# Research: Measuring the latency and jitter advantages of edge computing over cloud infrastructure

## 1. Introduction and motivation

Edge computing promises to bring computation closer to end users, reducing latency, improving reliability, and enabling real-time applications such as IoT control, video analytics, and AI inference.

However, the actual, measurable benefit of deploying services at the edge compared to traditional cloud hosting depends on network distance, workload type, and local conditions.

This project will conduct a controlled experiment to quantify the latency, jitter, and stability improvements achievable when moving an identical microservice from a remote cloud platform to a nearby edge node. The goal is to produce a reproducible baseline for evaluating when and why edge computing provides practical performance advantages.

## 2. Research objectives

1. Quantify latency and jitter differences between edge and cloud deployments of the same service.
2. Identify the conditions (payload size, computation cost, network quality) where edge computing offers a clear advantage.
3. Provide empirical evidence supporting or refuting common claims about the benefits of edge proximity.
4. Deliver a reproducible experiment framework suitable for future extensions (e.g., adaptive offloading or on-device AI).

## 3. Research questions

* How much does proximity (edge) reduce average and tail (p99) latency compared to a remote cloud server?
* Does the edge provide more stable (lower-jitter) response times under variable network conditions?
* For which types of workloads (small vs. large payloads, light vs. heavy computation) does the edge advantage persist?
* At what point do computation or data transfer costs outweigh latency savings?

## 4. Methodology and setup

### 4.1 Overview

A simple REST API will be deployed in two identical containers:

one on a local edge node (Raspberry Pi / Intel NUC) and one on a remote cloud platform (e.g., AWS, Render, or Railway).

A client laptop will act as the test generator, sending HTTP requests to both servers and recording detailed timing statistics.

### 4.2 Architecture diagram

A diagram of a computer server

AI-generated content may be incorrect.

### 4.3 Workload

Each request carries:

* A configurable payload size (0 KB – 1 MB).
* A compute delay parameter (cpu\_ms) to simulate processing effort.
* Both servers perform the same function.

### 4.4 Measurements

For each test condition, the script records:

* p50 / p90 / p99 latency (ms)
* Jitter (p99 − p50)
* Throughput (requests/s)
* Error rate

Network scenarios will be emulated using tc netem to represent:

* Good Wi-Fi (10 ms ± 2 ms)
* Congested network (40 ms ± 10 ms + 1 % loss)
* Mobile hotspot (80 ms ± 20 ms + 2 % loss)

All results will be exported as CSV and visualized in Python (pandas + matplotlib).

## 5. Expected outcomes

* Quantitative comparison showing how edge proximity affects average and tail latency.
* Identification of thresholds where edge computing provides measurable benefits (e.g., payload < 512 KB).
* Demonstration of stability improvements (lower jitter) for local edge nodes.
* A decision matrix for developers: when to deploy services at the edge vs. in the cloud.

## 6. Deliverables

1. Experiment scripts and Docker files (Git-versioned).
2. Raw and processed datasets (results/summary.csv).
3. Visual report with latency and jitter graphs.
4. Written report section discussing relevance, standards (ETSI MEC), open-source tools, limitations, and future work (e.g., adaptive offloading).

## 7. Resources required

* Hardware: 1 laptop (client), 1 Raspberry Pi.
* Software: Python 3.11, Flask, Docker / Docker Compose, hey, tc netem.
* Cloud: Free-tier deployment on Render / Railway / AWS EC2 t2.micro.

# Experiment: Design and implementation of a lightweight edge computing system with local processing and cloud visualization

## 1. Introduction and motivation

Edge computing enables data to be processed close to where it is generated, improving responsiveness, reducing network load, and ensuring operation even under intermittent connectivity.

This project aims to demonstrate a complete edge-to-cloud workflow by building a functional prototype where a Raspberry Pi performs data collection and local computation before synchronizing summarized results with a cloud-based visualization system.

Instead of continuously streaming raw data, the Pi will process information locally and transmit only essential summaries at periodic intervals.

This design emulates real-world IoT and industrial systems that balance local autonomy with centralized insight.

## 2. Objectives

1. Develop a working edge system prototype using a Raspberry Pi as the edge node.
2. Implement local data collection and processing on the Pi, with periodic result transmission to a cloud server.
3. Ensure reliable data synchronization and monitoring from the cloud side.
4. Visualize received data and system status through an interactive Streamlit web dashboard.
5. Evaluate the system’s performance using defined operational metrics.

## 3. System architecture

## A screenshot of a computer program AI-generated content may be incorrect.

## 4. Implementation plan

### Edge (Raspberry Pi)

* Collects data via sensors or camera (still to decide).
* Executes lightweight local processing (aggregation, feature extraction, or anomaly detection).
* Stores results in a local buffer when offline.
* Periodically transmits processed data to the cloud server through a REST API or MQTT.

### Cloud

* Receives and stores processed data in a Supabase database (PostgreSQL).
* Implements a simple verification service to ensure timely updates.
* Provides an endpoint for the Streamlit dashboard to access data.

### Dashboard (Python: Streamlit)

* Displays live and historical data in interactive charts.
* Visualizes trends, anomalies, and connectivity status.
* Issues alerts if updates are missing for longer than the expected interval.

## 5. Metrics to measure

Metrics that will be measured and shown in the dashboard next to the results of the sensor/camera.

| **Metric** | **Description** | **Purpose** |
| --- | --- | --- |
| **Local processing time** | Duration of data capture and edge-side computation per cycle. | Assess responsiveness of edge node. |
| **Network usage reduction** | Difference between raw data volume and processed/aggregated size transmitted to cloud. | Quantify efficiency of local preprocessing. |
| **Data delivery reliability** | Frequency of successful data transfers and missed uploads. | Evaluate communication robustness. |
| **Cloud data completeness** | Verify that the cloud receives at least one update per 10-minute window. | Check consistency of edge–cloud link. |
| **Offline resilience** | Ability of system to store and forward data during temporary connectivity loss. | Demonstrate fault tolerance. |
| **Dashboard latency** | Delay between data creation on edge and visualization in dashboard. | Assess end-to-end freshness of data. |

## 6. Expected deliverables

1. Working edge prototype running on a Raspberry Pi.
2. Cloud receiver service and database backend for data ingestion and storage.
3. Streamlit web dashboard displaying real-time and historical data trends.
4. Performance metrics summary based on recorded logs and system statistics.
5. Comprehensive report and presentation documenting architecture, results, and observations.

## 7. Resources required

| **Category** | **Item** |
| --- | --- |
| **Hardware** | 1 × Raspberry Pi (any recent model), optional sensors or camera module |
| **Software** | Python 3, Flask/FastAPI (for cloud API), Streamlit (for visualization), SQLite/PostgreSQL |
| **Networking** | Wi-Fi connection between Pi and cloud server |
| **Data** | Real or simulated sensor readings (temperature, video frames, etc.) |