Credit Task 4.3: Outcomes – Build initial Solution.

I. MVP Scope and Requirements

1. Define Scope and Requirements.

The focus of the MVP will revolve around incorporating essential functionalities of the Proposed Smart Gait Monitoring System, specifically targeting the Smart Bench and Smart Path components. The primary goal is to lay the groundwork for a thorough gait analysis while mitigating privacy concerns and ensuring practical applicability in healthcare. The framework will be categorized into two distinct modules, outlined as follows:

a. Smart Bench

- **NFC-Based User Identification:** The MVP will prioritize the implementation of NFC technology for user identification. This is a critical feature aimed at establishing a user-centric approach to gait monitoring, ensuring continuous and non-intrusive monitoring.
- **Capture of Sit-to-Stand Exercise Data:** Emphasizing the numeric data on sit-to-stand exercises is crucial for gaining insights into user mobility and transitions, contributing to a foundational understanding of individual gait patterns.
- **Recording Landing Coordinates and Forces:** The MVP will focus on capturing landing coordinates and forces during landing, providing essential data for the initial analysis of gait dynamics, and forming the basis for assessing posture and balance.

b. Smart Path

- **Collection of Gait Dynamics Data:** The MVP will highlight the measurement of gait dynamics, encompassing metrics such as speed, step length, and stride length. These metrics serve as foundational insights into user mobility patterns and are vital for comprehensive gait analysis.
- **Integration of 3D LIDAR for High-Resolution Gait Monitoring:** Implementing 3D LIDAR technology for capturing high-resolution point cloud data is a pivotal feature. This technology enables a detailed examination of joint movement, posture, and balance during gait, effectively addressing the limitations of traditional monitoring methods.
- **Real-time Evaluation of Handrail Reliance:** The Smart Path's surface sensor will conduct real-time assessments of handrail reliance, providing numeric data on activation duration and categorical data indicating frequently used sides. This information contributes to a nuanced understanding of user interactions.

2. Prioritise Features.

To effectively showcase the feasibility and efficiency of the proposed Smart Gait Monitoring System, the prioritization of features becomes crucial. The following fundamental components are prioritized to emphasize the key aspects of the solution:

a. Critical User Identification: NFC-based user identification takes precedence as it establishes the foundational element for a user-centric gait monitoring system. This prioritization facilitates non-intrusive continuous monitoring and the delivery of personalized feedback.

b. Essential Gait Dynamics Metrics: The measurement of gait dynamics on the Smart Path is prioritized to ensure the collection of indispensable data for initial gait analysis. This prioritization directly addresses the core issue of gait pattern assessment.

c. Foundational 3D LIDAR Implementation: The high-priority feature of implementing 3D LIDAR technology directly confronts the challenges associated with traditional camera-based gait detection limitations. This prioritization ensures the acquisition of high-resolution data for in-depth analysis.

d. Real-time Handrail Reliance Assessment: Prioritizing the real-time assessment of handrail reliance on the Smart Path adds a crucial layer of user interaction data. This priority contributes to a more comprehensive understanding of gait patterns and user behaviours.

3. Limitations and Trade-offs.

During the scoping and requirement-gathering process, certain limitations and trade-offs were encountered. While these factors pose challenges, they are integral to shaping a realistic and effective MVP for the Smart Gait Monitoring System.

a. Limitations

Machine Learning Implementation: Due to the scope of the MVP, certain advanced analytics, and machine learning algorithms, as proposed in the conceptual solution, may not be fully implemented initially due to time constraints.
User Feedback System: The MVP may not include a comprehensive user feedback system, focusing on foundational data collection and analysis.

b. Trade-offs

Simplified Machine Learning Models: Given the limited scope, machine learning models may be simplified initially, with a focus on foundational gait analysis. More sophisticated models can be introduced in subsequent iterations.
Feature Set: The feature set in the MVP is selected to demonstrate core functionalities, and certain features proposed in the conceptual solution may be deferred to later stages.

c. Challenges

Resource Constraints: Choosing features for the MVP required management constraints related to resources, encompassing considerations of time and personnel availability. Striking a balance between the aspiration for a comprehensive solution and the pragmatic feasibility poses a significant challenge.

