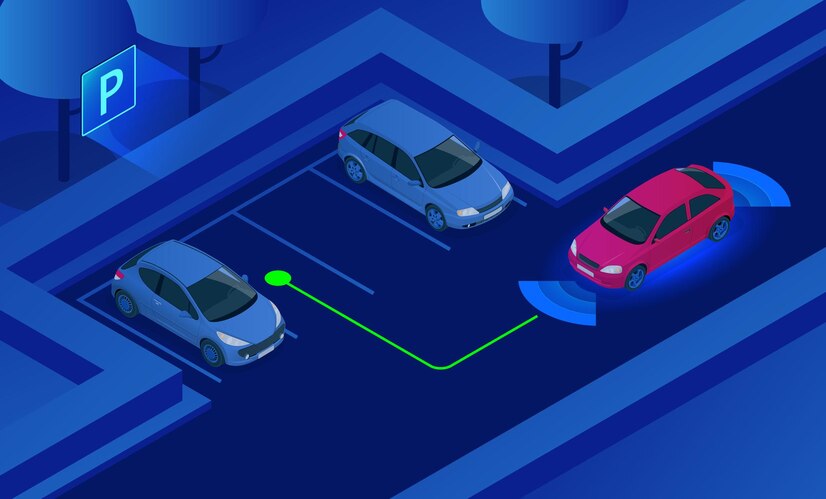
**SMART PARKING**

**INTRODUCTION AND DESIGN THINKING**

**Project Definition:**The project involves integrating IoT sensors into public transportation vehicles to monitor ridership, track locations, and predict arrival times. The goal is to provide real-time transit information to the public through a public platform, enhancing the efficiency and quality of public transportation services. This project includes defining objectives, designing the IoT sensor system, developing the real-time transit information platform, and integrating them using IoT technology and Python. 

**OBJECTIVES OF SMART PARKING:**

The objectives of smart parking using IoT, including real-time parking monitoring, mobile app integration, and efficient parking guidance, are :  
  
**1. Real-time parking monitoring:** One of the main objectives is to have a real-time monitoring system that continuously updates the availability of parking spaces. By using IOT sensors

**2. Mobile app integration:** The integration of smart parking systems with mobile applications allows drivers to access real-time parking information, reserve parking spots in advance, and make payments conveniently through their smartphones.

**3. Efficient parking guidance:** IoT-enabled smart parking systems can guide drivers to available parking spaces more efficiently. By leveraging sensors, cameras, or signage, the system can direct drivers to the nearest vacant parking spots, reducing the time

4. Traffic management and congestion reduction: Smart parking using IoT aims to contribute to better traffic management by reducing the number of vehicles circling around in search of parking.

**5. Revenue optimization**: Another objective of smart parking is to optimize revenue generation for parking operators. By implementing IoT systems, parking authorities can efficiently manage payment processes

* **Design and deployment of IOT sensors in parking spaces to detect occupancy and availability**   
    
  **1. Assess the parking area:** Begin by assessing the parking area's layout, size, and specific requirements. Identify the number of parking spaces and any unique features or challenges that need to be considered during design and deployment.  
   **2. Determine sensor placement:** Based on the assessment, determine the optimal locations to place the IOT sensors. Consider factors such as visibility, accessibility, power supply availability, and the type of parking spaces (e.g., parallel, angled, or perpendicular).  
    
  **3. Choose suitable IOT sensors:** Select IOT sensors that are reliable, accurate, and suitable for the parking space environment. Options include ultrasonic sensors, magnetic sensors, or camera-based sensors. Consider the cost, range, connectivity requirements, and compatibility with the chosen IOT platform.  
    
  **4. Establish connectivity infrastructure:** Determine the connectivity infrastructure required to transmit data from the sensors to the central system. This can be achieved through Wi-Fi, cellular networks, or dedicated IoT communication protocols like LoRaWAN or NB-IoT. Ensure sufficient coverage and network capacity.  
    
  **5. Plan power supply:** Decide on the power supply options for the IoT sensors. They can be battery-powered, solar-powered, or connected to a constant power source. Consider the maintenance requirements and the lifespan of the chosen power supply solution.  
    
