THOMAS MORE

Smart NV AI-Assisted Puzzle Solver

Realization document

Ibrahim Afkir
Student Bachelor Applied Computer Science

2025 - 2026

Table of Contents

1. List of Figures

2. List of Tables

3. Glossary

Term	Explanation
YOLO	An AI model used to detect objects in images.
ONNX	A model format that allows AI models to run on different platforms, including the browser.
WASM	WebAssembly, a technology that helps run fast code inside the browser.
Homography	A technique used to correct the camera angle of an image.
Instance Segmentation	A computer vision method that detects the exact shape of an object, not only a box around it.
Bounding Box	A rectangle drawn around an object detected in an image.
OpenCV	A computer vision library used for image processing tasks.
HSV	A color format used to detect objects based on color.
GPU	A graphics processor used to train AI models faster.
VRAM	Memory used by the GPU during model training.
Dataset	A collection of images and labels used to train an AI model.
Synthetic Data	Artificial images generated with software, such as Blender.
Real Data	Real photos taken with a camera and labeled manually.
Annotation	A label added to an image to show the model what object is present.
Mask	The exact pixel-level shape of an object in an image.
Grid Mapping	The process of converting detected pieces into positions on the digital puzzle grid.
Solver	The logic that checks if the puzzle can still be solved and can generate hints or a full solution.
Browser-only Application	An application that runs completely inside the browser without needing a backend server.
React	A JavaScript library used to build the user interface of the application.
Vite	A fast development tool used to build and run the React application.
Backtracking	An algorithm that tries solutions step by step and undoes the last step when it reaches a dead end, then tries a different option.
Web Speech API	A built-in browser feature that allows applications to generate spoken text without any external service.

Term	Explanation
Letterboxing	A technique that scales an image to fit a specific size while preserving its proportions, filling the remaining space with padding.
Convex Hull	The smallest shape that wraps around all the outer points of an object, like stretching a rubber band around it.
Sigmoid	A mathematical function that converts any value into a number between 0 and 1, used to determine if a pixel belongs to an object or not.

4. Introduction

This realization document describes the work carried out during my internship project at Smart NV. Smart NV makes physical logic puzzles and games for children and families, specializing in games such as IQ Waves and IQ Stars that challenge players to fit uniquely shaped pieces onto a board. These puzzles are easy to start with, but they become more difficult when a player gets stuck. The usual full solutions and challenges are printed in a booklet, and that booklet gives the full solution of a certain challenge. For a child, that can be too much help at once: instead of receiving a small hint, or checking if the pieces they have placed on the board still make the puzzle solvable, the puzzle is almost taken away. Moreover, not every possible challenge or board state exists in the booklet.

During my internship, I developed an AI-assisted computer vision application to address this gap. The goal is to allow users to take a photo of the physical board, have the application detect the pieces and their positions, convert the photo into a digital board state, and use solver logic to determine if the puzzle can still be solved. From that same board state, the app can also provide hints or a complete solution.

Smart NV is entering a new phase of product development by exploring how digital technology can enhance the physical play experience. This project represents their first foray into an AI-assisted computer vision application, with the strategic goal of keeping children and families engaged with the puzzles—preventing frustration and abandonment—while adding meaningful value to the existing product line.

The company had no internal computer vision capability and no labeled training dataset for their puzzle pieces. Building this system from scratch—including dataset generation, model training, and full application development—became the core challenge. To address this, we generated our own training data using a combination of approaches that we will detail throughout this document.

This document covers the realization of both the IQ Waves and IQ Stars pipelines, because that is where the dataset generation, YOLO model training, homography, grid mapping, and solver integration were developed in detail. Both puzzles are equally relevant as they demonstrate how the same idea can be extended across different Smart NV puzzles. This document covers the full workflow of the application, including the AI pipeline: creating the dataset with synthetic and real data, training and fine-tuning the model, correcting the detected board with homography, mapping detections to the digital grid, explaining the solver logic, and showing the results of the trained model. We also cover how the application itself was built, the tools used throughout the development process, and the complete integration of all components.

5. Analysis

The core idea behind this project is to bridge the gap between the physical puzzle and a digital assistant. The user opens the application, takes a photo of the physical board, and within seconds the application shows which pieces are placed, where they are positioned, and whether the puzzle is still solvable. From there, the user can request a hint to place the next piece, or ask the app to solve the entire puzzle. If the current placement leads to a dead end, the app will let the user know the puzzle is no longer solvable with the pieces in their current positions.

Behind this simple interaction lies a full computer vision and solving pipeline. The photo is processed by a YOLOv26 Nano instance segmentation model that identifies each piece and its exact shape on the board. The detected masks are then corrected for camera angle using homography, and mapped onto the digital grid. Finally, a backtracking solver computes whether the puzzle can still be completed, generates hints, or produces the full solution.

However, this workflow did not come ready. Before any of this could function, significant preparation was needed. The company had no existing dataset, no trained model, and no computer vision infrastructure. Everything — from generating synthetic training data in Blender, to labeling real images in Roboflow, to training the model, to building the detection pipeline and the solver — had to be created from scratch. The following sections describe this process in detail, covering the tools used, the choices made, and the alternatives that were considered along the way.

5.1. Dataset Creation

Since Smart NV had no existing dataset for their puzzle pieces, creating one from scratch was one of the first and most important steps of the project. The dataset needed to contain images of the IQ Waves puzzle pieces and board, annotated with instance segmentation masks so the model could learn not just where each piece is, but its exact shape at the pixel level. The dataset consists of nine classes: eight puzzle pieces (Red, Hot Pink, Dark Blue, Orange, Light Blue, Green, Yellow, Purple) and the Board itself.

Two types of data were used to build the dataset: synthetic images generated in Blender, and real photos taken with a camera and labeled manually in Roboflow.

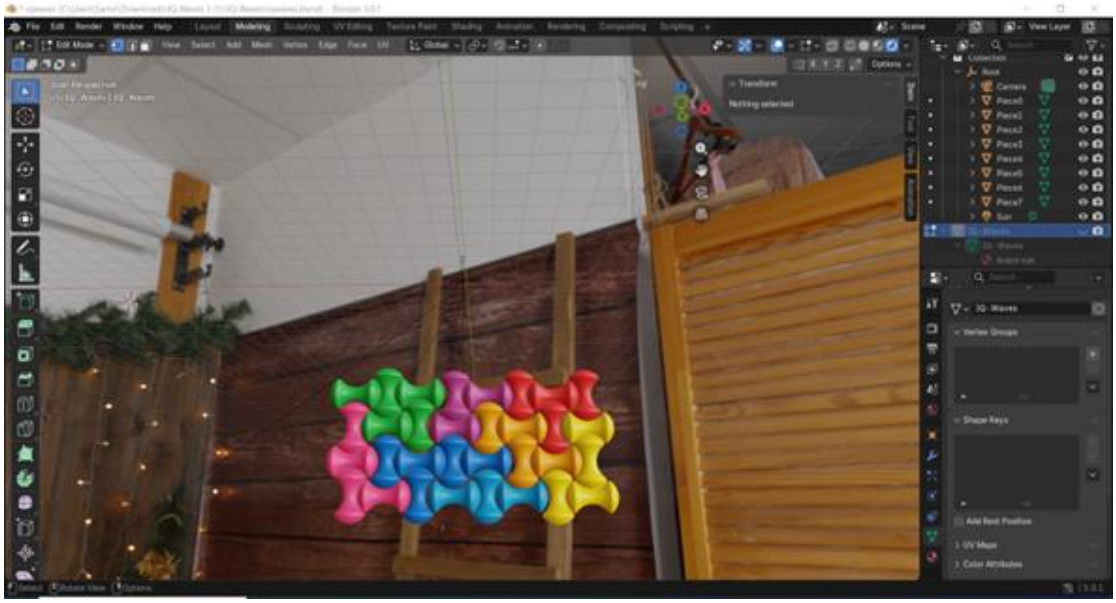
5.1.1. Synthetic Data

The first approach to generating synthetic data was using Trimesh and Pyrender, two Python libraries for 3D rendering. Smart NV provided STL files of all puzzle pieces, which were loaded and rendered programmatically. However, the results were poor, the rendered pieces looked flat and artificial, with unrealistic lighting and no resemblance to how the pieces look in real life as shown in Figure 1. This made the data unsuitable for training a model that would need to recognize pieces in real-world photos.

Figure 1: Trimesh & Pyrender

The decision was then made to switch to Blender. The same STL files were imported into Blender, where the pieces were manually colored and textured to match the real puzzle. A Blender Python script was written to automate the image generation process. The script rendered the pieces in various configurations — some images showing individual pieces separately, some showing only the board, and some showing pieces placed on the board — all with different backgrounds and from different angles. The script also automatically generated the corresponding segmentation annotations for each image, removing the need for any manual labeling of the synthetic data.

Figure 2: Blender Setup



This approach produced around 1800 images. The difference in quality compared to the Trimesh/Pyrender attempt was significant: the Blender-rendered images look close to real photographs, with realistic lighting, shadows, and textures, which made them far more useful for training as shown in Figure 2 and 3.

Figure 3: Examples of synthetic IQ Waves images generated in Blender, showing pieces on the board with varying backgrounds, an empty board, and individual pieces rendered separately.

This early experience showed that the quality of synthetic data directly impacts model performance, and investing time in a better rendering tool upfront saved significant effort later in the training process.

5.1.2. Real Data

In addition to the synthetic dataset, around 240 real photos of the physical puzzle board were taken with a camera. These images were uploaded to Roboflow, where they were labeled manually using instance segmentation with the pixel-level smart selector tool. This tool allowed for precise pixel-level annotations around each piece,

which is important because the wave-shaped pieces have curved and irregular edges that a simple bounding box would not capture accurately.

No augmentations were applied in Roboflow. The dataset split into training, validation, and test sets was handled later during the model training phase itself, rather than in Roboflow.

The combination of synthetic and real data resulted in a total dataset of approximately 2040 images. The synthetic data provided volume and variety, while the real data ensured the model could generalize to actual photographs taken in real-world conditions.

Figure 4: Examples of real IQ Waves photos taken with a camera, showing different piece configurations, lighting conditions, and camera angles.

5.2. Computer Vision Approach

Before any piece detection or grid mapping could work, several fundamental computer vision decisions had to be made. These decisions shaped the entire pipeline, from how the model detects the pieces, to how the board is located in the image, to how the camera angle is corrected before any analysis takes place. This section covers those decisions, the approaches that were considered, and the reasoning behind the choices that were made.

5.2.1. Detection Approach — Instance Segmentation vs Bounding Boxes

Before building the dataset and training the model, one of the first decisions to make was how the model should detect the puzzle pieces. Two approaches were considered:

- **Bounding Boxes**: A bounding box draws a simple rectangle around a detected object. While this approach is fast and straightforward, it is not suitable for IQ Waves. The puzzle pieces have irregular, curved, wave-like shapes, and when placed next to each other on the board, their bounding boxes would heavily overlap. This makes it impossible to accurately determine which grid cells each piece occupies, which is essential for the application to work correctly.

- **Instance Segmentation**: Instance segmentation goes further by detecting the exact pixel-level shape of each object. This means for each detected piece, the model returns a precise mask showing exactly which pixels belong to that piece. This is crucial for IQ Waves because it allows the application to accurately map each piece to the grid cells it covers, even when pieces are placed directly next to each other with no space in between.

The initial project plan considered bounding box detection, but after testing with pieces placed together on the board it became clear that segmentation was necessary. This decision added complexity to both the labeling process and the model, but it was essential for the accuracy the application needed.

5.2.2. Board Detection — OpenCV vs YOLO

Before the pieces can be mapped to the digital grid, the application first needs to locate the board in the photo. Two approaches were considered and tested for this:

- **OpenCV HSV Color Filter**: The first approach used OpenCV to detect the board by filtering the image based on the board's color using an HSV color range. While this worked under good lighting conditions, it proved to be

unreliable in practice. Any change in lighting, shadows, or camera angle caused the color range to shift, making the filter miss the board entirely or detect the wrong areas. This made it too fragile to rely on as the primary detection method.

- **YOLO Board Detection**: The second approach was to add the board as its own class (class 8) in the dataset and let the YOLO model detect it alongside the puzzle pieces. Since the model learned to recognize the board from thousands of images across different lighting conditions and angles, it is far more robust than a simple color filter. Both the pieces and the board are now detected in a single model pass, which simplifies the pipeline.

