# AISoC 2025 Proposal

## Unified API Creation for DIC Projects

By Team Stack Overload

## About Us

We are Stack Overload, a team of four dedicated students passionate about AI-driven solutions for workplace safety. Below are basic details for each member, to be customized with individual achievements.

- **Member 1(Leader):**
  - Name: Jasjot Singh
  - Degree: Bachelor of Technology in Computer Science and Engineering
  - University: UIET, Chandigarh
  - Email: jasjotsingh080405@gmail.com
  - LinkedIn: https://www.linkedin.com/in/jasjot-singh-a0916b281
  - GitHub: https://github.com/JASJOTSINGH08
  - Experience: I have hands-on experience in full-stack development, with a strong focus on React.js for frontend and Java for backend systems. I've successfully completed internships where I developed secure, scalable chat applications and responsive web interfaces, integrating API services, database authentication, and real-time communication features. Additionally, I've worked on AI-driven projects such as 2D to 3D image conversion, applying both frontend technologies and backend logic to deliver practical, innovative solutions.

- **Member 2:**
  - Name: Shivanshi Prashar
  - Degree: Bachelor of Technology
  - University:UIET, Chandigarh
  - Email: shivanshiprashar@gmail.com

- LinkedIn: www.linkedin.com/in/shivanshi-prashar-a356a92b8
- GitHub: Shivanshi-Prashar
- Experience:I have hands-on experience in backend and AI development, using Python, SQL, and deep learning frameworks like TensorFlow and PyTorch. During my internships, I contributed to scalable applications involving real-time data processing, API integration, and database management. I’ve contributed to build intelligent systems including a real-time Indian Sign Language (ISL) translator with Non-Manual Features (NMFs) using MediaPipe, Vision Transformers, and Generative AI, as well as models for audio classification and skin disease diagnosis employing CNNs, YOLO

- Member 3:
  - Name: Prerak Nagpal
  - Degree: Bachelor of engineering
  - University:UIET, Chandigarh
  - Email: prerak20coder@gmail.com
  - LinkedIn:https://www.linkedin.com/in/prerak-nagpal-1815ba24a/
  - GitHub:https://github.com/prerak20code
  - Experience: Hi, I’m Prerak Nagpal, a 3rd-year Computer Science student at UIET, Panjab University, with a strong foundation in backend development, API design, and cloud deployment using technologies like FastAPI, Flask, Docker, and AWS. I recently led the backend and infrastructure for my team’s Smart India Hackathon 2024-winning project, where we built a secure, scalable portal for the Ministry of AYUSH. Alongside this, I’ve worked on building modular systems, real-time dashboards, and production-ready solutions with clear documentation. I enjoy taking on technically challenging projects that require structured thinking, integration of complex systems, and real-world impact, and I’m always excited to collaborate and build something meaningful

● Member 4:
    - Name: Raghuvansh
    - Degree: Bachelor of Engineering
    - University: UIET, Chandigarh
    - Email: raghu12022005@gmail.com
    - LinkedIn: www.linkedin.com/in/raghuvansh-846899256
    - GitHub: Raghuvansh-12
    - Experience: Full-stack developer skilled in JavaScript | Node.js | React.js | Express.js | PostgreSQL | MongoDB | Cloudflare R2 | Firebase | GCP Pub/Sub | Hands-on experience building scalable microservices for video encoding using cloud infrastructure | Built and maintained REST APIs with authentication and file handling | Proficient in Git, Docker, and cloud deployments | Strong foundation in web security, documentation, and performance optimization | Practical exposure to project management and collaborative development through multiple internships and freelance work.

As Stack Overload, we combine expertise in AI, computer vision, and web development to deliver innovative solutions. We are excited to collaborate on this project to advance workplace safety.

## Expected Outcome

- A standardized API interface to wrap ML models
- Model compatibility with DIC repositories
- Initial integration with Whisper model
- Documentation for model deployment via API

## WEEK 1

1. Forked AISOC GitHub repository for project code and progress
2. Studied existing Machine Learning models  and API frameworks
3. Began model selection and initial experiments
4. Visualisation of obtained results

## 1. Forked AISOC GitHub repository for project code and progress

**Forked the AISOC-Template repo** – This created our own copy under our GitHub account, isolating our customizations and prototype work.