  **6. Develop the central system**: Design and develop the central system that will receive, process, and analyse the data from the IoT sensors. It should provide real-time parking information and status updates. Consider integrating it with mobile apps, web portals, or smart signage for user convenience.  
    
  **7. Test and validate:** Before deployment, conduct thorough testing of the IoT sensors, connectivity, and central system. Validate the accuracy of occupancy detection and the responsiveness of the system. Make necessary adjustments or refinements if required.  
    
  **8. Deploy sensors and infrastructure:** Install the IoT sensors in the determined locations using appropriate mounting hardware. Deploy necessary infrastructure such as gateways, routers, or access points for data transmission and connectivity. Ensure proper installation, weatherproofing, and security measures.  
    
  **9. Monitor and maintain:** Establish a monitoring and maintenance plan to ensure ongoing functionality. Regularly check sensor performance, battery levels, connectivity, and data accuracy. Address any issues promptly to maintain optimal system performance.  
    
  **10. Data analysis and optimization:** Continuously analyse the collected data to identify parking patterns, occupancy trends, and utilization rates. Use this information to optimize parking operations, improve efficiency, and make informed decisions regarding pricing, capacity planning, and resource allocation.  
  **A mobile app interface that displays real-time parking availability to users:**

**2. Map View:**- Upon entering a location, display a map view of the area with parking icons representing available spaces.  
- Show the user's current location as a marker on the map.  
- Color code the parking icons to indicate availability in real-time (e.g., green for available, orange for limited, and red for full).  
- Implement zoom and pan functionalities to allow users to explore the map.  
  
**3. List View:**  
- Provide an alternative list view of nearby parking spots alongside the map view.  
- Show the distance to each parking spot from the user's current location.  
- Display the availability status of each parking spot using color-coded indicators similar to the map view.  
- Include additional details such as parking rates, hours of operation, and any special features or restrictions.  
  
**4. Filter and Sorting Options:**- Include filter and sorting options to allow users to customize their search.  
- Provide filters for preferred parking types (e.g., covered, street, garages) and amenities (e.g., EV charging, disabled parking).  
- Offer sorting options based on distance, price, availability, or user ratings.  
  
5**. Detailed Parking Spot Information:**- Upon selecting a specific parking spot from the map or list view, display detailed information about that spot.  
- Show real-time occupancy information (e.g., the number of available spaces out of total capacity).  
- Provide directions to the parking spot from the user's current location, including estimated travel time.  
- Include parking rates, accepted payment methods, and any additional services or requirements.

