Automatic Photography System for Adventure Parks Using Computer Vision

Richard Faragó
Babeş–Bolyai University
Cluj–Napoca, Romania
richard.farago@stud.ubbcluj.ro

Árpád-Bence Fekete
Babeş–Bolyai University
Cluj–Napoca, Romania
arpad.fekete@stud.ubbcluj.ro

Dániel Ferencz
Babeş–Bolyai University
Cluj–Napoca, Romania
daniel.ferencz@stud.ubbcluj.ro

Zalán Ráduly
Codespring
Cluj–Napoca, Romania
raduly.zalan@codespring.ro

Örs-Krisztián Patakfalvi
Codespring
Cluj–Napoca, Romania
patakfalvi.krisztian@codespring.ro

Rudolf-Bálint Tunyogi
Codespring
Cluj–Napoca, Romania
tunyogi.rudolf@codespring.ro

Csaba Sulyok
Babeş–Bolyai University
Cluj–Napoca, Romania
csaba.sulyok@ubbcluj.ro

***Abstract*—In adventure parks, it is often difficult to photograph visitors while they are crossing obstacles, due to the design of the tracks. Hiring professional photographers to take photos of visitors for social media can incur significant costs. Photographers would require safety equipment and designated locations as it would not be possible for them to stand on the tracks.**

The Adrenaline Eye project aims to achieve an automatic camera system for photographing adventure park visitors.

Images are captured by track-mounted surveillance cameras and processed to recognize the numbers printed on the helmets of the riders. This number and the date of the visit associate the images with the participant. Customers can view a gallery of their pictures through a web application.

The current paper presents the structure of the project, including its microservice-oriented architecture, its communication patterns and its application of machine learning elements.

***Index Terms*—microservice, message queue, computer vision, helmet detection, number recognition**

I. Introduction

Adventure parks, being part of the leisure and attractions industry [1], tend towards digitizing their photography in order to satisfy their guests in the modern era. Employing photographers would sometimes not be possible as the trails are typically laid out in wooded areas, among trees, through which only one person can pass at a time. The solution may rely in an automatic camera system, whereas customers would not have to worry about bringing their gadgets for taking pictures. The system would ensure quality photos fit for social media, while the visitors can fully enjoy the adventure that the park offers.

There are various solutions for identifying individuals in public areas. One approach is AIPIX [2], which utilizes facial recognition, while others, like Sniper Action Photo [3] or Argus [4], are based on the use of RFID[1]. Using RFID tags and detector antennas is more facile to apply than computer vision methods. However, the maintenance of antennas and tags is time consuming, cumbersome, and not fully automated, as visitors have to touch RFID readers on the rails.

The Adrenaline Eye project aims to solve this problem by developing a system that automates the photography of adventure park participants using surveillance cameras. The system allows employees to introduce different cameras that take pictures of the courses from time to time. Images are processed based on the time recorded and the number recognized on the helmet, and then linked to participants. After completing the tracks, participants can access their gallery through a web application.

The project is primarily developed for the Adrenalin Park[2] located near Cluj-Napoca. The credit for the principal idea goes to Levente Szélyes, founder and managing director of the park.

II. Project Overview

In its current state the system supports a single user role, capable of accessing all its features, including camera and run management. A run represents a visit to the park, marked by the number of the safety helmet handed out, and the timestamp of their arrival. Besides common entity operations, runs may also be closed, and their associated images queried. Visitors can view their photos on the website accessible by scanning a QR code sent via e-mail, or by using the direct link to their gallery page.

The application is based on a microservice-oriented architecture [5] with a discrete microservice for each individual reusable task. The communication between them is predominantly message-based, although the web application uses synchronous API requests through HTTP[3]. There are two separate databases responsible for saving the model entities, and two storage volumes used to save and read images. The temporary volume stores unidentified photos whereas the persistent one contains all successfully processed by the