**Opened a GitHub Issue in the main repository** – To streamline communication, we filed a formal issue that outlines:

- Our purpose and goals for this Project
- The API's and the features we are implementing
- A link to our fork for review and collaboration

## 2. Studied existing Machine Learning models and API frameworks

In Week 1 of the project, we focused on understanding a wide range of machine learning models—both traditional and modern—and how they can be wrapped within a unified API interface. This involved not only studying the models themselves, but also evaluating their suitability for deployment, API responsiveness, and compatibility with different types of machine learning tasks such as classification, regression, and natural language processing.

This foundational work is critical to our overarching goal: to develop a single, standardized API interface that allows end-users and systems to access various models from different DIC projects seamlessly.

### 2.1. Natural Language Processing with SetFit and Hugging Face Models

We experimented with the **SetFit (Sentence Transformer Fine-Tuning)** framework, particularly using the Hugging Face model StatsGary/setfit-ft-sentinent-eval. This model is designed for sentiment analysis and is highly efficient for API-based deployment because of the following features:

- **Few-shot learning capability**: Allows good performance even with limited labeled data, reducing the need for large training pipelines in production.
- **Compact model size**: Optimized for fast inference, which is essential for real-time API response times.
- **Hugging Face Hub compatibility**: Enables direct loading and deployment via Python scripts or backend services without local training or model management.

We validated its use for text classification tasks (e.g., sentiment detection in user input) and explored how it can be accessed through a REST API. The basic API structure would include endpoints that accept user messages or text data and return classification outputs in JSON format. This fits into our API framework by offering a reusable text analysis module.

## 2.2. Classical Machine Learning Models for Structured Data

To support projects involving structured tabular data, we studied several commonly used machine learning algorithms, each with its own strengths in classification and regression contexts:

### Models Explored

- **Random Forest Classifier**: An ensemble learning method using multiple decision trees, known for high accuracy and resistance to overfitting. Suitable for fraud detection due to its robustness to noise.

- **Naive Bayes Classifier**: Based on Bayes' theorem with an assumption of feature independence. Efficient for real-time prediction due to its low computational complexity.

- **Logistic Regression**: A baseline binary classifier that offers interpretable outputs and probability scores, useful in risk prediction scenarios like fraud detection.

- **Decision Tree Classifier**: Easy to interpret and deploy; useful for proof-of-concept APIs due to its transparent decision rules.

### Use Cases Studied

- **Credit Card Fraud Detection**: These classifiers were tested for detecting anomalies and fraudulent transactions. APIs designed for this would take in transaction data and return binary classification results (fraud or not fraud).

- **House Price Prediction**: Regression-based deployment using similar API logic, where structured inputs like location, area, and number of rooms are processed and a predicted price is returned as output.

## API Relevance

As part of our initial experiments, we **began integrating these models into a standardized API format**. The objective was to evaluate how different models—ranging from NLP to structured data classifiers—can be accessed uniformly through RESTful APIs. Our approach included the following steps:

- We designed **API endpoints** that accept input data (text or structured features) via HTTP POST requests.
- Input data was **preprocessed as required**, depending on the model's format (e.g., vectorization for text, normalization for numeric features).
- The relevant model was **dynamically loaded or referenced**, allowing for flexibility in choosing models during runtime.
- **Predictions were returned in a structured JSON format**, which is easily consumable by frontend applications or other systems.
- We also ensured that each endpoint **follows a consistent request-response schema**, promoting modularity across different tasks.
- Where applicable, we included **confidence scores or probabilities** along with predictions for better interpretability.
- Additionally, we experimented with **handling multiple models under a common routing structure**, aiming to build the foundation for the unified API.

These initial trials have helped us understand practical deployment challenges and model compatibility with real-time APIs. This hands-on work directly supports our goal of building a scalable and unified API system for diverse DIC machine learning models.

This structure ensures all models can be accessed through a unified gateway, regardless of the problem domain.

## 2.3. Understanding API Frameworks for Deployment

To make these models accessible programmatically, we evaluated Python-based web frameworks for API development:

- **FastAPI**: A modern, high-performance web framework for building APIs with automatic documentation and async support for faster inference calls.
- **Flask**: A lightweight alternative suitable for simple proof-of-concept APIs and small-scale deployments.

These frameworks allow us to design modular APIs where each model (text, classification, or regression) is exposed through individual endpoints under a unified structure. This enables easy scaling and versioning while maintaining clean separation between different model types.

