Blocks - Gradio Docs, accessed on August 12, 2025, https://www.gradio.app/docs/gradio/blocks
33. Blocks And Event Listeners - Gradio, accessed on August 12, 2025, https://www.gradio.app/guides/blocks-and-event-listeners
34. Gradio Documentation, accessed on August 12, 2025, https://www.gradio.app/docs
35. Panel: The powerful data exploration & web app framework for Python - GitHub,

accessed on August 12, 2025, https://github.com/holoviz/panel
36. ngrok Python SDK Quickstart, accessed on August 12, 2025, https://ngrok.com/docs/getting-started/python/
37. alexdlaird/pyngrok: A Python wrapper for ngrok - GitHub, accessed on August 12, 2025, https://github.com/alexdlaird/pyngrok
38. Quickly share ML WebApps from Google Colab using ngrok for Free ..., accessed on August 12, 2025, https://medium.com/data-science/quickly-share-ml-webapps-from-google-colab-using-ngrok-for-free-ae899ca2661a
39. Using Voilà — voila 0.5.8 documentation, accessed on August 12, 2025, https://voila.readthedocs.io/en/stable/using.html
40. voila-dashboards/voila: Voilà turns Jupyter notebooks into standalone web applications - GitHub, accessed on August 12, 2025, https://github.com/voila-dashboards/voila
41. voila-dashboards/voila-material: Material design template for Voilà - GitHub, accessed on August 12, 2025, https://github.com/voila-dashboards/voila-material
42. Deploying Voilà — voila 0.5.8 documentation, accessed on August 12, 2025, https://voila.readthedocs.io/en/stable/deploy.html
43. Streamlit • A faster way to build and share data apps, accessed on August 12, 2025, https://streamlit.io/
44. Streamlit - GitHub, accessed on August 12, 2025, https://github.com/streamlit
45. How to Launch Streamlit App from Google Colab Notebook ..., accessed on August 12, 2025, https://discuss.streamlit.io/t/how-to-launch-streamlit-app-from-google-colab-notebook/42399
46. Community Cloud - Streamlit • A faster way to build and share data apps, accessed on August 12, 2025, https://streamlit.io/cloud
47. Connect your GitHub account - Streamlit Docs, accessed on August 12, 2025, https://docs.streamlit.io/deploy/streamlit-community-cloud/get-started/connect-your-github-account
48. en.wikipedia.org, accessed on August 12, 2025, https://en.wikipedia.org/wiki/FastAPI
49. FastAPI documentation - DevDocs, accessed on August 12, 2025, https://devdocs.io/fastapi/
50. Websockets with JavaScript (Jupyter notebook) | by Girish Venkatachalam | Medium, accessed on August 12, 2025, https://medium.com/@girish1729/websockets-with-javascript-jupyter-notebook-4daf98ba2739
51. Free Streamlit Dev Environment through Colaboratory - Show the Community!, accessed on August 12, 2025, https://discuss.streamlit.io/t/free-streamlit-dev-environment-through-colaboratory/2778
52. First Steps - FastAPI - Tiangolo, accessed on August 12, 2025, https://fastapi.tiangolo.com/tutorial/first-steps/
53. How to Create a WebSocket Client in Python? - Apidog, accessed on August 12,

2025, https://apidog.com/blog/python-websocket-client/
54. Build WebSocket Server and Client Using Python - GeekPython, accessed on August 12, 2025, https://geekpython.in/build-websocket-server-and-client-using-python
55. Websockets | ngrok documentation, accessed on August 12, 2025, https://ngrok.com/docs/using-ngrok-with/websockets/
56. how to use ws(websocket) via ngrok - Stack Overflow, accessed on August 12, 2025, https://stackoverflow.com/questions/49129451/how-to-use-wswebsocket-via-ngrok
57. Create data science web app with Streamlit library and Google Colab.ipynb, accessed on August 12, 2025, https://colab.research.google.com/github/tateemma/Streamlit-App-with-Google-Colab/blob/main/Create_data_science_web_app_with_Streamlit_library_and_Google_Colab.ipynb
58. 10_webapps.ipynb - Colab, accessed on August 12, 2025, https://colab.research.google.com/github/giswqs/geebook/blob/master/chapters/10_webapps.ipynb
59. Creating A Custom Chatbot With Blocks - Gradio, accessed on August 12, 2025, https://www.gradio.app/guides/creating-a-custom-chatbot-with-blocks
60. 04_Dashboard.ipynb - Colab - Google, accessed on August 12, 2025, https://colab.research.google.com/github/holoviz-community/HoloViz_KDD2022/blob/main/04_Dashboard.ipynb
61. Quickly deploy ML WebApps from Google Colab using ngrok - YouTube, accessed on August 12, 2025, https://www.youtube.com/watch?v=AkEnjJ5yWV0
62. Run streamlit app from a Google Colab Notebook, accessed on August 12, 2025, https://colab.research.google.com/github/mrm8488/shared_colab_notebooks/blob/master/Create_streamlit_app.ipynb
63. FastAPI in Google Colab with ngrok · GitHub, accessed on August 12, 2025, https://gist.github.com/SalihTalha/8b9be8662f239ffbac4b31b2022288ea
64. How to run FastAPI / Uvicorn in Google Colab? - Stack Overflow, accessed on August 12, 2025, https://stackoverflow.com/questions/63833593/how-to-run-fastapi-uvicorn-in-google-colab
65. Running Flask and FastAPI on Google Colab | by Sahil Ahuja - DataDrivenInvestor, accessed on August 12, 2025, https://medium.datadriveninvestor.com/flask-on-colab-825d2099d9d8
66. Quickstart — Requests 2.32.4 documentation - Read the Docs, accessed on August 12, 2025, https://requests.readthedocs.io/en/master/user/quickstart/
67. Python's Requests Library (Guide) – Real Python, accessed on August 12, 2025, https://realpython.com/python-requests/
68. Python Requests - GeeksforGeeks, accessed on August 12, 2025, https://www.geeksforgeeks.org/python/python-requests-tutorial/
69. Set up a WebSockets server in Python - Postman Quickstarts, accessed on August 12, 2025,

https://quickstarts.postman.com/guide/websockets-python/index.html?index=..%2F..index
70. websockets - PyPI, accessed on August 12, 2025, https://pypi.org/project/websockets/
71. Library for building WebSocket servers and clients in Python - GitHub, accessed on August 12, 2025, https://github.com/python-websockets/websockets
72. Quick examples - websockets 15.0.1 documentation, accessed on August 12, 2025, https://websockets.readthedocs.io/en/stable/intro/examples.html