For these reasons, YOLO was chosen as the primary method for board detection. However, OpenCV is still used in the pipeline — not for detection, but for image manipulation tasks such as perspective warping, masking, and grid mapping. The OpenCV HSV filter is kept as a fallback in case the YOLO model fails to detect the board, but in practice YOLO handles this reliably.

Switching to YOLO for board detection was one of the most impactful decisions in the project, but also one of the most time-consuming. It meant regenerating the entire synthetic dataset in Blender with the board included, and re-labeling all the real images in Roboflow with the board as an additional class. This set the project back by several days, but the improvement in reliability made it worth the effort. It was a clear example of how sometimes the right solution requires redoing earlier work rather than building on top of a fragile foundation.

5.3. Model Training

Training a computer vision model requires more than just feeding data into an algorithm. Several decisions had to be made before and during the training process, from choosing the right environment to run the training, to selecting the most suitable model variant, to figuring out the best way to combine the synthetic and real data. This section covers those decisions, the tools used, and the two training approaches that were explored and compared.

5.3.1. Training Environment

Training was done using Google Colab with a T4 GPU, which provides 15GB of VRAM. Running the training locally was not a practical option — the laptop available for this project has an NVIDIA GeForce GPU with only 4GB of VRAM, which is not sufficient for training a computer vision model efficiently. Google Colab provided a free and accessible solution with enough computational power to train the model in a reasonable amount of time.

5.3.2. Model Selection

The model used for this project is YOLOv26 Nano, the lightest variant of the YOLOv26 family. YOLOv26 is one of the most recent releases in the YOLO series, released just a few months ago, bringing improvements in speed and accuracy especially for edge and on-device deployment. The choice of Nano was driven by the fact that the model needs to be loaded directly into the browser application and run on the user's device. A heavier model would be too slow to load and run in real time, especially on mobile phones. The table below shows the different YOLOv26 variants and their characteristics:

Variant	mAP@50-95	CPU Inference	Params	Use Case
YOLO26-N	40.9%	38.9ms	2.4M	On-device / mobile
YOLO26-S	48.6%	87.2ms	9.5M	Server MVP, good accuracy/speed balance
YOLO26-M	53.1%	220.0ms	20.4M	Server production, best accuracy for latency budget

Table 1: Yolo Versions

For this project, YOLO26-N was selected because it is the fastest and lightest variant, making it suitable for running directly in the browser. Although it has a lower mAP than the larger variants, the accuracy is still sufficient for detecting the puzzle pieces reliably.

5.3.3. ONNX Export

After training, the model is exported from its original PyTorch format (.pt) to the ONNX format (.onnx). The reason for this is that the .pt file is a PyTorch-specific format that requires Python and PyTorch to run, which means it cannot be used directly in the browser. ONNX (Open Neural Network Exchange) is a universal model format that can run on many different platforms and runtimes. In this project, the ONNX model is loaded directly into the browser using ONNX Runtime Web, which runs the model via WebAssembly (WASM) — no Python, no backend, no server needed.

5.3.4. Training Methods

Two different training approaches were tested to find the best way to combine the synthetic and real data.

Method 1 — Pretrain on Synthetic, Finetune on Real: The model was first trained on 1800 synthetic images generated in Blender, using an 80/20 train/validation split. Once trained, the model was then finetuned on the 240 real images labeled in Roboflow, again with an 80/20 split. The idea behind this approach is to first let the model learn the general shape and color of the pieces from the synthetic data, and then adapt it to real-world conditions through finetuning.

Method 2 — Merged Dataset with Separate Validation: The synthetic and real images were merged into a single dataset. For the training split, 100% of the synthetic images and 80% of the real images were used for training. The validation set was composed entirely of the remaining 20% of the real Roboflow images. This approach ensures that the validation always reflects real-world performance, while still making full use of the synthetic data for training.

5.4. Application Architecture

The application architecture went through an important change during the project. What started as a full-stack system with a Python backend evolved into a fully browser-based application, following a request from Smart NV. This section explains the original architecture, why and how it changed, and the tools and technologies used in the final version.

5.4.1. Original Architecture — Backend + Frontend

The original architecture was agreed upon with Smart NV at the start of the project and aligned with the project charter. It consisted of two parts working together: a Python FastAPI backend and a React frontend.
The workflow was as follows: the user takes a photo of the physical board in the React app, the photo is sent to the backend via a REST API call, the backend runs the full computer vision pipeline using Python, YOLO, and OpenCV, and the resulting board state is returned to the frontend as JSON. The frontend then displays the board state and handles the solver logic.
This approach worked well and was fully functional. The backend ran on port 8000 locally, and the frontend proxied all API calls to it. However, it required two separate processes to be running at the same time, and users needed a Python environment set up on their machine to run the backend.

5.4.2. Move to Browser-Only

During the project, Smart NV requested that the backend be removed and that everything run in the browser instead. This was a significant architectural change, as the entire computer vision pipeline had to be rewritten from Python to JavaScript.

The key challenge was running the YOLO model in the browser. The solution was to export the trained model from its PyTorch format to ONNX, and then use ONNX Runtime Web to run it directly in the browser via WebAssembly (WASM). This allowed the model to run client-side without any server or Python environment.
The full CV pipeline, which was originally written in Python using OpenCV, was then ported to pure JavaScript. This included the image preprocessing, mask decoding, board corner detection, perspective warp, and grid mapping. The result was a fully browser-based application that requires no backend, no installation, and can be deployed as a simple static site.

Porting the entire Python pipeline to JavaScript was a challenge, since many of the functions that OpenCV provides out of the box in Python had to be reimplemented manually in JavaScript, such as the convex hull, minimum area rectangle, and the homography computation. This process also meant learning how ONNX Runtime Web works and understanding the differences between running a model in Python versus in the browser. In the end, moving to a browser-only architecture turned out to be one of the best decisions in the project, as it made the application much simpler to deploy and use.

5.4.3. Tools & Technologies

The frontend was built using React and Vite. Although the original project charter specified TypeScript, the decision was made to use JavaScript instead, as it was faster to develop with and sufficient for the scope of this project.
ONNX Runtime Web was used to run the YOLO model in the browser. It uses WebAssembly under the hood, which allows near-native performance for running the model client-side. The ONNX model file is served as a static asset and cached by the browser after the first load, meaning subsequent scans are fast and the app can even work offline.

The current version can be deployed as a static site. Looking ahead, Smart NV's goal is to publish the application on the Play Store and App Store as a native mobile app, which is planned for a future phase of the project.

6. Realization

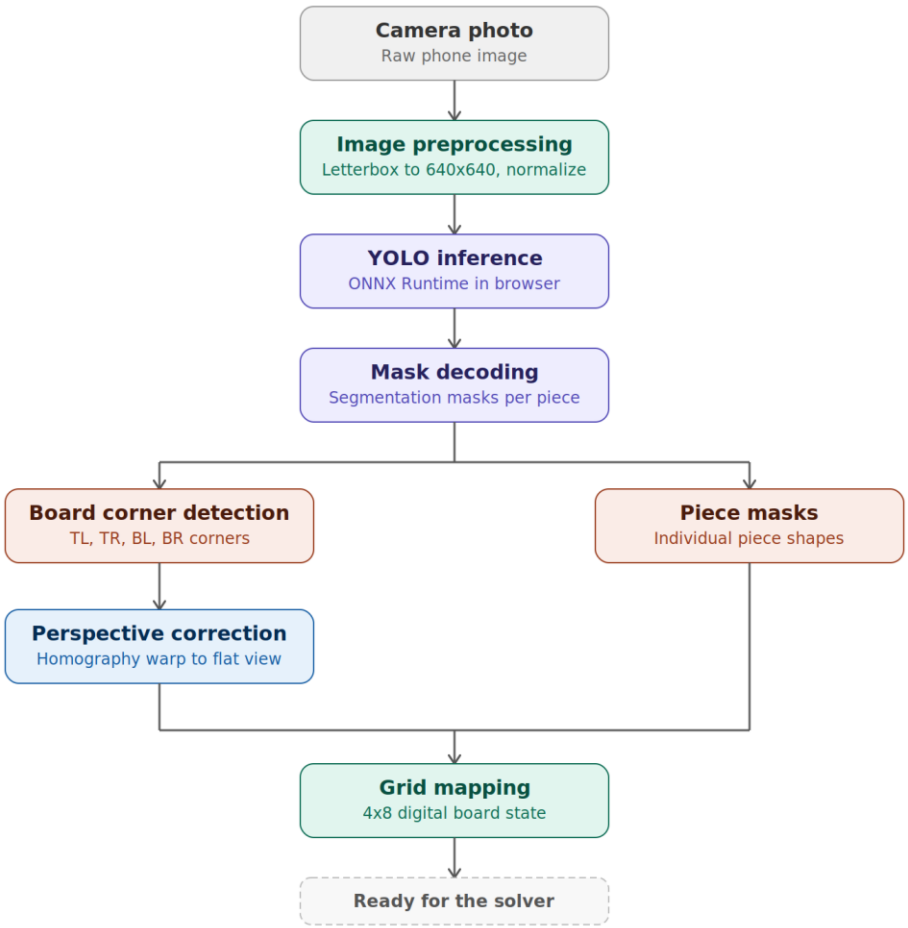
With the analysis decisions in place, including the dataset, the model, and the architecture, the next step was to turn all of that into a working application. The

following sections describe how each component was built in practice: the computer vision pipeline that processes a photo into a digital board state, the solver logic that determines if the puzzle is still solvable and generates hints, and the application that ties everything together into a usable interface.

6.1. Computer Vision Pipeline

The computer vision pipeline is the core of the application. Its job is to take a raw photo from the user's camera and convert it into a structured 4×8 board state that the solver can work with. This process goes through several steps: the image is first preprocessed to match the model's expected input, then fed through the YOLO model to detect pieces and the board, after which the masks are decoded, the camera angle is corrected using perspective transformation, and finally each detected piece is mapped to its position on the digital grid. The entire pipeline runs locally in the browser using ONNX Runtime, meaning no server or backend is involved.

Figure 5: Overview of the computer vision pipeline, from camera photo to digital board state.



6.1.1. Image Preprocessing

When a user takes a photo of the puzzle board through the application, the image comes in whatever resolution and aspect ratio the device's camera produces. A modern smartphone, for example, might capture a photo at 4032×3024 pixels. However, the YOLO model was trained on images with a fixed resolution of 640×640 pixels, which means every input image must be transformed to match that exact size before inference can begin.

Simply stretching the image to 640×640 would distort the proportions of the puzzle pieces, which could cause the model to miss detections or confuse one piece for another. To avoid this, the pipeline uses a technique called letterboxing. The image is scaled down proportionally so that its longest side fits within 640 pixels, and the remaining space is filled with a neutral gray padding. This preserves the original aspect ratio while producing the square input the model expects.

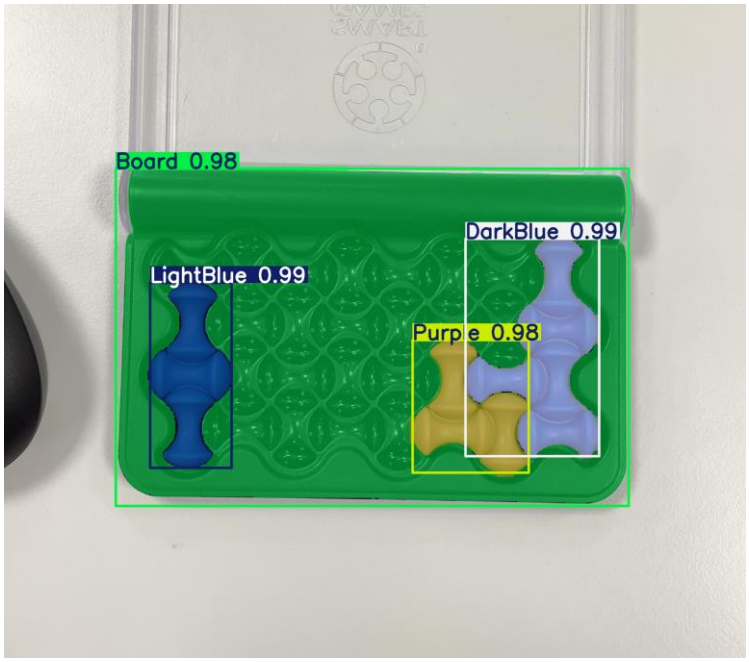
6.1.2. YOLO Inference in the Browser

Once the image is preprocessed into the correct tensor format, it is passed into the YOLO model for inference. The model runs directly in the browser using ONNX Runtime Web, which executes the neural network through WebAssembly. The model file and the ONNX runtime are loaded once when the application starts, so scanning is fast and no additional loading is needed between scans.

