EC535: Embedded Systems - Remote-Controlled Scouting Buggy with Live Camera Feed and GUI Control

varsingh, jknee {@bu.edu}

April 2025

Github: varsingh/ec535-final-project

Abstract

This project presents a browser-based remote control system for the HiWonder PuppyPi, a quadruped robot that integrates both live video streaming and sensor-based safety mechanisms. Utilizing manufacturer-specific Python libraries for precise control of servomotors and behavior sequencing, the system enables users to command the robot via keyboard inputs and UI buttons through a React frontend. Real-time feedback is achieved through an onboard USB camera streamed with MJPG-Streamer, while a LiDAR sensor continuously monitors the environment to override potentially unsafe movement commands. By combining vendor-specific APIs with a modular web-based interface and WebSocket communication, this project demonstrates how accessible and reactive robotics can be achieved even with hardware constraints, with applications in education, teleoperation, and autonomous exploration.

1 Introduction

Controlling mobile robots remotely often requires specialized software, proprietary protocols, or cloud-based services—barriers that limit flexibility and usability, especially in time-sensitive or infrastructure-constrained scenarios. In contrast, this project explores how modern web technologies and local sensing can be combined to create an accessible, reactive, and browser-based human-robot interface.[1]

Introducing: HiWonder's PuppyPi: a quadruped robot equipped with a USB camera, a Raspberry Pi 4B, a LiDAR sensor, and 8 servomotors.

