GhostMap: Real-Time Edge-Deployable UAV Geo-Projection Framework for Softwarized IoT Service Networks

Palla Jaswanthroyal
Computer Science and Engineering,

Indian Institute of Technology Tirupati

ViSAL lab, Indian Institute of Technology Tirupati

Tirupati, India
cs23b034@iittp.ac.in

Prof. Rama Krishna Sai Gorthi
Electrical Engineering,

Indian Institute of Technology Tirupati

ViSAL lab, Indian Institute of Technology Tirupati

Tirupati ,India
rkg@iittp.ac.in

***Abstract*— We present GhostMap, a lightweight, modular framework enabling real-time geo-projection of UAV detection outputs for immediate on-device visualization and integration into softwarized IoT services. Designed for edge deployment on resource-constrained platforms such as the Raspberry Pi 4B, GhostMap achieves a sustained throughput of 45 FPS and maintains an end-to-end latency of 22 ms per frame. Central to the system is a latency-optimized C++ geo-projection engine interfaced via PyBind11, which maps detected objects to GPS coordinates using a calibrated camera model. To ensure reliable operation under bursty network conditions, a dual-layer TCP protocol decouples processing and rendering streams, while a freshness-prioritized rendering logic isolates the most recent data for visualization, mitigating the effects of JSON payload concatenation and parse failures. Experimental results demonstrate GhostMap's ability to sustain performance and robustness where cloud offloading is infeasible, making it highly suitable for autonomous UAV operations and real-time IoT networks in latency-critical missions.**

***Code and documentation are available at:* https://github.com/JaswanthPalla149/GhostMap**

Keywords—UAV processing, edge computing, geo-projection, IoT services, real-time systems, softwarized networks, autonomous systems

I. Introduction

The proliferation of Unmanned Aerial Vehicles (UAVs) in IoT-driven domains-such as surveillance, agriculture, disaster response, and military reconnaissance-has led to an exponential increase in video and object detection data. Traditionally, this data is offloaded to cloud or fog servers for processing, visualization, and analytics. However, these architecture introduce critical limitations: high latency, network dependency, and bandwidth congestion, all of which severely impact mission-critical, real-time decision-making capabilities.

The demand for real-time, on-device analytics and visualization has thus emerged as a key requirement for the next-generation UAV systems. Unfortunately, most existing approaches either assume the availability of reliable edge/fog infrastructure or perform only lightweight processing on the UAV, offloading major computation externally.

To bridge this gap, we propose GhostMap a novel, edge deployable, lightweight framework designed for real-time object detection rendering and geo-projection directly on constrained devices like the Raspberry Pi 4B. The system achieves sub~22ms end-to-end latency and maintains ~45 FPS .

Key innovations introduced by GhostMap include:

- A Dual-layer TCP communication protocol that splits data ingestion and rendering, enabling independent control over each stage and tolerance to network jitter or lag.
- A lag-tolerant rendering engine that always displays the latest valid data, ensuring freshness without crashing UI pipeline.
- A modular and plug-and-play architecture that supports easy integration with different detection models via simple class mapping files, requiring no reconfiguration.
- A cross-platform backend optimized for low-latency GPS projection and on-device computation.

II. Related Work

A. Edge-Based UAV Video Analytics Systems

Hybrid edge-UAV systems perform onboard pre-processing and offload heavier analytics to nearby edge servers [1–3]. While effective for reducing bandwidth and latency, they rely on stable connectivity, limiting use in remote settings. **GhostMap** combines local autonomy with edge-augmented execution, delivering low-latency analytics even in disconnected environments.

B. UAV-Enabled Fog Computing Architectures

Fog architectures distribute computation across UAVs, mobile fog nodes, and gateways [4–6], improving scalability and responsiveness. However, they depend on dynamic coordination and stable links, which are fragile in mobile deployments. **GhostMap** operates as both fog consumer and provider, enabling task sharing across fleets while retaining full offline capability.

C. Cloud -Based Geospatial UAV Systems.

Traditional Cloud GIS platforms support spatial analytics like geo-projection and mapping [7–9], but require consistent uplinks, making them unsuitable for real-time field use. **GhostMap** embeds the entire geospatial stack onboard, enabling real-time, offline mapping with optional map sync when available.

