

Campus Crowd Management and Navigation System

Project Overview

The **Campus Crowd Management and Navigation System** is a smart solution designed to help students, faculty, and visitors efficiently navigate a university campus while avoiding overcrowded areas. This system uses **real-time crowd data, location tracking, and AI-based predictions** to ensure a smoother campus experience. It provides dynamic navigation routes, event crowd tracking, and facility usage insights, ultimately improving safety, convenience, and time management.

Problem Statement

University campuses often experience congestion in key areas such as classrooms, cafeterias, libraries, and event venues. This congestion leads to several challenges:

- ◆ **Time wastage:** Students and faculty struggle to reach their destinations on time due to overcrowding, leading to delayed classes, missed meetings, and reduced productivity.
- ◆ **Safety concerns:** Overcrowded areas may pose health and safety risks, especially during emergencies like fire drills, medical situations, or natural disasters, making evacuation difficult.
- ◆ **Inefficient facility usage:** Students may avoid important campus spaces such as libraries, study halls, and recreational areas because they expect them to be overcrowded, leading to **underutilization of resources**.
- ◆ **Navigation difficulties:** New students, visitors, and even faculty members may find it challenging to navigate the campus efficiently, particularly during the start of a semester or large events.

- ◆ **Limited accessibility for people with disabilities:** Overcrowding can make it harder for students with mobility challenges to access essential areas like ramps, elevators, and designated seating areas.
- ◆ **Increased waiting times:** High foot traffic in cafeterias, administrative offices, and transport hubs (such as shuttle bus stops) leads to long queues, frustration, and inefficiency.
- ◆ **Event mismanagement:** Large-scale campus events often result in sudden surges in crowd movement, causing **logistical challenges**, bottlenecks at entry/exit points, and increased risk of accidents.
- ◆ **Lack of real-time information:** Students and staff **lack access to real-time updates** about congested areas, leading to poor decision-making about when and where to go.
- ◆ **Strain on campus security & administration:** Managing large crowds manually can be overwhelming for security personnel and campus authorities, increasing the risk of mismanagement and conflicts.
- ◆ **Environmental impact:** Unregulated crowd movement leads to excessive **energy consumption** (lighting, air conditioning) in certain areas and increased **waste generation** in high-traffic zones.

Features

Feature: Real-Time Heatmaps for Campus Areas

This feature enables students and faculty to visualize **crowd density and movement patterns** across different areas of the campus in real time. The interactive heatmaps will display which areas are crowded (e.g., **cafeterias, corridors, libraries**) and which are less busy.

- ◆ **Purpose:** Helps users make informed decisions about where to go, avoiding congestion and improving campus mobility.
- ◆ **How it Works:** The system gathers real-time location data from various sources (Wi-Fi, GPS, sensors, CCTV) and dynamically updates heatmaps to reflect changing crowd levels.

◆ Key Benefits:

- Avoid crowded areas during peak hours
 - Find quiet study spaces easily
 - Improve safety by identifying overcrowded locations
 - Optimize campus facility usage
-

How to Implement It?

1. Data Collection (Gathering Real-Time Foot Traffic Data)

To generate heatmaps, we need real-time data on crowd movement. This can be done using:

- ✓ **Wi-Fi/Bluetooth Beacons:** Count the number of active Wi-Fi/Bluetooth connections in different campus zones.
- ✓ **GPS Data from a Mobile App:** If users opt in, collect their location within campus boundaries.
- ✓ **CCTV & AI-based Vision Analysis:** Use existing security cameras and AI models to estimate crowd density.
- ✓ **IoT Sensors:** Install motion sensors or pressure-sensitive floor mats in key locations (e.g., libraries, cafeterias).

2. Backend System (Processing and Analyzing Data)

Once data is collected, it must be processed to determine high and low foot-traffic areas.

◆ Data Processing Steps:

1. **Receive data** from multiple sources (Wi-Fi, GPS, CCTV).
2. **Apply clustering algorithms** (e.g., **DBSCAN**, **K-Means**) to detect crowded zones.
3. **Store & update heatmap data** in a database (e.g., **Firebase Firestore** for real-time sync).

◆ Tech Stack Suggestions:

- **Server:** Node.js or Python (FastAPI, Flask)
- **Database:** Firebase (for real-time updates) or PostgreSQL
- **AI/ML Models:** TensorFlow (for AI-based crowd detection via CCTV)

3. Frontend (Displaying Interactive Heatmaps on the App)

The real-time heatmaps should be visually interactive and easy to interpret.