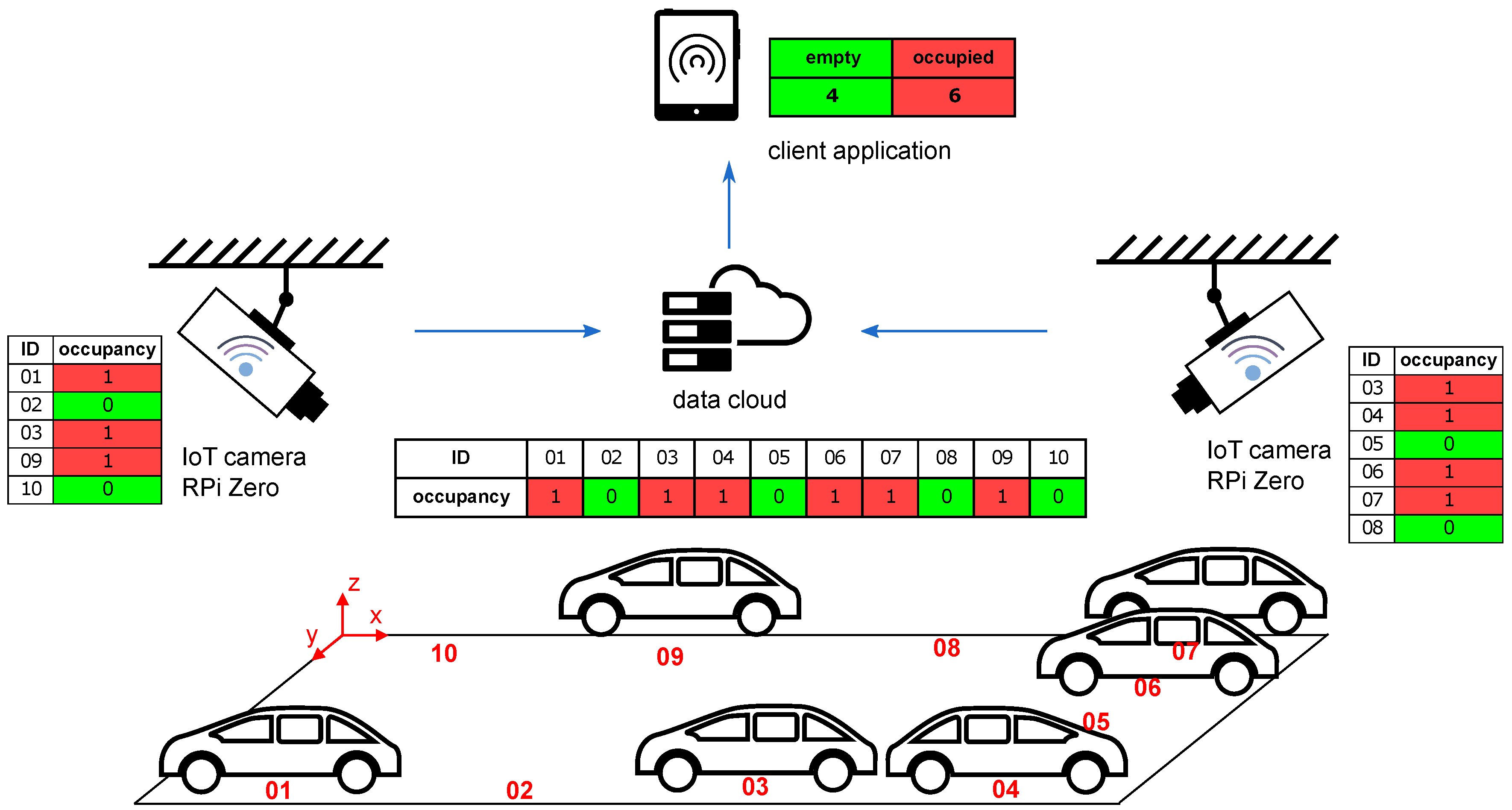
**8.Notifications and Alerts:**  
- Send push notifications or in-app alerts to inform users about parking availability updates, promotions, or time-limited discounts.  
- Notify users when their booked parking spot is close or when there are changes to the availability status.  
  
Remember to maintain a user-friendly and intuitive interface throughout the app. Use clear and concise language, provide visual cues, and incorporate common mobile app design patterns. Regularly gather user feedback and conduct usability testing to refine and improve the interface based on user needs and preferences.

**To collect data from sensors and update the mobile app using a Raspberry Pi,**

**1. Sensor Integration:**- Identify the necessary sensors for collecting the desired data (e.g., parking occupancy, temperature, humidity).  
- Connect the sensors to the appropriate GPIO (General Purpose Input/Output) pins on the Raspberry Pi.  
- Install any required libraries or drivers to enable communication with the sensors.  
  
**2. Data Collection and Processing:**  
- Write a script in a programming language like Python to read data from the connected sensors.  
- Use the appropriate libraries or APIs provided by the sensor manufacturers to access the sensor data.  
- Process the collected data as needed (e.g., calculate parking occupancy percentage).  
  
**3. Real-Time Data Streaming:**  
- Establish a connection between the Raspberry Pi and the mobile app for real-time data streaming. One option is to use MQTT (Message Queuing Telemetry Transport) protocol.  
- Set up an MQTT on the Raspberry Pi or a separate server to facilitate communication.  
- Publish the collected sensor data to specific MQTT topics.  
  
**4. Mobile App Integration:**  
- Develop the mobile app using a framework like React Native or Flutter.  
- Implement MQTT client functionality in the mobile app to subscribe to relevant MQTT topics and receive sensor data updates.  
- Upon receiving new data, update the app's UI to display the real-time information (e.g., parking availability, temperature).

**CAMERA BASED PARKING DETECTION**

**Camera based parking detection** in IOT refers to using cameras and internet of things (IOT) technology to monitor and analyse parking spaces in real time. This system uses cameras to capture images or videos of parking areas, and then utilizes computer vision algorithms and machine learning techniques to analyse these visuals and detect the presence or absence of vehicles in the parking spaces.