The model returns two outputs. The first contains a list of detections, where each detection holds a confidence score, a class ID identifying which piece or board it is, and 32 mask coefficients that are used later to reconstruct the exact shape of the detected object. The second output contains 32 prototype masks at a resolution of 160×160, which serve as building blocks for generating the final segmentation masks.

Since the model can return many detections with low confidence, the pipeline filters out anything below a confidence threshold of 0.25 and keeps only the highest scoring detection per class. Since each puzzle piece can only appear once on the board, there is no need to keep duplicate detections. The result is a clean list of detections, each representing one puzzle piece or the board, ready for mask decoding in the next step.

Figure 6: YOLO model detection output showing three detected pieces and the board with segmentation masks and confidence scores.



6.1.3. Mask Decoding

After inference, the model does not return ready-to-use masks for each detected piece. Instead, it returns 32 mask coefficients per detection and a set of 32 prototype masks at a resolution of 160×160 pixels. To get the actual shape of a detected piece, the pipeline combines these two outputs. Each detection's 32 coefficients are multiplied with the 32 prototype masks and summed together, producing a single 160×160 mask for that detection. A sigmoid function is then applied to convert the raw values into a range between 0 and 1, where values close to 1 represent pixels that belong to the piece and values close to 0 represent background.

Since 160×160 is too small for accurate grid mapping, the mask is upscaled to 640×640 using nearest-neighbour interpolation. During this upscaling, the mask is also clipped to the detection's bounding box area so that the shape does not bleed into neighbouring pieces. This clipping step is important because pieces are placed directly next to each other on the board, and without it, one piece's mask could overlap into another piece's area.

The result is a clean, pixel-level mask at full resolution for each detected piece and the board, ready to be used in the next steps for corner detection and grid mapping.

6.1.4. Board Corner Detection

Once the masks are decoded, the pipeline needs to locate the board in the image before it can map any pieces to the grid. It does this by taking the board's mask (class 8) and finding its four corners.

The board mask is a collection of pixels, so the pipeline first computes a convex hull around it, which is the smallest shape that wraps around all the board's pixels. From that shape, it calculates the smallest possible rotated rectangle that fits around it. This gives four corner points, which are then sorted into a consistent order: top-left, top-right, bottom-right, and bottom-left. This is what you see in the screenshot you showed earlier, with the TL, TR, BL, and BR labels.

These four corners are essential for the next step, because they tell the pipeline exactly how the board is positioned and angled in the photo, which is needed to correct the camera perspective.

Figure 7: Board corner detection showing the four corners (TL, TR, BL, BR) identified from the board's segmentation mask.

6.1.5. Perspective Correction

When a user takes a photo of the puzzle board, the camera is rarely perfectly centered above it. The photo is usually taken at an angle, which means the board

appears tilted or skewed in the image. If the pipeline tried to map pieces to the grid directly from this angled photo, the positions would be inaccurate.

To fix this, the pipeline uses a technique called homography. Using the four board corners detected in the previous step, it calculates a transformation matrix that maps those corners to a flat, rectangular shape. This is the same idea as when you straighten a photo of a document so it looks like it was scanned from directly above.

Instead of transforming the entire image pixel by pixel every time, the pipeline precomputes a lookup table that stores where each pixel in the corrected image comes from in the original image. This makes the warping fast enough to run in real time in the browser, because the heavy calculation only happens once and is then reused for every piece mask that needs to be corrected.

After this step, the board and all the piece masks are transformed as if the photo was taken from directly above, making it possible to accurately map each piece to the grid.

Early on, handling photos taken at different orientations caused some issues with the perspective warp. After testing a few approaches, including EXIF metadata and corner reordering, the mentor suggested a simpler solution: adding a camera overlay with the board silhouette so the user always positions the board correctly before taking the photo. This removed the need for any rotation handling in the code.

Figure 8: Board after perspective correction, showing the board as if photographed from directly above.

6.1.6. Grid Mapping

With the perspective corrected, the pipeline can now determine which grid cells each piece occupies. The IQ Waves board has a 4×8 grid, but the cells are not simple squares. They have an hourglass-like shape that matches the wave pattern of the puzzle. The exact shape and position of each cell is defined in a configuration file that the pipeline loads when the application starts.

For each cell, the pipeline knows exactly which pixels belong to it. To determine if a piece occupies a certain cell, it checks how many of that piece's mask pixels overlap with the cell's pixels. If the overlap exceeds a threshold of 55%, the cell is considered occupied by that piece. This threshold prevents small detection errors or mask edges from being counted as actual placements.

Once every piece has been checked against every cell, the pipeline produces a complete 4×8 board state: a grid where each cell is either empty or contains a specific piece. This is the final output of the computer vision pipeline, and it is exactly what the solver needs to determine if the puzzle is still solvable, generate a hint, or compute the full solution.

The grid mapping went through several iterations before reaching this final version. The first attempt used a uniform grid of squares, but these did not match the actual shape of the cells and caused neighbouring pieces to be misclassified. The second attempt used rectangles with alternating orientations to better fit the hourglass pattern, which improved accuracy but still had edge cases where cells were picking up parts of their neighbours. The final solution was to define each cell individually as a custom hourglass-shaped polygon, stored in a JSON configuration file. To make this process easier, a separate calibration tool was built that allowed fine-tuning the position and shape of each cell visually on top of a real board image.

Figure 9: Grid mapping result showing each cell of the 4×8 board labeled with the detected piece name.

6.2. Solver Logic

Once the computer vision pipeline produces a digital board state, the solver takes over. Its job is to determine whether the puzzle can still be solved with the remaining pieces, provide hints for the next move, or compute the full solution.

The board is represented as a 4×8 grid of 32 cells, where each cell is either empty or occupied by a specific piece. This is the same structure that the grid mapping step produces, so the solver can work directly with the output of the pipeline without any conversion.

6.2.1. Piece Orientations

Each puzzle piece has a unique shape, but it can be placed on the board in multiple orientations. A piece can be rotated by 90, 180, or 270 degrees, and it can also be flipped, which means each piece can have up to eight possible orientations. Some orientations may result in the same shape, for example a symmetrical piece might look the same after a 180 degree rotation. To avoid trying the same shape twice, the solver generates all eight possible orientations for each piece, normalizes them so they all start from the same reference point, and then removes any duplicates. This means the solver only works with the truly unique orientations of each piece, which keeps the solving process efficient.

6.2.2. Backtracking Algorithm

The solver uses a backtracking algorithm to find a solution. It works by placing one piece at a time onto the board and checking if the placement is valid, meaning the piece fits within the grid and does not overlap with any already placed piece. If the placement works, it moves on to the next piece. If at any point no valid placement can be found for the current piece, the solver backtracks: it removes the last placed piece and tries a different orientation or position for it.

To speed up this process, the solver does not simply try pieces in a fixed order. Instead, it always picks the most constrained piece next, which is the piece that has the fewest valid placements remaining. This is a common optimization in backtracking algorithms because if a piece can only fit in two positions, it makes sense to try that one first rather than a piece with twenty possibilities. This significantly reduces the number of attempts the solver has to make before finding a solution or determining that none exists.

6.2.3. Hint System and Solvability Check

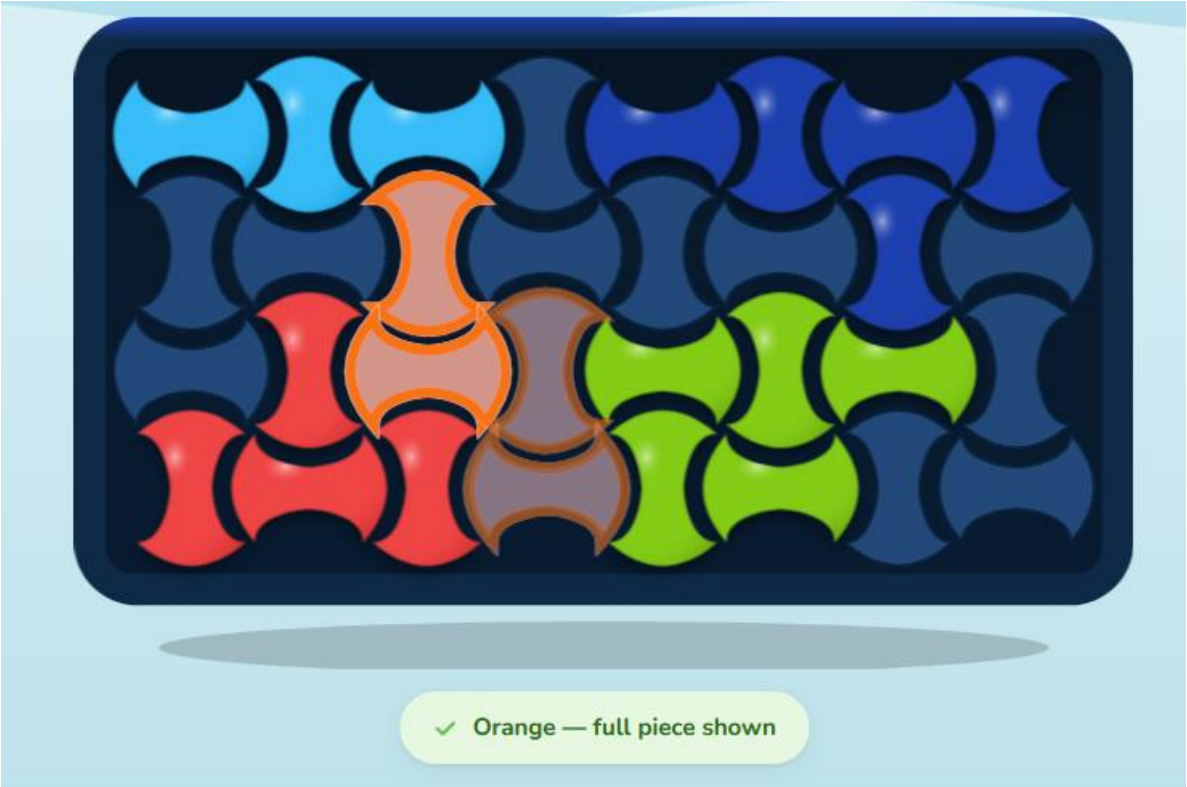
The solver provides two main features to the application: a solvability check and a hint system. The solvability check runs the solver from the current board state and

reports whether a solution still exists. If the current placement of pieces leads to a dead end, the application does not just tell the user the puzzle is unsolvable. Instead, it finds the smallest number of pieces that need to be removed to make the board solvable again, and highlights those pieces so the user knows exactly what to take out as shown in Figure 10

Figure 10: The removal hint feature, highlighting the Dark Blue piece in red to indicate it needs to be taken out to make the puzzle solvable again.

The hint system uses the same solver, but instead of returning the full solution, it returns only the next piece to place. The application then reveals this hint progressively: the first tap shows a single glowing cell to point the user in the right direction, a second tap reveals two cells to show the direction of the piece, and a third tap shows the complete piece placement. Figure 11 shows an example of the full piece placement revealed after the third tap. This way the user always gets the smallest amount of help needed, and can choose how much they want to see before placing the piece themselves.

Figure 11: The progressive hint system showing the full Orange piece placement after the third tap.



6.3. Application

With the computer vision pipeline handling the detection and the solver providing the logic, the final step was to bring everything together into an application that a child can actually use. The application is built entirely in the browser using **JavaScript** with **React** as the UI library and **Vite** as the build tool, meaning there is no backend or server involved. The user opens the app, scans their physical puzzle board with the camera, and from that point on everything happens locally on their device: the model runs the detection, the solver computes hints or checks solvability, and the interface updates in real time. The goal was to make the experience feel simple and playful, even though there is a full AI pipeline running behind it.

