POLYTECH
Peter the Great
St. Petersburg Polytechnic
University

COURSE WORK REPORT

In Intelligent Systems

Recognition of potholes on a road

Prepared by:

Alexander Syrcev
Alexei Antonov
Dmitrii Panfilov
Ekaterina Shirimetova
Mikhail Rozhdestvenskiy
Nikolai Glebov

Submitted to:

AmirHossein Bahrami

Saint Petersburg
2023

1. Introduction

Road maintenance is a critical aspect of urban management, directly influencing safety and comfort for all traffic participants. One of the common issues that plague drivers, cyclists, and even

pedestrians are potholes. Potholes can cause accidents, damage vehicles, and even impede emergency services. Addressing this issue promptly is essential for a safe and efficient transportation system.

The goal is straightforward and quantifiable: **to develop a multi-agents system capable of detecting potholes in real time with a high degree of accuracy and with a sufficiently high processing speed, marking detected potholes as dangerous or relatively safe by their width in the image**. This system should be capable of being deployed in a vehicle or roadside monitoring equipment to identify and report the location of potholes to a central database.

To achieve this goal, we need to accomplish several tasks:

1. **Collect and curate a dataset** of road images that include various pothole conditions.
2. **Train the YOLOv8 model** to recognize and accurately pinpoint potholes in these images.
3. **Develop a FastAPI backend** capable of receiving image data from the detection system, processing it, and updating the central database in real-time.
4. **Carry out the necessary testing** to ensure that the system is reliable and can work under different lighting and weather conditions.
5. **Show the ability to manage the system via Telegram bot** and serve various clients such as other bots, web applications or simple clients connected to the server through a single API.

To tackle this challenge, we turn to modern technology, harnessing the power of FastAPI and YOLOv8 and using of a multi-agent approach. FastAPI is a web framework for building APIs with Python, renowned for its speed and ease of use. It allows for the quick creation of a server that can handle requests and responses, essential for real-time applications. YOLOv8 (You Only Look Once version 8) is the latest iteration of a state-of-the-art, real-time object detection system, which can identify objects in images or video streams with remarkable accuracy and speed. A multi-agent approach is like having a team where each member is good at a specific task and they all work together to complete a big project. For our pothole detection system, this means having one main computer (the server with an API) that acts as a manager, and lots of smaller computers or devices (the clients) that act like scouts. These scouts go out, look for potholes, and send back information to the manager. This way, the work is done faster, and it doesn't overload any single scout or the manager.

Using a multi-agent system for pothole detection is smart because:

1. **It's Faster:** Many scouts can check different areas at the same time.
2. **It's Scalable:** You can start with a few scouts and add more as you need them.
3. **It's Flexible:** Scouts can be on different vehicles and use different routes every day.
4. **It's Robust:** If one agent fails, the others will still be able to do their job.
5. **It's Efficient:** The manager can organize the information and make sure it's used well.

By fulfilling these tasks, we aim to create a tool that not only improves road conditions but also enhances public safety and potentially saves on long-term road maintenance costs.

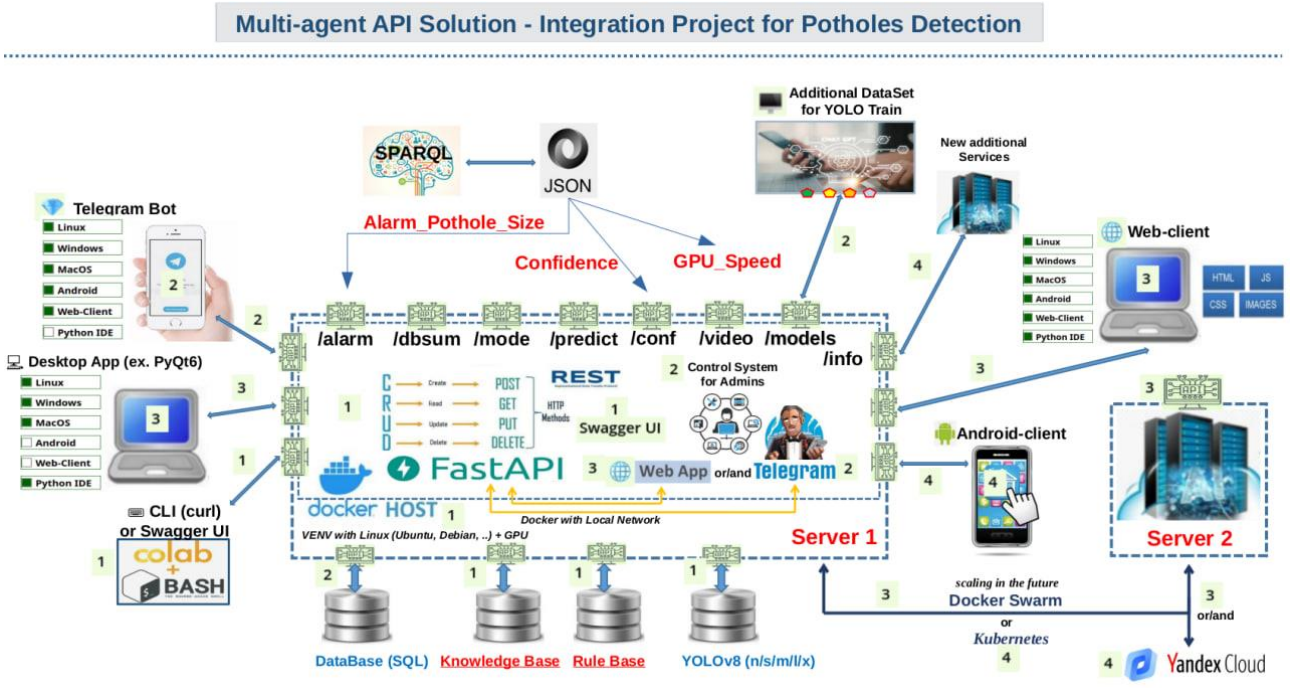
Here's how and where we could use a multi-agent system for detecting potholes:

1. **In City Buses:** They travel all around the city every day. Each bus can have a camera that sends pictures to the server when it sees a pothole.
2. **In Taxi Fleets:** Taxis can have an app that lets drivers report potholes with a button click, and the location is sent to the server.
3. **With Delivery Drones:** As drones fly overhead, they can take pictures and report back potholes from a bird's eye view.

4. **On Garbage Trucks:** They cover nearly every street once a week, making them great for scouting potholes.
5. **By Common People:** Ordinary people can install a simple application on their phones. If they notice a pothole, they can take a picture, and it will immediately go to the server with the geolocation of the place and with all the necessary details in a single database.
6. **By Various equipment of Road Services**: Vehicles and road maintenance equipment such as wipers and snow plows or dedicated drones can be equipped with cameras and sensors. As they drive through the city to perform their usual maintenance tasks, they can simultaneously detect potholes and report them. This integration improves the efficiency of road services by combining maintenance with pothole detection.
7. **By Google Glasses:** People wearing Google Glasses or similar augmented reality devices can contribute to pothole detection and be instantly alerted to potential hazards. When a pit is detected, the glasses can automatically capture an image and send it to the server. This method is especially effective in low-light conditions, as glasses can improve visibility, making it easier to detect potholes and report them at night or in poorly lit places, saving people from possible injury.
8. **Future Integration in Modern Cars:** Looking ahead, modern cars equipped with advanced sensors and cameras can be utilized for pothole detection. These vehicles, especially those with features like heads-up displays, can project signs and information about the road surface, including potholes, directly onto the driver's windshield. This not only contributes to the pothole detection database but also immediately informs drivers about potential road hazards, enhancing safety.

In each case, these agents work independently but contribute to a central system that collects, analyzes, and responds to the data on potholes, which helps the city fix roads faster and more efficiently. This can lead to safer driving, fewer accidents, and happy citizens.

The proposed architectural scheme of the interaction of the elements of this project (steps 1-2) and possible stages of further development (steps 3-4) is given below:



The diagram describes a multi-stage pothole detection project using a multi-agent system. The initial stages involve setting up a server (FastAPI Server 1) with a knowledge base (step 1) and database (step 2) and YOLOv8 for image processing, and a range of clients (like bot, web or/and Android apps) to send data. Early development includes integration of a Telegram bot for administration purposes, processing and markup of additional datasets for YOLO training and improving the quality of the neural model. Later stages focus on expanding services, such as alerts for pothole sizes, and scaling the system using Docker Swarm or Kubernetes, possibly incorporating a second server (Server 2 on stages 3-4) for enhanced performance and capacity in the future.

Objective function

The objective function described here is specifically designed to evaluate the performance of a model tasked with pothole detection. This function plays a central role in the model's training process, as it quantifies the model's accuracy in classifying potholes. The primary goal during training is to minimize this function, which, in essence, means reducing the number of errors the model makes in pothole prediction.
The objective function is defined as:

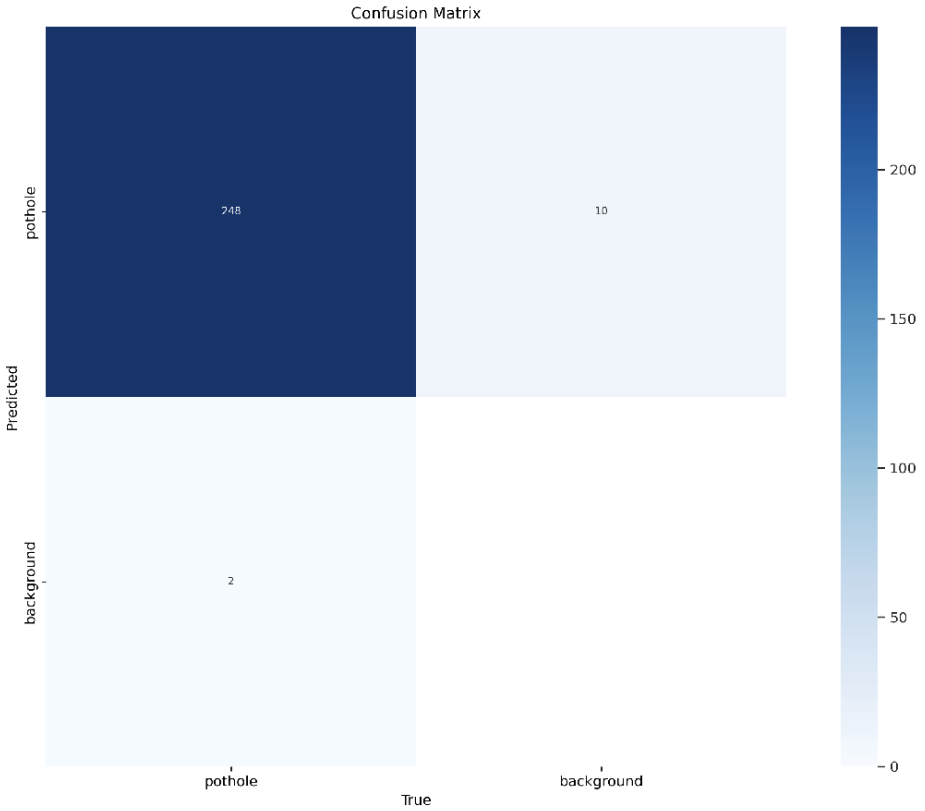
$$E(N_{ep}, \text{datasets}, \text{augmentation}, I_r, N_{pic}) = N_{cor} - N_{incor} \rightarrow \min$$

Where the components of the function are:

- **E**: This represents the pothole classification error. It's a measure of how well the model is performing, with a lower value indicating better performance.
- **Ncor (Correctly Detected Potholes)**: This is the number of potholes that the model has correctly identified. A higher count of correctly detected potholes contributes positively to model performance.

- **Nincor (Missed Potholes)**: This counts the potholes that the model failed to detect. The goal is to minimize this number, as each missed pothole is a classification error.
- **Nep (Number of Epochs)**: This refers to the number of complete passes through the training dataset. The number of epochs can affect the model's learning and its eventual performance.
- **Ir (Image Resolution)**: The resolution of images used in training can significantly impact the model's ability to detect potholes accurately. Different resolutions may be experimented with to find the optimal setting for the model.
- **Datasets**: These are collections of images used for training, encompassing various weather conditions like dry, rainy, snowy, etc. The diversity in the datasets ensures that the model is robust and can perform well under different real-world conditions.
- **Augmentation**: This is the process of artificially expanding the training dataset by altering the images, such as by rotating, cropping, changing brightness, etc. Augmentation helps in making the model more generalizable and less prone to overfitting.
- **Npic (Number of Elements in the Training Set)**: This is the total number of images or elements in the training dataset. A larger dataset can provide more comprehensive training, but it also requires more computational resources.

By focusing on minimizing this objective function, the training process aims to enhance the model's ability to accurately detect potholes while reducing the likelihood of missing any. This optimization leads to a more reliable and effective pothole detection model.



The confusion matrix in the image represents the performance of the YOLOv8 model on one of the training datasets for pothole detection.

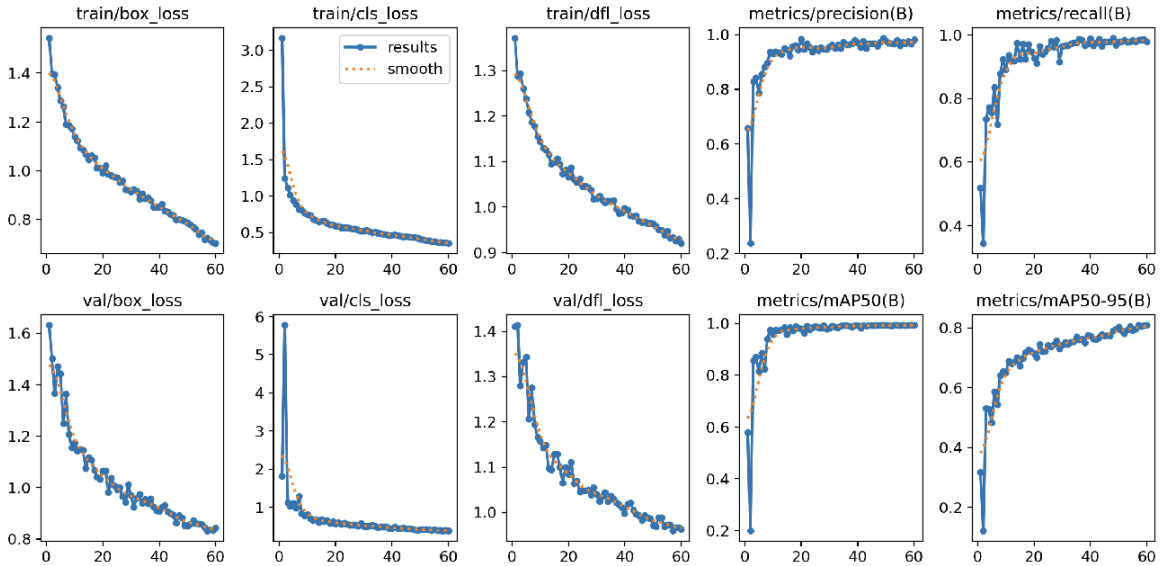
- **True Positives (Top Left Square)**: The model correctly identified 248 instances as potholes. These are the true positives, indicating cases where the model's prediction and the actual label agree that the potholes are present.

- **False Negatives (Bottom Left Square)**: There are 2 instances where the model incorrectly predicted the background (no pothole) when there was actually a pothole. These are false negatives, which suggest that the model missed detecting potholes in these cases.
- **True Negatives (Bottom Right Square)**: This cell of the matrix is not explicitly labeled, but it implies the number of true negatives, where the model correctly identified there were no potholes, and there were indeed none. Since it's not labeled, we assume it might have a high number consistent with a well-performing model, but the exact value is not visible.
- **False Positives (Top Right Square)**: There are 10 instances where the model incorrectly identified potholes when none were present. These are false positives, representing over-detection by the model.

In conclusion, the YOLOv8 model appears to be performing quite well on this dataset, with a high number of true positives and relatively low numbers of false negatives and false positives. However, the true negative count is missing from the confusion matrix, which is essential for a complete understanding of the model's performance, particularly its specificity. The high number of true positives and low number of false negatives and positives suggests that the model is effective at detecting potholes with a high degree of precision and recall.

Based on the example of provided confusion matrix data, the following quality metrics for the YOLOv8 model training on the pothole detection dataset are calculated:

- **Precision**: Approximately 96.12%, indicating a high accuracy of the model in classifying an image as containing a pothole when it does.
- **Recall**: About 99.2%, showing that the model is highly capable of identifying most of the positive pothole cases in the dataset.
- **F1 Score**: Approximately 97.64%, which is a balanced measure that takes into account both the precision and the recall. This high F1 score suggests that the model has a harmonious balance of precision and recall, making it very effective in the pothole detection task



The image shows a series of graphs showing various indicators and values of losses during the training of the YOLOv8 model for the task of detecting potholes on roads.

- **Box Loss (Train and Validation)**: These plots show the loss associated with the bounding box predictions. Both training and validation box losses decrease steadily over time, which indicates that the model is getting better at accurately predicting the location and size of the bounding boxes around the potholes.
- **Class Loss (Train and Validation)**: The classification loss represents the model's ability to correctly classify the objects within the bounding boxes. Both plots show a sharp decline and then level off, suggesting that the model quickly learned to classify the objects correctly and then made incremental improvements.