[1]Radio-frequency identification

[2]https://adrenalinpark.ro/en/

[3]HyperText Transfer Protocol

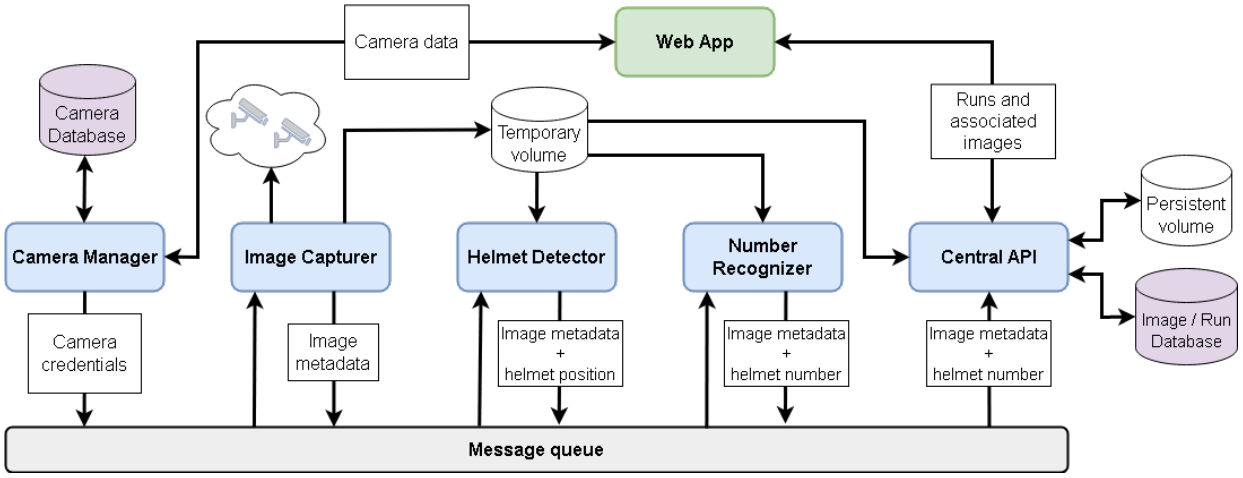


Fig. 1: The architecture of the system illustrating all the components and the communication between them.

computer vision modules, meaning that a helmet and its number is recognized on the picture.

The project uses a RabbitMQ broker [6] for managing message-based communication. The components and the one-way communication between the microservices are illustrated in Figure 1. The Camera manager on the left side of the flowchart acts as a producer, i.e. it generates messages, while the Central API on the right side only processes messages which means it functions as a consumer. At the same time, the Image capturer, Helmet detector, and Number recognizer services are prosumers, they both produce and consume.

III. Microservices

The following subsections present each microservice detailing their structure, functionalities and operations.

A. Camera manager

The camera manager is a server exposing a REST API [7], that makes managing cameras and scheduling tasks possible. These operations can be accessed through HTTP requests sent to the `/api/cameras` endpoint. The camera entities are stored in a PostgreSQL [8] database. After adding a new camera entity, a periodically scheduled task is also created that sends messages to the image capturer (see Section III-B) module to take a snapshot. When changes are made to a previously introduced camera, the scheduler modifies the related task or cancels it upon deletion. The messages contain the camera UUID, the protocol for capturing images, address and authentication data.

When the camera controller starts, it initializes the tasks for the cameras persisted in the database. Once connected to the message broker, the active jobs continuously send messages to the image capturer until the camera entity is deleted, or the service is shut down. When it stops, the scheduler automatically releases pending resources.

The service is based on the Spring framework [9] and uses the standard Controller-Service-Repository pattern for its multilayer architecture. It exposes a RESTful API and follows the DTO[4] design pattern [10]. Additionally, Spring Task Scheduler carries out the scheduling work, and Spring Cloud Stream provides interfaces to facilitate message-based communication with RabbitMQ.

B. Image capturer

The role of the image capturer service is to take pictures with given cameras and save the captured images to a temporary storage. It receives the required data from the camera manager and sends an API request to the appropriate camera, in response to which it receives a snapshot. On success, the resulting image gets a unique ID, which is transmitted to the helmet detector alongside the camera ID and a timestamp that marks the creation of the snapshot.

The Hikvision brand IP camera used to test the microservice captures 4MP images and connects to the Internet via Wi-Fi. It supports ISAPI[5] and those variants of the ONVIF[6] protocol which allow remote screenshots and/or streaming [11]. Upon receiving a request to snap an image, the camera validates the provided authentication information according to a pre-defined username and password. Accepted credentials can be entered in the administration interface of the camera upon enabling the ONVIF protocol.