6.3.1. Project Structure

The application is organized into a small set of files, each with a clear responsibility. The main file is App.jsx, which contains the entire user interface: the puzzle board, the camera modal, the buttons, and all the interaction logic. The computer vision

pipeline lives in pipeline.js, which handles everything from image preprocessing to grid mapping. The solver logic is in solver.js, containing the backtracking algorithm, hint generation, and board validation. A separate voice.js module handles the spoken hints and sound effects using the browser's built-in speech and audio APIs.
Alongside the code, the application loads two data files at startup. The first is model.onnx, which is the trained YOLO model that runs the piece detection. The second is grid_config.json, which defines the exact shape and position of every cell on the 4×8 hourglass grid, used by the pipeline to map detected pieces to their board positions. There is also a challenges.json file that contains preset puzzle configurations organized by difficulty level, allowing users to pick a challenge instead of scanning a physical board.

6.3.2. User Interface

The interface was designed to feel approachable for children while still being functional. The main screen shows the puzzle board rendered as an SVG element, where each of the 32 cells is drawn as an hourglass shape matching the real IQ Waves grid. The board sits inside a dark navy tray that resembles the physical puzzle, with a subtle shadow and rounded edges to give it a clay-like appearance. When a piece is detected on a cell, that cell is filled with the piece's color along with a radial gloss effect to make it look like the actual plastic piece.
Below the board, a status pill keeps the user informed about what is happening, such as how many pieces are placed, whether the puzzle is solvable, or if something went wrong during scanning. At the bottom of the screen, three main buttons provide the core actions: Scan to open the camera, Help me to request a hint, and Solved to fill in the complete solution.
The camera modal opens as a full-screen overlay with a live video feed from the device's camera. To help the user position the puzzle correctly, a semi-transparent silhouette of the board is drawn on top of the video, showing the exact outline of the tray, the hinge area, and all 32 hourglass cells. The user lines up their physical board with this guide and taps the capture button. If the device supports it, a torch button is also available to improve lighting conditions.
When something goes wrong during detection, the application shows friendly error modals instead of technical error messages. If the board is not detected, the modal suggests making sure the whole board fits in the frame. If a piece is detected in an impossible shape, the modal suggests trying again with better lighting.

Figure 12: The IQ Waves main screen showing the empty board, status pill, and the three action buttons: Scan, Help me, and Solved.

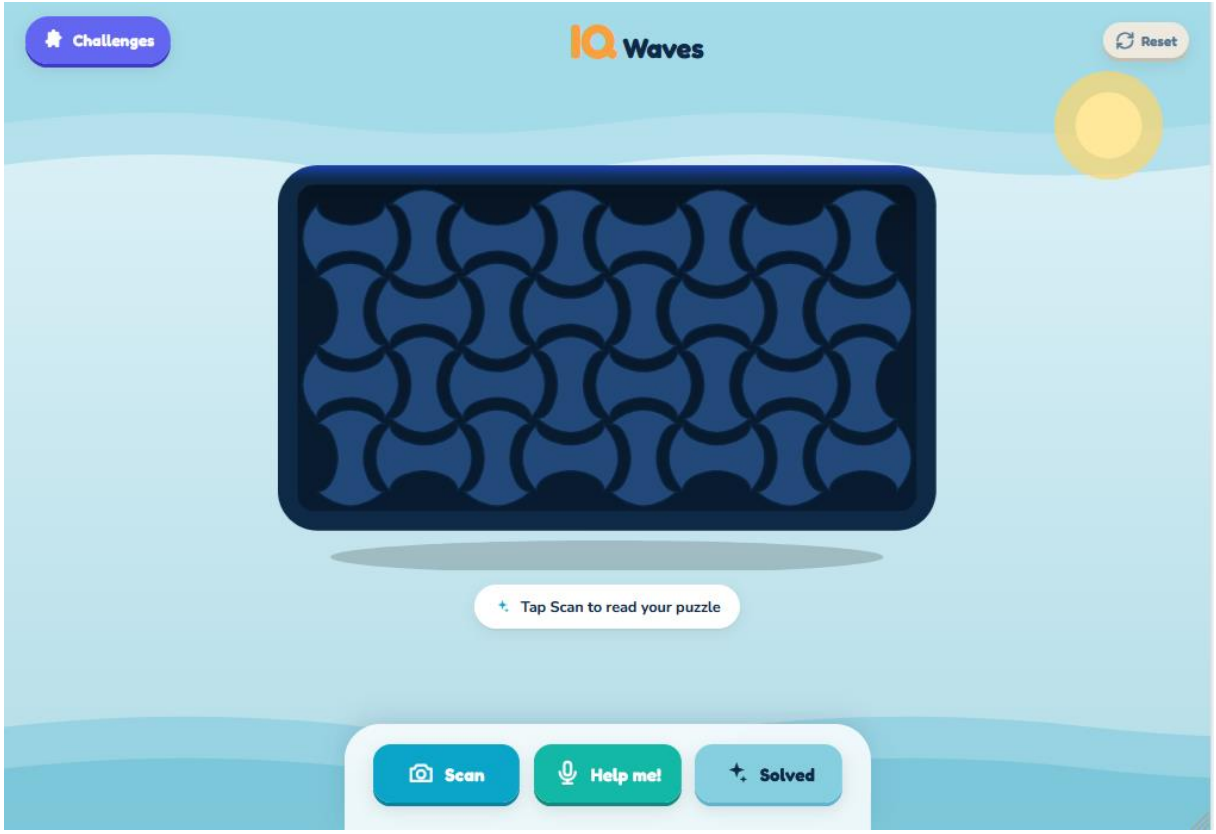
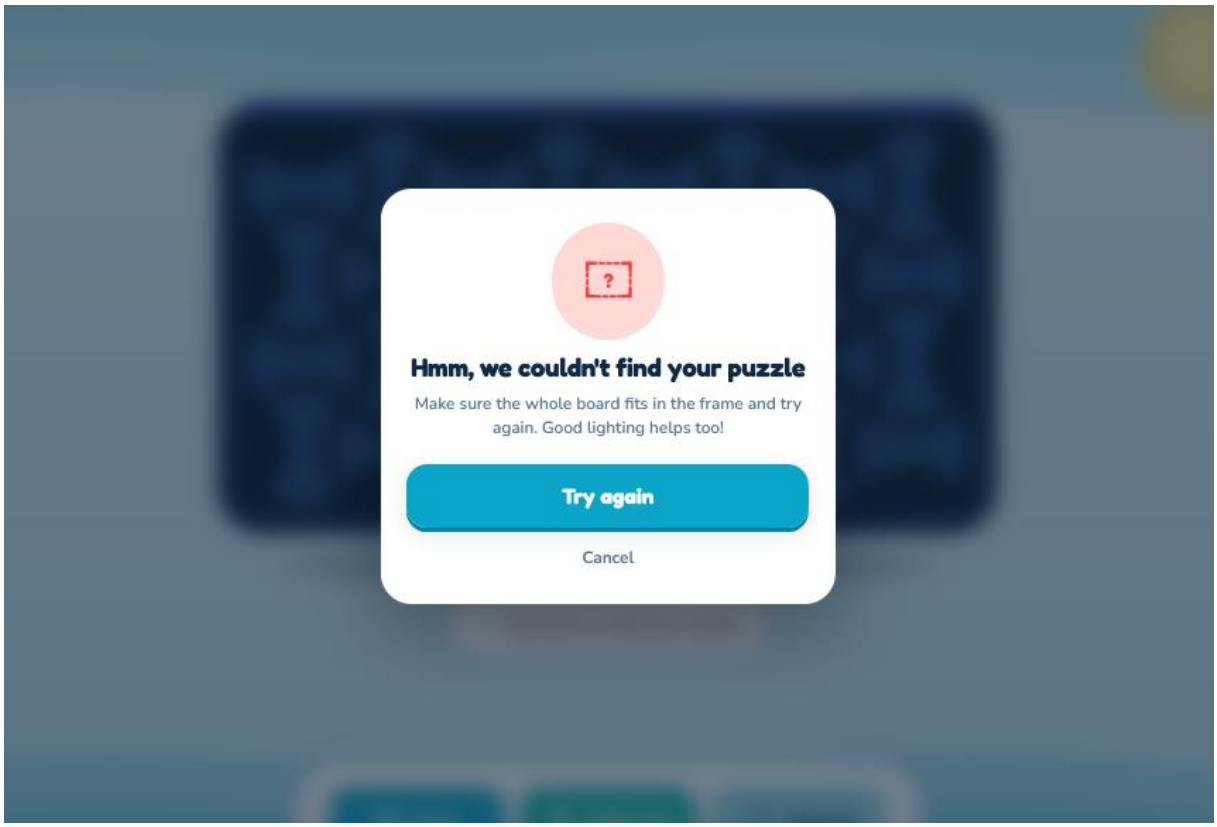


Figure 13: Error modal displayed when the board is not detected, suggesting the user to reposition the puzzle and try again.



6.3.3. User Flow

The typical flow starts when the user taps the Scan button, which opens the camera. After lining up the physical board with the silhouette guide and taking a photo, the pipeline processes the image and returns a 4×8 board state. The application validates this result by checking if every detected piece matches a real shape that exists in the game. If everything checks out, the board updates with the detected pieces shown in their correct positions.
From there, the user can tap Help me to get assistance. The hint system works progressively: the first tap highlights a single glowing cell on the board and a spoken voice tells the child which piece to look for and roughly where it goes. If the user needs more help, tapping again reveals two cells to show the direction of the piece, and a third tap reveals the complete placement. This way the child stays in control of how much help they receive, rather than getting the full answer at once.
Once the child places the physical piece on the real board, they tap the "I placed it!" button. The application then adds that piece to the digital board, plays a cheerful sound, and shows a brief glow around the newly placed cells so the child can visually confirm the placement is correct. The status updates to show progress, and the child can tap Help me again for the next piece.

If the solver determines that the current board state is unsolvable, the application does not simply show an error. Instead, it figures out the fewest pieces that need to be removed and highlights them in red with a pulsing outline. A spoken message tells the child which pieces to take out. After removing them and tapping the confirmation button, the board is corrected and the child can continue solving.
When all eight pieces are placed, the application celebrates with a confetti animation, a fanfare sound, and a spoken congratulation. From there, the user can pick a new challenge using the challenges menu, which offers preset puzzles organized into five difficulty levels ranging from Starter to Wizard.

Figure 14: The progressive hint showing two glowing cells for the Red piece, with the "I placed it!" and "More help" buttons visible.

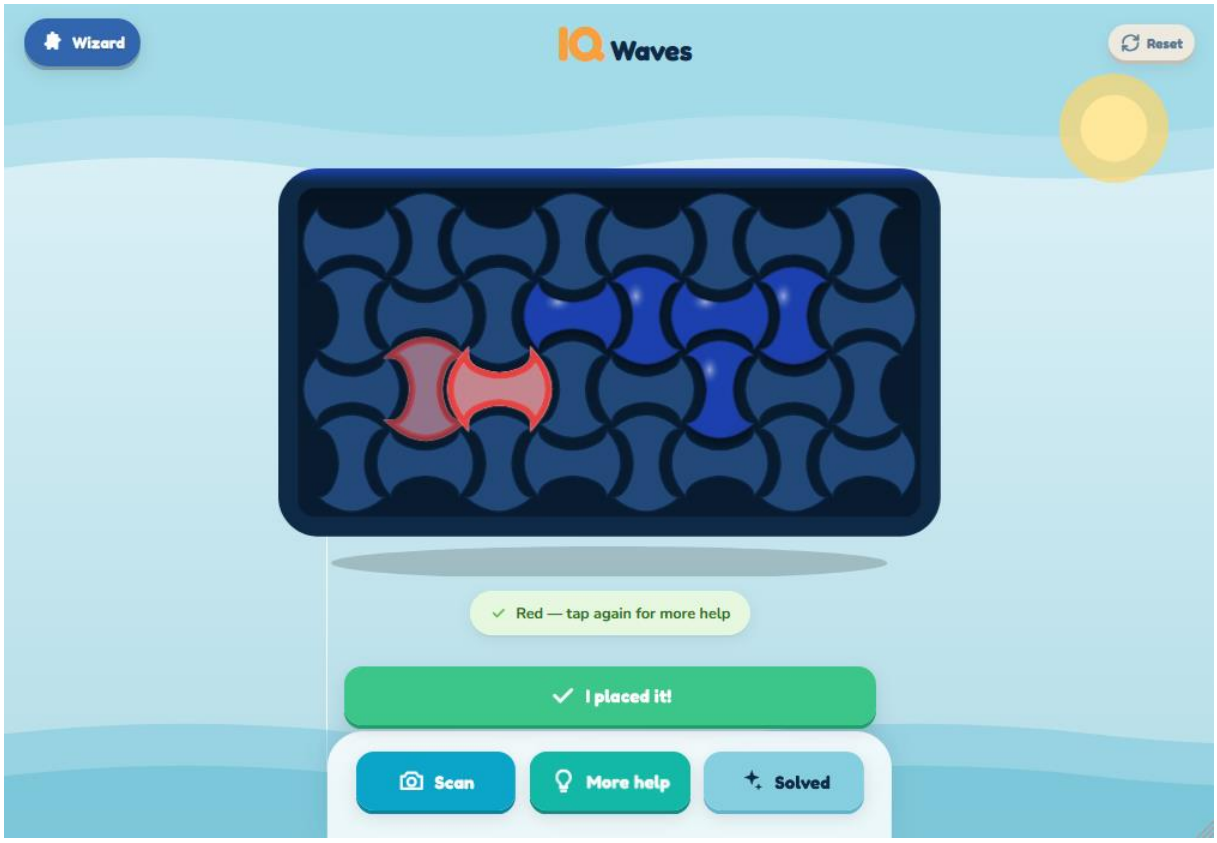


Figure 15: The board after placing the Red piece, with a glow effect confirming the placement and the status showing 2/8 done.