- **Objectness Loss (Train and Validation)**: This is depicted as "train/obj_loss" and "val/obj_loss" (though labeled as 'df1_loss' in the plots, which might be a specific notation for the dataset or framework used). These plots show the loss related to the confidence of the object presence within the bounding box. The trend is similar to the box and class losses, with a decrease over epochs indicating improving confidence in predictions.
- **Precision and Recall**: These metrics are crucial for understanding the model's performance. The precision plot shows that the model is consistently accurate in its predictions, and the recall plot indicates that the model is able to identify most of the relevant objects. Both metrics appear to have reached a high level, which suggests that the model has a good balance between precision and recall.
- **mAP (Mean Average Precision) at IOU=50 and IOU=50-95**: The mAP at IOU=50 is near perfect, which suggests that the model is very accurate when the Intersection Over Union (IOU) threshold is at 50%. The mAP at IOU thresholds between 50% and 95% shows a gradual increase, which suggests the model performs well across a range of strictness in overlap criteria, although it's less accurate at the highest thresholds.

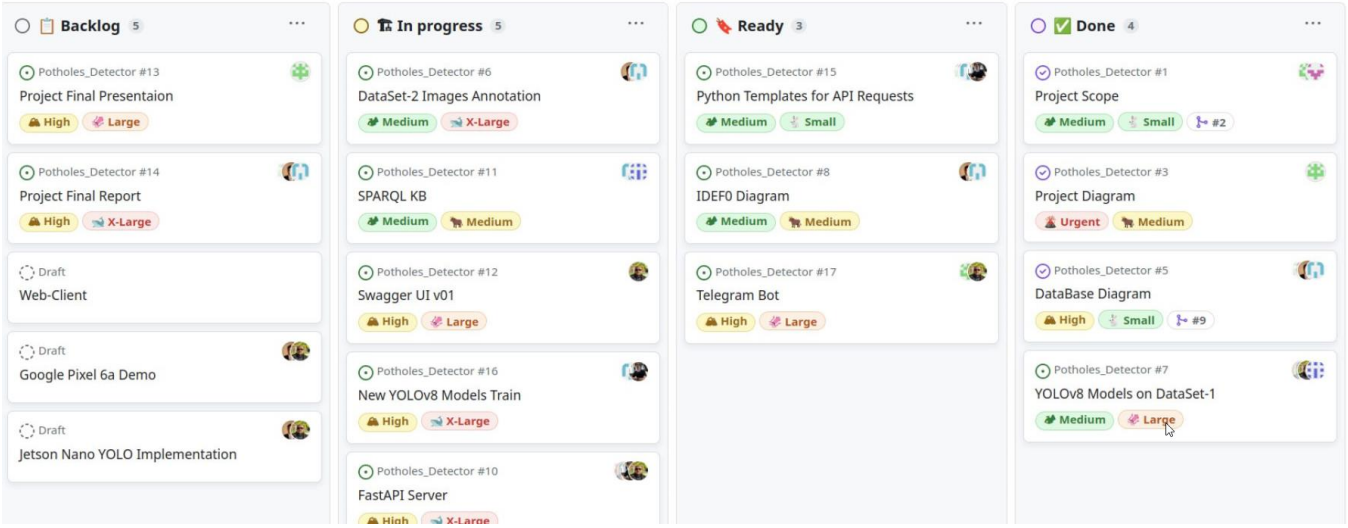
Overall, the plots indicate that the model has trained effectively, with loss metrics showing improvement and performance metrics indicating high precision and recall. The mAP scores suggest that the model is quite robust, performing well across different IOU thresholds. This suggests that the model would likely perform well in practical applications, such as detecting potholes in various conditions.

2. Approach for solution

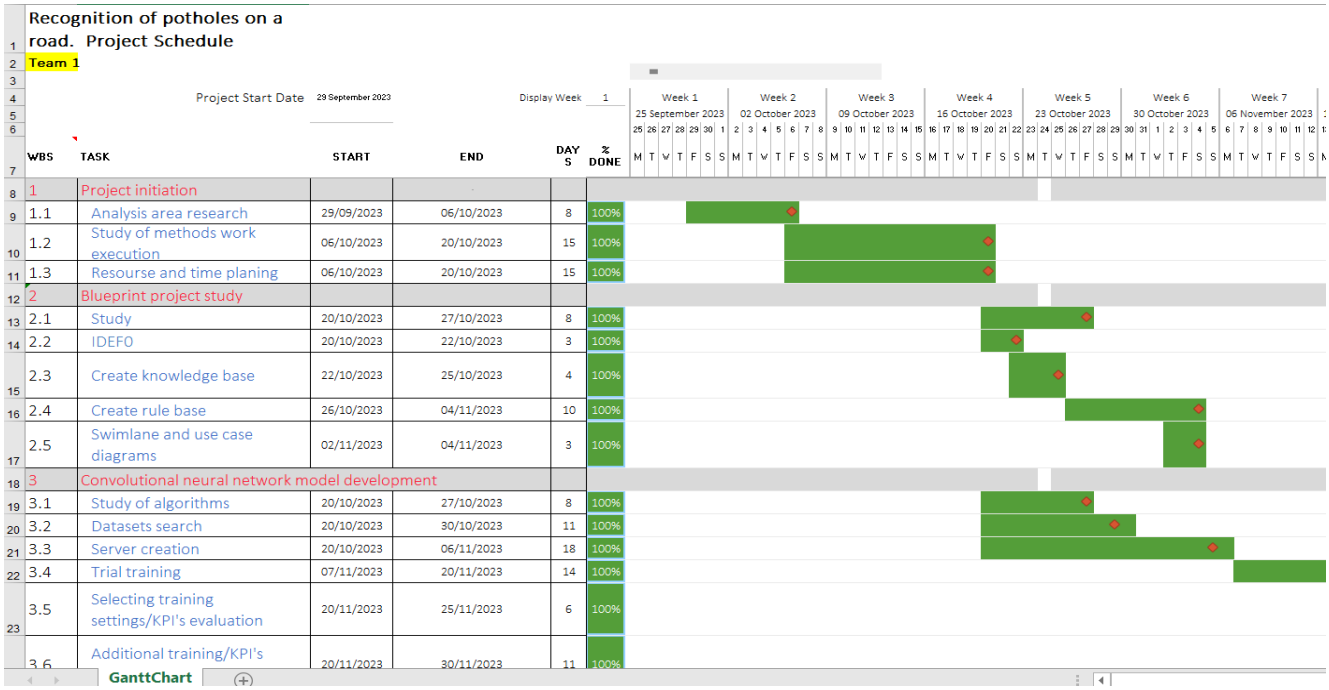
2.1 Work process organization

In the process of working on the project, our team used Kanban and MS Project to effectively organize work and track key points.

Kanban's visual board provided our team with an intuitive and real-time overview of ongoing tasks. Each team member could easily visualize their assignments, making it simpler to prioritize and stay aligned with project goals. The Kanban methodology allowed us to streamline our workflow by visualizing the entire process. Tasks moved through distinct stages, from "Backlog" to "In Progress" and finally to "Done," providing transparency into each task's status.



Microsoft Project played a key role in comprehensive project planning. Its Gantt chart capabilities enabled us to create detailed project timelines, including task dependencies and critical milestones. The integration of Kanban and Microsoft Project proved to be a winning combination for our team. It fostered a collaborative and organized work environment, enabling us to track key project points effectively and deliver successful outcomes.



Recognition of potholes on a road. Project Schedule

Team 1

Project Start Date: 29 September 2023 — Display Week: 1

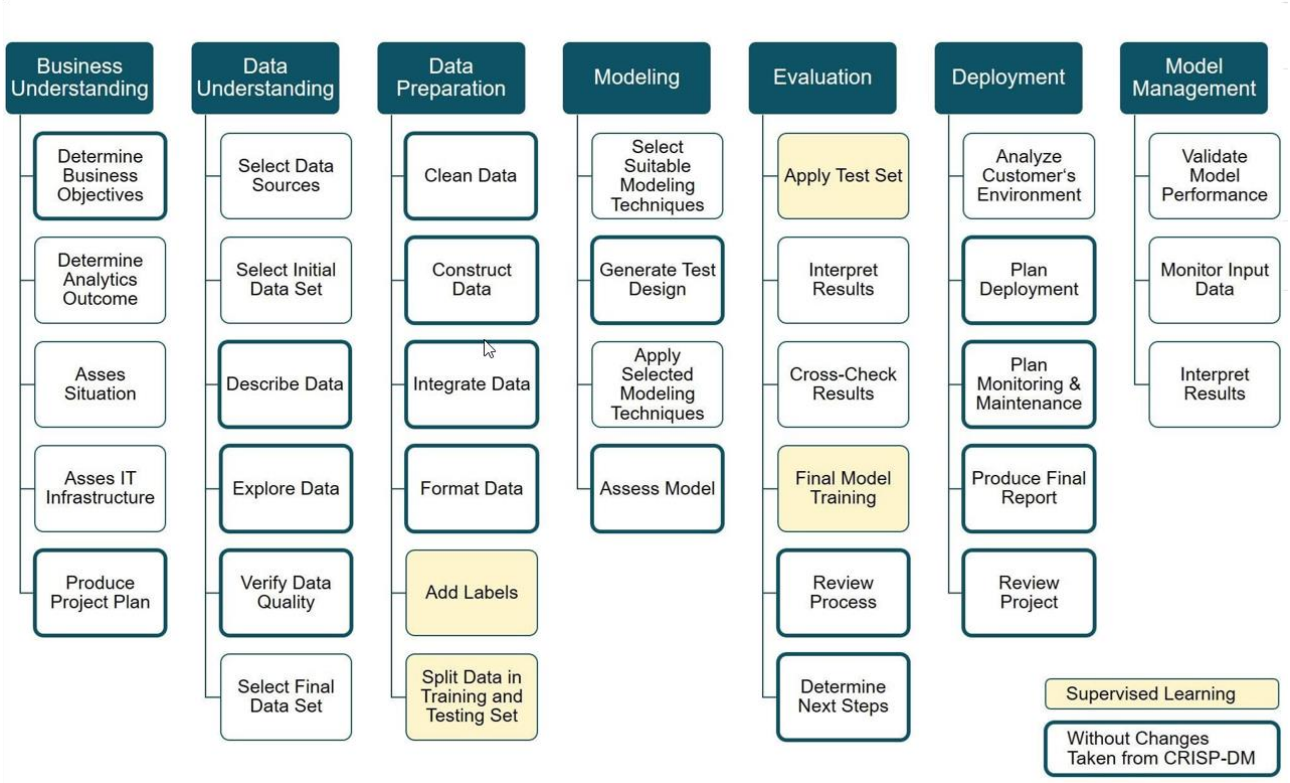
WBS	TASK	START	END	DAYS	% DONE
1	Project initiation				
1.1	Analysis area research	29/09/2023	06/10/2023	8	100%
1.2	Study of methods work execution	06/10/2023	20/10/2023	15	100%
1.3	Resourse and time planing	06/10/2023	20/10/2023	15	100%
2	Blueprint project study				
2.1	Study	20/10/2023	27/10/2023	8	100%
2.2	IDEF0	20/10/2023	22/10/2023	3	100%
2.3	Create knowledge base	22/10/2023	25/10/2023	4	100%
2.4	Create rule base	26/10/2023	04/11/2023	10	100%
2.5	Swimlane and use case diagrams	02/11/2023	04/11/2023	3	100%
3	Convolutional neural network model development				
3.1	Study of algorithms	20/10/2023	27/10/2023	8	100%
3.2	Datasets search	20/10/2023	30/10/2023	11	100%
3.3	Server creation	20/10/2023	06/11/2023	18	100%
3.4	Trial training	07/11/2023	20/11/2023	14	100%
3.5	Selecting training settings/KPI's evaluation	20/11/2023	25/11/2023	6	100%
3.6	Additional training/KPI's	20/11/2023	30/11/2023	11	100%

Week 1 25 September 2023 · Week 2 02 October 2023 · Week 3 09 October 2023 · Week 4 16 October 2023 · Week 5 23 October 2023 · Week 6 30 October 2023 · Week 7 06 November 2023

GanttChart

Also, the CRISP-DM methodology (Cross-Industry Standard Process for Data Mining) played a significant role in our project. CRISP-DM provides a structured approach to data-driven projects. It involves clear phases such as business understanding, data understanding, data preparation, modeling, evaluation, and deployment. This structure ensures that the project follows a systematic and organized path.

Given the substantial reliance of AI on data, CRISP-DM's emphasis on understanding and preparing data is particularly relevant. This ensures that the data used for model training is not only of high quality but also relevant and representative.



Multi-agent solution

The best approach for our project is to build a system with several smart parts working together. Using a multi-agent approach brings a lot of benefits. We'll have a main computer server that acts like a brain, and lots of smaller devices out in the world that act like eyes, spotting potholes. They will all talk to each other and share what they see. Each agent, whether it's a piece of software or a device, is really good at its job and works with the others to achieve our big goal: detecting potholes quickly and accurately. This way, we can quickly find out where the potholes are and fix them. It's like creating a team of experts, each with their own job, but all working towards the same goal of making our roads better and safer. Here's how it works in simple terms:

- **FastAPI**: This is like our project's coordinator. It's super fast and can handle lots of requests from the agents without getting tired. It talks to our databases, like SQLite, and SPARQL knowledge bases, to get or/and store information.
- **Databases and Knowledge Bases**: SQLite keeps our data organized and ready to use. The SPARQL knowledge base is like a smart library that helps us find specific information when we need it. In our implementation, when starting the server, we download the main KPIs from the knowledge base and store them along with the rest of the parameters in the config.json file.
- **Pre-trained YOLOv8 Models**: These are our experts in recognizing what a pothole looks like in different situations. They've learned from lots of pictures and can now spot potholes quickly.
- **Telegram Bot with AIOGram**: This is like a remote control for our project based on common Docker local network with FastAPI with fast communication. We can use it to change settings or check on things without having to be at the computer all the time.

- **Confidence, Skyline, Mode Settings**: These settings let us adjust how picky our pothole detector is and to select settings for a specific configuration of equipment on the agent for detecting potholes on the roads . We can make it more or less strict depending on what we need, or how the equipment is installed (for example, the angle of the surveillance camera), etc.
- **User Access and Server Choices**: We can decide who gets to use our system and which server should do the work. If one server is busy, we could have a backup ready.

All of these agents can work at the same time without waiting for each other. That's because they communicate asynchronously, which means they don't get in each other's way. So while one agent is spotting a pothole, another can be fetching data, and another can be learning from new pictures. It's efficient and scalable, which means our system can grow and handle more work without getting overwhelmed. This way, we can keep our roads safer and smarter.

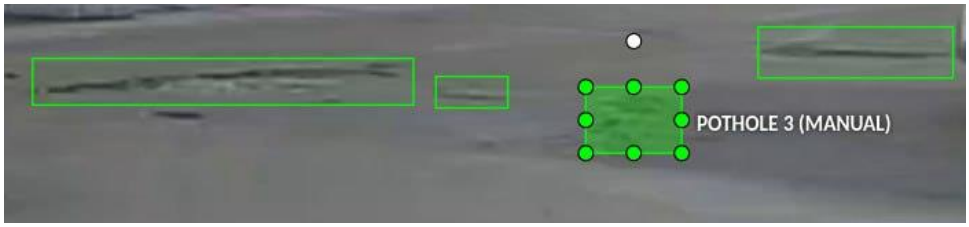
Image Labeling for extended DataSet

Our Team1 conducted training sessions using an available dataset of potholes on roads and concluded that we needed more data for training. After incorporating another small dataset that turned out to be suboptimal, the accuracy of our system decreased. The reason was straightforward—the problematic dataset included data augmentation with images flipped 180 degrees, resulting in upside-down cars, which is unrealistic. Consequently, we decided to gather additional data, manually annotate the dataset, and retrain the system on the new, combined dataset with the original one.

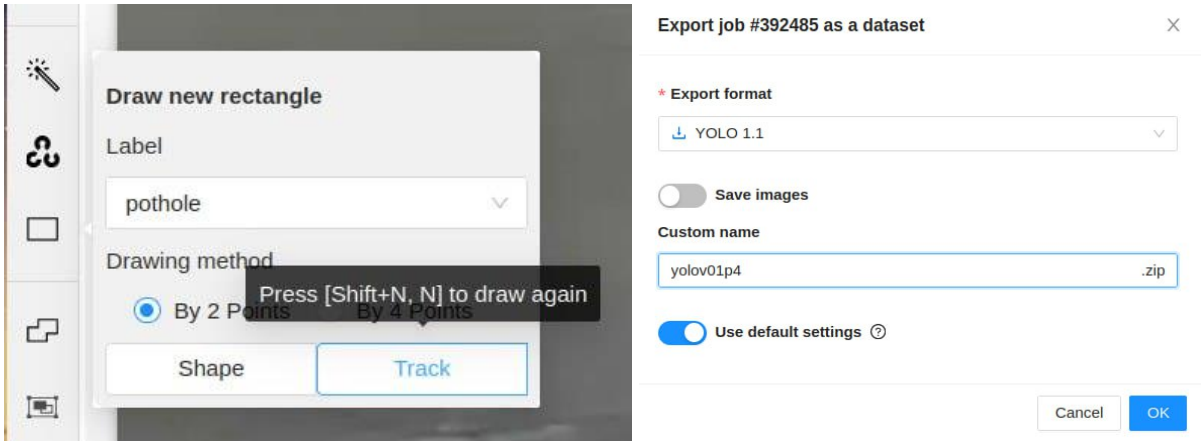
Our Team1 approached the video annotation task methodically to prepare data for training our YOLOv8 model. We first divided the entire video into several segments, ensuring that each segment contained various instances of the objects of interest. This division allowed multiple team members to work on different segments simultaneously, significantly speeding up the annotation process.

