insect detection systems [45], [46]. As described, compared to the algorithm described in [44], the new algorithm used in this study decreased mse from 50.8 to 13.6 and MAE from 5.1 to 2.1, and also increased recall around 34%, indicating a reduction of about $3.7\times$ in mse and $2.5\times$ in MAE. In comparison with YOLO methods, although the accuracy of YOLO methods is better than the proposed one, they are too computationally intensive, too large in size, and unable to run on resource-constrained MCUs.

Moreover, our analysis revealed that the presence of a high level of overlap among the captured insects in the obtained images, which occurs when new insects are trapped physically stuck overlapping the existing ones trapped previously, increases the error of the detection phase. This is due to the fact that the detected blobs are filtered out if their size is much larger than that assigned to the presence of a BMSB (overlapping increases the blob size). Thus, considering previously captured data and detected insects plays a significant role that impacts the system performance. To evaluate the impact of insects overlapping each other, we made some alterations to the dataset by removing the images that exceeded seven days between trap replacement. This time interval was selected based on the frequency of new insects being captured on the trap. This led to a significant reduction in error by decreasing the MAE from 2.07 to 1.05 and mse from 13.55 to 1.81. It also increased the recall by almost 10%. These results prove the impact of envisaging overlapping on the performance and that the recent changes in the algorithmic process are in the right direction to improve the efficiency of the system.

In terms of running time, the added step in the detection phase of the image processing algorithm takes up to $0.5\ \mathrm{s}$ per image. Therefore, the total runtime for a complete system operation, including image capturing (from two sides of the trap), detection (on two captured images), classification, and results, is approximately $17.5\ \mathrm{s}$. It is important to note that the running time varies depending on the number of detected areas identified during the detection phase, as the classification model runs separately for each detected area. The mentioned $17.5\ \mathrm{s}$ correspond to a trap with six insects on each side. Moreover, regarding power consumption, the device consumes up to $320\,\mathrm{mA}$ in operation mode. However, during nonoperation periods, the device enters deep sleep mode to reduce power consumption, consuming only $7\,\mathrm{mA}$.

VI. Client–Server Application

In this section, we describe a client–server architecture designed for bug detection that also facilitates farmer involvement through a feedback loop: subscribed farmers can send their images for inspection. Any RGB trained model can be integrated into this client–server application (briefly, app). In addition, also the UAV can use this app to autonomously fly inside the orchard to collect picture in order to augment the dataset of images, and enhance the trained ML models.

This client–server app is primarily utilized by farmers and autonomous UAVs for collecting RGB images to augment the image dataset, and presently, there is no direct connection with edge-based sticky traps. Nevertheless, it is technically feasible