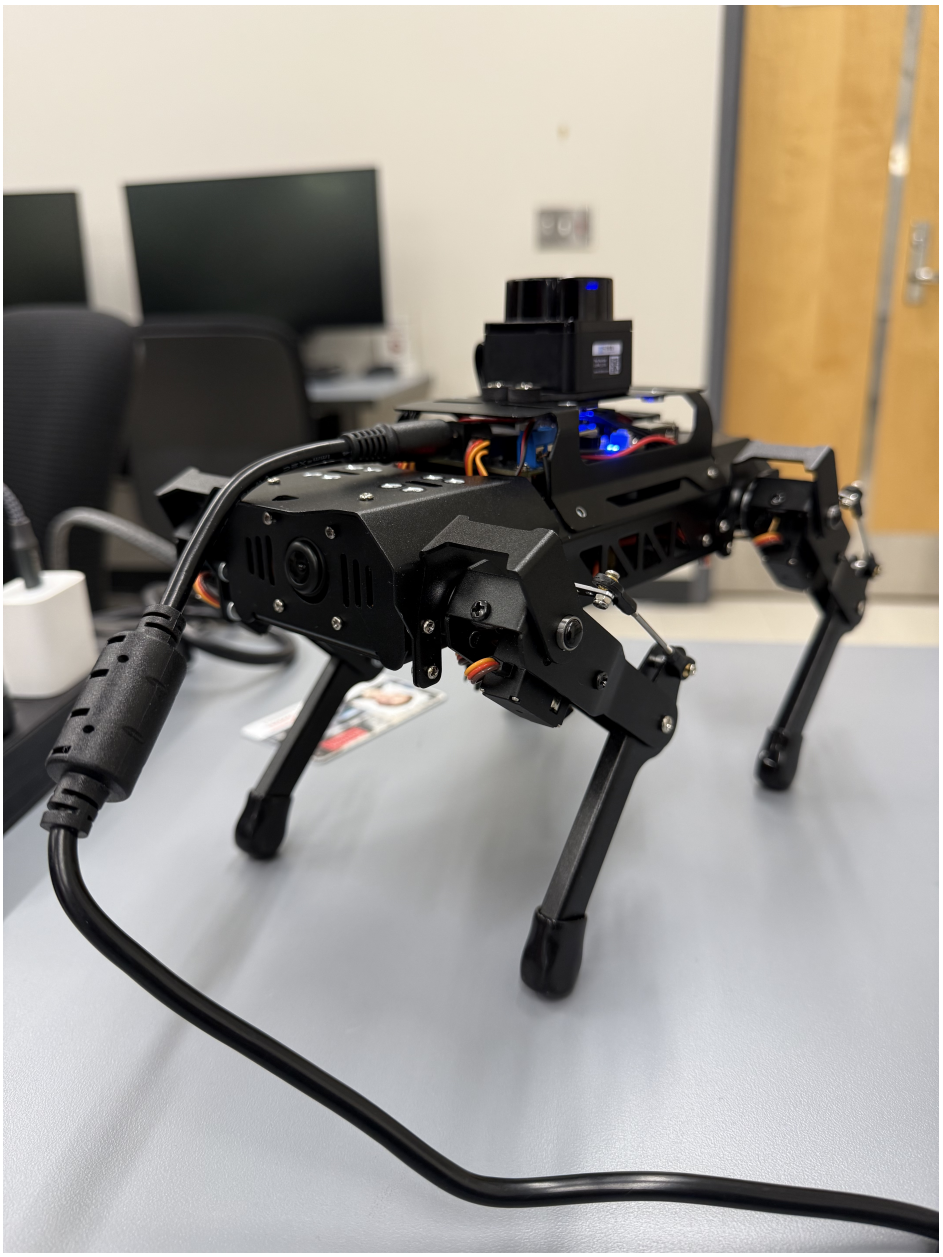

Figure 1: The HiWonder PuppyPi robot used in this project: a Raspberry Pi-powered quadruped with a USB camera and LiDAR sensor.

Through a React-based frontend, users can send movement commands (WASD) and trigger complex behaviors using the UI. This also incorporates simultaneous display of a live video feed stream from the robot's onboard camera. Commands are transmitted via WebSocket to ensure low-latency, bidirectional communication over a shared local network.

In addition to direct control, the robot incorporates basic autonomous safety logic via its LiDAR: if an object is detected too close in the front or rear, movement commands in that direction are overridden, preventing collisions. These constraints are reflected in the user interface through an asynchronous feedback queue from the robot, giving the user awareness of environmental conditions in real time.

The broader goal of this project is to demonstrate how robust, reactive robotic systems can be constructed using common, open technologies. While the low-level motor control relies on manufacturer-provided libraries

specific to the HiWonder platform, the higher-level system architecture—including the browser-based UI, network communication, and sensor-based overrides is platform-agnostic and can be generalized to similar robotic systems. This makes it suitable for both educational use and remote, infrastructure-limited deployments.

This paper details the system architecture, sensor integration, user interface design, and network protocol, followed by a discussion of performance, usability, and future directions such as cloud-based control or adaptive behavior based on video processing.

2 Method

The system architecture consists of two main components:

- Frontend: web-based control interface running on a client device (typically a laptop).
- Backend: embedded server hosted on the HiWonder PuppyPi robot.

These two components communicate over a local Wi-Fi network using the WebSocket protocol, allowing for real-time, bidirectional data exchange.

2.1 Robot Platform

The HiWonder PuppyPi is powered by a Raspberry Pi 4B and includes eight servomotors for locomotion, an onboard USB camera for live video streaming, and an LD19 LiDAR unit for environmental awareness. Servomotor commands and behavioral sequences (e.g., stand, nod, bow) are implemented using HiWonder's proprietary Python libraries, which interpret control packets and actuate the robot accordingly.

```
def runAction(actNum):
    global runningAction
    global stopRunning
    global online_action_times
    if actNum is None:
        return
    actNum = HomePath + "/PuppyPi_PC_Software/ActionGroups/" + actNum
    stopRunning = False
    if os.path.exists(actNum) is True:
        if runningAction is False:
            runningAction = True
            ag = sql.connect(actNum)
            cu = ag.cursor()
            cu.execute("select * from ActionGroup")

            puppy.servo_force_run()
            time.sleep(0.01)
            while True:
                act = cu.fetchone()
                if stopRunning is True:
                    stopRunning = False
                    break
                if act is not None:
                    if type(act[2]) is int:
                        for i in range(0, len(act)-2, 1):
                            setServoPulse(i+1, act[2 + i], act[1])

                    elif type(act[2]) is float:
                        rotated_foot_locations = np.zeros(12)
                        for i in range(0, len(act)-2):
                            value = act[i+2]
                            rotated_foot_locations[i] = float(value)
                        rotated_foot_locations = rotated_foot_locations.reshape(4,3)
                        rotated_foot_locations = rotated_foot_locations.T
                        rotated_foot_locations = rotated_foot_locations/100
```

```
                        joint_angles = puppy.fourLegsRelativeCoordControl(
                            rotated_foot_locations)

                        puppy.sendServoAngle(joint_angles, act[1])#, force_execute = True
                        # joint_angles = puppy.four_legs_inverse_kinematics_relative_coord(
                            rotated_foot_locations, puppy.config)
                        # puppy.send_servo_commands(PWMServoParams(), joint_angles, act[1])


                    time.sleep(float(act[1])/1000.0)
                else:
                    break

            runningAction = False
            cu.close()
            ag.close()
    else:
        runningAction = False
```

Listing 1: Programmed cervomotor-controlled commands

2.2 Frontend Control Interface

A custom React.js frontend acts as the user interface.