D. Gap

While existing UAV systems focus on edge processing, fog offloading, or GIS integration in isolation, few offer a unified architecture adaptable across varying network and compute conditions. **GhostMap** presents a modular framework that combines real-time video analytics, on-device geospatial mapping, and fog-level collaboration.

III. Ease of use and modularity

GhostMap is designed for rapid deployment and extensibility, enabling users to integrate the system into existing UAV pipelines with minimal effort. Key aspects include:

- Plug-and-play Model Support: Any YOLO-style detection model can be integrated by updating the classmap.txt file, which maps class IDs to labels and colors. No code changes required.
- Cross-platform Compatibility: The system supports Linux, windows, and ARM-based platforms like Raspberry-pi, making it suitable for both development and deployment environments.
- Minimal Configuration: Communication ports (9999 for processing and 12345 for rendering) are preconfigured and modular. Users can change them through config files or command-line arguments.
- Clear Separation of Components: The processor (Python + C++) and renderer (Qt/Qml via C++) are decoupled, making debugging, upgrading, and benchmarking easier.
- Lag-Tolerant Protocol: The dual-port architecture buffers lagged packets and renders only the latest valid frame, preventing UI crashes and reduced overhead.

IV. Proposed Approach

A. System Architecture

The proposed framework , GhostMap, is a modular, edge-deployable system designed for real-time UAV analytics nd geo-projection. It operates entirely on-device(e.g., Raspberry Pi 4B), eliminating reliance on cloud.

The architecture comprises of the following key modules:

1. Main.py(Processor): Parses composite messages (JPEG + JSON), separates image and detection info, and performs projection of detections to geo-coordinates using a C++ backend exposed via Pybind11. Sends processed GPS data to the renderer via another TCP socket(port 12345).
2. PDTG.cpp (Projection Engine): Implements the core geo-projection logic in C++, offering both accuracy and low latency. The module leverages a pinhole camera model and 3D-to-2D transformation, using UAV parameters (altitude, pitch, yaw, field of view, and camera intrinsics).
    - Semantic Labeling: Integrates runtime class and color mapping by loading a configuration file (classmap.txt), ensuring flexibility with various detection models.
    - Efficient Transformation: Converts normalized bounding boxes into pixel coordinates and subsequently to GPS points, factoring in UAV pose and flat-Earth geometry.

- Python Integration: Exposed as a Python module through PyBind11, enabling seamless invocation from the data processing pipeline.

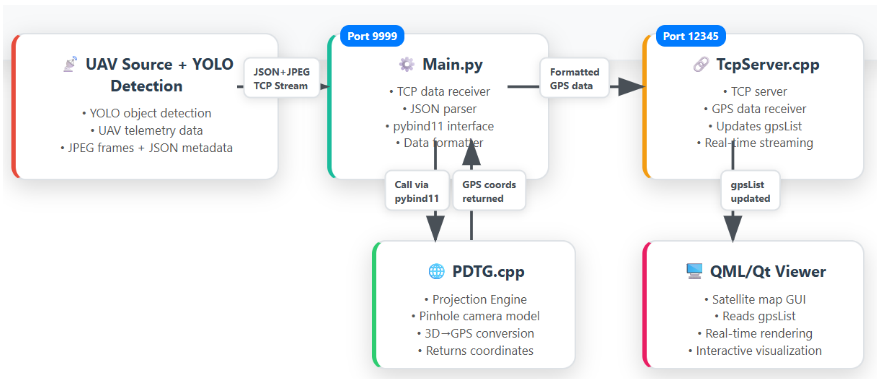
3. Qml/Qt Viewer (Renderer): A TCP Server listening on port 12345. It receives projected GPS data in JSON format, parses it and updates a QML-based frontend to display live geo-coordinates. Includes a lag-tolerant mechanism that handles malformed or merged packets by rendering only the latest valid frame, ensuring real-time freshness.

B. *Processing Flow*

1. Data Reception: Main.py receives JPEG + JSON metadata over TCP on port 9999. This could originate from any real-time UAV detection source or embedded vision pipeline.