◆ Implementation Steps:

1. **Use Google Maps API or Leaflet.js** to render campus maps.
2. **Overlay heatmaps** with color gradients (green = less crowded, red = highly crowded).
3. Allow users to **filter locations** (e.g., "Show me foot traffic in the cafeteria").
4. Enable **real-time updates** so the heatmap changes dynamically.

◆ Tech Stack Suggestions:

- **Mobile App:** React Native / Flutter
- **Web Dashboard:** React.js / Vue.js
- **Map Integration:** Google Maps API or Leaflet.js

4. Additional Features for Optimization

- ✓ **Push Notifications:** Notify users when an area is overcrowded.
- ✓ **Alternative Route Suggestions:** Suggest less crowded paths.
- ✓ **Predictive Analytics:** Use past data to predict crowd levels at different times of the day.

📌 Real-Time Heatmaps Using Thermal Sensors

This approach utilizes **thermal imaging sensors** to detect crowd density based on body heat signatures. Unlike cameras, **thermal sensors provide better privacy protection** because they do not capture facial details, only heat maps of human presence.

How It Works?

Thermal sensors detect **temperature variations** in different campus locations and generate real-time **heatmaps** showing crowded and empty areas.

◆ Where to Install Sensors?

- **Entry & Exit Points:** To track the number of people entering/exiting buildings.
- **Corridors & Cafeterias:** To detect movement and crowd density.
- **Libraries & Study Areas:** To indicate available seating.

◆ How Data is Processed?

- Sensors capture **infrared radiation** from human bodies.
- AI algorithms **filter out non-human heat sources** (e.g., hot surfaces).
- The system **calculates density** in each location based on **heat intensity and movement patterns**.
- Data is **sent to a cloud database** and displayed as a **color-coded heatmap** in a mobile app.

How to Implement It?

1 Hardware Setup (Thermal Sensor Installation)

You will need **thermal sensors** with IoT capabilities to collect temperature data in real-time.

✓ Sensor Choices:

- **FLIR Lepton** (Affordable, compact thermal sensor for IoT)
- **Melexis MLX90640** (High-resolution thermal imaging)
- **Seek Thermal Module** (For advanced real-time processing)

✓ Installation:

- Mount sensors **on ceilings or walls** in high-traffic areas.
 - Connect them to **Raspberry Pi/ESP32** for real-time data collection.
 - Ensure **Wi-Fi/Bluetooth connectivity** for transmitting data.
-

Data Processing & AI Analysis

Once the sensors capture thermal images, the data must be processed using **AI and cloud computing**.

Processing Steps:

1. **Preprocessing:** Convert **thermal data** into a structured format.
2. **AI Filtering:** Use **OpenCV + TensorFlow** to detect human heat patterns.
3. **Density Calculation:** Estimate the **number of people per zone** based on **heat intensity**.
4. **Real-Time Updates:** Send **crowd density data** to a cloud database (**Firestore, AWS IoT, Google Cloud**).




Tech Stack Suggestions:

- **Hardware:** Raspberry Pi + FLIR Lepton Thermal Camera
- **Backend:** Python (Flask/FastAPI) for processing
- **AI Model:** TensorFlow + OpenCV (for object detection)
- **Database:** Firestore (for real-time heatmap updates)

Crowd Density Forecasting Using Machine Learning

Feature Explanation

The **Crowd Density Forecasting** feature predicts **future crowd levels** in different campus areas based on:

-  **Historical crowd data** (past foot traffic trends).
-  **Event schedules** (upcoming exams, fests, or holidays).
-  **User behavior** (typical student movement patterns).

Benefits of Forecasting

-  **Helps students plan** their day efficiently.

- 📍 **Avoid overcrowded areas** like libraries and cafeterias.
- 🏫 **Optimize campus facilities** by predicting peak usage.
- 🚀 **Enhance safety** by preventing congestion.

📌 Step-by-Step Implementation

🔧 Step 1: Data Collection

To train a machine learning model, we need **historical data** about crowd density.

◆ Data Sources

Source	Data Collected	Example
Thermal Sensors	Heat signatures, foot traffic	"Library: 120 people at 3 PM"
Wi-Fi & Bluetooth	Device connections per area	"Cafeteria: 200 devices connected"
RFID Card Swipes	Entry/exit logs	"Main gate: 500 students entered"
Event Schedule	Exam dates, festivals	"Tech Fest: 5,000 attendees expected"

🔧 Step 2: Data Preprocessing

We need to **clean and structure** the collected data before training a model.