### 2.4. Vision for Unified API Integration

The ultimate goal is to create a **single API service** that can:

- Route incoming requests to the appropriate ML model based on input type or specified task
- Maintain consistent request/response formats across all endpoints
- Handle diverse input formats: text for NLP models, structured data for classifiers, and images for vision models (like those from Open Image Models)
- Offer real-time or batch processing options based on model complexity and latency constraints

This approach will make it significantly easier for different DIC project stakeholders to interact with models without worrying about model-specific intricacies, thus accelerating development, integration, and usability across teams.

| **Model Type** | **Models Studied** | **Application Context** | **Relevance to API Framework** |
|---|---|---|---|
| NLP | SetFit, Hugging Face Sentiment Model | Sentiment analysis, text classification | Lightweight, fast response via text API |
| Classification | Random Forest, Naive Bayes, Logistic Regression | Fraud detection, binary classification | Predictive APIs for structured input |
| Regression | Decision Tree, Linear Regression (for house pricing) | Price prediction based on numeric features | Output numeric predictions via API |
| Frameworks | FastAPI, Flask | Backend development and model deployment | Provides RESTful interface for all models |

By studying these models and deployment strategies, we’ve established the foundation for building a robust, unified API capable of serving multiple machine learning tasks in a modular, scalable, and accessible way.

## 3. Began model selection and initial experiments

## 3.1 WHISPER ASR MODEL

**Whisper** is an **Automatic Speech Recognition (ASR)** model developed by **OpenAI**. It is designed to convert spoken language (from audio files) into written text. Unlike traditional ASR models, Whisper is trained on **multilingual and multitask data**, which allows it to perform:

- **Transcription** of audio in multiple languages
- **Translation** of speech from other languages into English
- **Robust recognition** even in noisy or low-quality audio environments

Whisper is open-source and freely available, making it highly suitable for integration in academic, research, or production-level applications.