For the annotation, we used a cloud-based tool CVAT that enabled us to track objects across frames, which is especially useful for videos where objects move. Our team members drew shapes around the objects in the first frame, and the tool automatically tracked those objects in subsequent

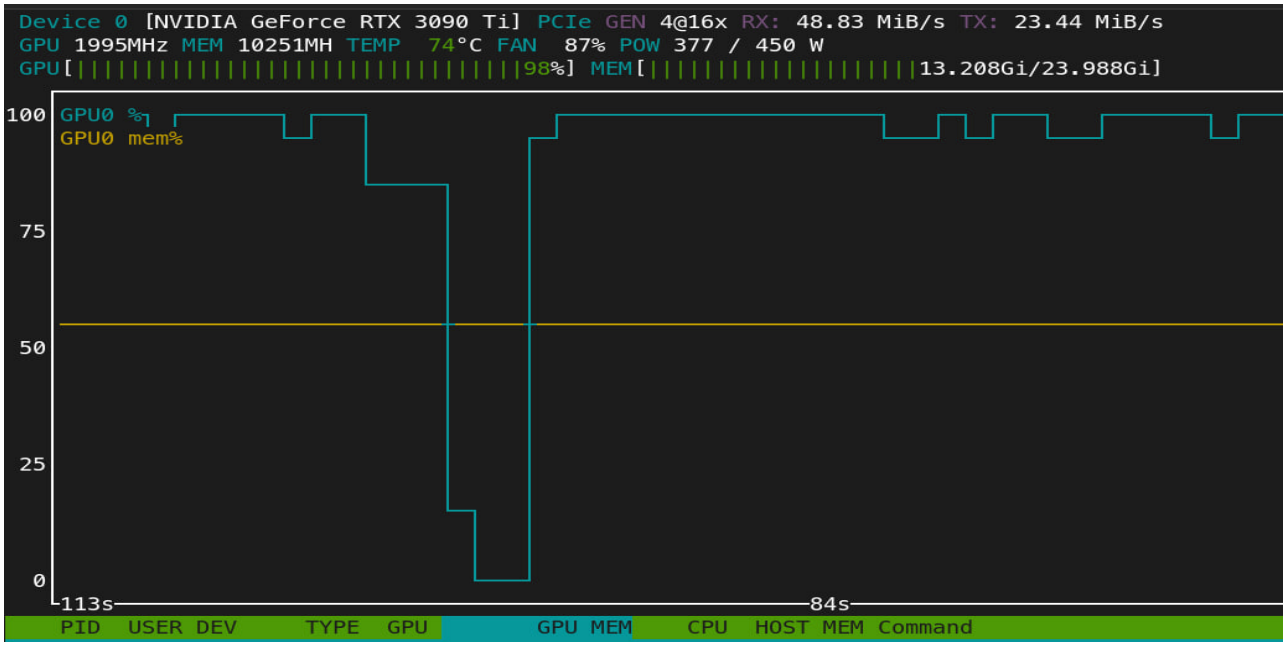
frames. Whenever the automatic tracking was not accurate, we manually adjusted the shapes to ensure consistency and precision.



Key frames were established at points where objects changed direction or appearance significantly, to guide the tracking algorithm. After completing the initial annotation, we thoroughly reviewed and refined the shapes to maintain alignment with the objects throughout the video.



This collaborative and systematic approach not only accelerated the annotation process but also ensured high-quality data was produced for the subsequent training of our YOLOv8 model, setting the stage for accurate and effective object detection.



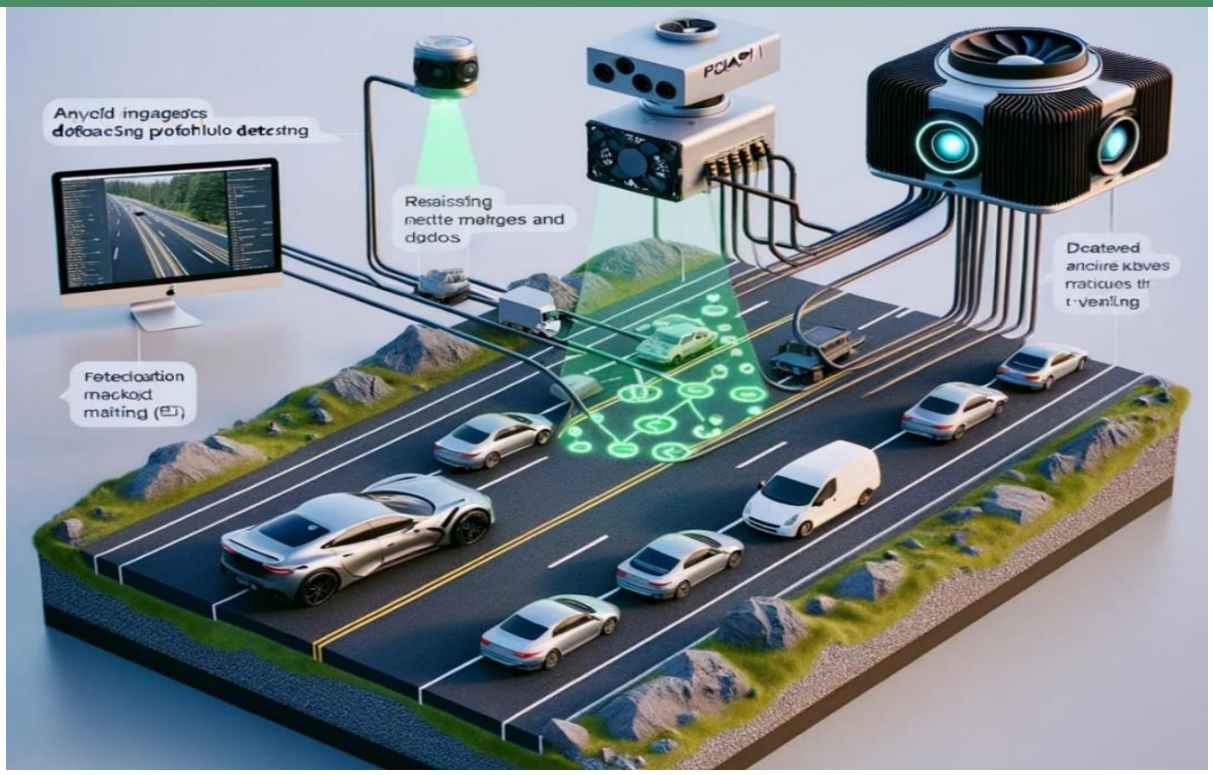
The digital twin concept

The digital twin is an advanced technological concept that is becoming increasingly important in various fields, including our pothole detection project. In fact, a digital twin is a detailed virtual

model of a real system or process. Think of it as a mirror that reflects not only the appearance, but also the behavior and dynamics of a physical object or system in a digital space.

As part of our pothole detection project, the digital twin now includes a number of features and can develop in various directions, including a lot of different components. To implement the various aspects of our road pothole detection project, we can consider the following approaches:

1. **Simulation of Environmental Factors**: This could involve replicating different weather conditions, traffic patterns, and lighting conditions that affect pothole formation and detection. We can model varying traffic conditions to understand how different levels of road usage impact pothole detection or can use virtual environments to simulate different weather conditions like rain, snow, or extreme heat that might affect pothole formation.
2. **Pothole Detection System Model**: A virtual representation of our detection system, including the hardware (like cameras and sensors) and the software (the YOLOv8 model and FastAPI server). We can create a virtual model of the detection system, including camera positions and angles, sensor placements, and the YOLOv8 algorithm's processing capabilities or simulate the detection process in diverse lighting and weather conditions to test the system's accuracy and robustness.
3. **Virtual Replication of Roads**: A digital model of the road networks where the application will be deployed, including various conditions such as urban streets, highways, and rural roads. In this case, we can include features such as road markings, signs and adjacent landscapes to increase the realism of the simulation, or develop 3D models of various types of roads, including urban streets, highways and rural paths, to test the effectiveness of the system on various sites by running artificially created video processing.
4. **User Interaction Scenarios**: Simulated interactions of various users, such as municipal authorities, road maintenance crews, and regular commuters, with our system. We can create scenarios where users receive and respond to pothole detection alerts, submit feedback, or request additional services or even simulate how different users, such as maintenance crews, city planners, and general users, will interact with the system.
5. **Data Flow and Processing**: A model of how data is captured, transmitted, processed, and stored, mirroring the real-world flow of information within our system. By compiling a special test procedure, we can test the data transfer rate, processing time and system response under various load conditions to ensure efficient data processing. For a deeper study of all processes, it is also possible to outline the entire data transmission path from the collection (using sensors/cameras) before processing (YOLOv8 and Facet API) and, finally, before storage and presentation (databases and user interfaces), making the necessary measurements on different equipment and at different load intensities.
6. **Infrastructure Requirements**: Virtual representation of the necessary infrastructure for the system’s deployment, including server specifications and network requirements. The digital twin concept also allows you to solve the following tasks: to simulate network data transfer requirements, paying special attention to areas with different connection levels or to model the server infrastructure needed to support the Rest API and the YOLOv8 system, taking into account factors such as computing power, memory and storage.



By virtually modeling and testing these aspects, we can, on the one hand, anticipate and reduce the risks of possible problems, ensuring a more reliable and efficient deployment of our pothole detection system, and on the other hand, create more complex information collection and processing systems by linking detailed information read by different agents and entering the database to a virtual map of cost and quality road surfaces. Thus, it can be a convenient solution for road services, and from the point of view of operation, an effective way to improve the quality of roads and increase safety for road users.

A neural network model for recognizing potholes on roads, based on YOLOv8, could be considered a form of a digital twin, but with some distinctions. In general, a digital twin is a virtual representation that serves as the real-time digital counterpart of a physical object or process. It is used for simulation, analysis, and control. In our case (at the initial stage of a comprehensive project), the YOLOv8-based model serves as a digital tool to replicate and analyze a specific aspect (potholes) of a physical environment (roads) with emulation of location coordinates.

However, it's important to note that the model mainly focuses on detection and recognition, rather than encompassing all aspects of the physical roads. A full digital twin of a road infrastructure would include not just pothole detection but also other features like traffic patterns, road wear and tear, weather effects, and more, what will be possible to implement potentially in the future.

IDEF0

In our project, we've created a detailed IDEF0 diagram to map out all the processes involved in our pothole detection system. IDEF0 is a method designed to model the decisions, actions, and activities of an organization or system. In simple terms, it's like drawing a map of what happens in our system, from start to finish.

The IDEF0 diagram we've developed clearly outlines the inputs, outputs, mechanisms, and controls for each process in our system. This includes everything from how we collect data, to how we train our neural network (YOLOv8), process requests, and eventually, how we deliver results.

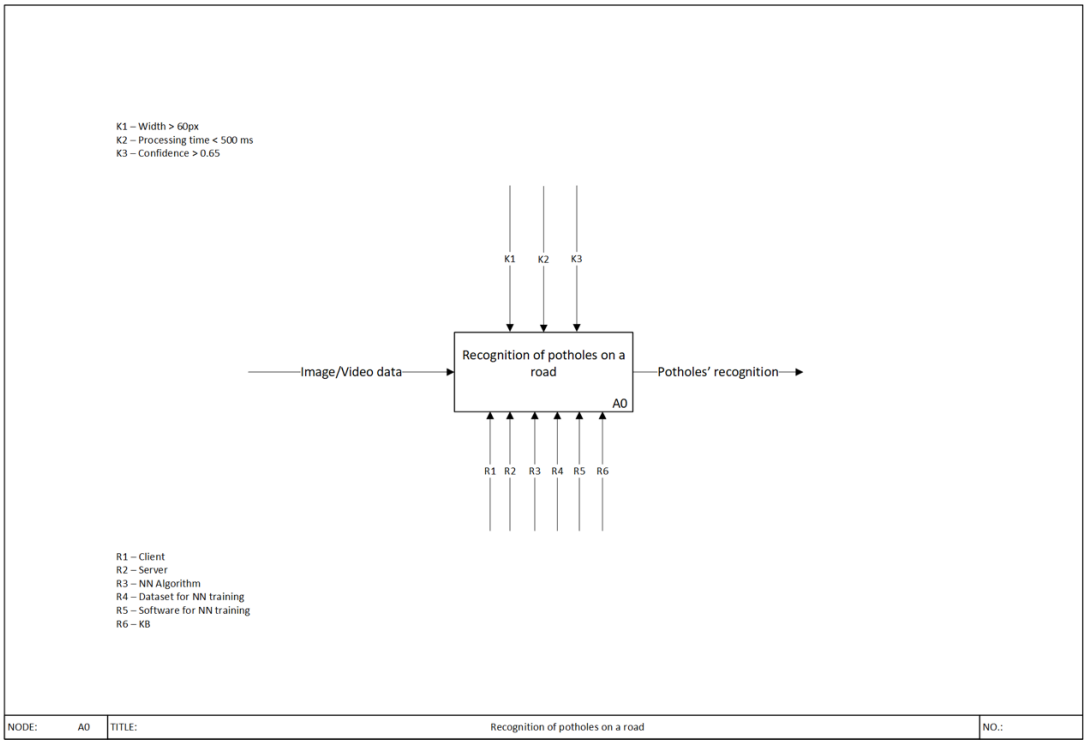
For instance, in the data collection stage, we look at what kind of data we need (like images of roads), where it comes from, and what we do with it. Then, in the neural network training stage, we feed this data into YOLOv8, tweaking it to learn and get better at spotting potholes.

Next, our diagram shows how we handle incoming requests – say, from a city worker or an automated sensor – and how these requests are processed by our system. Finally, it details how we deliver the results, like identifying a pothole's location, and possibly, its size and depth.

Preparing an IDEF0 diagram is a good opportunity to form a documentation for a project capturing the current state of processes Is can easily be transformed into another type of artifacts, e.g. Knowledge Base.

Overall, our IDEF0 diagram is a crucial tool in our project. It helps everyone involved understand the whole process clearly, makes sure we don't miss any important steps, and allows us to pinpoint areas where we can make our system even better.

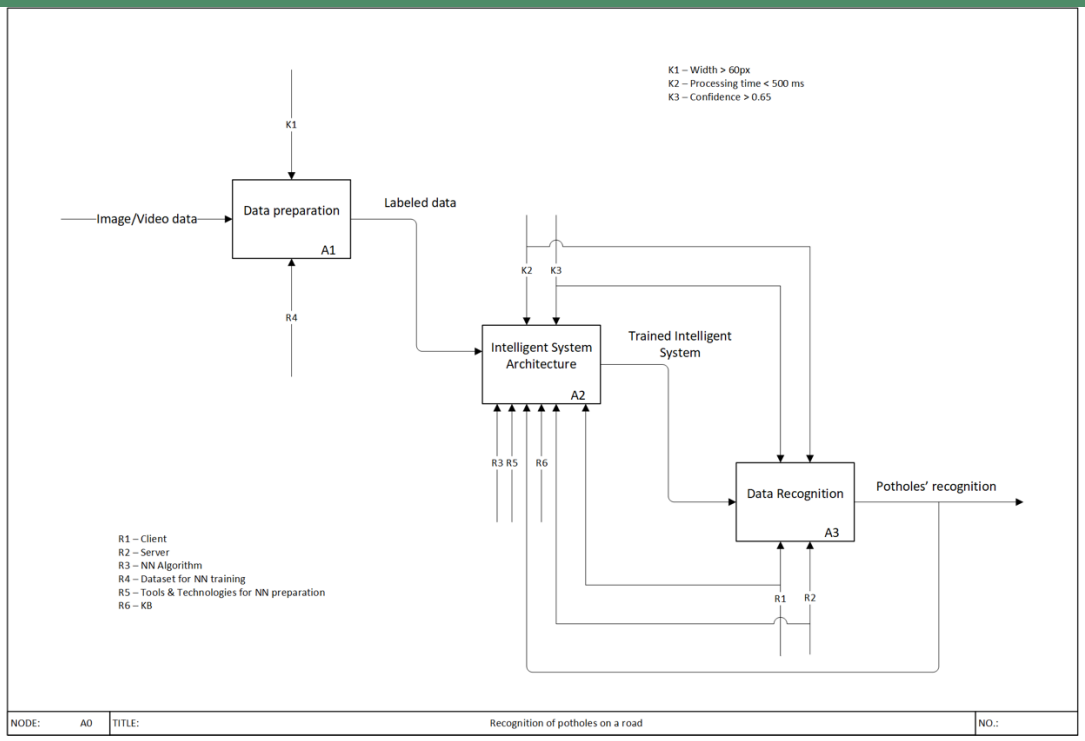
The main level of our IDEF0 diagram is as follows:



It demonstrates:

- Inputs;
- Outputs;
- Resources: R1-R6;
- KPIs: K1-K3.