◆ Steps

1. **Convert timestamps** into hourly or daily intervals.
2. **Remove anomalies** (e.g., sensor errors, missing data).
3. **Standardize data** (normalize values to a common scale).
4. **Label event-based spikes** (map crowd surges to specific events).

◆ Example: Converting Raw Data into Usable Format

```
import pandas as pd

# Load sample dataset
data = pd.read_csv("crowd_data.csv")
```

```
# Convert timestamp to datetime format
data['timestamp'] = pd.to_datetime(data['timestamp'])

# Aggregate data per hour
data['hour'] = data['timestamp'].dt.hour
hourly_data = data.groupby(['location', 'hour']).mean().reset_index()

print(hourly_data.head())
```

Step 3: Choosing the Right ML Model

We use **time series forecasting models** to predict future crowd density.

◆ Model Options

Model	Pros	Cons
ARIMA (AutoRegressive Integrated Moving Average)	Good for trend-based predictions	Struggles with sudden changes
LSTM (Long Short-Term Memory, Deep Learning)	Captures long-term patterns	Needs lots of data
XGBoost/Random Forest	Works well with multiple factors	Requires feature engineering

Step 4: Training a Machine Learning Model

We will use an **LSTM neural network** since it works well with time-series data.

◆ LSTM-Based Crowd Density Prediction

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```



```

# Load processed dataset
data = pd.read_csv("processed_crowd_data.csv")

# Prepare data for LSTM model
sequence_length = 24 # Use last 24 hours to predict the next hour
X, y = [], []

for i in range(len(data) - sequence_length):
    X.append(data.iloc[i:i+sequence_length, 1:].values)
    y.append(data.iloc[i+sequence_length, 1])

X, y = np.array(X), np.array(y)

# Build LSTM Model
model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(X.shape
[1], X.shape[2])),
    LSTM(50, activation='relu'),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')

# Train Model
model.fit(X, y, epochs=50, batch_size=16, validation_split=0.2

```

Step 5: Making Predictions

After training the model, we can **predict future crowd density**.

```

import matplotlib.pyplot as plt

# Predict next 6 hours
future_predictions = model.predict(X[-6:])

```

```
# Plot Predictions
plt.plot(range(6), future_predictions, marker='o', label="Predicted Crowd Den
sity")
plt.xlabel("Hours Ahead")
plt.ylabel("Estimated Crowd Count")
plt.legend()
plt.show()
```

Step 6: Integrating with Mobile App

Once we have predictions, we need to **display them in an app**.

◆ Firebase Integration (Real-Time Updates)

```
import firebase_admin
from firebase_admin import credentials, db

cred = credentials.Certificate("serviceAccountKey.json")
firebase_admin.initialize_app(cred, {"databaseURL": "https://your-db.firebaseio.com"})

def upload_forecast(location, forecast_data):
    ref = db.reference(f"/forecast/{location}")
    ref.set({"next_hours": forecast_data.tolist()})

upload_forecast("Library", future_predictions)
```

◆ Displaying in Mobile App (React Native)

```
import { useState, useEffect } from 'react';
import database from '@react-native-firebase/database';
```

```

const ForecastScreen = () => {
  const [forecast, setForecast] = useState([]);

  useEffect(() => {
    const ref = database().ref('/forecast/Library');
    ref.on('value', snapshot => {
      setForecast(snapshot.val()?.next_hours || []);
    });

    return () => ref.off();
  }, []);

  return (
    <View>
      <Text>Library Crowd Prediction</Text>
      {forecast.map((count, index) => (
        <Text key={index}>Hour {index + 1}: {count} people</Text>
      ))}
    </View>
  );
};

```

Step 7: Additional Features & Optimization

Adding Event Awareness

We can **incorporate campus event schedules** into our model.

```

# Adding event impact
data["is_event_day"] = data["date"].apply(lambda x: 1 if x in event_dates else
0)

```

Implementing Push Notifications

- **Send alerts** if a location is predicted to be overcrowded.

- Example: *"Library will be full in 2 hours. Consider alternative study areas!"*

Integration with Campus Scheduling Systems for Crowd Forecasting

◆ Feature Explanation

This feature integrates **crowd forecasting** with **campus scheduling systems** to predict how events like **classes, exams, and seminars** will impact foot traffic in different areas.

🌟 Why Is This Useful?

- ✓ **Real-time crowd projections** based on event schedules.
- ✓ **Warn students about high-traffic areas** before they go.
- ✓ **Optimize space utilization** in cafeterias, libraries, and lecture halls.
- ✓ **Improve safety & campus management** by preventing overcrowding.