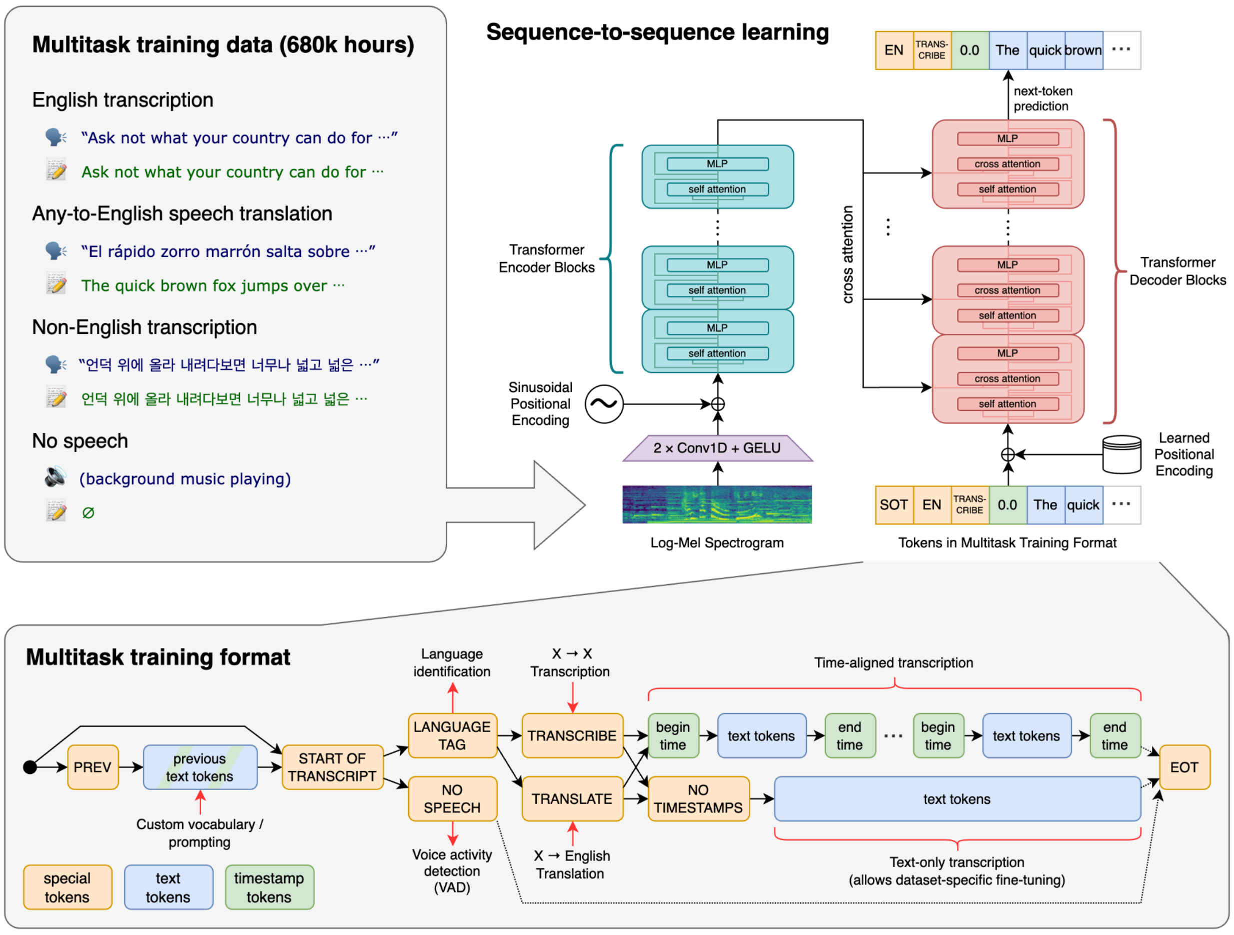
### Key Features of Whisper

1. **Multilingual Support**: Recognizes and transcribes speech in over 90 languages.
2. **Translation Capability**: Translates speech in non-English languages into English text.
3. **No Fine-tuning Required**: The pre-trained model works well on a variety of tasks out-of-the-box.
4. **Robustness**: Performs well even on poor-quality or noisy audio.
5. **Multiple Model Sizes**: Available in various sizes **(tiny, base, small, medium, large)** allowing users to balance accuracy and computational requirements.

### Model Architecture

Whisper is based on the **encoder-decoder Transformer architecture**. The audio input is first converted into log-Mel spectrograms (a visual representation of sound), which the encoder processes. The decoder then predicts the text sequence corresponding to the spoken content.

It is trained using a large dataset of 680,000 hours of multilingual and multitask speech data, which includes transcription, translation, and speech recognition tasks. This large-scale training makes it generalizable and powerful.


Multitask training data (680k hours)
English transcription
"Ask not what your country can do for ···"
Ask not what your country can do for ···
Any-to-English speech translation
"El rápido zorro marrón salta sobre ···"
The quick brown fox jumps over ···
Non-English transcription
"언덕 위에 올라 내려다보면 너무나 넓고 넓은 ···"
언덕 위에 올라 내려다보면 너무나 넓고 넓은 ···
No speech
(background music playing)
∅
Sequence-to-sequence learning
EN TRANS-CRIBE 0.0 The quick brown ···
next-token prediction
MLP
self attention
cross attention
Transformer Encoder Blocks
Transformer Decoder Blocks
Sinusoidal Positional Encoding
2 × Conv1D + GELU
Log-Mel Spectrogram
Learned Positional Encoding
SOT EN TRANS-CRIBE 0.0 The quick ···
Tokens in Multitask Training Format
Multitask training format
Language identification
X → X Transcription
Time-aligned transcription
PREV
previous text tokens
START OF TRANSCRIPT
LANGUAGE TAG
TRANSCRIBE
begin time
text tokens
end time
EOT
NO SPEECH
TRANSLATE
NO TIMESTAMPS
Custom vocabulary / prompting
Voice activity detection (VAD)
X → English Translation
Text-only transcription (allows dataset-specific fine-tuning)
special tokens
text tokens
timestamp tokens

## Using Whisper with APIs

Whisper's power is unlocked when it is wrapped inside an **API (Application Programming Interface)**. This allows developers to:

- Submit audio files through web or mobile apps
- Receive transcribed or translated text in response
- Integrate speech-to-text features into larger systems (e.g., educational tools, call centers, chatbots)

### Benefits of API Integration:

- **Modularity**: Whisper can be used as a service rather than embedded in every app.
- **Scalability**: Multiple clients can access it via a shared endpoint.

- **Platform Independence**: Mobile, desktop, or web apps can all make API calls.