II. MVP Development

1. Detailed Development Plan.

a. Key Tasks

- **Smart Bench Development**
  - Integrate NFC technology for user identification.
  - Implement numeric data capture for sit-to-stand exercises.
  - Capture landing coordinates and forces during landing.

- **Smart Path Development**
  - Emphasize measurement of gait dynamics, including speed, step length, and stride length.
  - Implement 3D LIDAR technology for capturing high-resolution point cloud data.
  - Assess handrail reliance in real-time, providing numeric and categorical data.

- **Data Collection, Pre-Processing & Logging**
  - Ensure real-time data collection post-acquisition.
  - Implement noise reduction and calibration for processed data.

- **Data Logging**
  - Log processed data using a microcontroller or computer and store it in Tabular/JSON Format.

- **Data Fusion**
  - Develop techniques for integrating various sensor data into a unified stream.
  - Establish communication protocols between components.

b. Activities

- Break down tasks into sprints for iterative development.
- Hold regular team meetings for progress updates and issue resolution.
- Maintain comprehensive documentation for code, processes, and decision-making.
- Plan user testing sessions to gather real-world feedback.

c. Resources

- Hardware: Acquire necessary hardware components for Smart Bench and Smart Path.
- Software: Choose development tools, IDEs, and version control systems.
- Testing Equipment: Procure tools for testing NFC, gait dynamics, and 3D LIDAR functionality.

2. Technology Stack or Tools.

a. Smart Bench and Smart Path Development

- **Programming Languages**
  - Python for versatility in data processing and system integration.
  - JavaScript for front-end development for user interfaces.

- **Main Framework**
  - Flask for building web applications in Python.
  - React.js for building interactive and responsive user interfaces.

- **User Database**
  - MongoDB for flexibility in handling diverse data types.
  - MongoDB Flask extension for seamless integration with Flask.

- **Sensor Integration**
  - Utilize sensor-specific libraries and APIs for NFC, surface sensors, and 3D LIDAR.
  - Ensure compatibility and reliability in data capture.

b. Data Collection, Pre-Processing & Logging

- **Real-time Data Collection**
  - MQTT (Message Queuing Telemetry Transport) protocol for real-time data transfer.
  - Microcontrollers to Process Data for Smart Bench and Smart Path.

- **Noise Reduction and Calibration**
  - Algorithms for noise reduction in sensor data.
  - Calibration processes for accurate data interpretation.

- **Data Logging**
  - Pandas library in Python for tabular data logging.
  - JSON/XML for versatile and structured data storage.

c. Data Fusion

- **Unified Data Stream**
  - Data fusion algorithms for seamless integration.
  - Message brokers like RabbitMQ for efficient communication.

3. Implement Prioritized Features

The implementation of prioritized features involved developing and testing pseudo-codes for various modules within the IoT framework for Gait Analysis, considering time constraints. Due to the absence of real-life data and software simulations, these codes will be refined and enhanced for practical applications in the future.

a. Critical User Identification.

```
# Database simulation for user identification
user_database = {
    "User123": {"name": "John Doe", "age": 30},
    # Add more user data as needed
}

# Simulated NFC class
class NFCReader:
    @staticmethod
    def read_data():
        # Simulate reading NFC data
        return "User123"
```

```
# Function to identify user using NFC
def identify_user(nfc_data):
    # Simulate database lookup
    user_id = nfc_data
    user_info = user_database.get(user_id, {})
    return user_info
```

This Python code simulates user identification through NFC technology. The user information is stored in a dictionary named user_database, where each user ID serves as a key mapping to a dictionary containing user details such as name and age. The NFCReader class has a static method, read_data(), which simulates reading NFC data. In this example, the method returns a hardcoded user ID ("User123").