How to Implement It

Step 1: Data Collection

To predict crowd surges, we need **two main types of data**:

◆ 1. Campus Schedule Data

- **Classroom Schedules** – List of class timings & room assignments.
- **Exam Timetables** – Predicts surges near exam halls.
- **Event Calendars** – Festivals, workshops, guest lectures.

API Integration with Campus Schedule Systems

If the campus uses **Google Calendar, Microsoft Outlook, or a university ERP system**, we can **connect via API** to pull event data.

```
import requests
```

```
# Example API call to fetch university schedule
```

```
api_url = "https://university-scheduler.com/api/events"
```

```
response = requests.get(api_url)

if response.status_code == 200:
    event_data = response.json() # Store schedule data
else:
    print("Failed to fetch schedule data")
```

Event-Centric Crowd Management System

◆ Feature Explanation

This system **manages crowd flow during campus events** such as **fests, sports events, and conferences** by:

- ✓ **Providing real-time crowd density updates** for event locations.
- ✓ **Guiding attendees to less crowded areas** (e.g., different food stalls, rest areas).
- ✓ **Suggesting alternate times** to attend events based on crowd forecasts.
- ✓ **Enhancing safety measures** by detecting overcrowding situations.

Campus Navigation with Dynamic Path Re-Routing

◆ Feature Description:

A smart campus navigation system that provides real-time route suggestions based on crowd density. This feature helps students, faculty, and visitors **avoid congested areas** and reach their destinations efficiently, whether they are **walking, biking, or using public transport**.

◆ Key Benefits:

- ✓ **Reduces travel time:** Users can take alternate routes to avoid delays caused by crowded areas.
- ✓ **Enhances user experience:** Helps students and faculty navigate the campus smoothly, especially during peak hours.

- ✅ **Improves accessibility:** People with disabilities can avoid congested paths, ensuring safer movement.
- ✅ **Optimizes campus infrastructure:** Distributes foot traffic efficiently, preventing overcrowding in certain areas.

◆ Implementation Approach:

1 Real-Time Crowd Data Collection:

- Use **IoT sensors**, **CCTV analytics**, or **WiFi/Bluetooth tracking** to estimate crowd density in various parts of the campus.
- Gather live **location data** from users via a **mobile app** or GPS-enabled devices.

2 AI-Powered Route Optimization:

- Develop an **algorithm** that dynamically calculates **optimal routes** based on real-time foot traffic.
- Use **graph-based shortest path algorithms** (like **Dijkstra's Algorithm** or *A Search*) to determine the best paths.

📌 Crowd-Sensitive Transport Scheduling - Smart Optimization

🚌 Feature Overview: Smart Campus Transport System

Integrate **real-time crowd data** with **campus shuttle/bus services** to:

- **Optimize Routes** 📍 → Adjust bus stops based on student movement patterns.
- **Dynamically Modify Schedules** ⌚ → Deploy extra buses during peak hours.
- **Avoid Congestion** 🚦 → Suggest alternate routes if main roads are blocked.

🎯 Key Benefits:

- ✓ Reduce waiting times at **bus stops** 📍
- ✓ Improve **shuttle availability** during events 🎉

✓ Prevent overcrowding in **buses & waiting areas** 🚶

🔧 How to Implement This?

◆ 1. Live Crowd Data Integration with Transport System

📌 Data Sources:

1 **IoT Thermal Sensors & CCTV AI Analysis** 📡 → Count people at bus stops.

2 **Wi-Fi/Bluetooth Beacons** 📶 → Track crowd density in real time

```
# Fetch live crowd density from sensors
def get_crowd_density(bus_stop):
    return sensor_data.get(bus_stop, 0) # Return crowd count

# Example: Get crowd at "Main Gate"
crowd_at_main_gate = get_crowd_density("Main Gate")
print(f"Crowd at Main Gate: {crowd_at_main_gate} people")
```

◆ 2. Dynamic Bus Scheduling (AI-Based Prediction)

- **Peak Hour Prediction** ⌚ → Detect rush periods & schedule extra buses.
- **Traffic-Based Route Adjustments** 🚦 → Avoid congested areas.