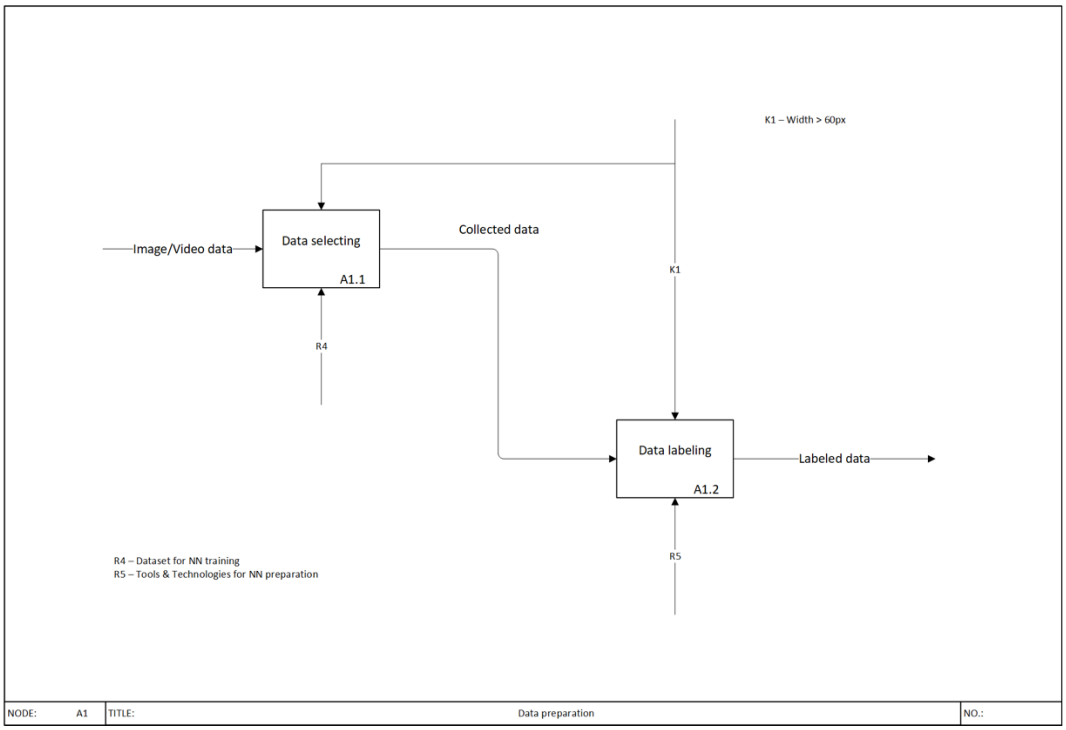
On A0 level the main level is decomposed to 3 processes as follows:



3 key processes are presented on this level:

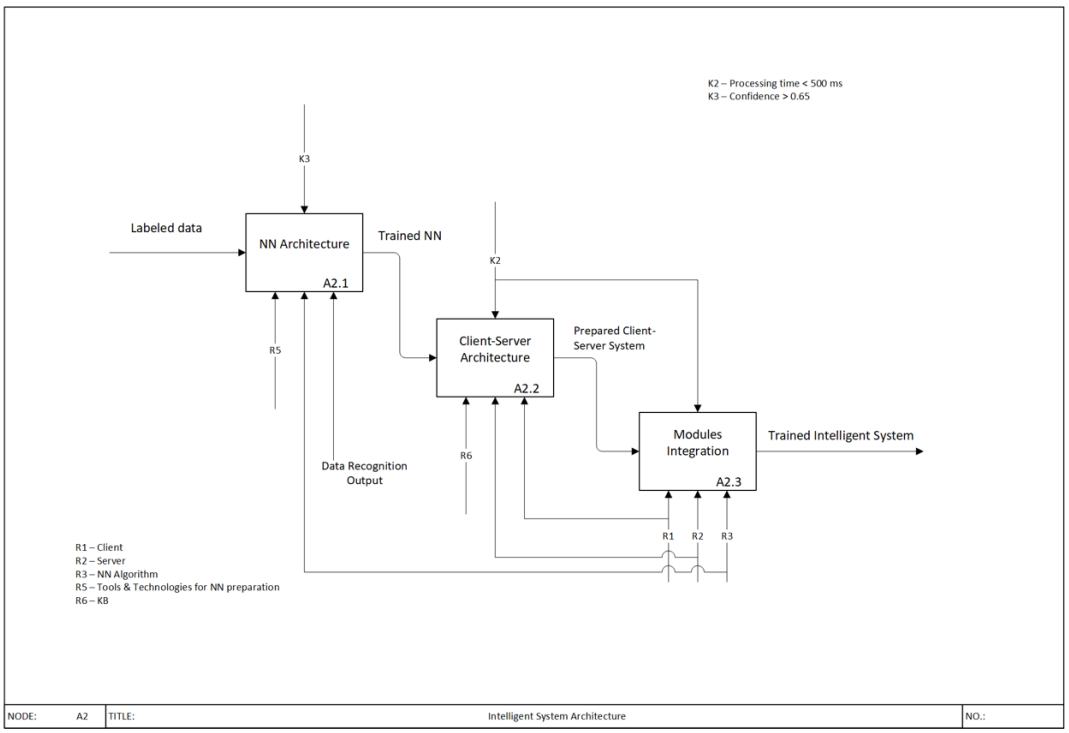
- A1 Data preparation – only K1 KPI and only R4 resource are used;
- A2 Intelligent System Architecture – K2-K3 KPIs and R1-R3, R5-R6 resources are used as well as the output from Data Recognition process;
- A3 Data Recognition – K2 -K3 KPIs and R1-R2 resources are used.

Details regarding the process A1 are given below:



The A1 process consists of 2 key sub-processes: Data selectin and Data labeling. The first one (A.1.1) includes set of activities related to looking for a proper dataset. The latter (A.1.2) includes a phase of labeling raw data in addition to the selected dataset.

The most complicated process is A2. The decomposition of it is presented below:



On this level three key sub-processes were described:

- A2.1 Neural Network Architecture – only K3 KPI and R3, R6 resources are used as well as the output from Data Recognition process;
- A2.2 Client-Server Architecture – only K2 KPI and R1-R2, R6 resources are used;
- A2.3 Modules Integration – only K2 KPI and R1-R3 resources are used.

Developing of the IDEFO diagram was useful for structuring project development processes, decomposing processes, and further building teamwork. IT significantly increased the understanding of all the team about the stages of the project.

Swimlane Diagram

A Swimlane diagram is a visual representation that illustrates the flow of processes or activities within a system, organization, or project. It could be useful to decompose and simplify complicated processes.

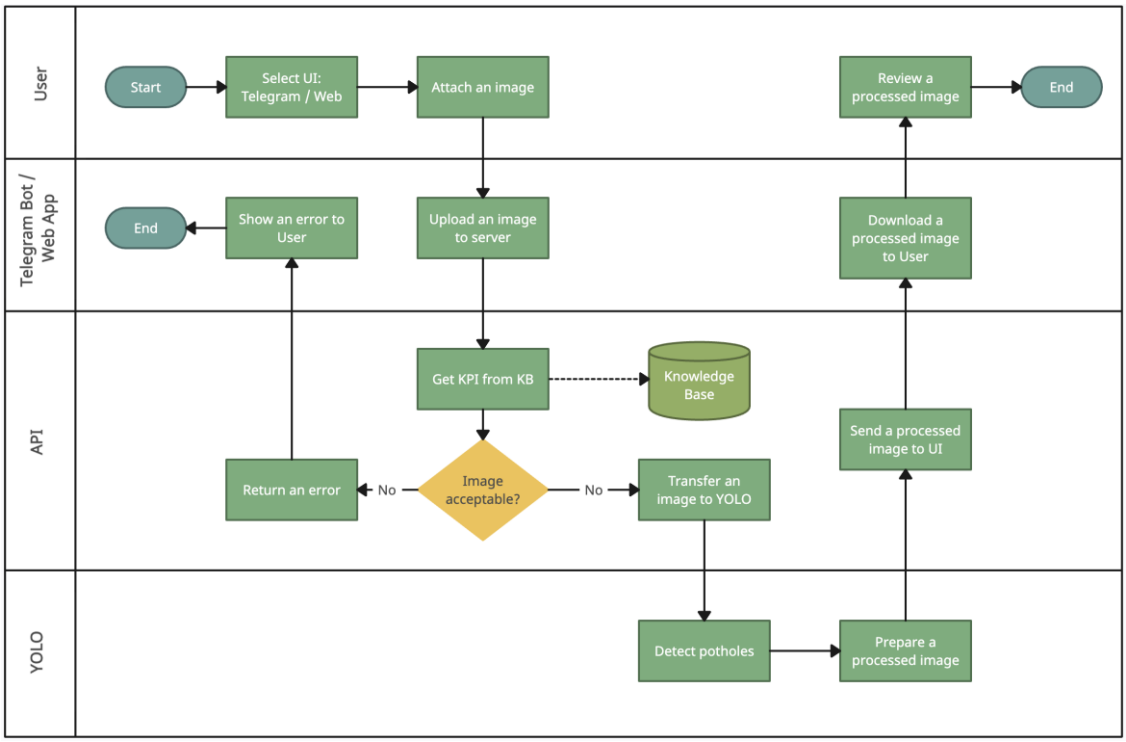
For IT projects, such as developing Intelligent Systems, swimlane diagrams could bring several benefits such as:

1. Clarity – Swimlane diagrams provide a transparent visual representation of workflows, making it easier for team members;
2. Processes optimization – Teams could identify and work with bottlenecks or/and areas for improvements;

3. Collaboration / Coordination (when used for describing project processes): Swimlane diagrams transparently delineate the responsibilities of individuals inside a team. By using this collaboration and coordination improve because participants understand their personal role and responsibilities;
4. Documentation – Preparing a swimlane diagram is a good opportunity to form a documentation of a future system, etc.

In the context of developing intelligent systems, where many people collaborate, and complex workflows are involved, a swimlane diagrams could introduce comprehensive view of a developing system.

The Team1 decided to demonstrate top-level details of general process of potholes detection. The diagram was developed as follows:



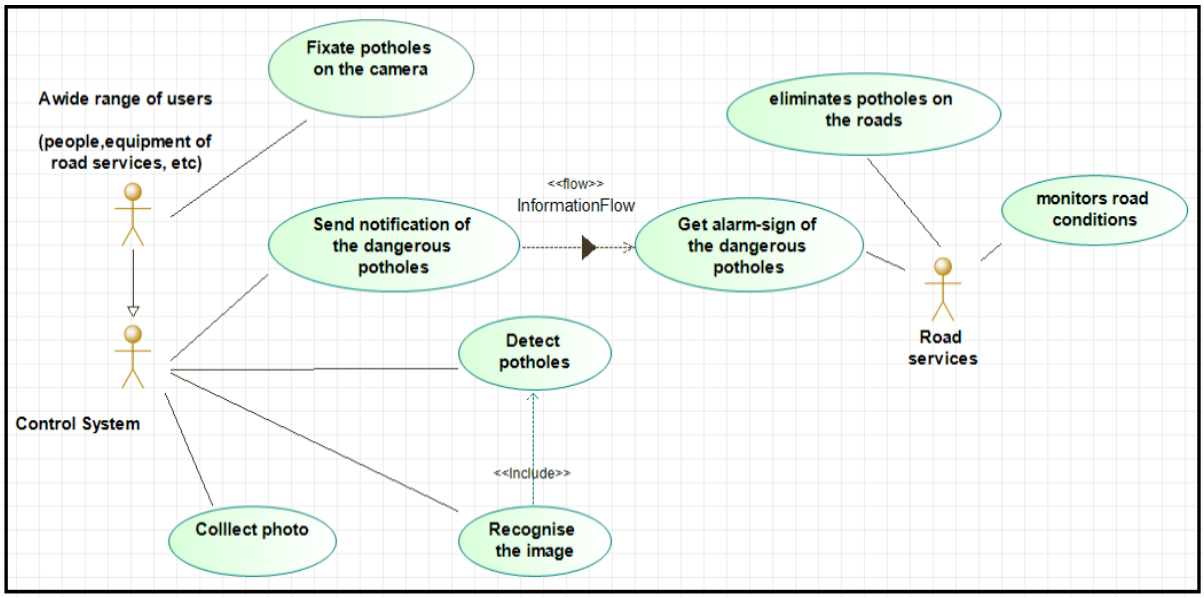
Is consists of 4 key actors along with actions they perform:

- A User that selects which one of user interfaces will be used and initiates the recognition process;
- Telegram Bot or Web Application represent two types of clients (UI);
- Server side (API) that includes Database, Knowledge Base, Rule Base, Code, etc.
- YOLO as an algorithm that is used to detect potholes.

After a user sends an image, a client uploads it to the server side where all manipulations with the image are performed. If the system cannot process an image it reruns an error to a user. After the processing stage backend sends the image to client which shows it to the user.

Use Case Diagram

A Use Case diagram has been prepared to illustrate the functionality and interactions between our system and users.



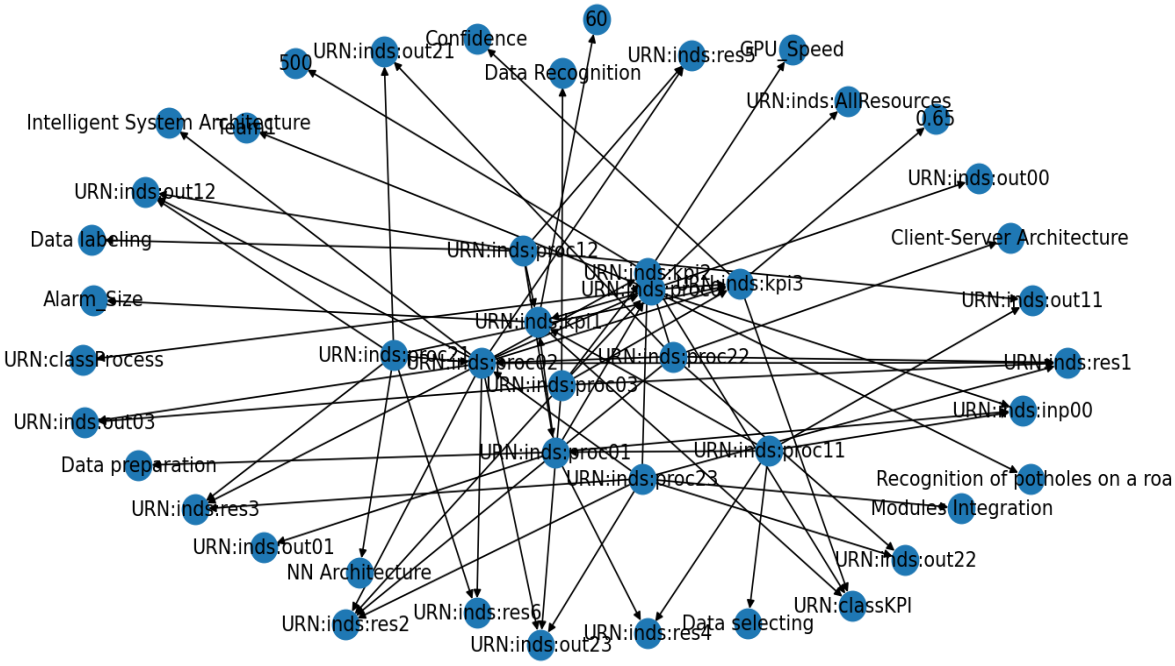
Knowledge base, Rule base, and SPARQL

Using the fundamental concepts to the field of artificial intelligence and knowledge representation, we created the Knowledge base and the Rule base for our intelligent system with RDF, RDFS and OWL technologies. RDF (Resource Description Framework), RDFS (RDF Schema), and OWL (Web Ontology Language) are key standards developed from the World Wide Web Consortium (W3C) for representing and modeling knowledge on the Semantic Web. Each of these standards plays a specific role in enabling the creation of structured, interoperable, and machine-readable data.

1. Knowledge base:

   A Knowledge Base (KB) is a crucial component in an expert system designed to store, organize, and manage knowledge in a structured manner. It acts as a repository for facts, information, rules, and relationships that represent the understanding of a particular domain. Knowledge bases play a crucial role in facilitating decision-making, problem-solving, and reasoning within intelligent systems. It has a dynamic nature and can be updated and expended over time to incorporate new information or adapt to changes in the environment.

   Our Knowledge Base contains all the information from the IDEF0 diagram, and it might be shown as a graph schema:



Furthermore, our KB is represented in RDF and RDFS which are standards for representing information about resources, properties, and relationships in triple-store format Subject-Predicate-Object. This triple structure forms the basis of representing relationships between resources in a machine-readable format (Sugumaran, 2016).

```
@prefix ind:<URN:inds:>.
@prefix prop:<URN:prop:>.
@prefix classes:<URN:class>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .


ind:proc0 a classes:Process ;
        rdfs:label "Recognition of potholes on a road" ;
        rdf:isDefinedBy "Team1" ;
        prop:hasKPI ind:kpi1 ;
        prop:hasKPI ind:kpi2 ;
        prop:hasKPI ind:kpi3 ;
        prop:hasResource ind:AllResources ;
        prop:hasInput   ind:inp00 ;
        prop:hasOutput  ind:out00 .
```

2. Rule base:

A Rule base is a collection of rules that govern the behavior or decision-making process of the system. Much of the knowledge in it is represented as rules, those are conditional sentences relating statements of facts with one another into the project. These rules are based on human expert knowledge that enable the system to process input data and produce a result. Rule base is flexible and allows for easy modification and addition of rules without major changes to the system architecture. It enables quick adaptation to new scenarios, changes in the environment, or updates in the knowledge base (Foster, 2023).

```
@prefix ind:<URN:inds:>.
@prefix prop:<URN:prop:>.
@prefix classes:<URN:classes:>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .


classes:Process a owl:Class .

prop:hasKPI a owl:ObjectProperty ;
       a owl:IrreflexiveProperty ;
       a owl:AsymmetricProperty .

prop:hasResource a owl:ObjectProperty ;
       a owl:AsymmetricProperty .
```

3. SPARQL:

SPARQL is a query language and protocol designed to query and retrieve data from RDF data stores. Most forms of SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable (The W3C SPARQL Working Group, 2013).