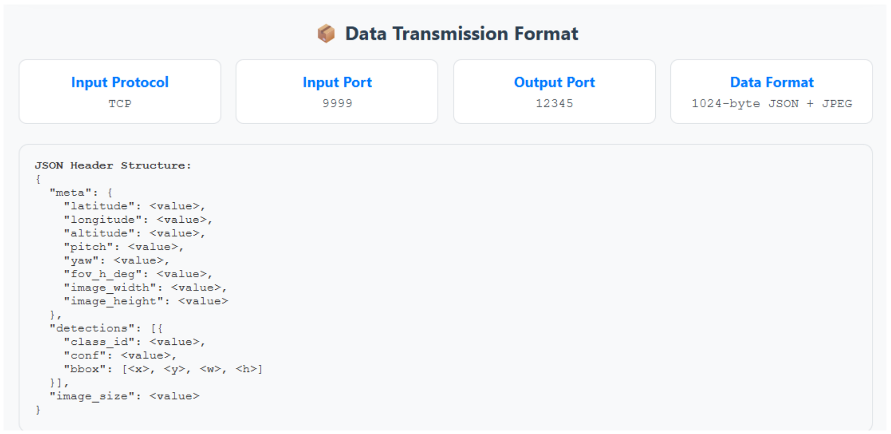
2. GPS Projection: The JSON metadata is parsed, and object detection bounding boxes are passed to the C++ projection engine (projector.so, formerly PDTG.cpp) using Pybind11. This engine transforms 2D image-space detections into real-world GPS coordinates using a **pinhole camera model**, assuming fixed-altitude, flat-Earth approximation, and constant intrinsic parameters. transmission to Viewer: The resulting GPS coordinates are serialized into JSON and streamed via TCP to TcpServer.cpp on port 12345.

   The GPS computation method is adapted from standard monocular projection models used in UAV vision tasks [10].



3. The renderer (TcpServer.cpp/QML) monitors incoming streams for well-formed JSON arrays. If multiple frames or malformed data are received, only the latest valid GPS data is used for display, preserving real-time responsiveness and robustness.

Fig.1- Modular Architecture diagram of the software


Data Transmission Format

Input Protocol	Input Port	Output Port	Data Format
TCP	9999	12345	1024-byte JSON + JPEG

```
JSON Header Structure:
{
  "meta": {
    "latitude": <value>,
    "longitude": <value>,
    "altitude": <value>,
    "pitch": <value>,
    "yaw": <value>,
    "fov_h_deg": <value>,
    "image_width": <value>,
    "image_height": <value>
  },
  "detections": [{
    "class_id": <value>,
    "conf": <value>,
    "bbox": [<x>, <y>, <w>, <h>]
  }],
  "image_size": <value>
}
```

Fig.2- Expected Data Input Format, works even if you don't send JPEG image.

V. RESULTS AND EVALUATION

A. *System setup and Test Configuration*

To validate the responsiveness and portability of GhostMap, we evaluated the framework on a Raspberry Pi 4B (8GB RAM, quad-core Cortex-A72). While this provides a low-power test environment, GhostMap is designed to be a platform-independent and performs well on x86 or Jetson-based edge devices too.

The test environment simulates a real-time UAV scenario where the detection data is streamed continuously at desired frequencies . The primary goal was to evaluate the system's responsiveness, robustness, and throughput behaviour under high-frequency input.

B. End-to-End Pipeline Latency and Processing Behavior

To measure responsiveness, we streamed input data at different frame rates and analysed how GhostMap processed and rendered each frame data.

We tested the system with varying frame rates from the simulator , analysing the behaviour across ports: the processor (9999) and the renderer (12345).

Processor Throughput Limits (Port 9999)

Input Frame Gap	Processor Behaviour	TCP Buffer Status	Outcome
(2-10) ms	Accept packets into Buffer with gap of 10ms	Fills up fast	Will overflow in short time
(~10) ms	Accepts packets with lag	Approaches Full	Potential for overflow
>= 20 ms	Stable processing	Clean	Frames are processed with no loss.