A camera simulator microservice is also implemented for testing the other components without any additional hardware dependencies. It mocks the API of the ONVIF cameras by returning randomly selected images from a given collection–these may be cherry-picked to create different testing scenarios.

The service is implemented in Python 3. The API requests are sent using the Requests library. Pika, an AMQP[7] library, is responsible for connecting to the message broker and transmitting messages.

[4]Data Transfer Object
[5]Internet Server Application Programming Interface
[6]Open Network Video Interface Forum
[7]Advanced Message Queuing Protocol

	Number of entities	Train	Validation	Test	Summary
Training dataset	Helmet	18196	4285	5455	27936
	Head	76852	18757	21671	117280
	Images	8045	2011	2515	12571
Fine-tuning dataset	Helmet	985	246	286	1517
	Head	789	146	350	1285
	Images	549	137	172	858
Distribution of images		64%	16%	20%	100%

TABLE I: The composition of the training and fine-tuning dataset used to train the helmet detector neural network

Network size	Augmentation	Precision (%)	Recall (%)	mAP@0.5 (%)	mAP@0.5:0.95 (%)
medium	default	93.8	88.2	91.9	74.0
	increased	90.2	91.8	93.7	72.4
small	default	91.3	90.2	92.8	73.0
	increased	92.5	89.5	93.0	70.1
nano	default	91.9	88.0	92.0	69.1
	increased	90.9	87.0	91.5	65.7

TABLE II: Performance of the fine-tuned networks on their corresponding test dataset

C. Helmet detector

The helmet detector microservice receives a JSON formatted message from the image capturer and processes the images taken by the cameras. Detection is performed by a YOLOv5 [12] neural network, which is able to locate head and helmet objects using less computing power than other object detectors. The message contains the name of the image the service attempts to read from the temporary volume. The message is discarded if the file does not exist. Whenever the detector locates a helmet on the image, a message with the bounding box coordinates of the object is forwarded to the number recognizer (see Section III-D).

On service startup, the neural network is loaded based on hyperparameters taken from environment variables. The component can be initialized in either CPU or GPU mode, impacting which unit performs performance-sensitive calculations, such as the object detection. For the detection, three different sized neural networks can be configured. The smallest `nano` detector has 1.9 million parameters and is the fastest but also the least accurate. A `small` version with 7.2 million parameters is slower but performs better, and a `medium` network with 21.2 million parameters is the slowest of the three, but provides the best results.

Training of the YOLOv5 networks consists of two stages: initial training and fine-tuning. In both stages, the experiments can be categorized based on the following criteria:

- *Network size*: `nano`, `small` or `medium`.
- *Augmentation*: Trainings with default augmentation values specified by YOLOv5 authors or experiments with increased augmentation parameters. The default augmentation includes the following transformations: increased values on HSV color channels by 1.5%, 70%, and 40%, respectively, random 10% translation and vertical axis mirroring with a 50% chance. The increased augmentation setting includes increasing each HSV color component by 100%, random 50% translation, and vertical axis mirroring with a 50% chance. It also applies -25 to 25 degree rotation, mosaic composition, mix-up, and segment copy-paste to each image.
- *Number of object classes*: Normal trainings with both helmet and head objects, or single class experiments, where only the helmet annotations are used.

Two separate datasets are created to train the models: a larger training set containing images taken mainly on construction sites; and a smaller set consisting of real-world adventure park scenes. The final training set is created by merging the Safety Helmet Detection [13] and Safety Helmet Wearing [14] datasets. Related statistics are presented in Table I.

Throughout the experiments the Adam optimizer is used with $\beta = 0.937$. The initial learning rate is 0.001, which is modified by a OneCycle learning rate scheduler [15]. The value of weight decay is 0.0005. Each experiment begins with 3 warmup epochs, with an initial momentum of 0.8 and an initial bias learning rate of 0.1.

The process starts with all 3 network sizes and both augmentation settings pre-trained for 300 epochs on the MS-COCO dataset. Based on training statistics it can be concluded that strong augmentation increases recall by an average of 22%, mAP@0.5 by 10.5%, and mAP@0.5:0.95 by 2.5%, but decreases accuracy by 2.6%. The medium sized model preformed best, reaching a mAP@0.5 and mAP@0.5:0.95 of 95.4% and 79.2%.