In order to be confident that your expert system works in a proper way, you must check if any inconsistencies exist. Finding inconsistencies between a knowledge base and a rule base in SPARQL involves querying the RDF data to identify instances where the information represented in the Knowledge base contradicts the rules defined in the Rule base.

Also, our intelligent system uses SPARQL query to extract the information about KPIs from the KB. When starting the server, the SPARQL query is sent to the knowledge base to retrieve information, and it returns JSON file with the existing values.

```
import rdflib
import os

# graph for Rule base
g_rb = rdflib.Graph()
RB_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), "RB_V3_2.n3")
g_rb.parse(file=open(RB_path, mode="r"), format="text/n3")

# graph for Knowledge base
g_kb = rdflib.Graph()
KB_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), "KB_V3_2.n3")
g_kb.parse(file=open(KB_path, mode="r"), format="text/n3")


3 usages
def rb_sparql_query(condition='owl:oneOf'):
```

Solution components

In our pothole detection project, each component plays a crucial role in the overall functioning of the system. Here's a breakdown of these components and why they are necessary:

1. **Neural Network (YOLOv8):**
    - **Purpose**: YOLOv8 is at the heart of our system. It's a powerful neural network designed for object detection. In our case, it's trained to identify potholes.
    - **Why It's Necessary**: YOLOv8 is known for its speed and accuracy. It can process images quickly, which is essential for real-time detection. Its accuracy means we can reliably identify potholes, which is crucial for road safety.
2. **Database (SQLite):**
    - **Purpose**: SQLite serves as our database solution to store and manage data.
    - **Why It's Necessary**: It's lightweight, requires minimal setup, and works well for applications that don't need the full power of a large database system. This makes it ideal for storing data like pothole locations, user reports, and system logs.
3. **Server/Client Architecture (FastAPI <-> Telegram Bot/Web-client):**
    - **Purpose**: FastAPI acts as the server backend, handling requests and responses. It connects with clients like a Telegram Bot or a web interface.
    - **Why It's Necessary**: This architecture allows for efficient communication between the user interface (like Telegram Bot or a web client) and our server. FastAPI is known for its high performance and easy-to-use features, making it ideal for real-time data processing and response.
4. **Telegram Bot (AIOGram) for Admin’s purposes:**
    - **Purpose**: The Telegram Bot provides a user-friendly interface for reporting potholes and accessing system information as well as for admin’s purposes.
    - **Why It's Necessary**: It's accessible, easy to use, and allows for quick reporting and feedback. This enhances user engagement and provides a direct channel for data collection and user interaction. In the basic configuration the main Telegram bot is implemented in a single configuration based on a Docker together with the FastAPI server part and serves to manage a comprehensive solution and to perform tasks of a digital twins.

Each of these components is chosen for its specific strengths, and together, they create a robust, efficient, and user-friendly system for pothole detection and reporting. This system not only helps in identifying potholes but also in managing the data and information flow, thereby contributing to safer and better-maintained roads.

Multi-agent Architecture based on FastAPI

FastAPI is a modern, fast, web framework for building APIs with Python 3.7+ based on standard Python type hints. Here's a simple explanation of FastAPI, its advantages, and its main components. FastAPI is a high-performance framework that's primarily used for creating web APIs. It's built on top of Starlette for the web parts and Pydantic for the data parts. This makes FastAPI really quick and efficient.

Advantages of FastAPI:

1. **Speed**: FastAPI is one of the fastest frameworks for building APIs in Python, thanks to its underlying ASGI (Asynchronous Server Gateway Interface) support. This makes it suitable for high-performance applications.
2. **Ease of Use**: It's designed to be easy to use and intuitive. This means you spend less time reading documentation and more time writing your code.
3. **Automatic Documentation**: FastAPI generates documentation for your API automatically. It uses standards like OpenAPI and JSON Schema, which means you get interactive API documentation and web user interfaces.
4. **Data Validation and Serialization**: Thanks to Pydantic, FastAPI validates incoming data, making sure it's correct. This reduces bugs and errors in your code.
5. **Asynchronous Code Support**: With FastAPI, you can write asynchronous code using Python's `async` and `await` which can improve performance and efficiency, especially when dealing with many simultaneous API calls.

6. **Type Checking and Editor Support**: FastAPI leverages Python's type hints. This not only helps with data validation but also improves editor support, giving you features like auto-completion, error checking, and more.
7. **Security and Authentication**: FastAPI includes tools and integrations to handle authentication and authorization, providing security features out of the box.

Main Components of FastAPI:

1. **Pydantic for Data Handling**: It's used for data validation and settings management, which simplifies parsing and validating JSON data.
2. **Path Operations**: FastAPI uses path operations like GET, POST, PUT, and DELETE, which are essential for building APIs.
3. **Dependency Injection System**: FastAPI has a powerful, but easy-to-use dependency injection system. It allows you to have reusable dependencies that you can inject into your route functions.
4. **Starlette for the Web Layer**: This handles all the web interactions - routes, requests, and responses. It's what allows FastAPI to be a fully-functional web framework.
5. **Background Tasks**: FastAPI allows you to run functions in the background. This is useful for operations that need to happen after returning a response.
6. **WebSockets Support**: FastAPI also supports WebSockets, which allows for real-time communication between the client and the server.
7. **Testing**: FastAPI provides easy testing with Pytest, making it simple to check the behavior of your applications.
8. **Swagger UI**: Swagger UI is a great tool for anyone developing or using an API. It's like an instruction manual and control panel for your API, all in one.

In summary, FastAPI offers a perfect blend of performance, ease of use, and robust features, making it a popular choice for modern API development in Python. It's particularly suitable for projects where speed, data validation, and automated documentation are important.

Advantages of Swagger UI:

1. **Interactive Documentation**: Swagger UI automatically generates user-friendly documentation for your API. This documentation is interactive, meaning you can test API calls directly from it.
2. **Ease of Use**: It presents the API endpoints and their required parameters in a clear, readable format. This makes it easy for developers to understand and use the API, even if they're not familiar with its codebase.
3. **Real-time Testing**: You can execute API requests directly in the browser and see the responses immediately. This real-time interaction helps in testing and debugging the API without needing separate tools or writing additional code.

4. **Visualization**: Swagger UI provides a visual representation of the API, including all available endpoints and operations. This helps in getting a quick overview of the API's capabilities and structure.
5. **Compatibility**: It works with any API that uses an OpenAPI Specification. This means it's widely compatible with modern web APIs.
6. **Easy Integration**: Integrating Swagger UI with an API is usually straightforward, making it a low-effort solution for enhancing an API's usability.

Overall, Swagger UI makes it easier to develop, test, and use APIs, improving collaboration between backend developers, frontend developers, and even non-technical stakeholders.

Models based on YOLOv8

YOLOv8, short for "You Only Look Once, version 8" is an advanced algorithm developed by Joseph Redmon and Ali Farhadi for real-time object detection and classification. It's an improvement over its predecessors, offering better accuracy and speed. Popular in applications ranging from autonomous vehicles to security systems, YOLOv8 stands out for its fast and precise capabilities.

Architecture of YOLOv8:

- Backbone Network: YOLOv8's foundation, using a ResNet-50 Convolutional Neural Network (CNN) to extract high-level features from images.
- Neck Network: Connects the backbone to the head network, balancing accuracy and speed by refining object maps and resolutions.
- Head Network: Predicts the class and location of objects using anchor boxes and Intersection Over Union (IoU) optimization, including Non-Maximum Suppression (NMS) to filter overlapping bounding boxes.

- **Real-World Applications**: Consider how YOLOv8's technology can be beneficial in various industries for object detection tasks.
- **Parallel Structure for Predictions**: YOLOv8 operates on different scales simultaneously, enhancing accuracy and the detection of objects of various sizes.

Key Features of YOLOv8:

- Improved Accuracy: New network architecture and multi-scale predictions for better detection.
- Faster Processing: Streamlined network structure for quicker real-time detection.
- Versatile Usage: Suitable for detection, classification, and segmentation of objects in various applications.

"YOLOv8 is a game-changer in object detection, classification, and segmentation, offering unmatched speed, accuracy, and versatility."

Database (SQLite)

SQLite is a widely-used database engine, chosen for its simplicity and efficiency. Here's a brief overview of its advantages and the potential for easy migration to more powerful database systems as our system scales:

Advantages of SQLite:

1. **Lightweight and Self-contained**: SQLite is renowned for being a lightweight database that requires minimal setup and configuration. It's a self-contained system with no external dependencies, making it ideal for development and testing.
2. **Ease of Use**: It offers a simple and user-friendly way to store and manage data. SQLite databases are stored in a single file, making them easy to manage and transfer.
3. **Reliability**: Despite its simplicity, SQLite is reliable and powerful enough for many small to medium-sized applications. It supports standard SQL queries and is ACID-compliant, ensuring data integrity.
4. **Low Resource Requirement**: SQLite doesn’t require a separate server process or system to run, making it a low-resource and cost-effective option for handling data.

5. **Wide Application**: It's used in a variety of applications, from small local databases to larger applications due to its portability and ease of integration.

Prospects for Migration to More Powerful Databases:

As our system grows, we might reach a point where SQLite's capabilities are outpaced by our needs, especially in terms of handling concurrent accesses and managing large volumes of data. Here's how we can plan for an easy migration:

1. **Migrate to SQL-Based Databases**: Transitioning from SQLite to more robust SQL-based databases like PostgreSQL or MySQL is relatively straightforward because they all use SQL. This means most of the queries and database structures can be transferred with minimal changes.
2. **Modular Design**: By designing our system with modularity in mind, especially in how we interact with the database, we can ensure that switching out SQLite for a more powerful database system in the future will be simpler.
3. **Scalability Consideration**: We can start planning for scalability by considering how our data structures, indices, and queries will translate to a more complex database system. This forward-thinking approach will make the transition smoother when the time comes.
4. **Migration Tools and Services**: There are tools and services available that facilitate the migration process from SQLite to other database systems, handling much of the heavy lifting and minimizing downtime.
5. **Testing Migration**: As part of our development process, we can periodically test migration to more powerful databases to ensure compatibility and to understand the challenges and adjustments needed for a full-scale migration.

In summary, while SQLite serves as an excellent starting point for its simplicity and efficiency, we have a clear pathway for migrating to more powerful database solutions as our system's demands grow, ensuring scalability and robust data management.

Telegram Bot (AIOGram)

Choosing Telegram Bot (AIOGram) for our project comes with several advantages, especially considering the evolving role of Telegram as a "superapp," similar to China's WeChat. Here's why it's a great choice:

1. **Wide Reach and Accessibility**: Telegram has a massive, growing user base, making it accessible to a wide range of users. By integrating our system with Telegram, we can reach a larger audience easily.
2. **AIOGram for Asynchronous Programming**: AIOGram is an asynchronous framework for Telegram Bot API. This means it can handle many operations at once, making it efficient for real-time interactions, which is crucial for our system's responsiveness.
3. **User-Friendly Interface**: Telegram is known for its user-friendly and intuitive interface. Users familiar with Telegram will find it easy to interact with our bot, lowering the learning curve.
4. **Versatility of Telegram as a Superapp**: Following the trend of superapps like WeChat, Telegram is expanding its functionalities beyond messaging. It's becoming a one-stop

platform for various services, from payments to comprehensive apps, making it an ideal platform for hosting our bot.

5. **Notification and Instant Communication**: Telegram bots are excellent for sending notifications and instant updates to users. This feature can be utilized for real-time alerts about pothole detections.
6. **Security and Privacy**: Telegram is known for its strong focus on security and privacy, which is essential when handling user data and sensitive information in our project.
7. **Customizable and Scalable**: Telegram bots can be highly customized and scaled according to the needs of the project, whether it's for handling a small user group or a city-wide deployment.
8. **Cost-Effective**: Using Telegram Bot is a cost-effective solution as it doesn't require additional infrastructure for basic messaging and interaction capabilities.

AIOGram is a popular asynchronous framework for Telegram Bot API, and it offers several advantages, especially for building high-performance and responsive bots. Here's a brief rundown of its key benefits:

1. **Asynchronous Capabilities**: AIOGram is built to support asynchronous programming. This means it can handle multiple tasks and operations simultaneously, leading to faster response times and better handling of concurrent users or requests.

2. **Efficient Handling of Real-Time Data**: With its asynchronous nature, AIOGram is excellent for bots that require real-time data processing, making it ideal for applications like instant notifications, live updates, and interactive user communications.
3. **Scalability**: AIOGram allows for easy scaling of Telegram bots. Whether you're dealing with a small group of users or a large audience, AIOGram can manage increased workloads efficiently.
4. **User-Friendly API**: AIOGram provides a straightforward and intuitive API, making it easier for developers to create complex bot functionalities with less effort and more readable code.
5. **Rich Feature Set**: AIOGram supports a wide range of Telegram Bot API features, including custom keyboards, inline queries, and various bot commands, allowing for rich and interactive bot experiences.
6. **Community and Support**: Being a popular framework, AIOGram has a strong community and good documentation, which makes it easier for developers to find help and resources when needed.

AIOGram's combination of asynchronous processing, scalability, and a rich feature set makes it an excellent choice for developing responsive and feature-rich Telegram bots. By leveraging Telegram Bot, we are not just using a messaging platform; we are tapping into a rapidly growing ecosystem that offers a range of functionalities and a wide user base. This makes our system more accessible, efficient, and user-friendly.

🔹 **Situational Uncertainty -** situational uncertainty arises when the system lacks sufficient data about the current situation. **Strategies**:

- **Use of Pre-trained Models**: To mitigate data scarcity, the system can utilize pre-trained models that have been trained on large, diverse datasets. This approach helps in making reliable predictions even with limited data specific to the current context.
- **Data Augmentation Methods**: Implementing techniques to artificially expand the dataset, like rotating, zooming, or altering lighting conditions in images, can enhance the model's ability to generalize from limited data.
- **Continuous Monitoring and Feedback**: Establishing a system of ongoing monitoring and feedback allows for the constant gathering and integration of new data, improving the model's performance over time.
- **Cross-Functional Collaboration**: Encouraging collaboration among various departments can pool different types of expertise and resources, helping to address data gaps effectively.
- **Reserve Communication Channels**: Setting up alternative communication channels ensures stakeholders are informed and can contribute to resolving data-related issues promptly.

🔹 **Algorithm uncertainty** – Algorithm uncertainty occurs when the system lacks a specific algorithm to process incoming data or cases with a command which does not exist in the code. **Strategies**:

- **Fallback Mechanisms**: In cases where the system encounters an unknown command or data format, a fallback mechanism, such as ignoring the unrecognized input and continuing operation, ensures system stability.

- **Passive protection:** At this stage of development of our expert system, it can process the most types of photo and video formats. If an unknown type or damaged file is received, the system will ignore it and continue functioning.
- **System Resiliency Planning**: Developing plans for scenarios where existing algorithms are insufficient. This may include creating guidelines for quickly developing or integrating new algorithms as needed.

🔹 **Model uncertainty** – model uncertainty refers to situations where the system lacks a model of object behavior, often due to changes in equipment or operational conditions. **Strategies**:

- **Comparative Analysis**: When new equipment is introduced, compare its performance with previous setups. This helps in understanding the behavior of the new equipment and its impact on the system.
- **System Tuning and Modification**: Analyze which parts of the system need adjustments. For instance, if a new camera is installed and accuracy decreases, it might be due to a default setting like low resolution. Adjusting these settings can optimize performance.
- **Update Procedures**: Establish procedures for regularly updating the model in response to changes in equipment or operating conditions. This includes retraining the model with new data and adjusting system parameters as necessary.

By implementing these strategies, the pothole detection system can effectively handle various types of uncertainties, ensuring it remains efficient, accurate, and adaptable to changing conditions and requirements.