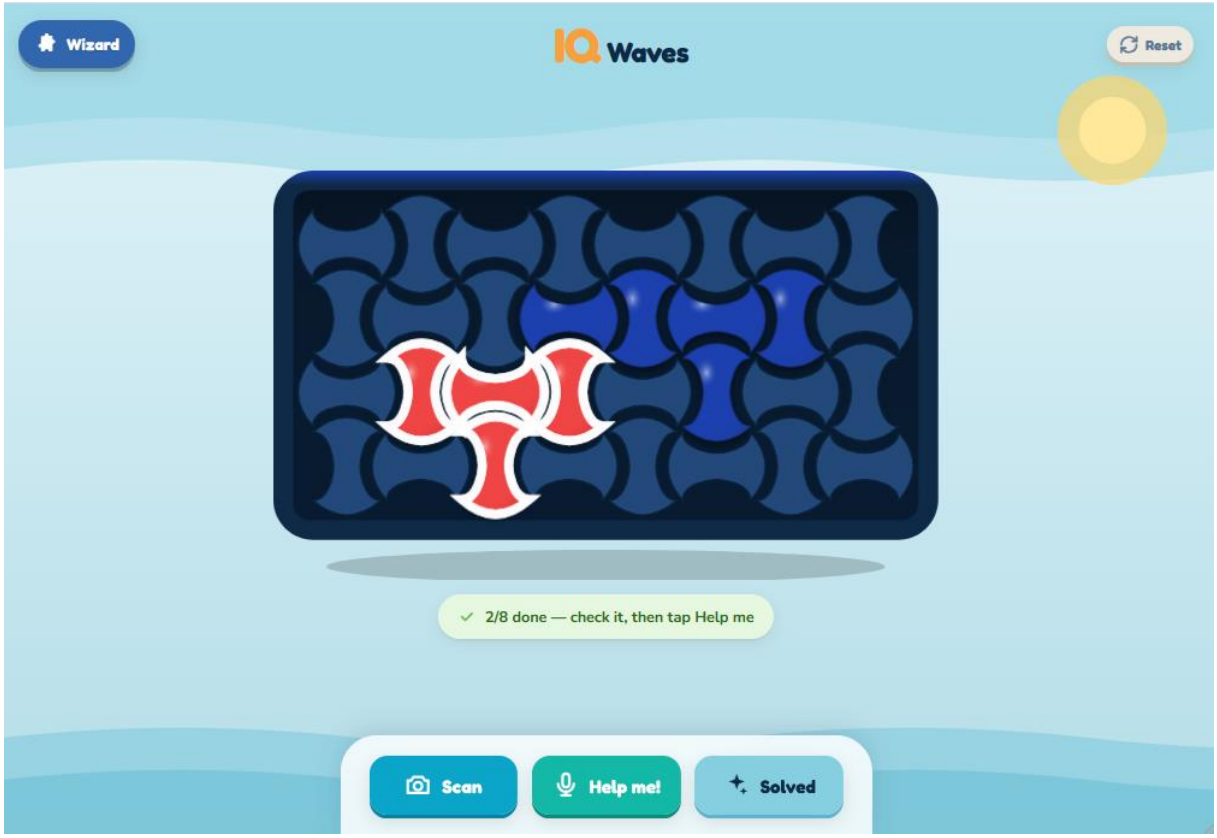


Figure 16: The solved state with all 8 pieces placed, confetti animation, and the "New challenge" button appearing.

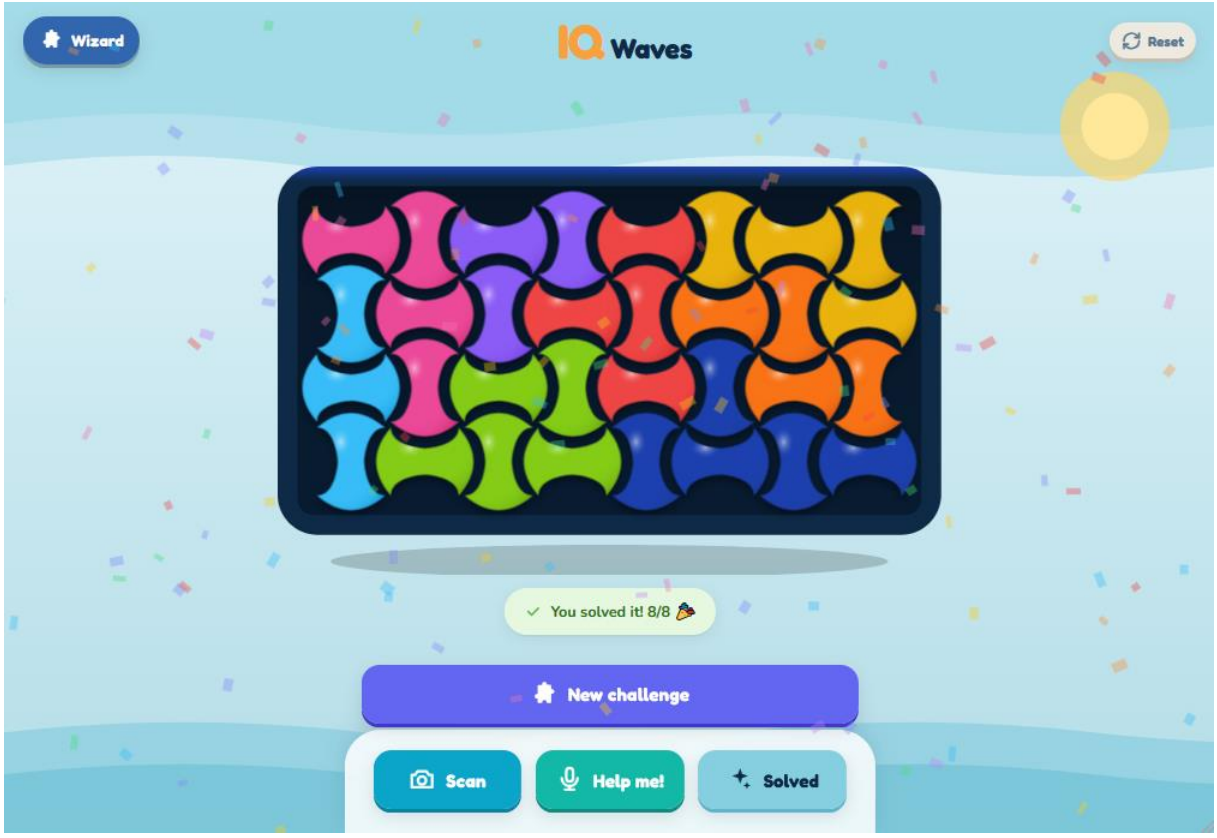
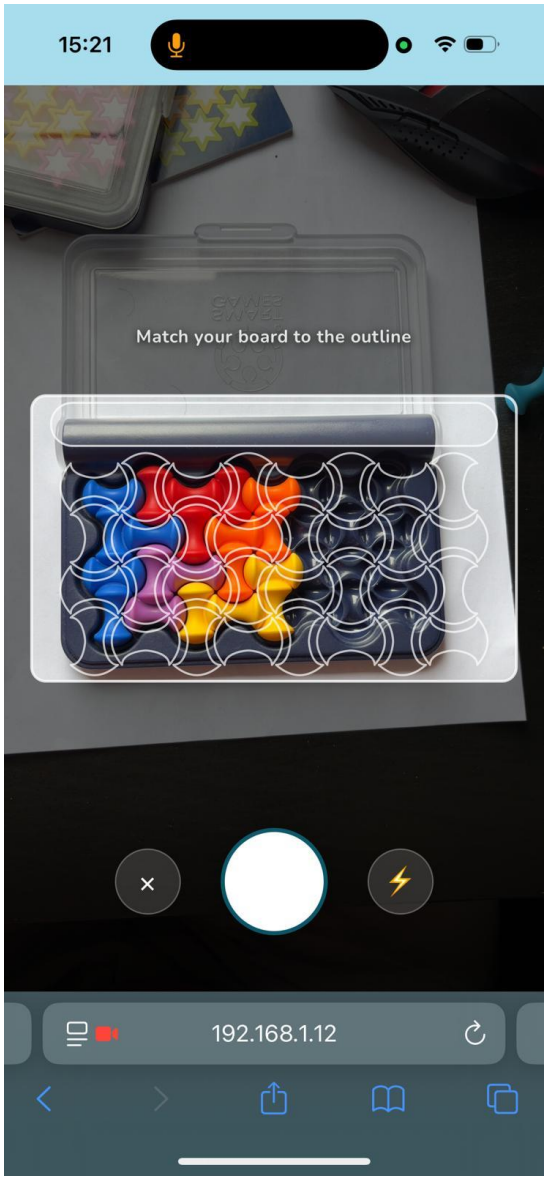


Figure 17: The IQ Waves camera modal with the hour-glass silhouette guide on a mobile device.



6.3.4. Voice Assistant

To make the application more accessible and engaging for young children, a voice assistant was added that speaks all hints and feedback out loud. This is built using the browser's built-in Web Speech API, which means it works offline and does not require any external service or API key.
When a hint is given, the voice describes which piece to look for and where on the board it goes, using playful and encouraging language. The phrases are randomized so the voice never says the same thing twice in a row, which keeps it feeling natural rather than robotic. When the child places a piece, the voice celebrates with a cheerful line and tells them how many pieces are left. If the puzzle is complete, it gives a bigger congratulation.

Alongside the spoken hints, the application uses the Web Audio API to generate simple sound effects. A short ding plays when a piece is placed, a soft pop accompanies each hint, and an ascending fanfare plays when the entire puzzle is solved. These sounds are synthesized directly in the browser, so no audio files need to be loaded.
The voice automatically selects the most natural sounding English voice available on the device, preferring newer neural or online voices over the default system voice. The pitch and speed are slightly randomized with each sentence to avoid a flat, repetitive tone.

7. Training Results

Training a model that works reliably in the real world was not a one-shot process. Two different approaches were tested to find the best way to combine the synthetic and real data. The first method pretrained the model on the synthetic dataset and then finetuned it on the real images, while the second method merged both datasets together and trained the model on everything at once. The following sections present the results of each approach and compare them to determine which one was used in the final application.

7.1. Method 1: Pretrain and Finetune

The first training approach was built around the idea of learning in two stages. Instead of showing the model both synthetic and real data at the same time, the model was first trained entirely on the synthetic dataset to learn the general shapes, colors, and structure of the puzzle pieces. Once that foundation was in place, the model was then finetuned on the smaller set of real images so it could adapt to how the pieces actually look in real photographs, with real lighting, shadows, and camera angles. The advantage of this approach is that the model gets a strong baseline from the large synthetic dataset, and only needs to make small adjustments when it sees real data.