b. Smart bench Features.

```
class SmartBench:
    def __init__(self, user_id):
        self.user_id = user_id
        self.sit_to_stand_data = []
        self.landing_data = []

    def identify_user(self):
        # Simulate user identification using NFC technology
        print(f"User {self.user_id} identified using NFC.")

    def capture_landing_data(self, coordinates, forces):
        # Capture landing coordinates and forces during landing
        landing_data = {"coordinates": coordinates, "forces": forces}
        self.landing_data.append(landing_data)
        print(f"Landing Data Captured: {landing_data}")

    def count_long_exercises(self, duration_threshold=30):
        # Count the number of sit-to-stand exercises of 30 seconds or more
        long_exercises = [exercise for exercise in self.sit_to_stand_data if exercise['duration'] >= duration_threshold]
        count = len(long_exercises)
        print(f"Count of Sit-to-Stand Exercises of 30 seconds or more: {count}")
        return count

# Example Usage:
user_id = 123
smart_bench = SmartBench(user_id)

# Simulate user identification
smart_bench.identify_user()

# Simulate capturing landing coordinates
landing_coordinates = {"x": 10, "y": 5, "z": 2}
landing_forces = {"force_x": 20, "force_y": 15, "force_z": 8}
smart_bench.capture_landing_data(coordinates=landing_coordinates, forces=landing_forces)

# Count the number of sit-to-stand exercises of 30 seconds or more
count_long_exercises = smart_bench.count_long_exercises(duration_threshold=30)
```

This Python code defines a SmartBench class representing a smart bench module for gait monitoring. The class has methods for user identification through NFC technology (identify_user), capturing landing data during exercises (capture_landing_data), and counting the number of sit-to-stand exercises lasting 30 seconds or more (count_long_exercises). In the example usage, an instance of SmartBench is created for a user with ID 123. Simulated functions showcase the identification of the user, capturing landing data, and counting long sit-to-stand exercises based on a specified duration threshold. The code aims to simulate the functionalities of the Smart Bench module within the proposed Smart Gait Monitoring System.

c. Essential Gait Dynamics Metrics on Smart Path.

```
class SmartPath:
    def __init__(self):
        self.gait_dynamics_data = []

    def measure_gait_dynamics(self, speed, step_length, stride_length):
        # Capture gait dynamics data
        dynamics_data = {"speed": speed, "step_length": step_length, "stride_length": stride_length}
        self.gait_dynamics_data.append(dynamics_data)
        print(f"Gait Dynamics Data Captured: {dynamics_data}")

    def analyze_gait_dynamics(self):
        # Analyze gait dynamics data
        average_speed = sum(data['speed'] for data in self.gait_dynamics_data) / len(self.gait_dynamics_data)
        average_step_length = sum(data['step_length'] for data in self.gait_dynamics_data) / len(self.gait_dynamics_data)
        average_stride_length = sum(data['stride_length'] for data in self.gait_dynamics_data) / len(self.gait_dynamics_data)

        print("Gait Dynamics Analysis:")
        print(f"Average Speed: {average_speed}")
        print(f"Average Step Length: {average_step_length}")
        print(f"Average Stride Length: {average_stride_length}")

# Example Usage:
smart_path = SmartPath()

# Simulate capturing gait dynamics data
smart_path.measure_gait_dynamics(speed=1.2, step_length=0.8, stride_length=1.5)
smart_path.measure_gait_dynamics(speed=1.0, step_length=0.7, stride_length=1.4)
smart_path.measure_gait_dynamics(speed=1.5, step_length=0.9, stride_length=1.7)

# Analyze gait dynamics
smart_path.analyze_gait_dynamics()
```

This Python code defines a SmartPath class representing a smart path module for gait monitoring. The class has methods for measuring gait dynamics (measure_gait_dynamics) and analyzing the captured gait dynamics data (analyze_gait_dynamics). The example usage creates an instance of SmartPath and simulates capturing gait dynamics data with different speed, step length, and stride length values. The subsequent analysis computes and prints the average speed, step length, and stride length based on the collected gait dynamics data. The code aims to simulate the functionalities of the Smart Path module within the proposed Smart Gait Monitoring System, providing insights into user mobility patterns.