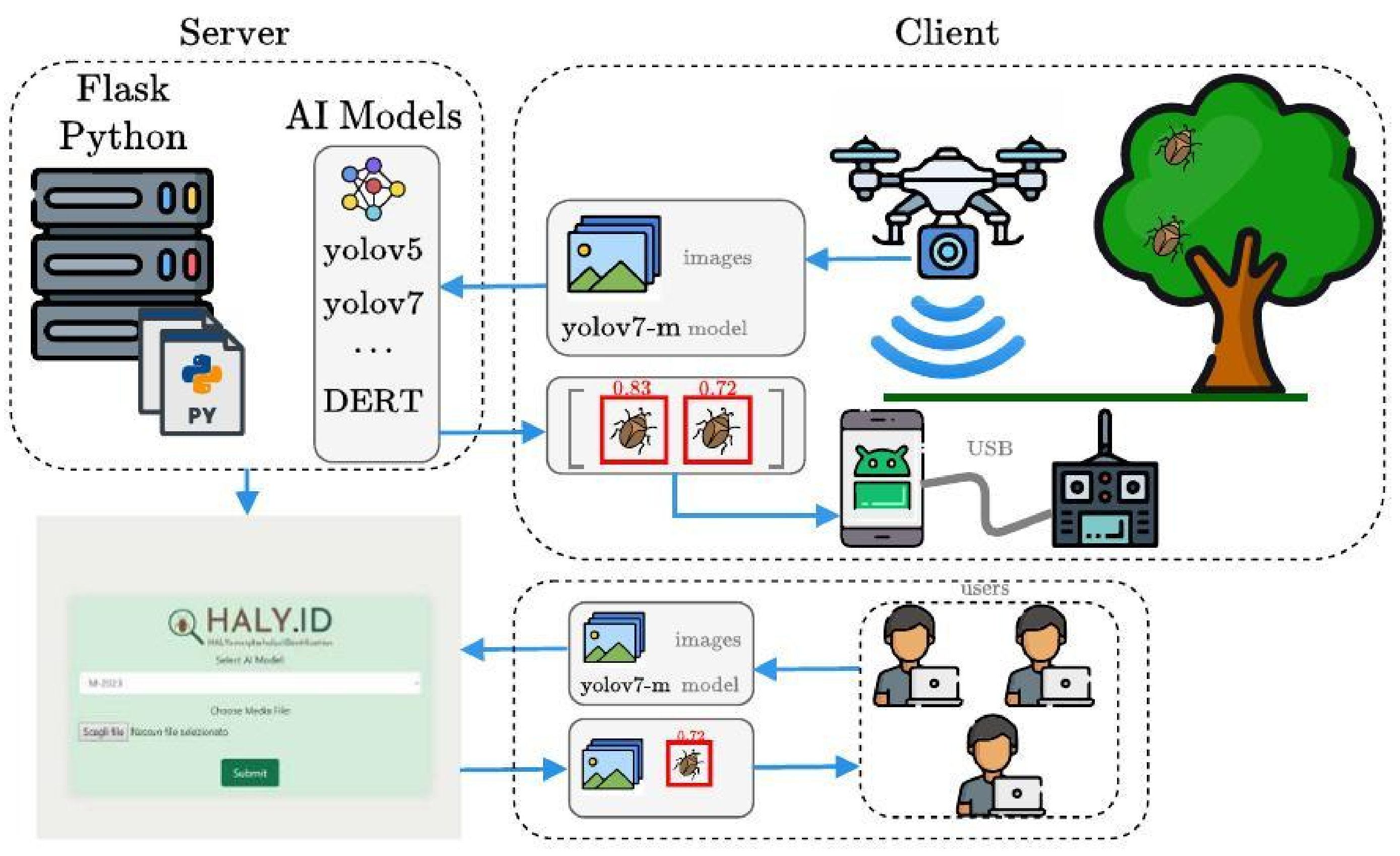

Fig. 5. Application architecture illustration. The server hosts a Python Flask service where AI YOLO models are stored. The models are accessible via exposed APIs, allowing any client device, such as smartphones or drones, to access and utilize them.

to enable the trap to trigger exposed application programming interfaces (APIs) from the server side and this will be considered in future work.

A. Overview of the Architecture

The primary objective of the app is to create an *almost real-time* mechanism capable of detecting and counting BMSB in the field. We use the term "almost" because the detection does not occur in real time; it starts after the clients take pictures. A client can be any device equipped with a camera (such as smartphones, tablets, or drones) and network connectivity. The app consists of two main components: a *server* and a *client*. The server hosts the trained models discussed in Section III and listens to *requests* sent by one or more clients. The concept of request splits into two different manners: API request or GUI request. Upon receiving a request, the client sends the captured images to the server for evaluation using one of the various ML algorithms stored within. Once the server completes processing the request, it responds to the client by transmitting the potential bug detections in the form of bounding box coordinates, along with their associated confidence levels. At the client level, also autonomous UAVs can perform these requests. The architecture of the client–server app is depicted in Fig. 5.

B. Server Side

The server component is responsible for providing BMSB detection to clients that make requests. It remains passive until a client submits a request consisting of a set of captured images and specifies a particular model for querying. The server exposes a Flask service, which is a lightweight web framework written in Python. Flask falls into the category of microframeworks due to its minimalistic requirements and flexibility [47]. We opted for Flask due to its simplicity and lack of dependencies on specific tools or libraries.

Flask listens for incoming requests via HTTP POST requests. This allows clients to include captured images in the body of the request message, which is essential for our app. Clients can trigger a POST request using a specific URL format (`url/model:port`), with the enclosed images in