3. Application development

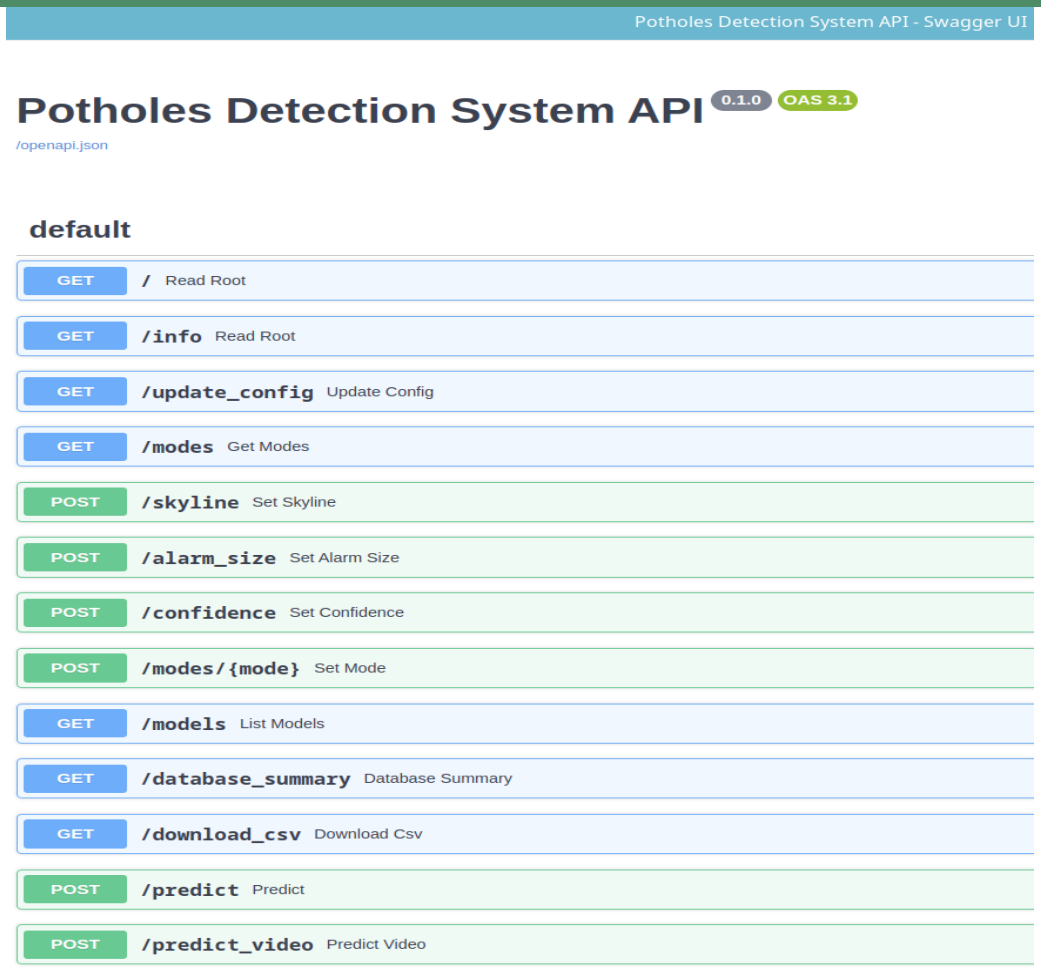
Information regarding all developed modules is included in the Section 2.

FastAPI stands out as a highly efficient server solution, renowned for its speed and user-friendly design. Its performance is notably impressive, owing to the Starlette framework that adeptly manages asynchronous operations, making it a top choice among Python web frameworks. The ease of use is another significant aspect; FastAPI's intuitive nature caters well to both novice and seasoned developers, streamlining the development process. Additionally, the framework's integration of Python type hints enhances code quality and readability while ensuring automatic request validation. This feature significantly reduces the likelihood of errors and bugs in APIs, bolstering overall system reliability.

The most important code listings

Our project on GitHub is available by the following link: LINK TO GITHUB

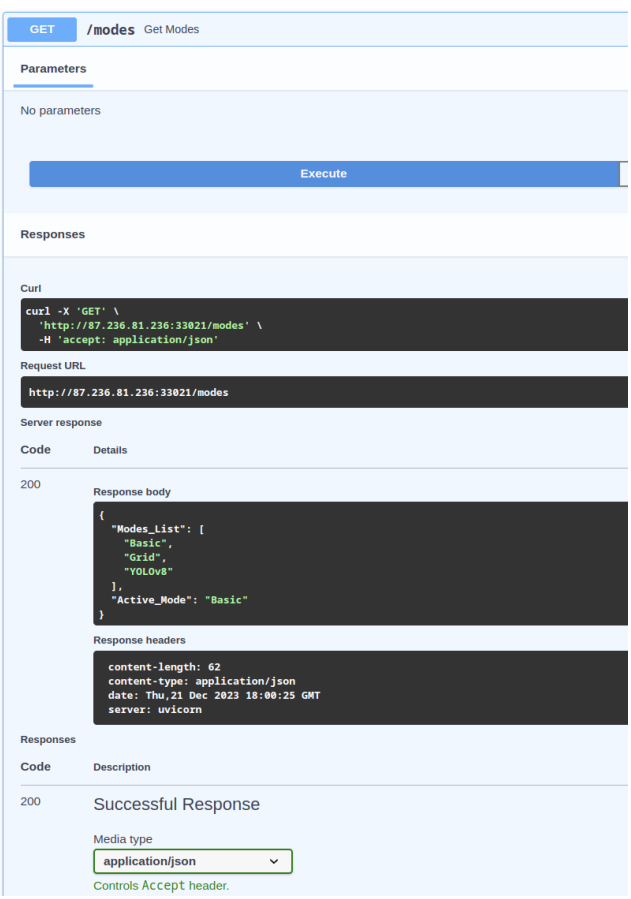
Screenshots of Swagger UI are presented below:



Complementing FastAPI's capabilities, Swagger UI introduces a layer of interactivity and convenience, particularly in API documentation and testing. This tool automatically generates a web-based user interface for the API, reflecting updates in real-time based on FastAPI's code. This feature not only simplifies the testing and debugging process by allowing direct interaction with the API but also eliminates the need for writing additional documentation, saving considerable time and effort. For Python developers, Swagger UI coupled with FastAPI expedites the development of simple requests, enabling quick experimentation and response analysis. Moreover, it fosters improved collaboration between frontend and backend teams by providing easily accessible and understandable documentation, facilitating a smoother development cycle and enhancing team synergy. Together, FastAPI and Swagger UI create a robust environment for both rapid development and effective long-term project management.

One of the advantages of Swagger is that it provides simple commands for accessing endpoints, which can be easily executed even from the command line.

```
curl -X 'POST' \
  'http://xx.xx.xx.xx:port/predict' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@img00.jpg;type=image/jpeg' \
  -F 'mdl_name=./default/best.pt'
```



GET /modes Get Modes

Parameters

No parameters

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://87.236.81.236:33021/modes' \
  -H 'accept: application/json'
```

Request URL

```
http://87.236.81.236:33021/modes
```

Server response

Code	Details
200	Response body

```
{
  "Modes_List": [
    "Basic",
    "Grid",
    "YOLOv8"
  ],
  "Active_Mode": "Basic"
}
```

Response headers

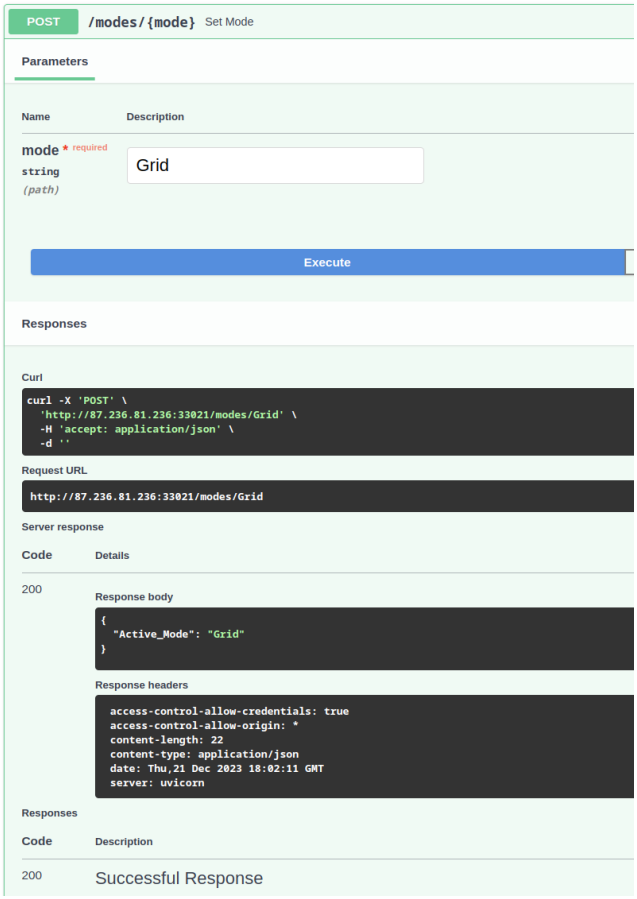
```
content-length: 62
content-type: application/json
date: Thu,21 Dec 2023 18:00:25 GMT
server: uvicorn
```

Responses

Code	Description
200	Successful Response

Media type: application/json

Controls Accept header.



POST /modes/{mode} Set Mode

Parameters

Name	Description
mode * required string (path)	Grid

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://87.236.81.236:33021/modes/Grid' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://87.236.81.236:33021/modes/Grid
```

Server response

Code	Details
200	Response body

```
{
  "Active_Mode": "Grid"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
content-length: 22
content-type: application/json
date: Thu,21 Dec 2023 18:02:11 GMT
server: uvicorn
```

Responses

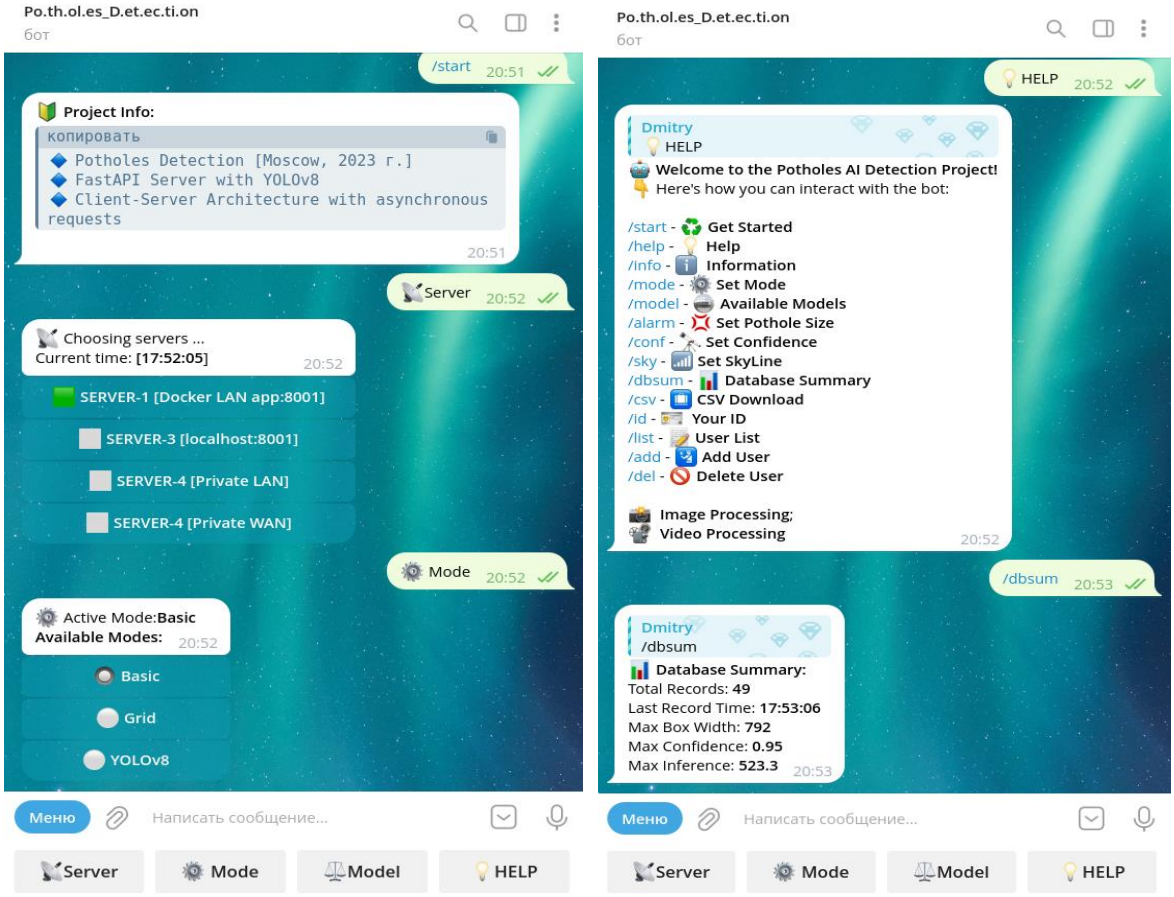
Code	Description
200	Successful Response

Welcome | simple_request.py

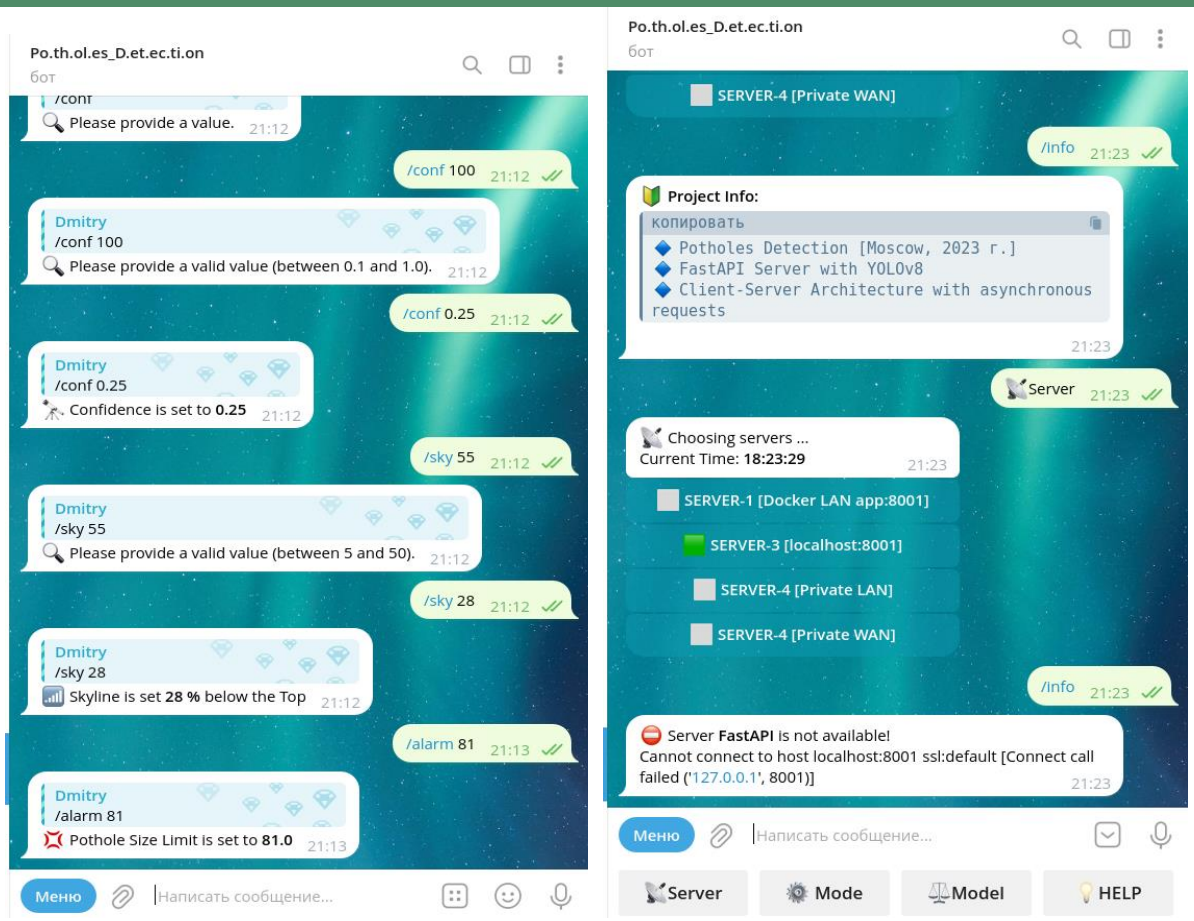
simple_request.py > ...

```python
import requests
from pathlib import Path

# URL of your FastAPI server
url = 'http://localhost:8000/predict'

# Path to the image file you want to upload
file_path = Path('/path/to/your/image.jpg')

# Optional: Model name, if you want to specify a particular model for prediction
model_name = 'your_model_name.pt'  # Replace with your model name or leave as None

# Prepare the file in the correct format for uploading
files = {'file': (file_path.name, open(file_path, 'rb'), 'image/jpeg')}

# Prepare additional data if needed
data = {'mdl_name': model_name} if model_name else {}

# Make the POST request to the FastAPI endpoint
response = requests.post(url, files=files, data=data)

# Check the response
if response.status_code == 200:
    print("Success:", response.json())
else:
    print("Error:", response.status_code, response.text)

```

The Python script provided is a practical example of how to create and send HTTP requests to a FastAPI server, specifically targeting an endpoint for image processing. It utilizes the popular requests library in Python to interact with the server, demonstrating a simple yet effective way to communicate with a FastAPI endpoint. The script is tailored to handle a common use-case: sending an image file to a server for processing. It's designed to be user-friendly, with clear definitions for the server's URL and the path to the image file. Additionally, it offers the flexibility to specify an optional model name, allowing users to choose a particular model for processing the image.
In terms of functionality, the script prepares the image file for upload and constructs a POST request to send this file to the FastAPI endpoint. This process includes setting the correct MIME type for the file and packaging any additional data, like the model name, if provided. Upon sending the request, the script handles the server's response effectively. It checks the status code to determine if the request was successful and then either displays the result or error information. This approach exemplifies a straightforward method for external applications or services to interact with FastAPI endpoints, highlighting the framework's suitability for tasks like image processing. The script's structure and its ability to facilitate easy communication with FastAPI endpoints make it an excellent tool for developers looking to integrate and automate interactions with web servers in their applications.