## Typical Workflow for Whisper as an API:

1. A client application sends an audio file (e.g., .mp3 or .wav) to the server via an HTTP POST request.

2. The server receives the file and passes it to the Whisper model for transcription.

3. The model processes the audio and returns the text output.

4. The client receives the text and uses it as needed.

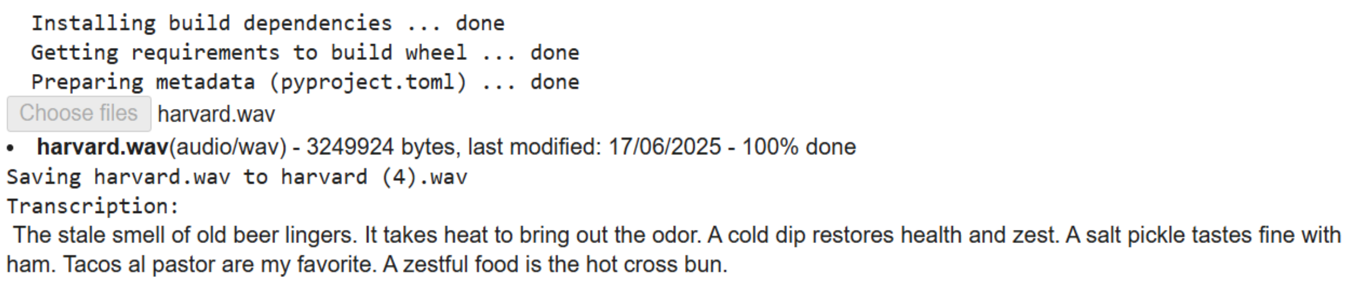
This allows seamless integration of speech recognition into any system, without requiring end-users to manage or run the model themselves.

| Aspect | Details |
|---|---|
| **Model Name** | Whisper |
| **Developed by** | OpenAI |
| **Main Function** | Speech-to-text transcription and translation |
| **Languages Supported** | 90+ |
| **Special Strengths** | Multilingual, noisy environment performance, translation |
| **Ideal Use Case** | As a backend service accessible through APIs |
| **Deployment Strategy** | Wrap model in REST API (e.g., using FastAPI or Flask) |
| **Integration Benefits** | Platform independence, centralized logic, easy maintenance |

**OUTPUT-**

**Test Audio Used-** harvard.wav

```python
# Install whisper
!pip install -q git+https://github.com/openai/whisper.git
!pip install -q torchaudio

# Import and load model
import whisper
from google.colab import files

uploaded = files.upload()  # Upload sample_audio.wav
filename = list(uploaded.keys())[0]

model = whisper.load_model("base")
result = model.transcribe(filename)
print("Transcription:")
print(result['text'])
```



```
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
```

Choose files harvard.wav

- **harvard.wav**(audio/wav) - 3249924 bytes, last modified: 17/06/2025 - 100% done

```
Saving harvard.wav to harvard (4).wav
Transcription:
```

The stale smell of old beer lingers. It takes heat to bring out the odor. A cold dip restores health and zest. A salt pickle tastes fine with ham. Tacos al pastor are my favorite. A zestful food is the hot cross bun.

## Open Image Models (OIM) and API Integration

**Open Image Models (OIM)** is a high-level Python library that provides pre-trained **computer vision models** for tasks such as:

- Object detection
- Image classification
- License plate detection
- Face detection
- OCR (Optical Character Recognition)

OIM is built to make state-of-the-art models accessible through simple interfaces, allowing developers and researchers to integrate advanced vision capabilities without writing complex code.

It supports backends such as **PyTorch**, **ONNX**, and **TorchScript**, making it flexible and production-ready.

**Key Features**

1. **Plug-and-Play Models**: Load and run pre-trained models with just a few lines of code.

2. **Multi-Backend Support**: Compatible with PyTorch, ONNX, and TorchScript Helpful for optimizing and deploying models on different platforms.

3. **Optimized for Real-Time**: Includes models that are efficient and fast (like YOLO variants), suitable for real-time applications.

4. **Modular Design**: Each task (like license plate detection) has its own class and methods for prediction and visualization.

5. **Auto-Inference Engine**: Automatically handles pre-processing, inference, and post-processing under the hood.

## Example Use Case: License Plate Detection

OIM includes specialized models like **yolo-v9-t-256-license-plate-end2end** for **automatic license plate detection and recognition**. This is particularly useful for:

- Parking management systems
- Traffic surveillance
- Toll booth automation
- Vehicle access control

The model returns bounding boxes around license plates and often the recognized text as well.

## Using Open Image Models with an API

Although OIM is a Python library, it becomes even more powerful when wrapped inside a **REST API**. This allows developers to send images to a central service and receive detection results without embedding the detection logic in every application.