7.1.1. Pretraining on Synthetic Data

The first stage of training used the synthetic dataset generated in Blender, consisting of 1800 images split 80/20 into 1440 training images and 360 validation images. The model used was YOLOv26 Nano Segmentation, trained for up to 200 epochs with a batch size of 16 and an image size of 640×640 pixels. Early stopping was enabled with a patience of 25 epochs, meaning the training would stop automatically if the model showed no improvement for 25 consecutive epochs.

Several augmentations were applied during training to help the model generalize better. Color augmentations such as hue, saturation, and brightness shifts were used to prevent the model from relying too heavily on exact colors, encouraging it to also learn the shapes of the pieces. Geometric augmentations included horizontal and vertical flips, small rotations of up to 10 degrees, slight translations, and random scaling. Mosaic augmentation was disabled because the puzzle images already contain multiple objects in a fixed layout, and combining four images into one would create unrealistic training samples.

The training ran for 96 epochs before early stopping was triggered, with the best results recorded at epoch 71. The model achieved a mask mAP50 of 0.995 and a mask mAP50-95 of 0.99 overall, with precision at 0.995 and recall at 0.989. Every individual piece scored above 0.98 in both precision and recall, showing that the model learned to distinguish all eight pieces and the board reliably from synthetic data alone.

The confusion matrix confirms this: every piece is classified correctly with no confusion between classes. The only false positives come from background regions occasionally being predicted as pieces, most notably LightBlue with 4 and Purple with 3 false positives from background. These are minor and get filtered out by the confidence threshold in the application.

Although the results on synthetic data were strong, the model had only seen rendered images at this point. The next step was to finetune it on real photographs so it could handle actual lighting conditions, camera angles, and real-world variations.

Figure 18: Per-class results of the pretrained model

Class	Images	Instances	Box(P	R	mAP50	mAP50-95)	Mask(P	R	mAP50	mAP50-95):
all	776	3383	0.995	0.989	0.995	0.994	0.995	0.989	0.995	0.99
Red	394	394	0.997	0.989	0.995	0.995	0.997	0.989	0.995	0.99
HotPink	408	408	0.998	0.988	0.995	0.995	0.998	0.988	0.995	0.991
DarkBlue	418	418	0.999	0.988	0.995	0.995	0.999	0.988	0.995	0.994
Orange	416	416	0.984	1	0.995	0.995	0.984	1	0.995	0.993
LightBlue	389	389	0.997	0.999	0.995	0.995	0.997	0.999	0.995	0.981
Green	412	412	0.994	0.993	0.995	0.995	0.994	0.993	0.995	0.995
Yellow	419	419	1	0.997	0.995	0.991	1	0.997	0.995	0.989
Purple	380	380	0.997	0.983	0.995	0.995	0.997	0.983	0.995	0.984
Board	147	147	0.987	0.966	0.994	0.994	0.987	0.966	0.994	0.992

Figure 19: Training and validation metrics of the pretrained model.

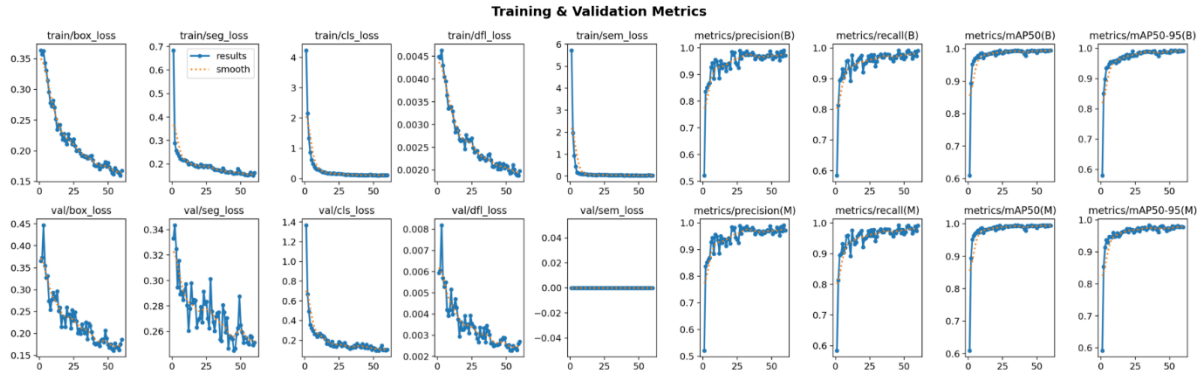
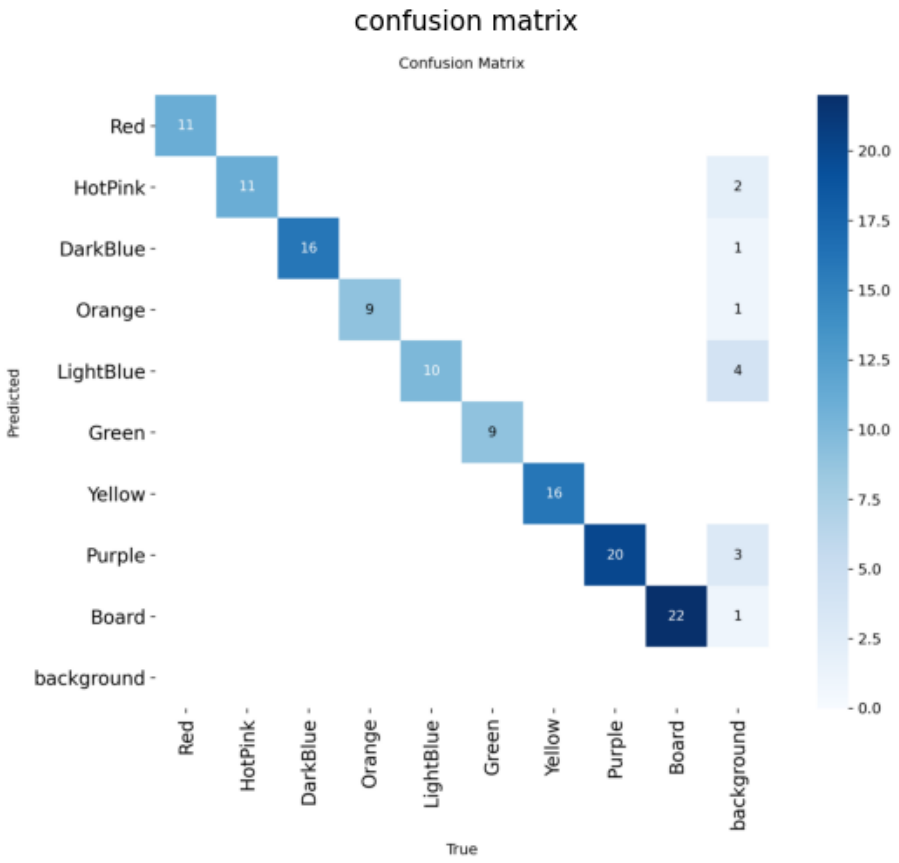


Figure 20: Confusion matrix of the pretrained model on the validation set



7.1.2. Finetuning on Real Data

With the pretrained model already performing well on synthetic data, the next step was to finetune it on real photographs so it could handle actual lighting, shadows, and camera angles. The finetuning dataset consisted of 205 training images and 33 validation images, all taken with a real camera and labeled manually in Roboflow.

To preserve the knowledge the model had already learned from the synthetic data, the first 16 layers of the network were frozen during finetuning. This means only the later layers were updated, allowing the model to adapt to real images without forgetting the shapes and patterns it already knew. The learning rate was also lowered to 0.001 with a final learning rate factor of 0.01, which prevents the model from making large changes too quickly and overwriting useful weights.

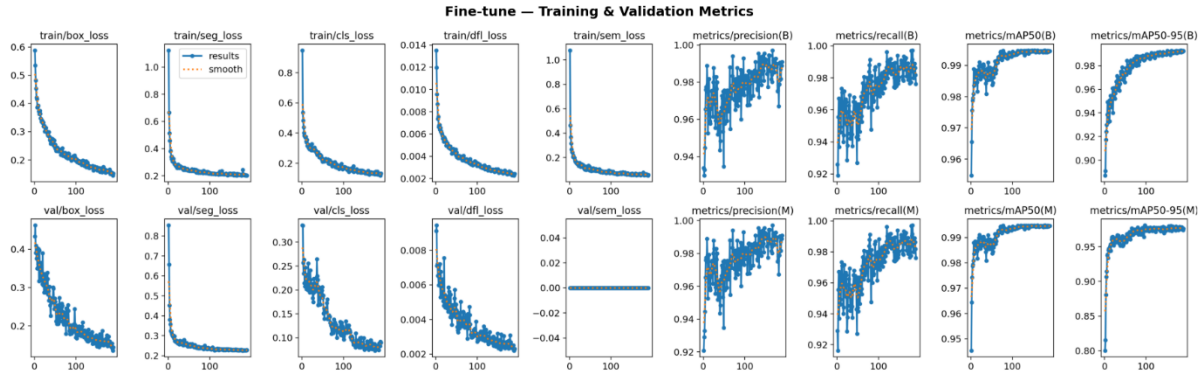
The augmentations were slightly adjusted compared to the pretraining stage. Color augmentations remained similar, with small hue, saturation, and brightness shifts. Horizontal flips were kept, but vertical flips were disabled since the real photos are always taken right-side up. Small rotations, scaling, and no mosaic were carried over from the pretraining configuration. The model was trained for up to 200 epochs with a batch size of 16, image size of 640, and early stopping patience of 30 epochs.

Figure 21: Per-class results of the finetuned model on the validation set.

Class	Images	Instances	Box(P	R	mAP50	mAP50-95)	Mask(P	R	mAP50	mAP50-95):
all	72	304	0.995	0.977	0.995	0.992	0.995	0.977	0.995	0.98
Red	28	28	0.992	1	0.995	0.995	0.992	1	0.995	0.98
HotPink	28	28	1	0.942	0.995	0.995	1	0.942	0.995	0.972
DarkBlue	32	32	0.991	1	0.995	0.995	0.991	1	0.995	0.978
Orange	38	38	0.977	1	0.995	0.995	0.977	1	0.995	0.995
LightBlue	28	28	1	0.988	0.995	0.995	1	0.988	0.995	0.978
Green	30	30	0.992	0.967	0.994	0.994	0.992	0.967	0.994	0.994
Yellow	34	34	1	0.94	0.994	0.994	1	0.94	0.994	0.975
Purple	38	38	0.999	1	0.995	0.99	0.999	1	0.995	0.967
Board	48	48	1	0.957	0.993	0.977	1	0.957	0.993	0.984