It displays a live MJPG video stream from the robot's onboard camera and provides buttons and keyboard mappings (WASD) to send motion commands. Predefined commands are also triggered from the UI, as well as commands for changing the height of the PuppyPi and movement speed. This interface runs in any modern browser on a device connected to the same local network as the robot.

Now that we have a frontend, we need to figure out a way to have commands sent from the frontend to the PuppyPi.

2.3 Communication Protocol

The core of our communication architecture is built around the WebSocket protocol, which allows two-way, low-latency messaging between the operator's laptop and the PuppyPi robot. Unlike traditional HTTP-based polling or REST APIs, WebSockets establish a persistent connection over a single TCP socket, enabling instantaneous data exchange in both directions.

We previously discussed the frontend, which serves as the control interface for the user. When a key/button is pressed (e.g., W/A/S/D or a behavior trigger), the React client emits a WebSocket event through the browser using the native `WebSocket` API. This event contains a command payload containing the action type (movement or behavior) and (implicitly) the timestamp.

```
# WebSocket command handling
async def command_handler(websocket):
    global PuppyMove
    global set_mark_time_srv
    async for message in websocket:
        try:
            data = json.loads(message)
            key = data.get("key", "")
            rospy.loginfo(f"Received key command: {key}")
            '''
            ...
            '''
            PuppyVelocityPub.publish(**PuppyMove)
        except json.JSONDecodeError:
            rospy.logwarn(f"Invalid JSON: {message}")

async def start_websocket_server():
    async with websockets.serve(command_handler, "0.0.0.0", 8765):
        await asyncio.Future()

```

```
def websocket_thread():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(start_websocket_server())
```

Listing 2: PuppyPi-side Websocket

On the robot side, the Raspberry Pi runs a lightweight Python-based WebSocket server built using the `websockets` library. This server listens for incoming commands and parses each payload to either:

- Trigger servo motion sequences (e.g., using the HiWonder `.d6ac` motion control interface).
- Send movement commands directly to the servos via the HiWonder Python SDK.
  - Can also reject movement commands based on real-time proximity data from the LiDAR sensor (e.g., object too close).

In parallel, the robot asynchronously sends status updates—such as LiDAR proximity readings or motion acknowledgments—back to the frontend via the same WebSocket channel. These updates are displayed in the browser UI, giving users responsive visual feedback.

Both devices (laptop and PuppyPi) are network clients connected to a common local hotspot. This architecture avoids the security measures placed on *eduroam* and keeps latency minimal, making it suitable for time-sensitive robotic control tasks.

2.4 Sensor Integration and Safety Logic

The LD19 LiDAR mounted on the robot continuously scans the environment for nearby objects.