**Parking enforcement:** By detecting the presence or absence of vehicles in parking spaces, this system can be used to enforce parking regulations and prevent unauthorized parking. It can alert authorities when a vehicle is parked in a restricted area or exceeds a time limit 

**1. Camera Monitoring:** Cameras installed in parking areas continuously capture images or videos of the parking spaces. The cameras may be fixed in position or have the ability to pan, tilt, and zoom for a wider coverage area.

**2. Data Transmission:** The captured visuals are transmitted over a network connection, such as Wi-Fi or Ethernet, to a centralized server or cloud platform. This allows for remote access and monitoring of the parking spaces.

**3. Image Processing:** On the centralized server or cloud platform, computer vision algorithms and machine learning models are applied to process the captured visuals. These algorithms analyse the images or videos to detect the presence or absence of vehicles in the parking spaces.

**4. Vehicle Detection:** The computer vision algorithms utilize techniques such as object detection, image classification, or pattern recognition to identify and locate vehicles within the parking areas. These algorithms have been trained on large datasets to accurately identify vehicles in different lighting conditions and angles.

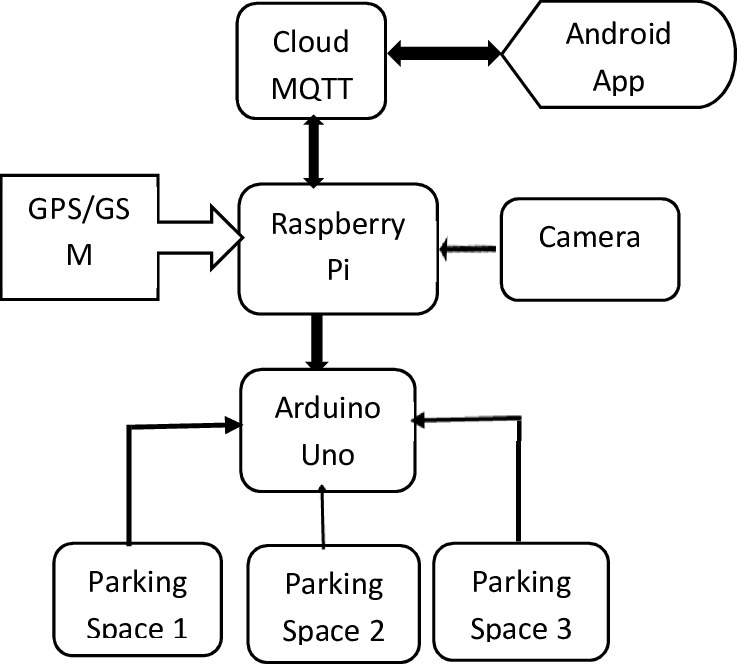
**5. Real-time Monitoring:** As the detection process occurs, the system provides real-time monitoring of the occupancy status of the parking spaces. It can indicate whether a space is vacant or occupied, and update this information continuously.

**6. Alerts and Notifications:** The system can generate alerts and notifications based on specific conditions. For example, it can notify authorities or parking operators when a violation,

**7. Integration with IOT Devices:** The parking detection system can be integrated with other IOT devices or systems, such as smart parking meters or mobile applications. This integration allows for seamless connectivity and automation, enhancing the overall parking experience.

**9. Maintenance and Updates:** Regularly maintain and update the cameras, algorithms, and cloud platform to ensure the system's optimal performance and accuracy. This may involve monitoring the camera hardware, training the machine learning models with new data, and applying software updates.

**10. Security:** Implement proper security measures to protect the data transmitted and stored by the parking detection system. This may involve encryption, access controls, and monitoring for any potential security breaches.



Remember that camera based parking detection in IoT is a complex system that involves multiple technologies and components. It is important to thoroughly plan and test each step to ensure a reliable and accurate parking detection system.

Overall camera based parking detection in IOT offers a smart and efficient solution for managing parking spaces, improving the overall parking experience, and optimizing parking operations in cities and other urban environments.