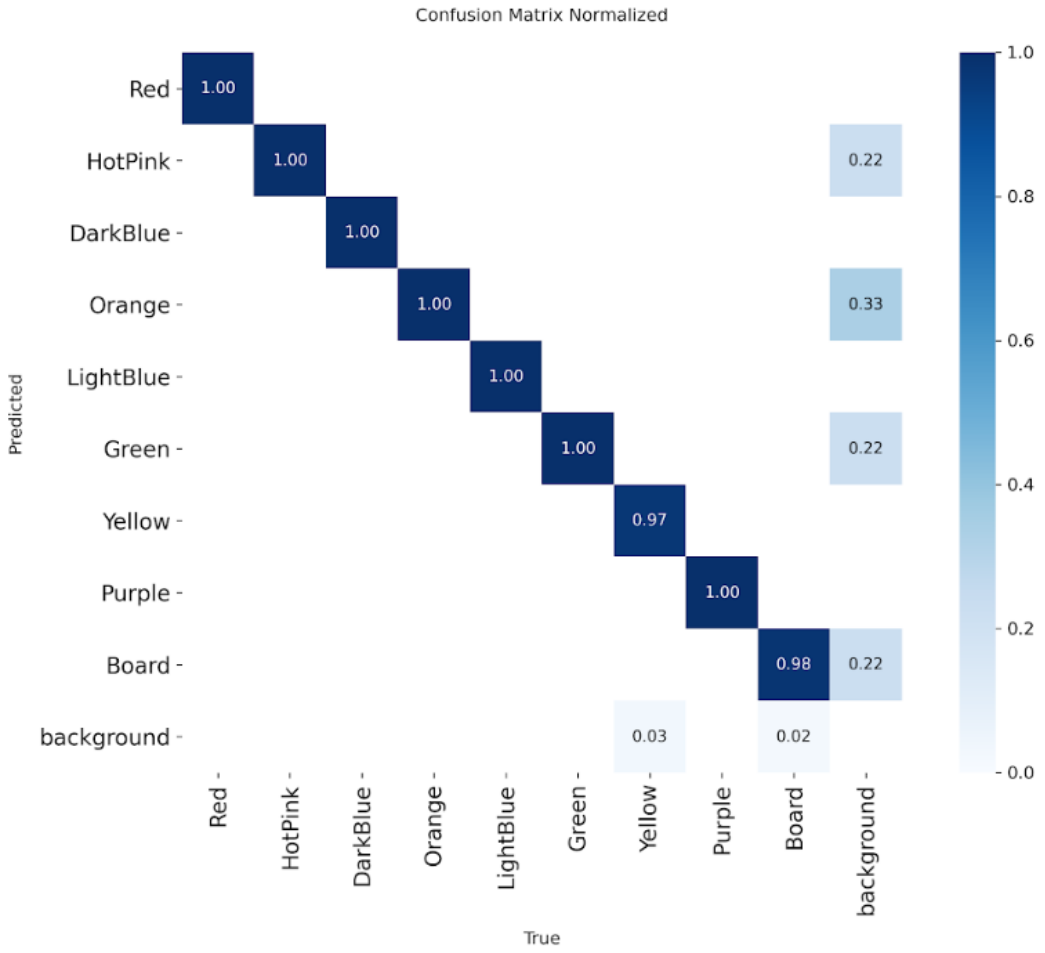
The training ran for 190 epochs before early stopping was triggered, with the best results recorded at epoch 160. The final model achieved a mask mAP50 of 0.995 and a mask mAP50-95 of 0.98 overall, with precision at 0.995 and recall at 0.977. Every individual piece scored above 0.94 in recall and above 0.97 in precision. The lowest recall was Yellow at 0.94 and HotPink at 0.942, while the highest were Red, DarkBlue, Orange, and Purple all reaching 1.00.

Figure 22: Training and validation metrics during finetuning on real data.



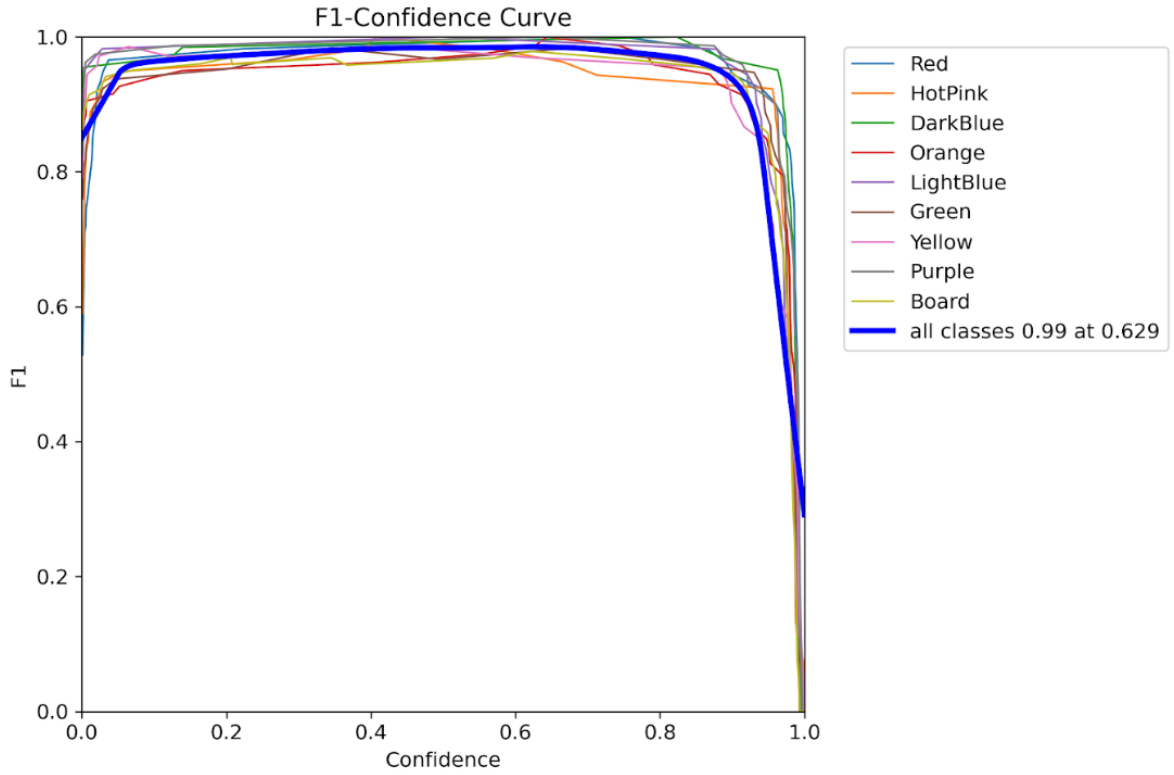
The results graph shows the full training progress of the finetuned model. On the training side (top row), all losses decrease smoothly over time, indicating the model is learning from the real images without issues. The segmentation loss and classification loss both drop steeply in the first 25 epochs and then continue to decrease gradually, showing that the model quickly adapted to real data thanks to the pretrained weights.
On the validation side (bottom row), the losses follow the same downward trend but with noticeably more fluctuation. This noise is caused by the small validation set of only 33 images, where a single misdetection between epochs causes a visible jump in the metrics. The precision and recall for masks stay consistently above 0.94, with the mAP50 reaching 0.99 and mAP50-95 settling around 0.95. Despite the spiky appearance, the overall trend is stable and upward, confirming that the model is genuinely improving and not overfitting.

Figure 23: Normalized confusion matrix of the finetuned model.



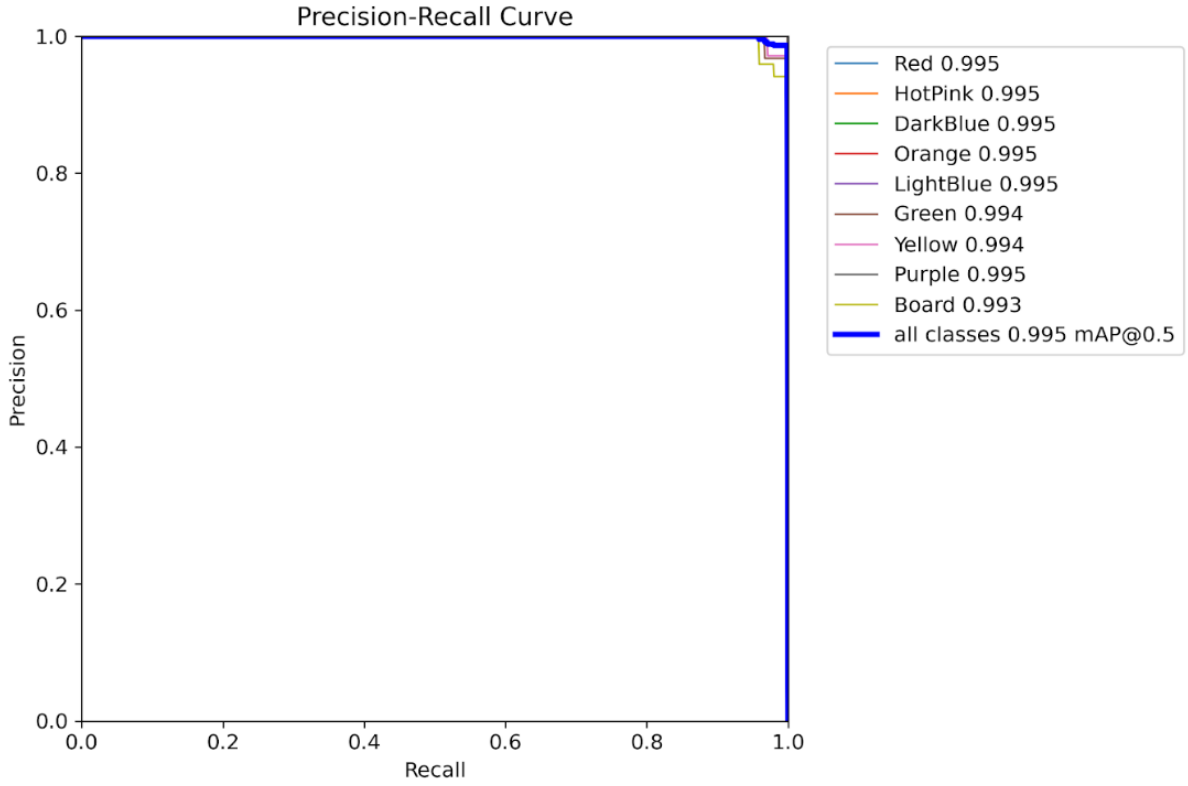
The confusion matrix shows that the finetuned model correctly classifies every piece with high accuracy. Red, HotPink, DarkBlue, Orange, LightBlue, Green, and Purple all score 1.00, meaning they are correctly identified 100% of the time. Yellow is the only piece slightly below at 0.97, and the Board scores 0.98. There is no confusion between any of the piece classes, which means the model never mistakes one piece for another. The only false positives appear in the background column, where the model occasionally detects a piece where there is none. Orange has the highest background false positive rate at 0.33, but these are filtered out by the confidence threshold in the application and do not affect the user experience.

Figure 24: F1-Confidence curve of the finetuned model per class.



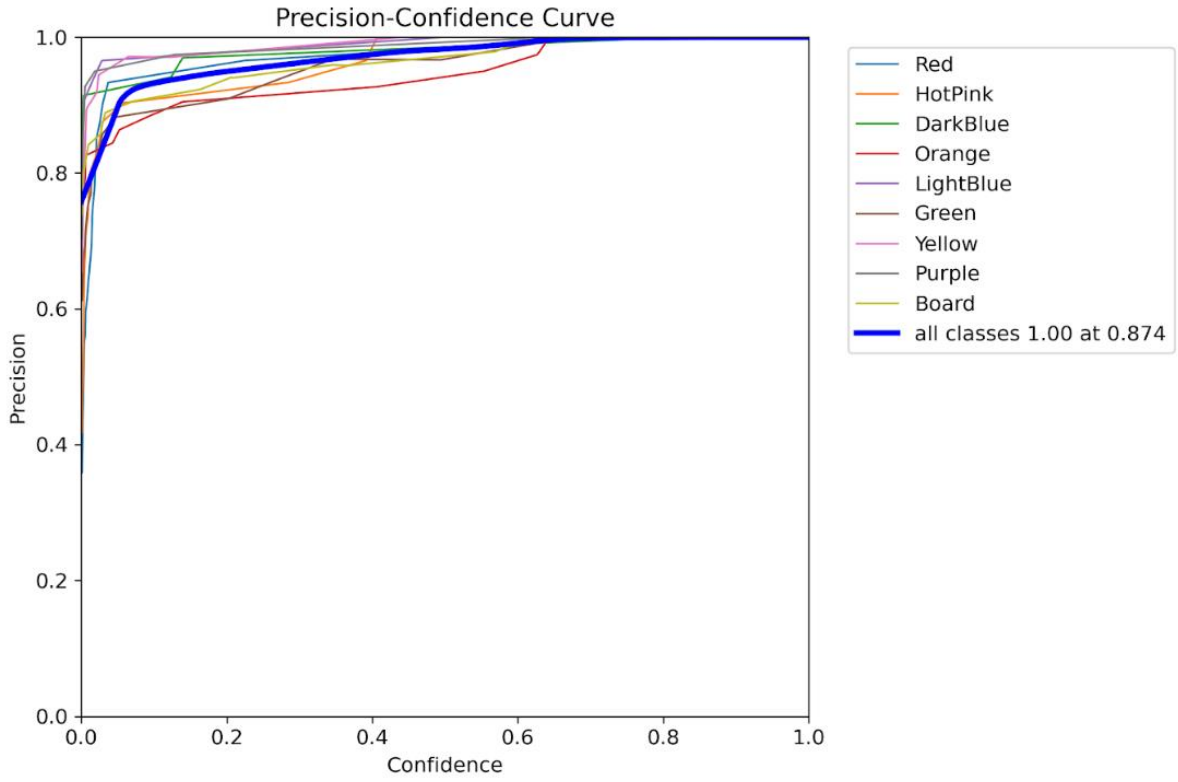
The F1 score combines precision and recall into a single metric, where a score of 1.0 means the model is both finding every piece and never making a wrong detection. The curve shows that across all classes, the F1 score reaches 0.99 at a confidence threshold of 0.629. All nine classes are tightly grouped near the top of the graph, meaning no single piece is significantly harder to detect than the others. Yellow and HotPink sit slightly lower than the rest, but still well above 0.90. The flat plateau between confidence 0.1 and 0.85 shows that the model performs consistently across a wide range of thresholds, which makes it robust in practice. In the application, a confidence threshold of 0.25 is used, which sits comfortably within this high-performance range.