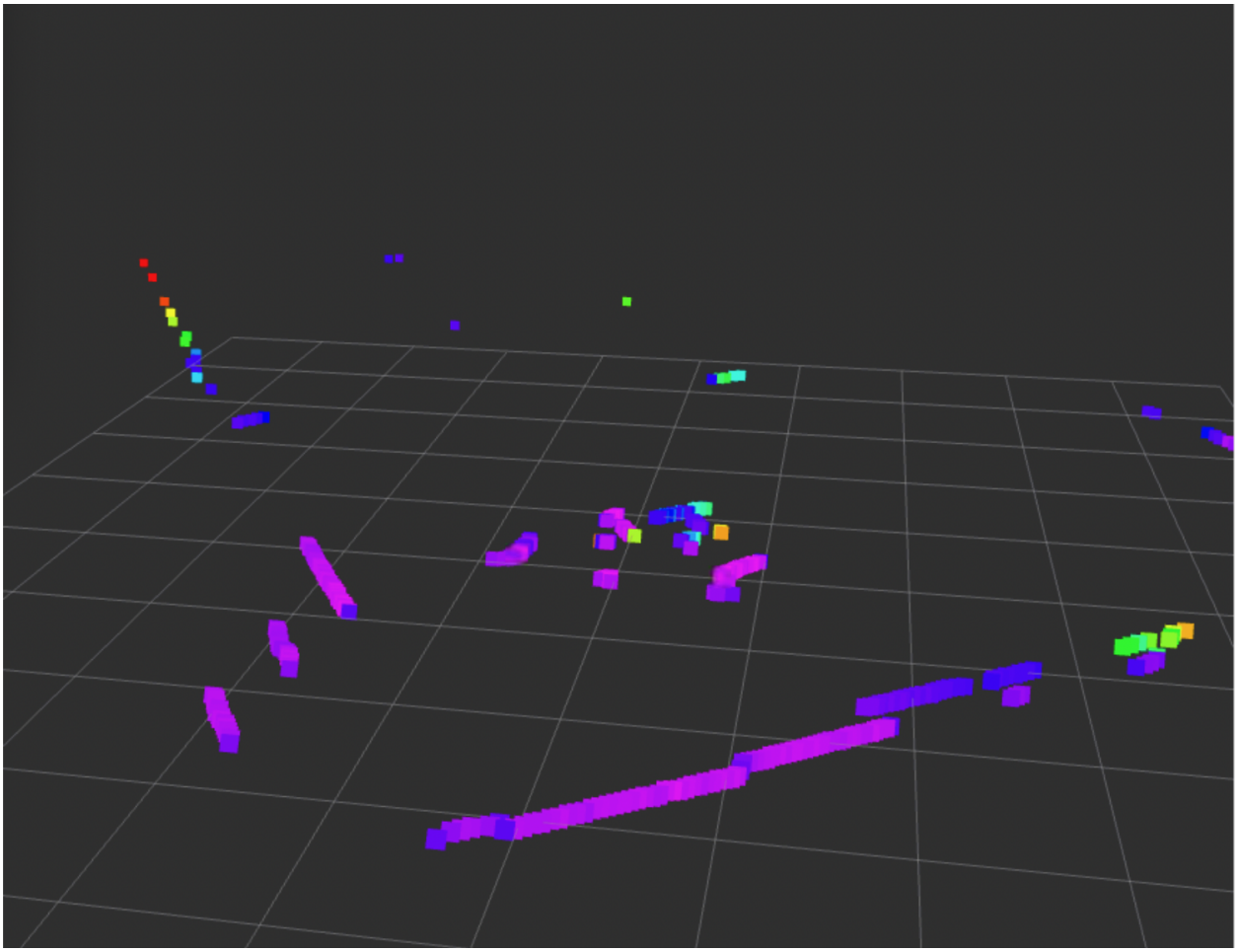

Figure 2: LiDAR object detection visualization.

If an obstacle is detected within a critical distance threshold in the front or rear direction, the robot's onboard Python server prevents execution of forward or backward movement commands. This logic is implemented asynchronously (launching multiple threads) and runs in parallel with command listening. We can view this through a `rospy.loginfo` message:

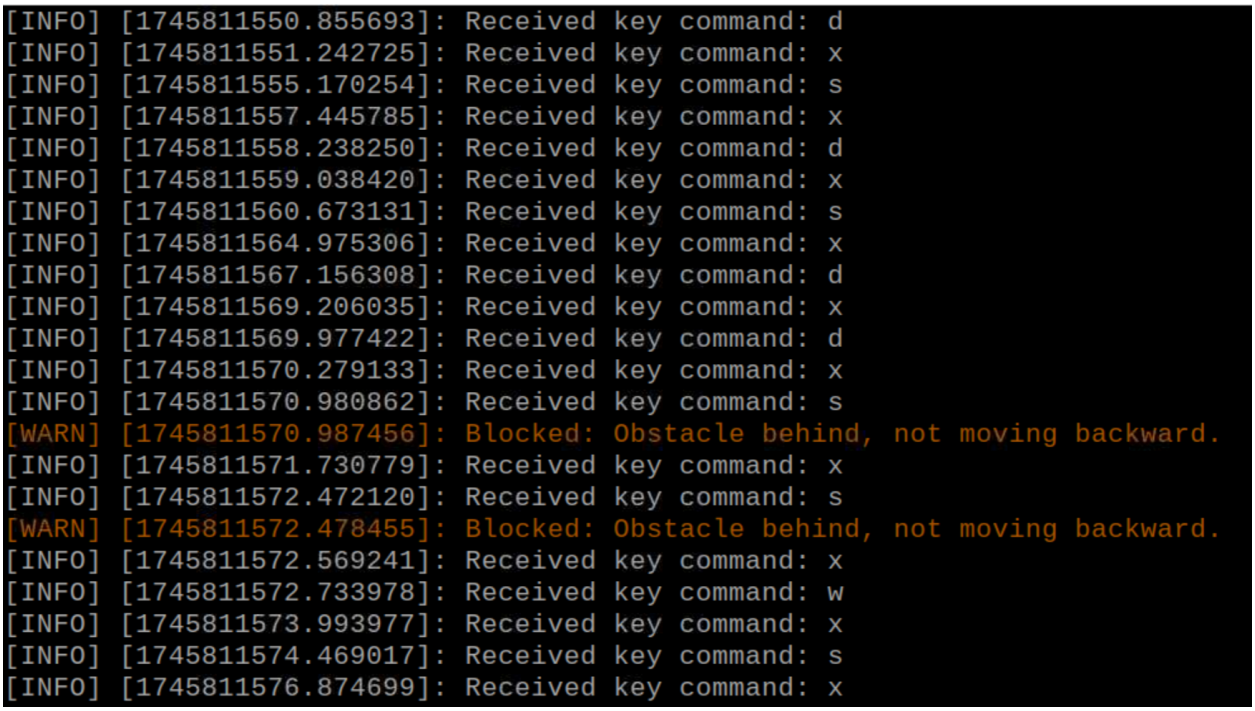

Figure 3: PuppyPi-side dangerous command acknowledgment.

Additionally, the robot sends feedback packets to the frontend indicating whether commands were executed or blocked. This feedback is rendered in the UI so the user can easily understand when commands are ignored due to proximity constraints.

2.5 System Summary

Overall, this architecture enables responsive, video-guided, and sensor-aware teleoperation of a quadruped robot using only a browser and a shared local network. While the low-level motor control depends on vendor-specific APIs, the communication layer, safety features, and UI design are modular and generalizable to other robotic platforms.

3 Results

3.1 Experimental Setup

To evaluate our system, we conducted a series of real-time interaction trials in a closed environment. The HiWonder PuppyPi plugged into it's power source at 8.4V and placed on a flat surface. We connected the robot and the control laptop to a local Wi-Fi hotspot, to ensure no noise from other connections could interfere with our experiment. The React frontend was served locally in a browser, and the Raspberry Pi onboard the PuppyPi ran both the WebSocket server and MJPG Streamer for video streaming. No internet access was required.

The experimental environment was designed to simulate realistic remote operation scenarios, including:

- Simple directional movement tests using W/A/S/D keys,.
- Execution of complex behavior sequences from `.d6ac` files.
- Obstacle detection and prevention (with static and dynamic obstacles) using real-time LiDAR feedback.
- Live video monitoring through MJPG Streamer integrated into the frontend.

3.2 Evaluation Criteria

We evaluated system performance across the following dimensions:

- **Latency:** Time between user input and robot response.
- **Reliability:** Command delivery and correct execution rate.
- **Safety:** LiDAR-based obstacle prevention accuracy.
- **User Experience:** Subjective responsiveness and usability.

Each test was repeated over 10 trials to ensure consistent behavior.

3.3 Quantitative Results

3.3.1 Latency Considerations

For real-time control, it is essential to keep the system's latency below the threshold where users can perceive delays. According to research by the Nielsen Norman Group, response times exceeding **100 ms** begin to noticeably affect the user experience in real-time control scenarios [2]. As such, we designed our system to ensure that the latency between user input and robot response remains within this upper bound to maintain a smooth and responsive interaction.

To get accurate measurements: we send a packet with the current timestamp to the PuppyPi with `pong`, and it will send a packet to the server containing the same timestamp after executing the movement. With this, we calculate the difference between the time the packet is received and the payload message from that packet.