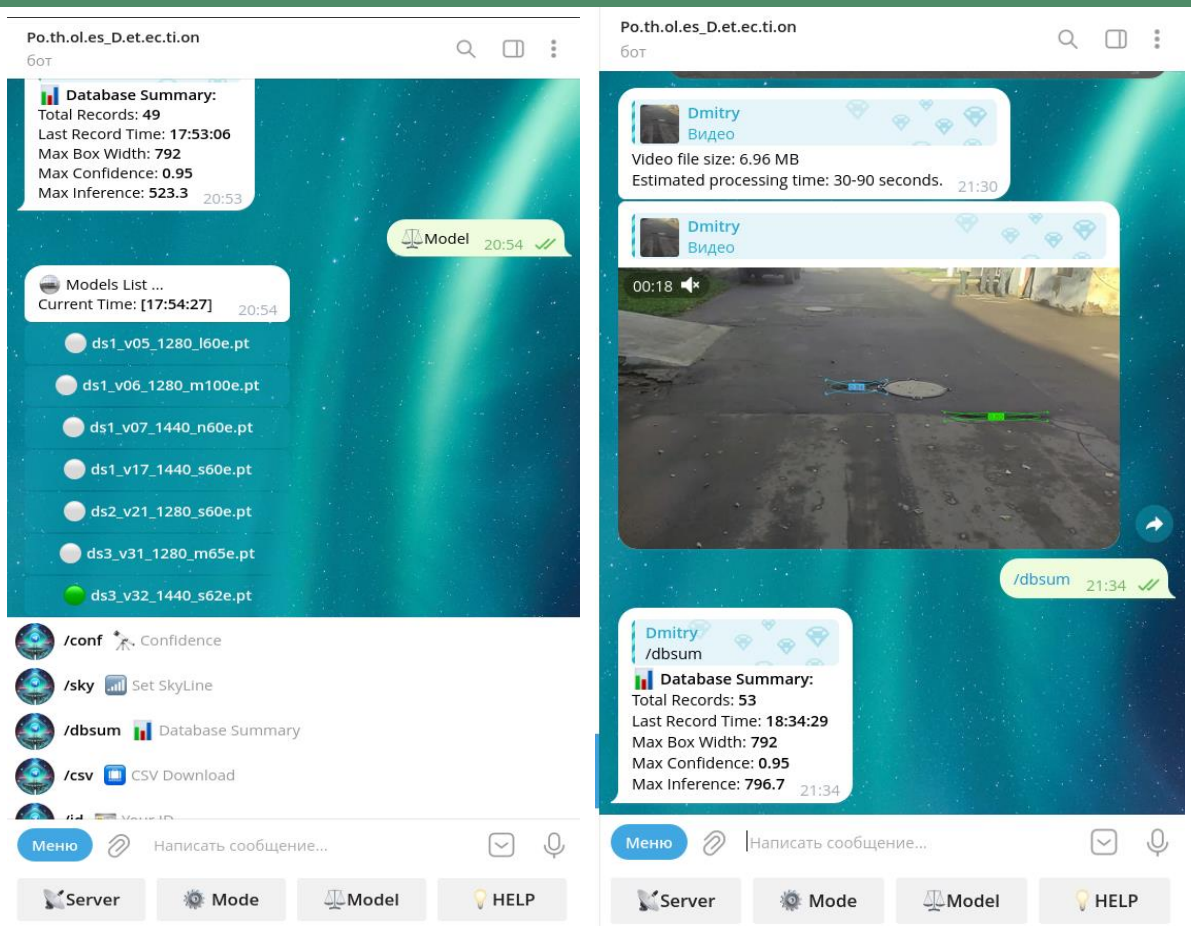


In our project, the FastAPI server and the main Telegram bot are deployed in two separate Docker containers, interconnected by a virtual network. This setup offers notable advantages, such as speed and the ability to function without a dedicated external IP address. The primary Telegram bot acts as an administrative tool, enabling the input of basic parameters. Other clients can directly interact with the system without needing to specify intricate details and promptly receive processed images based on the predefined parameters.



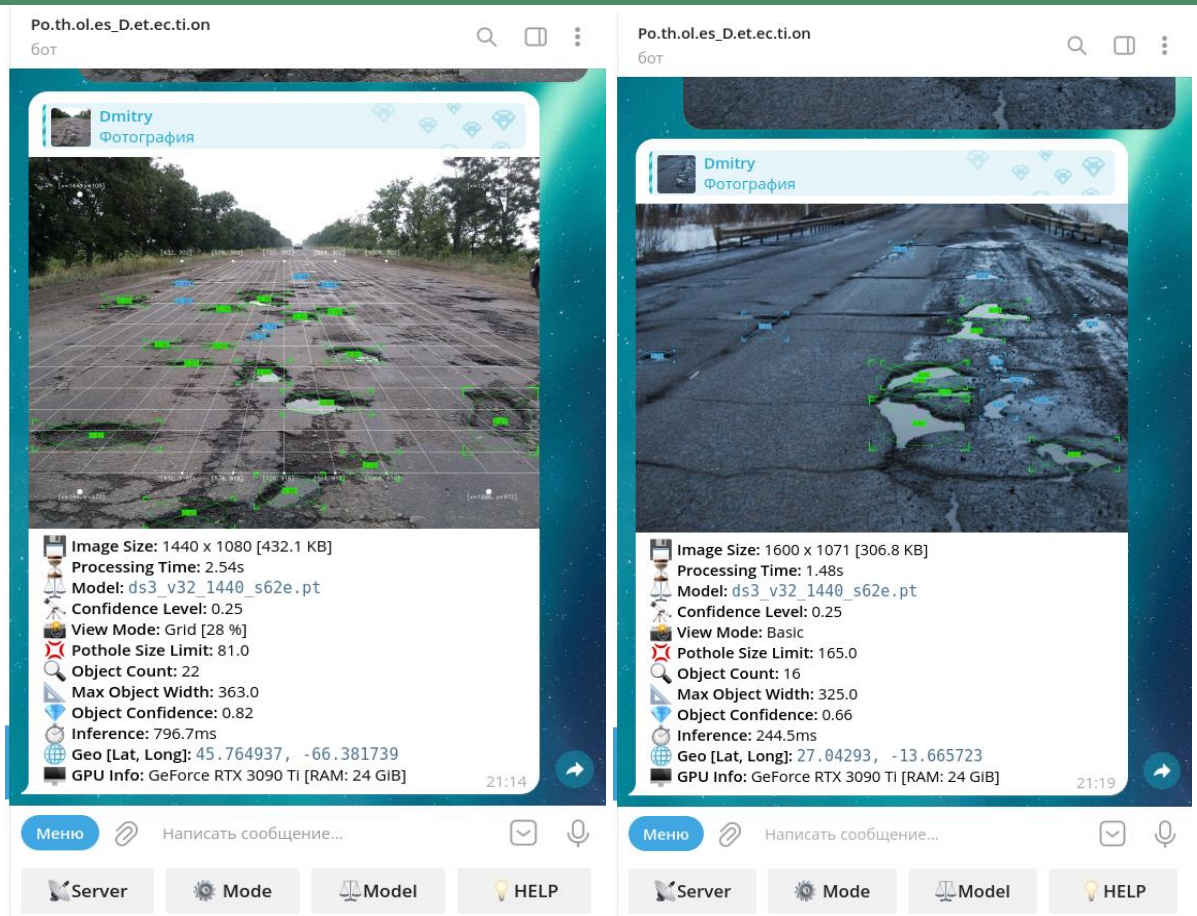
At the launch of this solution, necessary Key Performance Indicators (KPIs) are loaded from a SPARQL knowledge base into a JSON file. These KPIs are then used to monitor and track the system's performance. In case of any issues with the FastAPI server, the Telegram bot is equipped with a mechanism to switch to alternative paths or backup servers. This ensures continuity and reliability of the service. There are four basic buttons within the bot for user interaction: selecting the server, choosing the interaction mode, picking the model for processing, and accessing a help guide that describes all the bot's commands. This user-friendly interface simplifies the process for clients, making the system not only efficient but also accessible for users with varying levels of technical expertise.

The Potholes AI Detection Project features a highly interactive and multifunctional Telegram bot designed to assist users in detecting potholes using AI technology. This bot provides a range of commands, each tailored to enhance the user experience and facilitate efficient pothole detection and analysis.
Upon initiation with the '/start' command, users are introduced to the bot and can begin their interaction. For assistance and clarification on the bot's functionalities, the '/help' command provides necessary guidance. The '/info' command offers detailed information about the project and the bot's capabilities. Users can adjust operational parameters such as the detection mode with '/mode', choose from a variety of available AI models using '/model', set the size threshold for pothole detection with '/alarm', and specify the confidence level for AI detection accuracy with '/conf'. Additionally, the '/sky' command allows users to set the SkyLine parameter, which could be crucial for image analysis.



The bot also includes commands for database interaction and data management. With '/dbsum', users can access a summary of the database, while '/csv' allows for the downloading of data in CSV format. For user management, the bot provides '/id' to display the user's ID, '/list' to view a list of users, '/add' to add new users, and '/del' to remove existing users.
Moreover, the bot extends its capabilities to include both image and video processing, enabling users to analyze various media formats for pothole detection. This comprehensive set of features makes the bot an invaluable tool for anyone involved in road maintenance, research, or interested in the applications of AI in real-world scenarios.

In the pothole detection process using the YOLOv8 model, the system is designed to highlight detected potholes in different colors based on the set parameters, specifically emphasizing the potential danger levels. This color-coding feature enhances the visual interpretation of the results, making it easier for users to quickly assess the situation.
When the model identifies a pothole that may pose a danger, it is outlined in a bright green color. This distinct color choice is intended to immediately draw attention to more critical findings, signifying areas that might require urgent attention or intervention. The use of bright green as an indicator for potentially hazardous potholes ensures that these areas are not overlooked and can be prioritized for repair or further inspection.
Other detected potholes, which are deemed less critical based on the model's analysis and the predefined parameters, are marked in different colors. This differentiation helps in categorizing the potholes based on severity and potential risk. Users can quickly discern which potholes are of immediate concern and which are less urgent, aiding in efficient decision-making for road maintenance and safety measures.



In the FastAPI application, some distinct API endpoints have been developed to modify specific configuration settings, each handling POST requests for different parameters. The first endpoint, /skyline, is dedicated to adjusting the 'Skyline' value within the application's settings. It rigorously checks that the input is an integer falling between 5 and 50. Should the input stray from this range, the system defaults the value to 28 and issues an HTTP 400 error, flagging the input as "Invalid skyline". The endpoint then responds by returning the set or defaulted 'Skyline' value.

The second endpoint, /alarm_size, focuses on setting the 'Alarm Size' parameter. It is designed to accept floating-point numbers ranging from 10.0 to 500.0. Similar to the first endpoint, it enforces this range strictly; any value outside of this specified range triggers an HTTP 400 error, specifying "Invalid Alarm Size value". Upon successful validation, it updates the configuration with the new 'Alarm Size' value and conveys this to the user.

Lastly, the /confidence endpoint is responsible for establishing the 'Confidence' level in the system's configuration. It accepts floating-point values, but only within the strict bounds of 0.1 to 1.0. If a value outside this bracket is received, the endpoint responds with an HTTP 400 error, citing an "Invalid confidence value". Upon receiving a valid input, it updates the 'Confidence' level in the configuration and provides the user with this updated value. Collectively, these endpoints play a pivotal role in allowing users to personalize and refine key operational aspects of the FastAPI application, ensuring values remain within acceptable and effective ranges to maintain the system's integrity and efficacy

fastapi.py > set_confidence

Codeium: Refactor | Explain | Generate Docstring | ×

```python
@app.post("/skyline")
def set_skyline(skyline: str):
    config = load_config()  # Load actual configuration
    if int(skyline) >= 5 or int(skyline) <= 50:
        config['Skyline'] = int(skyline)  # Set skyline
    else:
        config['Skyline'] = 28
        raise HTTPException(status_code=400, detail="Invalid skyline")
    save_config(config)
    return {"Skyline": skyline}
```

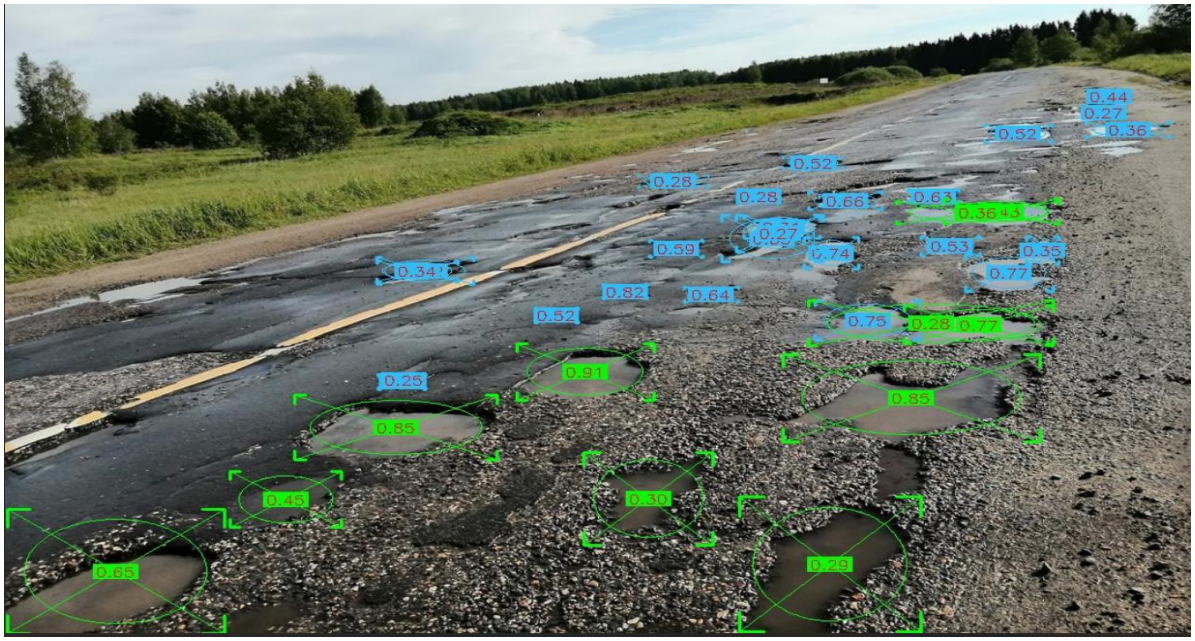
Codeium: Refactor | Explain | Generate Docstring | ×

```python
@app.post("/alarm_size")
def set_alarm_size(alarm_size: float):
    config = load_config()  # Load actual configuration
    if 10.0 <= alarm_size <= 500.0:
        config['Alarm_Size'] = alarm_size  # Set Alarm Size
    else:
        raise HTTPException(status_code=400, detail="Invalid Alarm Size value")
    save_config(config)  # Save Config
    return {"Alarm_Size": alarm_size}
```

Codeium: Refactor | Explain | Generate Docstring | ×

```python
@app.post("/confidence")
def set_confidence(confidence: float):
    config = load_config()  # Load actual configuration
    if 0.1 <= confidence <= 1.0:
        config['Confidence'] = confidence  # Set Confidence Level
    else:
        raise HTTPException(status_code=400, detail="Invalid confidence value")
    save_config(config)  # Save Config
    return {"Confidence": confidence}
```

After processing an image using the YOLOv8 model, the bot provides a comprehensive set of information that details various aspects of the analysis. This output is designed to give users a clear understanding of the processing parameters and results.



The color-coded visualization provided by the system is an integral part of its user-friendly interface. It not only adds clarity to the detection results but also aids in effectively communicating the severity of each detected pothole, ensuring that users can make informed decisions regarding road safety and maintenance priorities.

The bot starts by displaying the size of the processed image, including its dimensions and file size, for instance, "2276 x 1272 [165.8 KB]". This is followed by the processing time, indicated in seconds, such as "1.85s", providing insight into the efficiency of the model. The specific model used for the analysis, like "ds3_v32_1440_s62e.pt", is then listed, giving users information about the AI tool behind the processing.