📌 Implementation:

1 Train an **AI model** on **historical crowd & transport data**.

2 Predict **when and where** extra buses will be needed.

3 Send **automated schedule updates** to the transport system.

```
from sklearn.linear_model import LinearRegression

# Sample historical data: [hour, crowd size] → buses needed
X = [[8, 50], [9, 70], [10, 30], [11, 90]] # Time & crowd size
y = [2, 3, 1, 4] # Buses required

# Train model
model = LinearRegression().fit(X, y)
```

```
# Predict buses needed for current crowd size at 10 AM
predicted_buses = model.predict([[10, 80]])
print(f"Recommended Buses: {int(predicted_buses[0])}")
```

◆ 3. Smart Routing & Traffic Avoidance

- **AI suggests best routes** based on real-time congestion.
- **Bus drivers receive real-time alerts** 📱.

```
# Detect high congestion zones
def suggest_route(crowded_areas):
    if "Main Road" in crowded_areas:
        return "Use Alternative Route via Park Road"
    return "Regular Route"

# Example: Main Road is crowded
route = suggest_route(["Main Road", "Library"])
print(f"Suggested Route: {route}")
```

◆ 4. Passenger Load Balancing

- **Limit bus boarding** if it reaches max capacity.
- **Direct passengers to next available bus.**

```
MAX_CAPACITY = 50 # Max passengers per bus

def check_bus_load(current_passengers):
    return "Full - Board Next Bus" if current_passengers >= MAX_CAPACITY else "Seats Available"

status = check_bus_load(52)
print(status)
```


Queue Management for Popular Campus Facilities

Feature Overview: Smart Queue Monitoring

Optimize **high-demand campus services** like:

- **Food courts** 🍕 → Reduce wait times by suggesting less crowded hours.
- **Bookstores** 📖 → Notify students when lines get shorter.
- **Gym & Sports Centers** 🏋️ → Show real-time occupancy and best time slots.

Key Benefits:

- ✓ Minimize **waiting times** ⌚
- ✓ Improve **service efficiency** ⚡
- ✓ Enhance **student experience** 🎓

How to Implement This?

◆ 1. Real-Time Queue Length Detection

Data Sources:

- 1 **Thermal Sensors & AI Cameras** 📡 → Detect queue lengths.
- 2 **Wi-Fi/Bluetooth Beacons** 📶 → Track mobile devices in the queue.
- 3 **Student Check-ins via App** 📱 → Voluntary queue status updates.

```
# Fetch real-time queue data from sensors
def get_queue_length(location):
    return sensor_data.get(location, 0) # Return people count

# Example: Get queue at Cafeteria
queue_at_cafeteria = get_queue_length("Cafeteria")
print(f"Queue at Cafeteria: {queue_at_cafeteria} people")
```

◆ 2. AI-Based Waiting Time Prediction

- **Estimate wait times** ⌚ based on queue length.
- **Predict peak & off-peak hours** 📊 using past data.

📌 Implementation:

1 Train a **Machine Learning Model** using:

- Queue length 📊
- Average service time per person ⌚
- Historical wait times 🕒

```
from sklearn.linear_model import LinearRegression
# Sample data: [queue length] → wait time in minutes
X = [[5], [10], [15], [20], [25]] # People in queue
y = [5, 10, 15, 20, 25] # Predicted wait time (minutes)

# Train model
model = LinearRegression().fit(X, y)

# Predict wait time for current queue length (18 people)
predicted_wait = model.predict([[18]])
print(f"Estimated Wait Time: {int(predicted_wait[0])} minutes")
```

◆ 3. Smart Queue Alerts & Notifications

- **Push notifications** 📧 → Alert users when queues are short.
- **Digital signage** 📺 → Display wait times at locations.
- **Live tracking on campus app** 📱 → Show real-time queue status.

```
# Notify students when queue is short
def notify_users(queue_length):
    if queue_length < 5:
```

```
    return "Queue is short! Visit now."
    return "High wait time. Check later."
```

```
status = notify_users(4)
print(status)
```

◆ 4. Self-Check-In for Queue Optimization

- **Students can join a virtual queue** via app.
- **Receive an alert** when it's their turn.

```
queue_list = []

# Add user to queue
def join_queue(user):
    queue_list.append(user)
    return f"{user}, you have joined the queue. Current position: {len(queue_list)}"



print(join_queue("Srijan"))
```

Multi-Mode Campus Navigation: Walking, Cycling, and Parking

Feature Overview: Multi-Mode Navigation

This feature provides **dynamic navigation options** tailored to various modes of transport: walking, cycling, and driving. It utilizes **real-time crowd data** to suggest **optimal routes** and **available parking** in real-time.




Why Is This Useful?

- ✓ Reduce travel time 
- ✓ Avoid crowded areas 

✓ Help with parking availability 

✓ Enhance campus mobility 


Modes of Transport Covered:

- **Walking**  → Short distances, suitable for crowded paths.
 - **Cycling**  → Faster routes, ideal for less crowded paths.
 - **Driving**  → For users heading to distant locations or parking lots.
-

How to Implement It?