d. Foundational 3D LIDAR Implementation.

```
class LidarSensor:
    def capture_point_cloud_data(self):
        # Simulated 3D Lidar data generation
        num_points = 100
        x = [random.uniform(-1, 1) for _ in range(num_points)]
        y = [random.uniform(-1, 1) for _ in range(num_points)]
        z = [random.uniform(0, 3) for _ in range(num_points)]

        return {"x": x, "y": y, "z": z}

class LidarVisualizer:
    @staticmethod
    def visualize_point_cloud(point_cloud_data):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(point_cloud_data['x'], point_cloud_data['y'], point_cloud_data['z'], c='r', marker='o')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        plt.show()

# Usage of 3D Lidar
lidar_sensor = LidarSensor()
lidar_data = lidar_sensor.capture_point_cloud_data()

# Visualize 3D Lidar data
LidarVisualizer.visualize_point_cloud(lidar_data)
```

This Python code defines two classes, LidarSensor and LidarVisualizer, simulating the functionality of a 3D Lidar sensor and its visualizer. The LidarSensor class has a method capture_point_cloud_data that generates simulated 3D Lidar data, including random points in three-dimensional space (x, y, z). The LidarVisualizer class contains a static method visualize_point_cloud that uses Matplotlib to create a 3D scatter plot visualizing the captured point cloud data.

The example usage demonstrates the instantiation of a LidarSensor object to capture 3D Lidar data and then utilizes the LidarVisualizer to visualize the generated point cloud data in a 3D scatter plot. This code snippet simulates the integration of 3D Lidar technology within the proposed Smart Gait Monitoring System, emphasizing its capability to capture high-resolution point cloud data for detailed analysis of joint movement, posture, and balance during gait.

e. Real-time Handrail Reliance Assessment.

```
class HandrailSensor:
    def __init__(self):
        self.activation_threshold = 0.5  # Threshold for handrail activation

    def assess_handrail_reliance(self, sensor_data):
        # Simulated sensor data (numeric values between 0 and 1)
        activation_duration = sum(sensor_data)  # Summing up the sensor data for simplicity

        # Assess handrail reliance based on activation duration
        if activation_duration > self.activation_threshold:
            return True  # User relies on the handrail
        else:
            return False  # User does not rely on the handrail

    @staticmethod
    def categorize_support_level(total_time_on_handrails):
        # Categorize support level based on total time spent on handrails
        if total_time_on_handrails > 60.0:  # 60 Seconds
            return "High Support"
        elif 30.0 <= total_time_on_handrails <= 60.0:
            return "Moderate Support"
        elif 10.0 <= total_time_on_handrails < 30.0:
            return "Low Support"
        else:
            return "No Significant Support"

# Usage of real-time handrail reliance assessment and support level categorization
handrail_sensor = HandrailSensor()

# Simulating multiple assessments to calculate total time on handrails
total_time_on_handrails = sum(RealTimeAssessment.simulate_real_time_assessment(handrail_sensor) for _ in range(5))

# Categorize support level based on total time on handrails
support_level = HandrailSensor.categorize_support_level(total_time_on_handrails)
```

This Python code defines a class named HandrailSensor, simulating the functionality of a sensor assessing handrail reliance and categorizing support levels based on the total time spent on handrails. The assess_handrail_reliance method takes simulated sensor data and assesses handrail reliance, returning a Boolean value. The categorize_support_level static method categorizes support levels (High, Moderate, Low, or No Significant Support) based on the total time spent on handrails.

The usage example creates an instance of HandrailSensor and simulates multiple assessments using the simulate_real_time_assessment function. The total time spent on handrails is calculated by summing the simulated assessments, and the support level is then categorized using the categorize_support_level method. This code snippet represents the real-time assessment of handrail reliance and the subsequent categorization of support levels within the proposed Smart Gait Monitoring System.

f. Data Collection, Pre-Processing & Logging.

```
class DataProcessor:
    def __init__(self):
        self.raw_data = []
        self.processed_data = []

    def collect_data(self, sensor_data):
        # Simulate real-time data collection
        time_stamp = time.time()
        data_point = {"timestamp": time_stamp, "sensor_data": sensor_data}
        self.raw_data.append(data_point)
        print(f"Data Collected: {data_point}")

    def preprocess_data(self):
        # Simulate noise reduction and calibration
        self.processed_data = self.raw_data  # For simplicity, we are assuming no pre-processing in this example
        print("Data Preprocessed")

    def log_data(self, file_format='json'):
        # Log processed data to a file
        if file_format == 'json':
            file_name = f"data_log_{int(time.time())}.json"
            with open(file_name, 'w') as file:
                json.dump(self.processed_data, file, indent=2)
        else:
            print("Unsupported file format. Logging as JSON.")
            self.log_data()

# Example Usage:
data_processor = DataProcessor()

# Simulate data collection
data_processor.collect_data({"sensor_1": 25, "sensor_2": 30, "sensor_3": 15})

# Simulate pre-processing
data_processor.preprocess_data()

# Log processed data to a JSON file
data_processor.log_data()
```