3.3.2 Results

- **Latency:** Average round-trip time between key press and robot movement confirmation was measured at:
  - Idle: 13ms
  - Walking: 24ms
  - Action Button: 23ms
  - LiDAR blocking: 44ms
- **Reliability:** Command transmission success rate was 100% over a stable LAN. All sent commands were received and interpreted correctly.
- **LiDAR Safety:** Obstacle detection correctly blocked forward/backward commands in 10 out of 10 trials when an object was within the 30cm threshold.

3.4 Qualitative Observations

We report a high degree of perceived responsiveness, with the robot's behavior matching input expectations. The MJPG video feed was effective for remote navigation, although occasional frame rate drops were observed when the CPU load on the Raspberry Pi spiked (especially during concurrent video streaming and behavior execution). Additionally, real-time camera feed tends to exceed real-time rates when the server (hotspot) is placed far away. This is expected, as both the laptop and the Puppy connect through this server.

The combination of WASD-based control and visual feedback provided an intuitive control experience. The addition of programmed motions (via `.d6ac` files) added expressive capability beyond raw locomotion, making the robot feel more responsive and lifelike.

4 Limitations and Future Work

4.1 Limitations

While the system demonstrated strong real-time performance, several limitations were encountered during development and testing.

- **Raspberry Pi Processing Power:** The Raspberry Pi, while sufficient for many tasks, struggled to maintain a high frame rate for the video stream when handling concurrent processes (e.g., motion control and video streaming). This led to occasional frame drops and a slight decrease in video quality, especially during more complex movements or when the robot executed `.d6ac` behaviors. We saw this happen quite frequently when the power source was removed, as the PuppyPi would be able to operate (poorly) for a few minutes before completely *browning out*.
- **Limited Range of Obstacle Detection:** While the LiDAR sensor successfully blocked forward and backward movement when obstacles were detected within a 30cm range, we decided to avoid preventing side collisions or detect obstacles in order to avoid locked movement.
- **User Interface Complexity:** The React frontend provided basic functionality, but as the system evolved (and can evolve in the future), additional complexity was introduced (e.g., video feed, sensor feedback). A more streamlined UI could enhance usability, especially for new users, by reducing cognitive load when operating the robot.

4.2 Future Work

Despite these limitations, the project offers several exciting avenues for future enhancement:

- **Hardware Upgrade:** Switching to a more powerful hardware platform, such as a Raspberry Pi 4 with a better GPU or a dedicated video capture device, could significantly improve video quality and frame rate, providing a smoother remote control experience. This would be a great starting step for dynamic object avoidance.
- **Advanced Obstacle Detection:** Adding camera-based vision functionality could expand the robot's ability to detect and identify objects. This improvement would be essential for applications regarding safety and autonomy, such as a scouting robot searching for humans after a disaster.
- **Enhanced Control Features:** Integrating machine learning algorithms for more autonomous navigation (e.g., object avoidance, path planning) could reduce the reliance on human input for obstacle avoidance, making the robot more independent and adaptable to varying environments. We originally aimed to have an *autonomous rover* setting, where the PuppyPi would be able to try navigating and interacting with the environment on its own. However, we could not get to this step due to time constraints.
- **Dynamic IP Addressing:** Implementing dynamic IP addressing would eliminate the need for manual configuration of the IP address. By using technologies like mDNS (Multicast DNS) or DHCP (Dynamic Host Configuration Protocol), the system could automatically discover and connect to the robot without requiring user intervention. This would make the system more user-friendly and scalable, particularly in environments with multiple devices or where IP addresses may change frequently.

4.3 Conclusion

In conclusion, this project demonstrated the feasibility of using a browser-based interface for real-time control of a quadruped robot, leveraging WebSocket communication for low-latency interaction. Despite encountering challenges with interfacing with the robot, processing power and obstacle detection, the system performed well within the limits of human perceptible delay. Future work will focus on enhancing hardware capabilities, expanding sensor coverage, and incorporating more autonomous features to improve both usability and functionality.

5 References

References

[1] Hiwonder. *Hiwonder GitHub Repositories*. Available at: https://github.com/Hiwonder?tab=repositories. Accessed: May 6, 2025.

[2] Nielsen Norman Group, *Response Times: 3 Important Limits*, 2025. Retrieved from: https://www.nngroup.com/articles/response-times-3-important-limits/