The 100 epoch long fine-tuning starts from the weights of the networks trained in the previous phase. The images in the afferent dataset are resized to 640x640 resolution. All three network sizes are further trained with default and increased augmentation. Based on the experiments performed in the second phase, it can be stated that fine-tuning

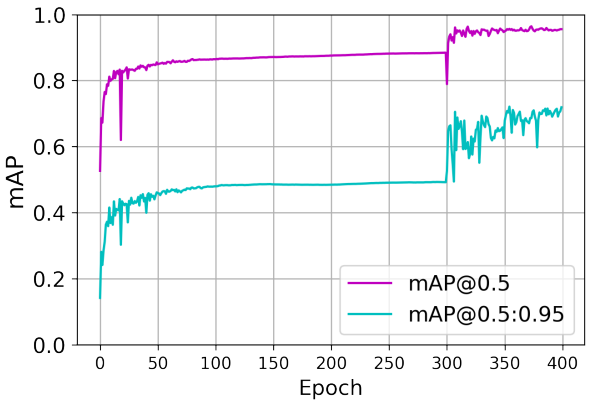


Fig. 2: The mAP metrics during the training and fine-tuning of the best performing, medium sized model using increased augmentation. At the end of the fine-tuning phase, the mAP@0.5 and mAP@0.5:0.95 on the validation set was 95.5% and 71.9% respectively.

improves performance. The recall of these networks on their corresponding test dataset is higher by an average of 16.9%, mAP@0.5 by 12.2%, and mAP@0.5:0.95 by 31.7% compared to networks trained in the initial phase. The precision increases only by 7.6%. Since it is already high after the initial training, fine-tuning cannot further increase it. Table II contains the performance statistics of these models, while Figure 2 shows the mAP metrics during the experiments with the best performing model.

If it is possible to track the position of a helmet between consecutive images, then the system is able to assign a previously detected helmet number to an image, even if the number is not visible. The position of the helmet can be determined with MOSSE [16], meanshift and CAMshift [17], Kálmán-filter [18], and ROLO [19]. Testing the MOSSE tracker shows it cannot perform well in the low FPS environment that is created by the periodic image capturing, because the displacement of the helmet between two consecutive images is too large.

D. Number recognizer

The number recognizer is the second service that performs image processing based on images that contain detected helmets. Its technology stack and startup behavior resembles that of the helmet detector, with the addition of the EasyOCR [20] library for number recognition, and OpenCV [21] for pre-processing.

Messages arriving from the helmet detector contain the name of the photograph, the time it was taken, the ID of the camera and the bounding box coordinates of the helmet. The image is retrieved from the shared storage, cropped around the rectangle that defines the helmet, and then enlarged. After this pre-processing, it is evaluated by the neural network. The result may contain several predictions; the one with the highest confidence score is retained. If the confidence level obtained for the prediction exceeds a configured threshold value, the result is forwarded to the central API (see Section III-E) message queue.

The EasyOCR library is tested on a dataset of 30 images gathered from the Adrenalin Park website and augmented with public stock photos. The average input size is 399.3 x 433.9 pixels. The tests reveal that the position of the numbers greatly influences the prediction of the EasyOCR neural network. Better accuracy is achieved when the numbers are in a horizontal position. Since it is not known exactly at what angle the person is rotated in relation to the camera, a general rotation is introduced into the system. Images are evaluated by rotating 0, 30, and -30 degrees, in order. If the confidence level of a prediction exceeds the threshold value, the evaluation no longer continues for the other rotation variants. This method increases the initial 85% accuracy on the test dataset to 93.3%.

E. Central API

The central API is principally responsible for managing runs and identified images. A run entity contains the e-mail address provided by the visitor, the assigned helmet number and the start time of the visit. Once the run is completed, an end date is also attached to it. The `/api/runs` RESTful endpoint exposes CRUD[8] operations on the run entities. When a run is created, the service generates a QR code containing the URL of the gallery page, and sends it as an e-mail to the visitor.

The server additionally listens for messages from the number recognizer. Upon receiving one, it copies the image from the temporal volume to the persistent volume and creates an identified image entity in the database. The saved images are available as static content through the `/images` endpoint.

The API is built with Spring Boot, with the same technologies, tools, and dependencies as the camera manager. The QR code generation is provided by ZXing[9] [22] library and the e-mails are sent via the SendGrid [23] Java Client API library. Using the configuration mechanism provided by Spring, the two packages mentioned above can be configured based on environment variables.