**Benefits of API-Based Deployment**

| Feature | Benefit |
|---|---|
| **Decoupled Logic** | The detection system is separate from the user interface |
| **Cross-platform** | Apps on web, mobile, or desktop can call the same API |
| **Maintainability** | Easier to update the model or logic centrally |
| **Scalability** | Can serve many clients through load-balanced servers |

**API Workflow**

1. A client sends an image (e.g., .jpg or base64-encoded) via an HTTP request.
2. The API receives the image, loads the OIM license plate detector.
3. The model processes the image and returns predictions (e.g., bounding boxes and plate text).
4. The client displays or stores the result.

**API Design**

| Endpoint | Method | Description |
|---|---|---|
| /detect_plate | POST | Accepts an image file and returns detection results |
| /healthcheck | GET | Checks if the API is running |

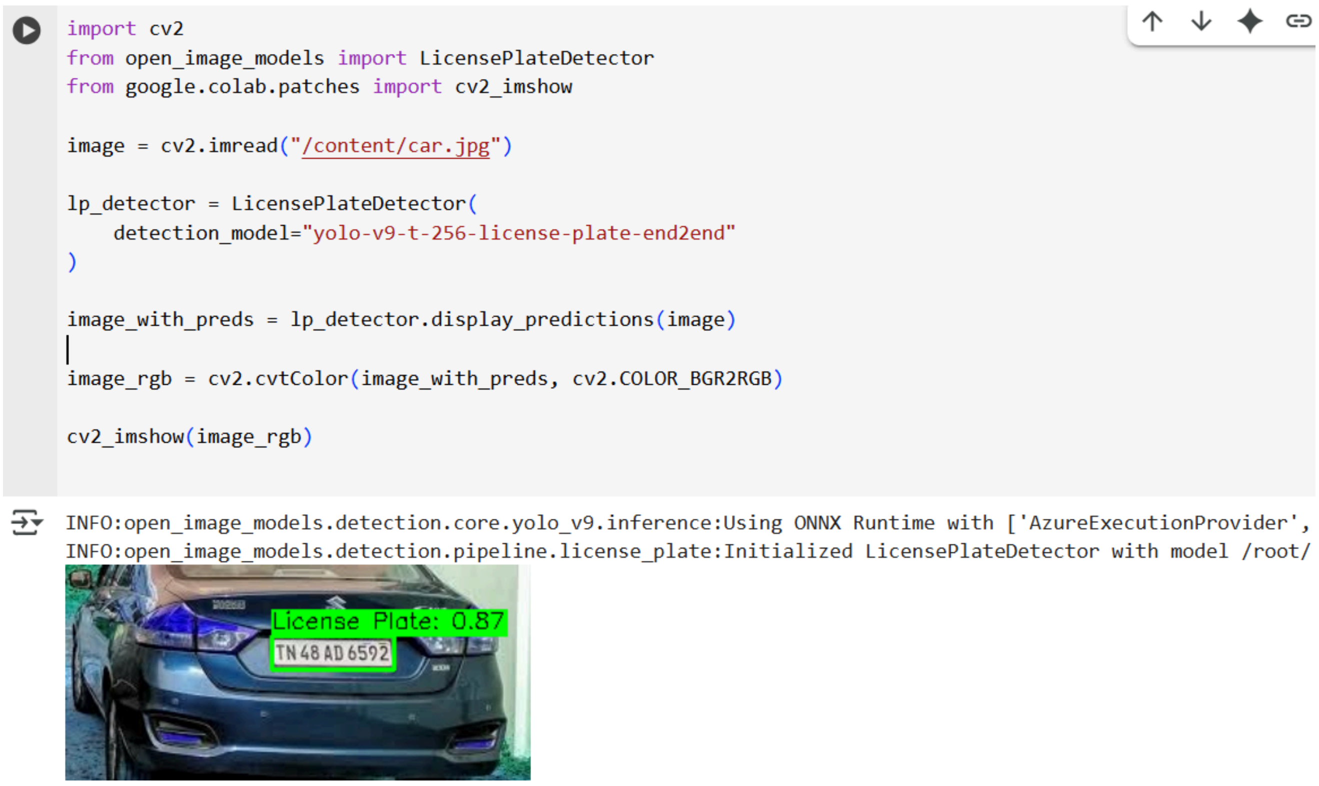
**Expected Response:**

```
{
 "plate_text": "MH12AB1234",
 "bounding_box": [x1, y1, x2, y2]
}
```

This makes OIM a plug-and-play detection system in any application — especially useful in smart cities, transportation, and law enforcement.