1. Multi-Mode Navigation Integration

Data Sources:

1 Crowd Data  → Gathered from thermal sensors, cameras, and app check-ins.

2 Traffic & Route Data  → Collected from GPS, traffic sensors, or user reports.

3 Parking Data  → Real-time availability of parking spots.


4 Campus Map  → Detailed map of campus buildings, paths, and bike racks.

2. Route Suggestions Based on Mode of Transport



Walking Routes:

- **Shorter routes** through less crowded areas.
- **Sidewalk data** for avoiding blocked paths.
- **Real-time crowd data** to avoid congested corridors.

Cycling Routes:

- **Bike-friendly paths**  → Suggest routes that avoid traffic and pedestrian-heavy zones.
- **Bike racks & stations** → Show available parking for bikes.

Driving Routes:

- **Optimal driving routes**  → Avoid congested parking lots.
- **Parking Availability**  → Show available parking spaces near the destination.

◆ 3. Real-Time Data Integration for Route Optimization

📌 Walking Mode:

1. Use **crowd density data** to suggest less crowded pathways.
2. Consider **real-time data** to update paths that are blocked or under construction.

📌 Cycling Mode:

1. Show **bike lanes** and **bike parking stations**.
2. Avoid pedestrian-heavy areas to ensure smooth travel.

📌 Driving Mode:

1. Use **parking availability** data to guide users to available spaces.
2. Integrate **traffic data** to recommend the quickest route, avoiding congested roads.

```
# Example of integrating route suggestions based on mode
def get_route(mode, start, end):
    if mode == 'walking':
        # Get walking-friendly path, avoid crowded areas
        return f"Optimal walking route from {start} to {end} via less crowded paths."
    elif mode == 'cycling':
        # Get bike lanes, avoid pedestrian zones
        return f"Optimal cycling route from {start} to {end}, bike lanes available."
    elif mode == 'driving':
        # Show driving route with available parking
        return f"Drive from {start} to {end}. Parking available at nearest lot."

# Example use case
print(get_route("walking", "Library", "Cafeteria"))
```

◆ 4. Parking Availability Feature

- Integrate **real-time parking data** to show available spots.

- Use sensors or crowd data to suggest parking lots that have fewer cars and are closer to the destination.

```
# Example: Check parking availability
def check_parking_availability(location):
    parking_spots = {"Lot A": 10, "Lot B": 5, "Lot C": 0} # Mock data
    return f"Available parking spots at {location}: {parking_spots.get(location, 'Not Available')}"

# Example use case
print(check_parking_availability("Lot A"))
```

◆ 5. Multi-Mode Navigation User Interface

📌 App Features:

1. **Mode Selection:** Allow users to select walking, cycling, or driving mode in the app.
2. **Crowd Status:** Show crowd density in different routes and suggest the least crowded paths.
3. **Parking Information:** Show availability of parking near the destination and suggest lots with fewer cars.
4. **Real-Time Updates:** Continuously update users with new traffic, parking, and crowd data.

```
# Function to choose mode of transport
def select_mode(mode):
    return f"Selected transport mode: {mode}"





# Example use case
print(select_mode("cycling"))
```

Interactive Campus Dashboard for Students





Feature Overview:

The **Interactive Campus Dashboard** is designed to give students a personalized experience for managing and tracking crowd conditions on campus. Students can customize their dashboard preferences, select frequently visited areas, and receive real-time crowd alerts. This feature aims to empower students by offering them control over their campus navigation and alerting them about crowd conditions in advance.

Why Is This Useful?

- ✓ **Personalized Experience**  — Tailor alerts and settings to each student's preferences.
 - ✓ **Crowd Awareness**  — Help students avoid crowded areas and plan their visits accordingly.
 - ✓ **Real-Time Updates**  — Access live data on crowd density, events, and popular campus areas.
 - ✓ **Enhanced Productivity**  — Optimize time spent on campus by avoiding congested zones.
-

Key Features:

1. **Customization Options**  : Allow students to choose the types of alerts they want, like low/high crowd density or real-time updates for specific areas.
 2. **Crowd Heatmaps**  : Visualize the crowd density in various campus areas, helping students decide where to go.
 3. **Real-Time Crowd Data**  : Display live data from thermal sensors, cameras, and other sources showing the current crowd status across the campus.
 4. **Frequent Area Selection**  : Let students mark and save their most-visited areas (e.g., library, cafeteria, gym) for quick access and alerts on those spots.
-

How to Implement It?