Conclusion:

- The processor gracefully handles even high-speed input using TCP buffering.
- But real-time responsiveness is guaranteed only when input frame gap >= 10 ms.
- Processor bottleneck is effectively ~100 FPS.

Renderer Throughput Limits (Port 12345)

Input Frame Gap	Renderer Behaviour	Packet Loss %	Notes
~10 ms	Packets come too close, many of the packets get merged	~50%	With the filtering at TcpServer.cpp only the latest data will be uploaded to renderer if they come in merged form, thereby missing some of the buffer.
10-20 ms	Partial Success	~20%	With Filtering enabled the packets loss can be less.
>= 22 ms	Full rendering	~0%	One packet per cycle, no merges, no losses.

Conclusion:

- Since the data to Renderer has to come from Processor , it can only be possible with time gap >10 ms.
- Renderer is the true bottleneck.
- It requires at least 22ms gap between packets for stable rendering.
- Anything faster leads to parsing errors due to JSON merge in TCP stream.
- Filtering logic in TcpServer.cpp allows robustness by handling parsing errors and showing only the latest valid data.

C. Packet Loss and TCP Behaviour Summary

Component	Bottleneck Threshold	Crash Risk(High-Speed Input)	Recovery Mechanism
Processor	~10 ms	High Load = buffer lag	TCP Buffer absorbs burst.
Renderer	~22ms	Parse failure at high input	JSON filtering for last packet.

Design Insight:

- We enforce a minimum 22 ms frame gap at the source to match the renderer's capabilities.
- This aligns processing, transfer, and rendering into a smooth 45 FPS pipeline.

D. *Final Throughput and Behaviour Table*

Stage	Frame Gap Needed	Behaviour	Filtering Required
Input to Processor	>= 10 ms	Processed without delay	No
Processor to Renderer	>= 22 ms	Rendered Completely	No
Input < 10 ms	-	Processor lags & merges packets	TCP buffered & Filtering at Port: 12345 for not losing more packets
Input < 22 ms	-	Renderer skips/merges packets	Filtering at Port: 12345 for not losing more packets

E. *Protocol Robustness Insight:*

One of the core novelties in GhostMap is the drop-tolerant, freshness-aware protocol implemented via custom filtering logic. When two JSON messages arrive merged in the same TCP chunk (e.g., [..O1..]\n[..O2..]\n), the renderer previously crashed.

With the update:

- We discard stale data, retaining only the latest valid GPS message.
- This ensures no crashes and reduced packet losses due to merging and visual freshness even in congested networks.

F. *Summary of Observations*

Metric	Value/Insight
Effective System Latency	~22 ms per frame(end-to-end)
Sustained FPS (stable)	~ 45 FPS rendering
Processor bottleneck	At ~10 ms per frame input
Renderer bottleneck	At ~ 22 md per frame input
TCP crash-resilience	Robust, handled via buffering
Result freshness	Always show latest object detection frame
Protocol behavior	Lag-tolerant, model-agnostic, modular

Test Conclusion:

GhostMap achieves near real-time, crash-resilient rendering of object detections streamed over TCP from UAVs or edge processors.

By introducing minimal Delays (~22ms), enabling freshness-filtered rendering, and maintaining processor throughput above real-time, GhostMap proves effective for edge scenarios like surveillance, tactical intelligence, and low-bandwidth monitoring. Unlike standard TCP viewers or cloud based visualizers, GhostMap ensures that every frame shown is timely, accurate, and rendered without visual artifacts or latency buildup.

G. *Output: The backgound map can be set to an image of a satellite ".tiff" file. You will be asked to set the GPS coordinates of base; the map will cover a 2000m radius around it. The test data we have used is data from [Roboflow]*

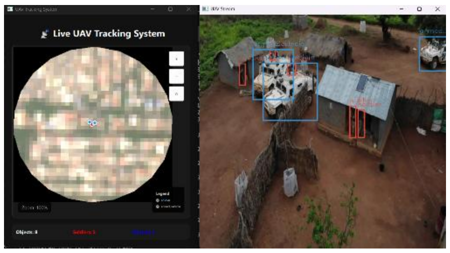


Fig.3 – Output of GhostMap with show enabled and background map selected.