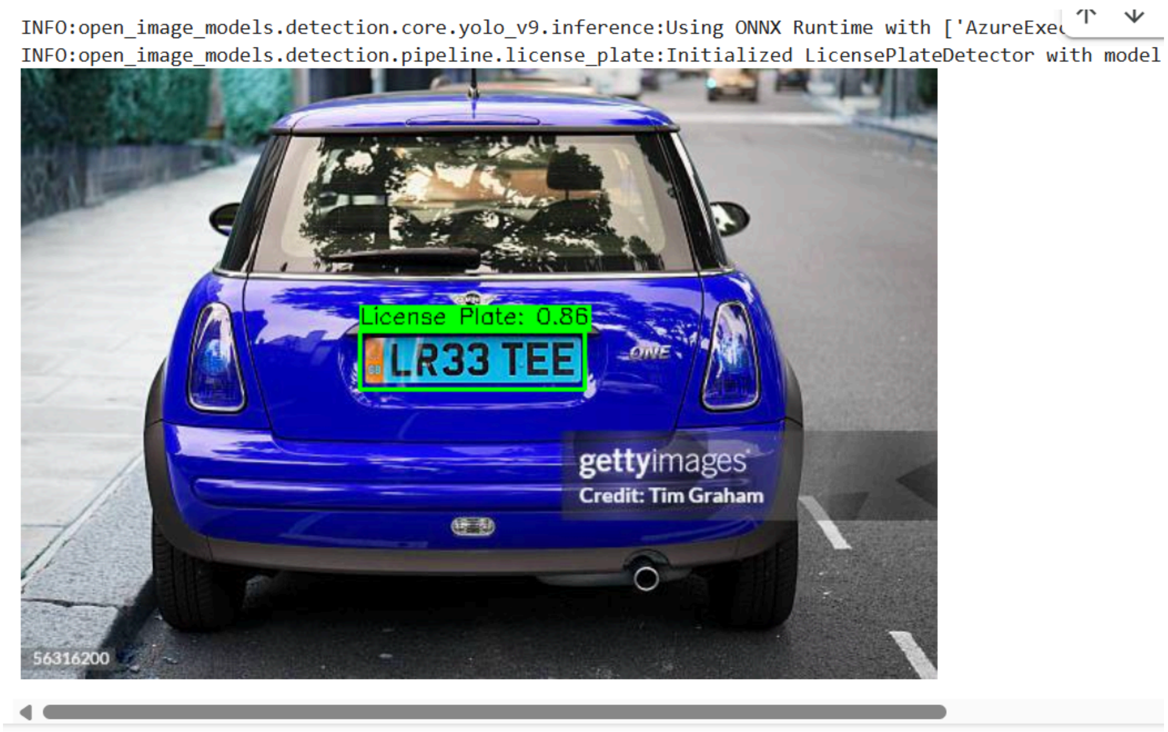
| Aspect | Description |
|---|---|
| **Library Name** | Open Image Models (OIM) |
| **Main Use** | Easy access to pre-trained CV models |
| **Typical Tasks** | Object detection, OCR, license plate recognition |
| **Supported Models** | YOLO variants, OCR, face detection, etc. |
| **Backend Support** | ONNX, PyTorch, TorchScript |
| **Ideal Usage** | Wrap as an API to allow remote image analysis |
| **Applications** | Smart parking, surveillance, LPR systems, mobile apps |