1. Dashboard Interface

The main feature of this implementation is the **interactive dashboard**, where students can:

- View real-time crowd data.
- Set crowd alerts.
- Access heatmaps of crowd density.

App Interface Features:

- **User Profile:** Allow students to log in and save preferences like frequently visited areas.
- **Crowd Alerts:** Enable students to select thresholds for crowd density alerts (e.g., "Alert me when crowd density exceeds 50%").
- **Heatmap Display:** Provide a visual representation of the campus with areas color-coded based on crowd density.

```
# Example of dashboard structure
class Dashboard:
    def __init__(self, student_name):
        self.student_name = student_name
        self.fav_areas = []
        self.alert_preferences = {}

    def add_fav_area(self, area):
        self.fav_areas.append(area)
        return f"Area {area} added to your favorites."

    def set_alert(self, area, threshold):
        self.alert_preferences[area] = threshold
        return f"Alert for {area} set to {threshold}% crowd density."

    def show_dashboard(self):
        return f"Dashboard for {self.student_name} - Favorite Areas: {self.fav_areas}, Alerts: {self.alert_preferences}"

# Example use case
```



```
student_dashboard = Dashboard("John")
print(student_dashboard.add_fav_area("Library"))
print(student_dashboard.set_alert("Cafeteria", 60))
print(student_dashboard.show_dashboard())
```

◆ 2. Crowd Heatmap

The **crowd heatmap** visualizes real-time crowd data and displays which areas of campus are crowded. This can be generated using crowd data collected from thermal sensors, cameras, or app check-ins.

- **Heatmap Logic:** Map out the campus with areas color-coded based on real-time crowd data.
 - Green = Low density (few people)
 - Yellow = Moderate density
 - Red = High density (crowded)

```
# Example of heatmap color-coding based on crowd data
```

```
def get_crowd_color(density_percentage):
```

```
    if density_percentage < 30:
```

```
        return "Green"
```

```
    elif 30 <= density_percentage < 70:
```

```
        return "Yellow"
```

```
    else:
```

```
        return "Red"
```

```
# Example: Display heatmap for an area
```

```
area = "Cafeteria"
```

```
crowd_density = 75 # Example data (percentage)
```

```
print(f"The crowd density at {area} is {crowd_density}%. The color for this area is {get_crowd_color(crowd_density)}%")
```

◆ 3. Real-Time Data Integration

Integrate real-time crowd data sourced from various sensors (e.g., thermal sensors, cameras) to provide up-to-date crowd information on the dashboard.

- **Crowd Density Calculation:** Use sensor data to calculate crowd density for specific areas.
- **Data Fetching:** Fetch real-time data at regular intervals and update the dashboard with the latest crowd density and heatmap information.

```
# Example: Real-time crowd density update
import random

def update_crowd_density(area):
    # Simulating real-time crowd data (percentage of crowd in an area)
    density = random.randint(0, 100)
    return f"Current crowd density at {area}: {density}%"

# Example use case
print(update_crowd_density("Library"))
```

◆ 4. Notifications and Alerts

- Students can **set alerts** based on crowd density thresholds. For example, they could get a notification if the crowd density in a specific area exceeds their preferred limit.
- **Push Notifications** can be used to inform students when their selected areas reach the crowd density threshold they've set.

```
# Example: Sending notifications based on alert preference
def send_crowd_alert(area, density, alert_threshold):
    if density > alert_threshold:
        return f"ALERT: Crowd density at {area} is {density}%, which exceeds yo
ur set threshold of {alert_threshold}%."
```

```

else:
    return f"Crowd density at {area} is {density}%, within your acceptable range."

# Example use case
print(send_crowd_alert("Cafeteria", 80, 60))

```

◆ 5. Dashboard Personalization

Students should be able to:

- **Customize the dashboard:** Select and save areas they visit most often.
- **Set crowd preferences:** Choose when and how often they want to be alerted about crowd conditions.
- **View historical data:** Track crowd conditions over time to understand trends and adjust preferences.

```

# Example: Personalize the dashboard for a user
def personalize_dashboard(student_name, fav_area, alert_pref):
    return f"{student_name}'s Dashboard - Favorite Area: {fav_area}, Alert Preference: {alert_pref}"

# Example use case
print(personalize_dashboard("Alice", "Gym", "Alert me when crowd density exceeds 50%.")

```





Smart Campus Infrastructure for Crowd Management

Feature Overview:





The **Smart Campus Infrastructure** feature integrates real-time crowd data with campus infrastructure like **elevators**, **staircases**, and **hallways** to optimize crowd management. By dynamically adjusting access to these facilities, this feature

ensures smoother flow of foot traffic, minimizes congestion, and improves overall campus experience for students and staff.