VI. PORTABILITY AND FLEXIBILITY

GhostMap is engineered for seamless deployment across a variety of edge devices and detection pipelines, emphasizing modularity and runtime configurability.

A. *Runtime Configurability via Launch Flags:*

GhostMap's launch script (./run.sh) provides fine-grained control over input and visualization modes, allowing users to switch between lightweight and full-feature configurations without altering source code.

Command Variant	Description
./run.sh nc-y show	No-confidence YOLO input with video stream rendering
./run.sh nc-y no-show	No-confidence YOLO input, video rendering disabled
./run.sh c-o show	Confidence-based detection with video rendering
./run.sh c-o no-show	Confidence-based detection, no video (minimal mode)

B. *Model-Agnostic Operation with Dynamic Class Mapping*:
GhostMap decouples label and color mapping from core logic. The backend dynamically loads class metadata from an external classmap.txt, supporting plug-and-play integration of new models:

```
        // projector.so or projector.pyd
bool loadClassMap(const string& filepath);
```

Example classmap.txt:

```
soldier       #e74c3c
civilian      #f1c40f
armed_vehicle   #3498db
```

VII. Limitations and Applicability Scope

Although GhostMap is designed for real-time edge scenarios, it has two practical constraints:

1. **Geographic Projection Range:**
Due to the projection algorithm's planar approximation, accurate rendering is guaranteed only within a **2 km radius** of the GPS anchor point. Beyond this, the lat-lon error grows non-linearly and should be corrected with a spherical projection model.
2. **Altitude Restriction:**
The model assumes a UAV altitude under **120 meters**, aligning with civil aviation and drone laws. Higher altitudes may require deeper reprojection correction or LIDAR-based terrain estimation.

Acknowledgment

The authors would like to express their sincere gratitude to **Sk. Danish Dada, Department of Computer Science and Engineering, Indian Institute of Technology Tirupati**, for his invaluable support in drafting this paper and for his meticulous assistance in recording and validating experimental results. His insights and collaboration were instrumental in shaping the direction and clarity of this work.

The authors also gratefully acknowledge **K. Janith Siddartha Roy**, **Department of Mechanical Engineering, Indian Institute of Techology Tirupati**, for his technical input and continued encouragement throughout the development and testing phases of the system.

References

[1] Hu et al. (2018). *"Bandwidth-Efficient Live Video Analytics for Drones via Edge Computing."* IEEE Transactions on Network and Service Management.
[2] Zhang et al. (2024). *"Privacy-Preserving Live Video Analytics for Drones via Edge Computing."* Applied Sciences.
[3] Chen et al. (2023). *"Democratizing Drone Autonomy via Edge Computing."* ACM/IEEE Symposium on Edge Computing.
[4] Liu et al. (2020). *"Multi-task Offloading Scheme for UAV-Enabled Fog Computing Networks."* EURASIP Journal on Wireless Communications and Networking.
[5] Gupta et al. (2024). *"Unmanned Aerial Vehicles in Collaboration with Fog Computing Networks for Improving Quality of Service."* International Journal of Communication Systems.
[6] Wang et al. (2023). *"Dynamic Offloading in Flying Fog Computing: Optimizing IoT Network Performance with Mobile Drones."* Drones.
[7] Rodriguez et al. (2023). *"Edge Computing in IoT Ecosystems for UAV-Enabled Early Fire Detection."* Sensors.
[8] Kumar et al. (2024). *"UAV-Enabled Mobile Edge-Computing for IoT Based on AI: A Comprehensive Review."* Drones.
[9] Thompson et al. (2024). *"Trajectory-Aware Offloading Decision in UAV-Aided Edge Computing: A Comprehensive Survey."* PMC.
[10] J. Kim, S. Park, and H. Kwon, "Real-time geo-localization using UAV video feed with deep object detection and GPS projection," *IEEE Transactions on Geoscience and Remote Sensing*.

[11] "Tracking Military RCA Dataset." Roboflow Universe, Escola Naval. [Online]. Available: Roboflow