## OUTPUT:

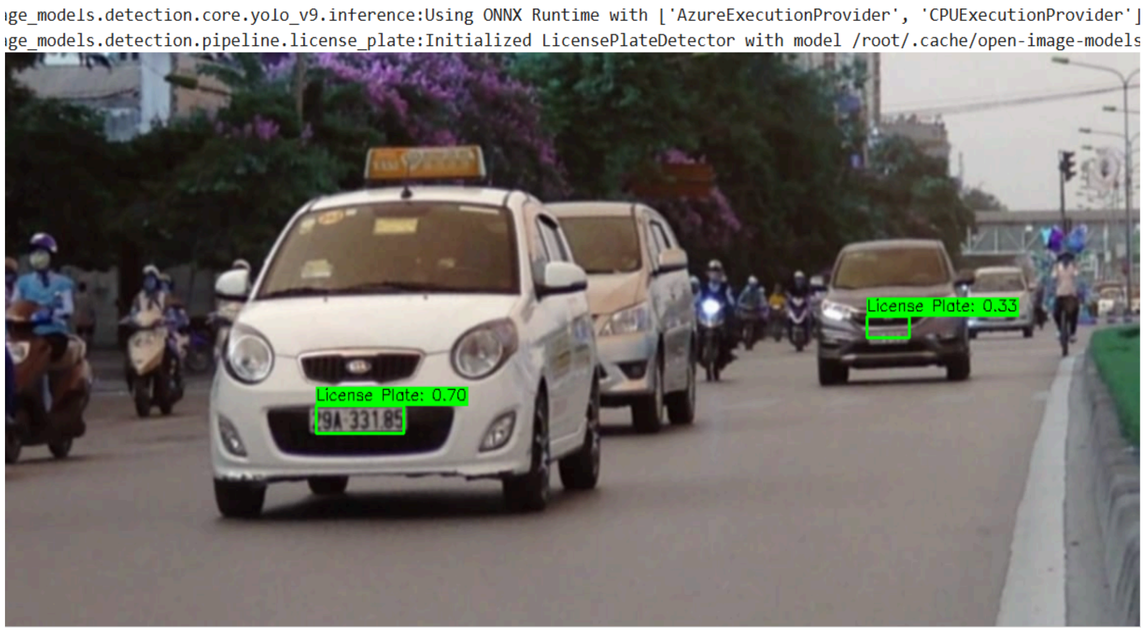


```
import cv2
from open_image_models import LicensePlateDetector
from google.colab.patches import cv2_imshow

image = cv2.imread("/content/car.jpg")

lp_detector = LicensePlateDetector(
    detection_model="yolo-v9-t-256-license-plate-end2end"
)

image_with_preds = lp_detector.display_predictions(image)

image_rgb = cv2.cvtColor(image_with_preds, cv2.COLOR_BGR2RGB)

cv2_imshow(image_rgb)
```

```
INFO:open_image_models.detection.core.yolo_v9.inference:Using ONNX Runtime with ['AzureExecutionProvider',
INFO:open_image_models.detection.pipeline.license_plate:Initialized LicensePlateDetector with model /root/
```



```
INFO:open_image_models.detection.core.yolo_v9.inference:Using ONNX Runtime with ['AzureExec
INFO:open_image_models.detection.pipeline.license_plate:Initialized LicensePlateDetector with model
```



```
ige_models.detection.core.yolo_v9.inference:Using ONNX Runtime with ['AzureExecutionProvider', 'CPUExecutionProvider']
ige_models.detection.pipeline.license_plate:Initialized LicensePlateDetector with model /root/.cache/open-image-models
```

## Conclusion

In the first week of our project, we laid a strong technical foundation for the development of a unified API system that can integrate machine learning models across diverse domains such as NLP, structured data analysis, and computer vision. Our work focused on understanding not only the performance and design of individual models, but also their deployment feasibility within an API architecture.

We began by setting up our development environment through GitHub, forking the AISoC template repository, and opening structured issues to track progress and goals. This allowed us to maintain transparency and version control, ensuring smooth collaboration throughout the project lifecycle.

We then studied a variety of **machine learning models** suited to the types of problems commonly encountered in DIC projects. In NLP, we experimented with the SetFit framework and Hugging Face-hosted models for sentiment analysis, identifying their suitability for lightweight, real-time inference within APIs. For structured data tasks, we evaluated classical models such as Random Forest, Logistic Regression, Decision Tree, and Naive Bayes, using them in domains like credit card fraud detection and house price prediction. We began testing these models with standard inputs and output formats in mind, to ensure their adaptability to a unified API structure.

In parallel, we explored **modern web frameworks** like FastAPI and Flask to understand the technical considerations behind building scalable and modular API services. These frameworks enable standardized routing, asynchronous processing, and consistent response handling—capabilities that are essential for our unified API vision.

We also initiated **model experimentation and visualization**, testing inference pipelines for real-world applications. This included the integration of OpenAI's Whisper model for speech-to-text transcription and translation, as well as Open Image Models for tasks like license plate recognition. Both models were analyzed for their readiness for RESTful deployment, and we designed conceptual API workflows to demonstrate how such models could serve end-users via centralized endpoints.

A key achievement during this phase was the creation of a standardized API prototype structure that can:

- Handle various input types (text, structured data, audio, images)
- Route requests to the correct model based on task or format
- Return structured and interpretable responses suitable for integration into larger systems

In conclusion, Week 1 has enabled us to bridge theory with practical deployment strategies, aligning well with our project objective of building a unified, scalable, and intelligent API framework for DIC machine learning models. The groundwork laid during this week ensures we are well-positioned to begin structured development, feature integration, and testing in the upcoming phases of the project.