Why Is This Useful?

- ✓ **Reduced Congestion**  — Avoid bottlenecks and crowded hallways by managing access.
 - ✓ **Improved Safety**  — Direct traffic away from overcrowded areas to prevent accidents.
 - ✓ **Efficient Resource Use**  — Optimize the use of elevators and staircases based on crowd density.
 - ✓ **Enhanced Campus Experience**  — Provide a seamless and hassle-free campus mobility system.
-

Key Features:

1. **Dynamic Elevator Control**  : In high-density areas, elevators can be restricted, rerouted, or directed to alternate floors based on crowd data.
 2. **Smart Staircase Management**  : Direct students to staircases when elevators are overloaded, and vice versa, based on crowd conditions.
 3. **Real-Time Foot Traffic Monitoring**  : Use crowd density data to monitor hallways and ensure smooth flow of foot traffic.
 4. **Access Control in Crowded Zones**  : In high-traffic areas, limit or stagger access to elevators or staircases to avoid overcrowding.
-

How to Implement It?

◆ 1. Real-Time Crowd Data Integration

- Use sensors, cameras, and app check-ins to track crowd density in different areas of the campus.
- Implement algorithms to determine when crowd levels in certain areas exceed a comfortable threshold, and adjust access to elevators, staircases, or hallways accordingly.

Example Implementation:

For example, if a building has a high crowd density, **elevators** can be rerouted to less crowded floors, while **staircases** can be used more effectively by directing students to them through push notifications or signs.

```
# Example: Control elevator access based on crowd data
def adjust_elevator_access(crowd_density, max_capacity=70):
    if crowd_density > max_capacity:
        return "Elevators rerouted to less crowded floors."
    else:
        return "Elevators operating normally."

# Example use case
print(adjust_elevator_access(80)) # In case of high density
```

◆ 2. Smart Staircase Management

- When **elevators** are at capacity or in high-density zones, encourage the use of **stairs** by dynamically displaying directional signage or sending notifications to guide students.

```
# Example: Direct students to stairs when elevators are crowded
def manage_foot_traffic(area, crowd_density, stair_access=True):
    if crowd_density > 75:
        return f"High crowd density detected at {area}. Please use stairs for faster movement."
    elif stair_access:
        return f"Elevators are operational, but stairs are available for faster movement at {area}."
    else:
        return "Stair access temporarily unavailable."

# Example use case
print(manage_foot_traffic("Library", 80)) # High density suggests using stairs
```

◆ 3. Foot Traffic Monitoring and Hallway Optimization

- Use crowd data to monitor and direct foot traffic. For example, display real-time hallway traffic updates or guide students to alternate paths to avoid congested areas.

```
# Example: Monitor hallway foot traffic and recommend optimal paths
def hallway_traffic_monitor(area, crowd_density):
    if crowd_density > 70:
        return f"Traffic is heavy in {area}. Consider using alternate hallways."
    else:
        return f"{area} is clear. Proceed as normal."

# Example use case
print(hallway_traffic_monitor("Main Corridor", 80)) # Suggest alternate paths
when crowded
```

◆ 4. Access Control and Staggered Access

- **Elevator control:** In high-density scenarios, prevent multiple users from accessing the same elevator. Stagger access to reduce wait times and prevent overcrowding.
- **Staircase management:** Encourage students to use stairs during peak times, especially when elevators are unavailable.

```
# Example: Stagger access to elevators based on crowd density
def stagger_elevator_access(crowd_density, max_density=75):
    if crowd_density > max_density:
        return "Elevator access is temporarily restricted. Please use stairs or wait
for the next elevator."
    else:
        return "Elevator access is available."

# Example use case
print(stagger_elevator_access(80)) # High density triggers restrictions
```

◆ 5. User Interface for Access Control and Alerts

- **Real-Time Notifications:** Send notifications to students informing them of congestion in certain areas and recommending alternative paths (stairs or elevators).
- **Signage Integration:** Digital signage around the campus can show the availability of elevators, suggest using stairs, or guide students to less crowded areas.

```
# Example: Notify users of congestion and recommend alternative routes
def send_congestion_alert(area, crowd_density):
    if crowd_density > 80:
        return f"ALERT: High crowd density detected at {area}. Please consider using stairs or alternate routes."
    else:
        return f"Traffic at {area} is normal."

# Example use case
print(send_congestion_alert("Main Hall", 85)) # Alerts user when congestion is high
```