F. Frontend

The project allows users to manage cameras and runs using a web application. Adventure park visitors can use the app to view the images taken of them by the cameras in the park. This data is provided by the central server and camera controller services.

The communication with the camera manager and central API is achieved via the Axios [24] library; separate environment variables are responsible for the request paths to the connected backend services.

The application is written in TypeScript [25] using the Next.js [26] framework. The presentation layer is made up of reusable styled React components. Next.js handles navigation by mapping top-level component paths to URL patterns, retaining the Single-Page Application nature of the website.

The user interface (UI) consists of MUI [27] components, MUI being a widely used, customizable React-based UI library which guarantees a unified look with predefined components. These provide responsive and user-friendly operation for an application.

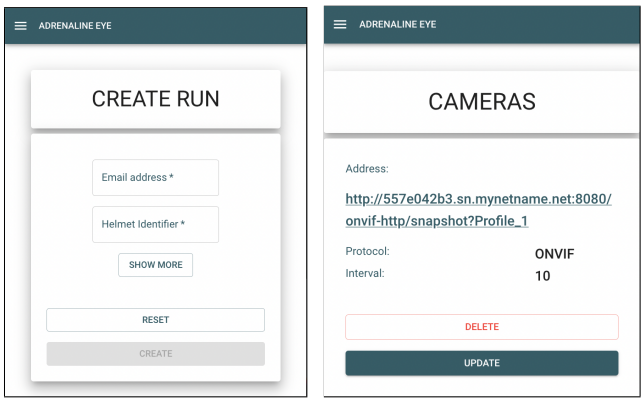
IV. The Usage of the Web Application

When the application is launched, the user is greeted by a home page. The web page has a navigation bar that can be accessed using the icon on the left.

The first functionality of the web application is the management of the cameras. Separate pages are provided for their listing and modification (see Figure 3). To add cameras, the user needs to fill out a form with its access and identification information, and an interval value which defines how many seconds apart it is going to take pictures of the track. Once completed, the app will send the information to the camera manager service. After uploading, the user receives feedback from the web application in the form of a message.

[8]Create Read Update Delete

[9]Zebra Crossing



(a) Run creation form (b) Listing cameras

Fig. 3: Run and camera management pages.

Upon success, the rest of the system starts processing the images received from the camera, as described in Section III.

The next functionality is the registration of the runs. A form is presented where the participant e-mail address and helmet number must be entered. By default, the form sets the start date of the run to the time the page is loaded and leaves the end time blank. Upon completion, the website sends the information to the central server and receives feedback on the success of the process. When the user enters a run, the central server saves the information and starts associating images based on the messages it receives from the number recognizer. The server then sends a QR code to the e-mail address of the participant.

On the run management page the already created runs are displayed, which can also be modified and deleted. The Finish Run button closes the active runs, signalling the current time as the end time to the server.

Galleries for each run may also be accessed here. Alternatively, they can be searched by their unique IDs, or by using the QR code received in the e-mail.

V. Conclusion and Further Development

Within the Adrenaline Eye project, an application has been developed that recognizes the numbers that appear on helmets based on images taken by the cameras and associates them with a particular run based on time.

The system is created with a microservice-based architecture, in which the camera manager service is able to manage the cameras and schedule the photo-taking tasks, the image capturer service captures and temporarily stores the images taken by the different cameras. Among the image processing microservices, the helmet detector is able to locate heads and helmets, while the number recognizer identifies the numbers on the helmets. The central API server takes care of the management of runs, persists and makes identified photos accessible for the web application.

The web application provides a user-friendly platform to manage cameras and runs, and displays images associated with the latter in a gallery form.

Message-based communication between services is achieved with the help of a RabbitMQ message broker, and each service is containerized, giving way to virtualized deployment strategies with tools such as Kubernetes.

Several possibilities for further development have emerged during the design and development of the various microservices:

- User management for the API servers and the web application: segregating features from admins and adventurers by creating separate roles. The admin user could manage the cameras, runs and images, while the participants could view their own runs and associated images.
- Introduction of a mobile application for adventure park staff to manage the system.
- Applying edge detection before number recognition to make numbers horizontal, or use an OCR library more robust to the issue of rotation angle.
- Use helmet tracking, which allows the system to associate a visitor with an image that does not display the helmet number.
- Utilize further functionalities of smart cameras, such as motion detection and event notifications, which could replace the current scheduled snap shooting with a more applicable, less resource-demanding service.