The Python code defines a class named DataProcessor representing a data processing module within the proposed Smart Gait Monitoring System. The class includes methods for collecting data (collect_data), preprocessing data (preprocess_data), and logging processed data (log_data).

In the example usage, an instance of DataProcessor is created, and data is simulated to be collected using the collect_data method. The collected raw data includes a timestamp and sensor data. The preprocess_data method is then simulated to perform noise reduction and calibration, and the processed data is logged to a JSON file using the log_data method.

This code snippet illustrates the basic functionality of the data processing module, simulating the collection, preprocessing, and logging of sensor data. The actual implementation would involve more sophisticated preprocessing techniques depending on the characteristics of the sensor data in the real system.

4. Iterative and Incremental Nature.

a. Feedback Mechanism

- Implement user-friendly interfaces for users to provide feedback.
- Plan regular feedback collection sessions during user testing.

- Integrate analytics tools for tracking user interactions and system performance.

b. Iterative Development

- Conduct regular sprint reviews and retrospectives for continuous improvement.
- Address any identified issues or bugs promptly in subsequent iterations.
- Plan for bi-weekly releases to incorporate new features and improvements.

c. User Testing

- Organize user testing sessions at key development milestones.
- Gather feedback on usability, performance, and overall user experience.
- Analyze feedback to identify areas for improvement and refinement.

III. MVP Evaluation

1. Assess Performance and Functionality.

The MVP (Minimum Viable Product) of the Smart Gait Monitoring System has been designed to focus on key features related to the Smart Bench and Smart Path components. The performance and functionality of the implemented features were evaluated based on their ability to address the core challenges identified in the conceptual solution. The NFC-based user identification on the Smart Bench demonstrated effective user-centric gait monitoring and non-intrusive continuous monitoring. The capturing of sit-to-stand exercise data and landing coordinates and forces provided valuable insights into mobility patterns and initial gait dynamics analysis. On the Smart Path, gait dynamics data collection, 3D LIDAR implementation, and real-time assessment of handrail reliance contributed to a more comprehensive understanding of user interactions and detailed gait analysis.

2. Testing & Evaluation.

It's important to note that the testing process for the MVP involved a preliminary phase where the functionality and performance were assessed through pseudo-code and hard-coded values. Simulated data inputs were used to assess the system's response to various user interactions, and real-time data collection was mimicked to validate the practical applicability of the Smart Bench and Smart Path components. However, a more comprehensive testing and simulation phase is planned in the coming weeks. This phase will involve a more systematic and rigorous evaluation of the system's performance under various scenarios, both simulated and real-world. This iterative testing process will be crucial in refining the system, identifying potential issues, and ensuring its robustness in diverse environments.

3. Challenges and Lessons Learned.

Throughout the development and preliminary evaluation of the MVP, a significant challenge was the constraint of time. To meet project deadlines, pseudo-codes were developed and tested to check the functionality and working of the modules, serving as the foundational structure of the Smart Gait Monitoring System. It is crucial to note that these pseudo-codes, while valuable for assessing core functionalities, may not function optimally in real-world scenarios due to external factors and

complexities not accounted for in the initial development phase. Recognizing this, future iterations will focus on improving the codes using real data and simulations with dedicated computer software, providing a more robust proof of concept. Another notable challenge is the lack of available real-world data during the current phase, emphasizing the need to gather data from users in real-life scenarios for more accurate testing and validation of the system's performance. This iterative approach and the willingness to adapt to evolving challenges are key takeaways for ensuring the system's effectiveness in practical healthcare applications.