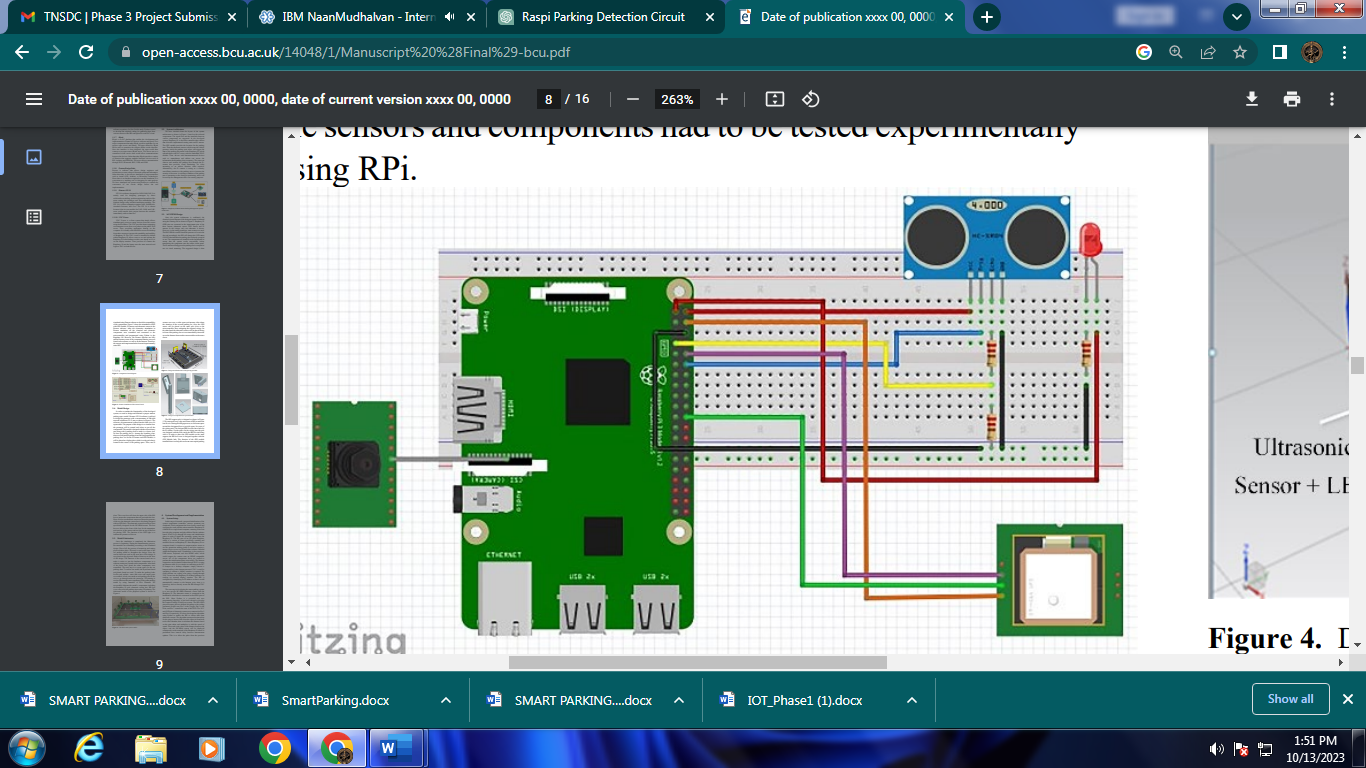
While programming is required to develop and implement the necessary software components for the camera-based parking detection system, the **Raspberry Pi** provides a flexible and cost-effective platform for running the required algorithms and managing the system. Whether it's capturing visuals, processing data, networking, or integrating with other IoT devices, the Raspberry Pi can effectively perform these tasks in a camera-based parking detection system in IoT.

**IOT sensor system and Raspberry Pi integration.**

* ultrasonic sensors or distance measuring sensors to measure the distance between the vehicle and nearby objects. This information can then be used to determine if a parking space is available or occupied.
* Additionally, Raspberry Pi can also utilize image processing techniques with the help of a camera module to analyze images or video footage and identify vacant parking spaces based on certain criteria, such as the absence of a vehicle or the presence of specific markings.  E.