References

[1] B. Slatter, "How social media is transforming the leisure and attractions industry," 2015. [Online]. Available: https://blooloop.com/theme-park/in-depth/how-social-media-is-transforming-the-leisure-and-attractions-industry/

[2] J. Kleiman, "Polin Introduces Image Recognition Technology for Amusement Parks and Waterparks," *IPM InPark Magazine*, 2020. [Online]. Available: https://www.inparkmagazine.com/polin-aipix/

[3] RFID Enabled Automated Photograpy. [Online]. Available: https://sniperactionphoto.com/

[4] Z. Szécsi, K. Simon, and L. Szélyes, "Argus: Hardware and software system for automatic or semi-automatic photo taking," in *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*, 2015, pp. 37–41.

[5] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.

[6] G. Roy and J. Titcumb, *RabbitMQ in Depth*. Manning, 2017.

[7] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[8] L. Ferrari and E. Pirozzi, *Learn PostgreSQL: Build and manage high-performance database solutions using PostgreSQL 12 and 13*. Packt Publishing, 2020.

[9] F. Gutierrez, *Pro Spring Boot 2*. Springer, 2019.

[10] P. Ansari, *Learn MapStruct: Give You Enough Understanding on the Various Functionalities of Mapstruct with Suitable Examples*. Independently Published, 2021.

[11] S. Cheruvu, A. Kumar, N. Smith, and D. Wheeler, *Demystifying Internet of Things Security: Successful IoT Device/Edge and Platform Security Deployment*. Apress, 2019. [Online]. Available: https://books.google.ro/books?id=YSKpDwAAQBAJ

[12] G. Jocher, A. Stoken, A. Chaurasia, J. Borovec, Y. Kwon, K. Michael, L. Changyu, J. Fang, V. Abhiram, P. Skalski *et al.*, "Ultralytics/yolov5: V6.0 — YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support," *Zenodo Tech. Rep.*, 2021.

[13] "Safety helmet detection dataset." [Online]. Available: https://www.kaggle.com/datasets/andrewmvd/hard-hat-detection

[14] "Safety helmet wearing dataset." [Online]. Available: https://github.com/njvisionpower/Safety-Helmet-Wearing-Dataset

[15] L. N. Smith and N. Topin, "Super-convergence: Very fast training of neural networks using large learning rates," 2017. [Online]. Available: https://arxiv.org/abs/1708.07120

[16] D. Bolme, J. Beveridge, B. Draper, and Y. Lui, "Visual Object Tracking Using Adaptive Correlation Filters," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2544–2550.

[17] G. Bradski, "Real Time Face and Object Tracking as a Component of a Perceptual User Interface," in *Proceedings Fourth IEEE Workshop on Applications of Computer Vision. WACV'98 (Cat. No. 98EX201)*. IEEE, 1998, pp. 214–219.

[18] Q. Li, R. Li, K. Ji, and W. Dai, "Kalman filter and its application," in *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, 2015, pp. 74–77.

[19] G. Ning, Z. Zhang, C. Huang, Z. He, X. Ren, and H. Wang, "Spatially supervised recurrent convolutional neural networks for visual object tracking," *arXiv preprint arXiv:1607.05781*, 2016.

[20] A. Rosebrock, "Getting Started with EasyOCR for Optical Character Recognition," *pyimagesearch*, 2020. [Online]. Available: https://pyimagesearch.com/2020/09/14/getting-started-with-easyocr-for-optical-character-recognition/

[21] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[22] M. Winter, *Scan Me - Everybody's Guide to the Magical World of QR Codes*. Westsong Publishing, 2011.

[23] R. Costa and D. Hodun, *Google Cloud Cookbook*. O'Reilly Media, 2021.

[24] "Promise based HTTP client for the browser and node.js." [Online]. Available: https://axios-http.com/docs/intro

[25] D. Choi, *Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL*. Packt Publishing, 2020.

[26] "The React Framework for Production." [Online]. Available: https://nextjs.org/

[27] A. Boduch, *React Material-UI Cookbook: Build Captivating User Experiences Using React and Material-UI*. Packt Publishing, 2019.