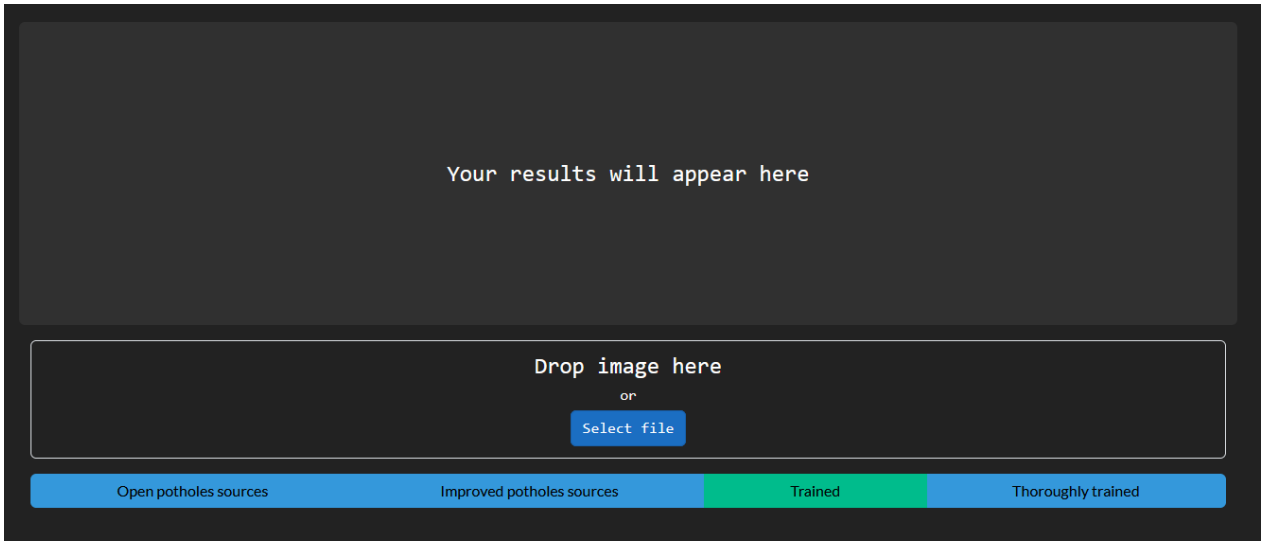
bot.py > process_image

```python
async def process_image(message: types.Message):
    API_URL = os.getenv("API_URL")  # Remote server URL for FastAPI (for local Bot: http://app:8001)
    mdl_name = os.getenv("mdl_name")  #
    file_id = message.photo[-1].file_id  #
    file = await bot.get_file(file_id)  #
    file_path = file.file_path  #

    image_data = await bot.download_file(file_path)  #

    url = f'{API_URL}/predict'  #

    timeout = ClientTimeout(total=30)  #
    async with aiohttp.ClientSession(timeout=timeout) as session:  #
    # async with ClientSession() as session:  #
        form = aiohttp.FormData()  #
        form.add_field('file', image_data, filename='input_image.jpg', content_type='image/jpg')  #
        form.add_field('mdl_name', mdl_name)
        try:
            # async with session.post(url, data=form, timeout=30) as response:
            async with session.post(url, data=form) as response:
                if response.status == 200:
                    response_json = await response.json()  #
                    img_str = response_json['image']
                    results = response_json['results']  #
                    # Create additional information string with additional data
                    add_info = f"\n<b>💾 Image Size:</b> {results['image_size']}" \
                               f"\n<b>⏳ Processing Time:</b> {results['processing_time']}s" \
                               f"\n<b>⚖️ Model:</b> <code>{results['model_name']}</code>" \
                               f"\n<b>🔭 Confidence Level:</b> {results['conf']}" \
                               f"\n<b>📸 View Mode:</b> {results['mode']}" \
                               f"\n<b>💢 Pothole Size Limit:</b> {results['alarm_size']}" \
                               f"\n<b>🔍 Object Count:</b> {results['object_count']}" \
                               f"\n<b>📐 Max Object Width:</b> {results['max_box_width']}" \
                               f"\n<b>💎 Object Confidence:</b> {results['confidence']}" \
                               f"\n<b>⏱️ Inference:</b> {results['inference']}ms" \
                               f"\n<b>🌐 Geo [Lat, Long]:</b> <code>{results['latitude']}, {results['longitude']}</
                               f"\n<b>🖥️ GPU Info:</b> {results['gpu_info']}"
                    output_image_data = BytesIO(base64.b64decode(img_str))
                    output_image_data.seek(0)
                    await message.reply_photo(photo=output_image_data, caption=add_info, parse_mode="HTML")
        except KeyError as key_error:
            print(f"Key error: {key_error}")
```

Key parameters set for the image analysis are also shown. These include the confidence level, represented as a percentage like "0.65", which reflects the accuracy threshold for object detection. The view mode, such as "Grid [10 %]", indicates the format in which the image was analyzed. The pothole size limit, like "141.0", specifies the minimum size for pothole detection in the image. Crucial results from the analysis include the object count, highlighting how many potential potholes were detected, such as "3". The maximum width of any detected object, like "507.0", provides a sense of the largest pothole size identified. Additionally, the highest confidence level for an object, indicated as a percentage like "0.85", shows the certainty of the largest pothole's detection.

The inference time, such as "444.4ms", demonstrates the time taken by the model to analyze the image. Geographical coordinates, displayed as latitude and longitude, like "-86.644018, 103.765661", may also be provided, indicating the location related to the image. Finally, the GPU information, for example, "GeForce RTX 3090 Ti [RAM: 24 GiB]", offers technical details about the hardware used in processing, highlighting the computational power involved in the analysis. This detailed summary provided by the bot gives users a thorough understanding of both the process and the results of the image analysis for pothole detection.

In addition to the Telegram bot, our project also features a web application that functions as a client. This web application is designed to interact with the FastAPI server's endpoints. The core interaction between the web client and the FastAPI server involves the submission and retrieval of images. Specifically, the web application allows users to upload images of road surfaces, which are then sent to the FastAPI server. On the server side, these images are processed by a YOLOv8 model, which is specialized in detecting potholes on roads. Once the image is processed, the results, including the detected potholes, are sent back to the web application. This seamless integration between the web client and the FastAPI server provides a user-friendly interface for effective pothole detection and analysis.

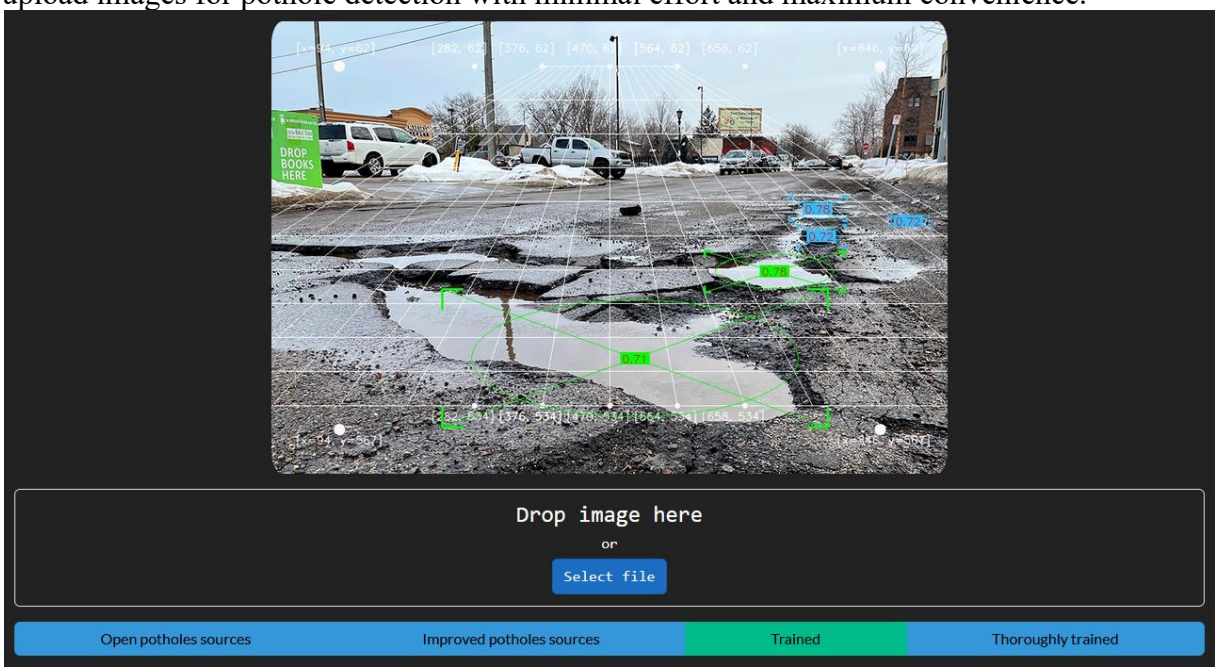


In this web application, users have a convenient and user-friendly interface for uploading images for processing. There are two primary methods for image upload:

- **Selecting from File List**: Users can choose an image from their device by clicking on a designated area or button within the application. This action typically opens the device's file explorer, allowing users to navigate through their folders and select the desired image file from their stored documents.
- **Drag-and-Drop Feature**: Alternatively, the application offers a drag-and-drop functionality, which is an intuitive and quick method for uploading images. Users can simply drag an

image file from their device and drop it into a specified area on the web application's interface. This area is usually highlighted or marked with instructions like "Drop files here" to guide users.

Both methods are designed to provide ease of use, catering to different user preferences. The select-from-list option offers a more traditional approach, while the drag-and-drop feature enhances the user experience with a more interactive and modern interface. These options ensure that users can upload images for pothole detection with minimal effort and maximum convenience.



The use of React for web application development brings several key advantages:

- **Component-Based Architecture**: React's modular structure allows developers to build reusable and manageable components, promoting efficient code organization and maintainability. This component-based approach enables easier scaling and updating of applications.
- **Rich Ecosystem and Community Support**: React benefits from a robust ecosystem of tools and libraries, as well as strong community support. This wealth of resources aids developers in solving common problems and implementing advanced features.
- **High Performance**: React's virtual DOM (Document Object Model) improves application performance. It optimizes rendering processes, ensuring that only components that have changed are re-rendered, leading to faster and more efficient performance, especially in dynamic, data-intensive applications.
- **Strong Backing by Facebook**: Being developed and maintained by Facebook, React receives regular updates and enjoys long-term support, making it a stable and reliable choice for web development.
- **Versatility and Flexibility**: React's design and capabilities make it a versatile tool for building a wide range of web applications, from simple static pages to complex, interactive user interfaces.

In addition to these benefits, React can be used to create Progressive Web Apps (PWAs). PWAs are web applications that offer a user experience similar to native mobile apps. By leveraging React, developers can build PWAs that are capable of functioning across different platforms, including Android and iOS. This approach allows for the creation of app-like experiences, complete with offline capabilities, push notifications, and a home screen icon, without the need for separate native app development for each platform. This cross-platform compatibility significantly reduces development time and costs while providing a seamless and engaging user experience.

4. Conclusion

When reflecting on a project like the Pothole Detection using YOLOv8 and FastAPI, it's important to consider both the successes and challenges, as well as the insights gained and future plans.

Successful Implementations

- **Effective Pothole Detection**: We successfully integrated the YOLOv8 model, enabling accurate and efficient detection of potholes in various road conditions. This was a key achievement, as it forms the core of our project's functionality.
- **User-Friendly Interface**: The creation of an intuitive interface through a Telegram bot and a web application was another significant success. These platforms allow users to easily interact with our system, upload images or videos, and receive processed results.
- **Real-Time Processing and Feedback**: Implementing FastAPI for real-time image processing and immediate feedback was a crucial aspect we managed to achieve. This ensures that the system operates efficiently and responds quickly to user requests.

Challenges and Unsuccessful Attempts

- **Handling Varied Lighting and Weather Conditions**: One of the problems we faced was the inconsistency of the model's characteristics under different lighting and weather conditions. Achieving consistent accuracy in all scenarios remains a work in progress and requires more careful selection of the dataset and image labeling.
- **Scalability Issues**: As our user base has grown, we have encountered some scalability issues, especially when processing simultaneous requests for large images and videos, which we intend to eliminate in future updates. At the very beginning, we mentioned that the project provides for scaling and load balancing on multiple servers with an increase in the number of requests.

Insights Gained

- **Importance of Data Quality**: The learning curve was steep regarding data quality and diversity. The effectiveness of the model hinges significantly on these factors, shaping our approach towards future data collection and model training processes. Through training YOLOv8 on various datasets, we discovered the profound impact of data quality, quantity, and the balance between different types of objects. Focusing on the single class of 'potholes', we observed that the model's ability to detect typical or specific types of potholes varied based on the dataset composition. For example, skewing the dataset towards images of wet or snow-covered roads altered the model's detection capabilities, clearly demonstrating how the weights in the model training are influenced. We also realized the importance of avoiding unnatural augmentations, such as 180-degree flipped images of vehicles or road surfaces, which can detrimentally affect the model's learning process. The more precise and balanced the image selection and annotation are, the higher the quality of the results and the confidence level of the detections.
- **Importance of API in Progressive Development**: The project underscored the significance of using APIs for phased designing and incremental functionality additions, especially after the system becomes operational. In the initial stages, the server could be accessed using simple methods like command-line tools (e.g., curl), basic Python scripts in Google Colab or an IDE, and even through Swagger UI without writing a single line of code. This flexibility proved highly advantageous for team collaboration, system debugging, and developing the client-side of the application.

- **User-Centric Design**: User feedback illuminated the crucial role of a user-friendly interface and the necessity for results to be easily comprehensible. This insight has profoundly influenced our design approach and feature implementations, ensuring that the system is not only efficient but also accessible and intuitive for users.

Future Plans and Developments

- **Enhancing Model Robustness**: We plan to retrain the model with a more diverse dataset, including images under various environmental conditions, to improve its accuracy and reliability. There is also an idea to use various trained models on highly specialized datasets - in winter with priority on snow-covered roads, in rainy weather. weather with priority on wet asphalt and puddle pits, and we also plan to consider the option of night roads in poor lighting.
- **Scaling the System**: To address scalability, we aim to optimize our server infrastructure and explore more efficient data processing techniques. One of the possible options may be effective in sending a short video to the processing model instead of a single image to pre-select a frame with better characteristics and clarity.
- **Expanding Functionality**: Future developments include adding additional features to the bot, such as custom alerts when serious potholes are detected, and integration with mapping services for better geographical tracking of road conditions. It is also planned to use additional sensors to measure the depth of potholes on roads.

In summary, while we have achieved key milestones in pothole detection and user interaction, we acknowledge the challenges faced, particularly in model consistency and system scalability. The insights gained from this project are invaluable and will significantly shape our future endeavors to enhance the system's effectiveness and expand its capabilities.

4. Conclusion (For corporate Inf Syst)

The project outlined in the document involves creating a multi-agent system for detecting potholes using FastAPI and YOLOv8. The system includes various clients such as a Telegram Bot, web applications, and simple clients connected to a server through an API. The goal is to identify potholes in real-time with high accuracy and processing speed. Key tasks include collecting and curating a dataset, training the YOLOv8 model, developing a FastAPI backend, testing under various conditions, and managing the system via a Telegram bot.

To calculate labor costs for this project, consider the following aspects:

1. Dataset Collection and Curation: Time and resources spent on gathering and preparing the road image dataset.
2. Model Training (YOLOv8): Hours required for training the model, including the time of data scientists and computational resources.
3. Backend Development (FastAPI): Development time for creating the backend, including coding, integration, and testing.
4. System Testing: Hours spent on testing the system under different conditions.
5. Management and Client Interface Development: Development time for the Telegram bot and other client interfaces.

For each aspect, you would assign an estimated number of hours and multiply it by the hourly rate of the professionals involved (data scientists, developers, testers). This will give you the total labor cost for each component, which can then be summed up to estimate the overall labor cost for the project.

5. Appendices

References

1. **Telegram-Bot https://t.me/po_th_ol_es_d_et_ec_ti_on_bot**
2. Sugumaran V. (2016). Semantic technologies for enhancing knowledge management systems. Elsvier (https://doi.org/10.1016/B978-0-12-805187-0.00014-0).
3. Foster, E. (2023, August 2). Choosing between a rule-based vs. machine learning system. Enterprise AI (https://www.techtarget.com/searchenterpriseai/feature/How-to-choose-between-a-rules-based-vs-machine-learning-

system#:~:text=In%20AI%2C%20rule%2Dbased%20systems,data%20and%20produce%20a%20result).
4. The W3C SPARQL Working Group. (2013, March 21). SPARQL 1.1 Overview. (https://www.w3.org/TR/sparql11-overview/).
5. State Standard Of Russia (2000). Methodology Of Functional Analysis Modeling IDEF0 (https://advanced-quality-tools.ru/assets/idef0-rus.pdf).
6. CVAT. Open Data Annotation Platform (https://www.cvat.ai/).
7. Habr (2022). CVAT – Marking instructions (https://habr.com/ru/articles/677484/).
8. Ultralytics Inc. (2023). Model Training with Ultralytics YOLO (https://docs.ultralytics.com/modes/train/#arguments).
9. YOLOv8 (https://yolov8.com/).
10. Keras (2023). Efficient Object Detection with YOLOV8 and KerasCV (https://keras.io/examples/vision/yolov8/).
11. Smart Vision Europe (2023). What is the CRISP-DM methodology (https://www.sv-europe.com/crisp-dm-methodology/).
12. FastAPI (2023). Official website (https://fastapi.tiangolo.com/).
13. Telegram LLC (2023). Official bot API documentation (https://core.telegram.org/schema).
14. Docker (2023). Official website (https://www.docker.com/).
15. Microsoft Learn (2023). Docker images for ASP.NET Core (https://learn.microsoft.com/ru-ru/aspnet/core/host-and-deploy/docker/building-net-docker-images?view=aspnetcore-8.0).