**Pin connections:**

* **Ultrasonic Sensor**:
  + VCC (Power): Connect to 5V (Pin 2 or Pin 4) on the Raspberry Pi.
  + Trig (Trigger): Connect to a GPIO pin (e.g., GPIO17, Pin 11 on Raspberry Pi).
  + Echo: Connect to another GPIO pin (e.g., GPIO18, Pin 12 on Raspberry Pi).
  + GND (Ground): Connect to Ground (Pin 6, Pin 9, or Pin 14) on the Raspberry Pi.



**Python script for parking detection using an ultrasonic sensor. This script measures the distance and detects if a car is in a parking space based on a distance threshold:**

**import RPi.GPIO as GPIO**

**import time**

**# Set GPIO mode and pins**

**GPIO.setmode(GPIO.BCM)**

**TRIG\_PIN = 17**

**ECHO\_PIN = 18**

**GPIO.setup(TRIG\_PIN, GPIO.OUT)**

**GPIO.setup(ECHO\_PIN, GPIO.IN)**

**def distance():**

**GPIO.output(TRIG\_PIN, True)**

**time.sleep(0.00001)**

**GPIO.output(TRIG\_PIN, False)**

**while GPIO.input(ECHO\_PIN) == 0:**

**pulse\_start = time.time()**

**while GPIO.input(ECHO\_PIN) == 1:**

**pulse\_end = time.time()**

**pulse\_duration = pulse\_end - pulse\_start**

**return pulse\_duration \* 17150**

**try:**

**while True:**

**dist = distance()**

**print(f"Distance: {dist} cm")**

**# Adjust this threshold as needed for your parking space**

**threshold = 30 # Change this distance as needed**

**if dist < threshold:**

**print("Parking space occupied")**

**else:**

**print("Parking space available")**

**time.sleep(1)**

**except KeyboardInterrupt:**

**GPIO.cleanup()**

**Connect to an application or cloud service :**

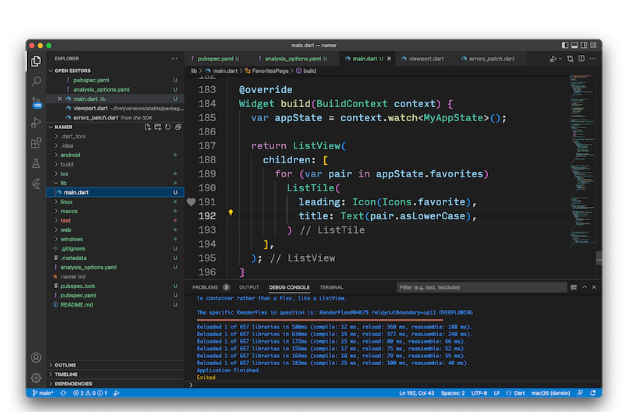
* Raspberry Pi can connect to applications or the cloud in a few different ways. One common method is by using Wi-Fi or Ethernet connectivity to establish a network connection
* To connect to an application or cloud service, you can use various protocols such as HTTP, MQTT, or Web Socket. These protocols enable the Raspberry Pi to send and receive data to and from the application or cloud

.

* You can use APIs (Application Programming Interfaces) provided by the application or cloud service to integrate the Raspberry Pi with specific functionalities or data. This allows for seamless communication and interaction between the Raspberry Pi and the application or cloud.

**DEVELOPING MOBILE APP USING PYTHON AND FRAMEWORK (FLUTTER)**

import 'package:english\_words/english\_words.dart';  
import 'package:flutter/material.dart';  
import 'package:provider/provider.dart';  
  
void main() {  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  const MyApp({super.key});



  @override  
  Widget build(BuildContext context) {  
    return ChangeNotifierProvider(  
      create: (context) => MyAppState(),  
      child: MaterialApp(  
        title: 'Namer App',  
        theme: ThemeData(  
          useMaterial3: true,  
          colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepOrange),  
        ),  
        home: MyHomePage(),  
      ),  
    );  
  }  
}  
  
class MyAppState extends ChangeNotifier {  
  var current = WordPair.random();  
}  
  
class MyHomePage extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    var appState = context.watch<MyAppState>();  
  
    return Scaffold(  
      body: Column(  
        children: [  
          Text('A random idea:'),  
          Text(appState.current.asLowerCase),  
        ],  
      ),  
    );  
  }  
}