Figure 25: Precision-Recall curve of the finetuned model per class.



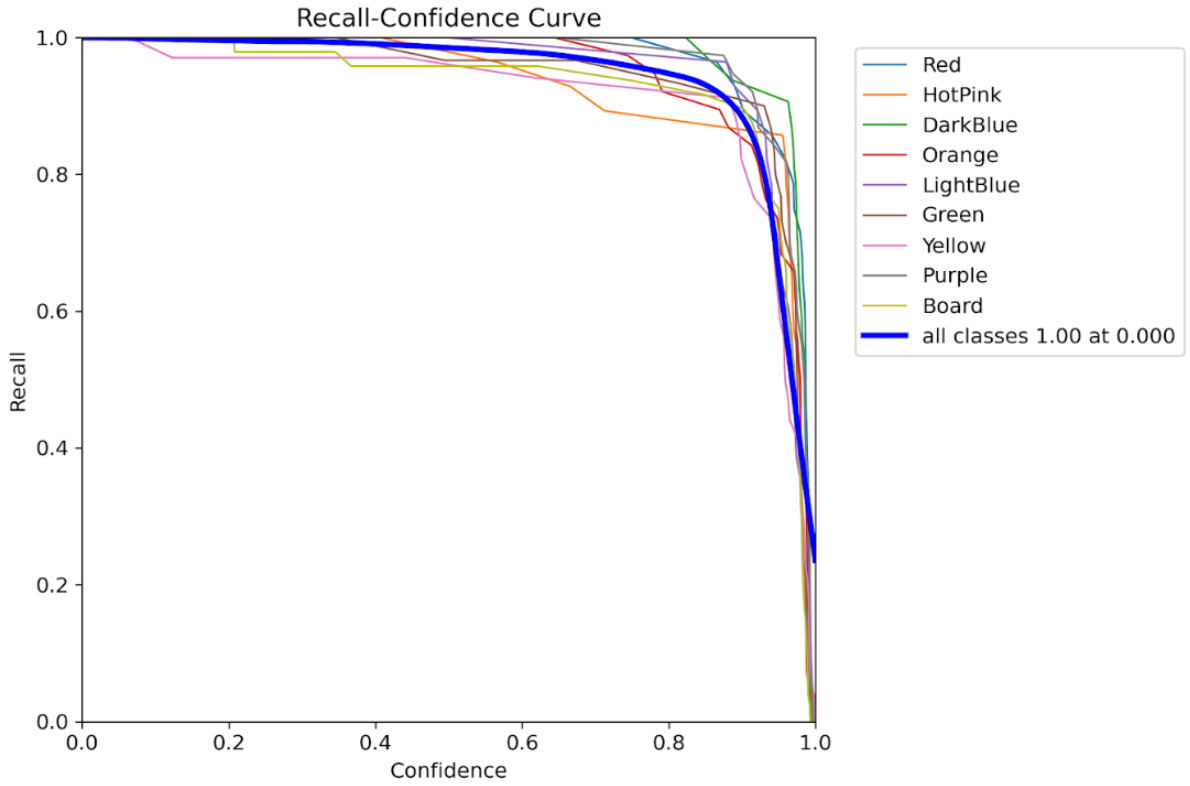
The precision-recall curve shows the relationship between how many detections are correct (precision) and how many actual pieces are found (recall). An ideal model sits in the top-right corner, meaning it finds everything and makes no mistakes. The finetuned model achieves exactly that, with all classes hugging the top-right corner at nearly perfect scores. The overall mAP@0.5 is 0.995 across all classes, with every individual piece scoring between 0.993 and 0.995. The Board sits slightly lower at 0.993 but is still nearly perfect. This curve confirms that the model can reliably detect and correctly classify all pieces in real photographs.

Figure 26: Precision-Confidence curve of the finetuned model per class.



The precision curve shows how accurate the model's detections are at different confidence thresholds. A precision of 1.00 means that every detection the model makes is correct. The curve reaches 1.00 across all classes at a confidence threshold of 0.874, meaning that above this threshold the model never produces a false detection. Even at lower confidence levels, precision stays above 0.80 for all classes. Orange and DarkBlue climb slightly slower than the other pieces, but still reach perfect precision before the 0.6 mark. This confirms that the model is highly reliable and rarely labels a piece incorrectly.

Figure 27: Recall-Confidence curve of the finetuned model per class.



The recall curve shows how many of the actual pieces the model manages to find at different confidence levels. At low confidence thresholds, recall reaches 1.00, meaning the model finds every piece in the image. As the confidence threshold increases, recall gradually decreases because the model becomes stricter about what it considers a valid detection. Most classes maintain a recall above 0.85 up to a confidence of 0.85. Yellow and HotPink drop off slightly earlier than the other classes, which aligns with the confusion matrix where Yellow scored 0.97 instead of 1.00. In the application, the confidence threshold is set to 0.25, where all classes have a recall close to 1.00, ensuring that no pieces are missed during scanning.

7.2. Method 2: Merged Dataset

The second training approach took a different direction. Instead of training in two stages, all the data was merged into a single dataset and the model was trained on everything at once. The merged training set consisted of 2250 images: the full 1800 synthetic images combined with the real images and their 90-degree rotated copies. The validation set contained 114 real images only, since the goal is to evaluate how well the model performs on actual photographs.

Unlike Method 1, no layers were frozen and no custom learning rate was set. The optimizer was automatically selected by ultralytics, which chose AdamW with a learning rate of 0.000769. The model started from the default COCO pretrained

weights rather than from a previously trained model. The augmentations were similar to Method 1, with horizontal and vertical flips, small rotations of up to 10 degrees, random scaling, and no mosaic. The color augmentations were slightly stronger, with higher saturation and brightness shifts, to encourage the model to generalize across the mix of synthetic and real images.

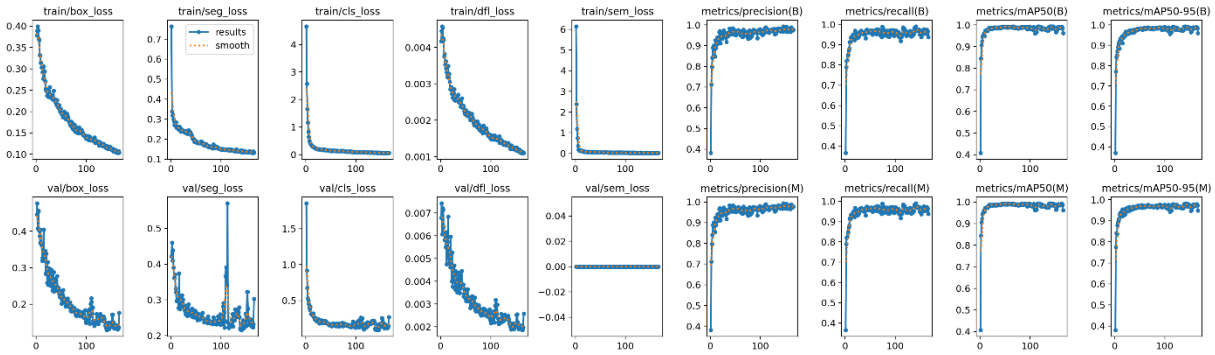
The training ran for 166 epochs before early stopping was triggered, with the best results recorded at epoch 141. The final model achieved a mask mAP50 of 0.994 and a mask mAP50-95 of 0.982 overall, with precision at 0.977 and recall at 0.987.

Figure 28: Per-class results of Method 2 on the validation set.

Class	Images	Instances	Box(P	R	mAP50	mAP50-95)	Mask(P	R	mAP50	mAP50-95):
all	114	462	0.977	0.987	0.994	0.993	0.977	0.987	0.994	0.982
Red	44	44	0.986	1	0.995	0.995	0.986	1	0.995	0.993
HotPink	44	44	0.977	0.971	0.991	0.991	0.977	0.971	0.991	0.969
DarkBlue	56	56	0.931	0.964	0.991	0.991	0.931	0.964	0.991	0.982
Orange	52	52	0.967	1	0.995	0.994	0.967	1	0.995	0.989
LightBlue	50	50	0.977	1	0.995	0.995	0.977	1	0.995	0.962
Green	42	42	1	0.997	0.995	0.995	1	0.997	0.995	0.992
Yellow	50	50	0.979	0.949	0.992	0.991	0.979	0.949	0.992	0.984
Purple	56	56	0.988	1	0.995	0.995	0.988	1	0.995	0.976
Board	68	68	0.988	1	0.995	0.994	0.988	1	0.995	0.992

The per-class results show that most pieces perform well, but with slightly more variation than Method 1. DarkBlue has the lowest precision at 0.931 and recall at 0.964, while Yellow has a recall of 0.949. Green and Red perform the strongest, both reaching 1.00 in recall or precision. The overall mask mAP50 is 0.994 and mAP50-95 is 0.982, which are high scores but slightly below Method 1's 0.995 and 0.98 respectively. The Board class performs well with a precision of 0.988 and recall of 1.00.

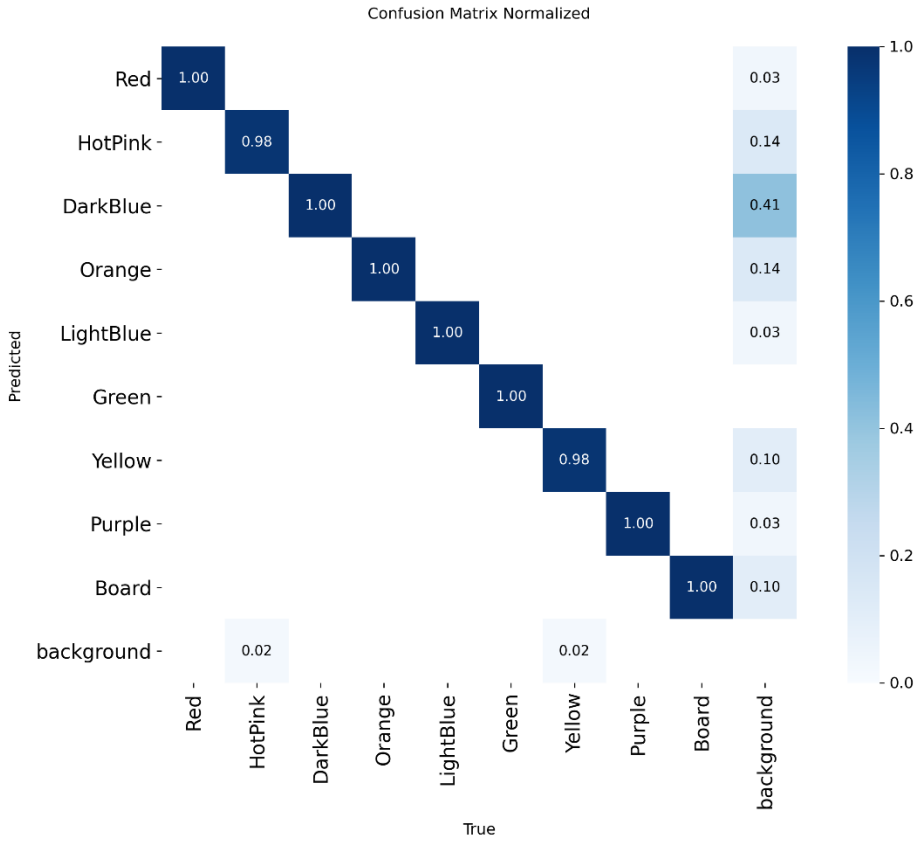
Figure 29: Training and validation metrics during Method 2 training on the merged dataset.



The training losses on the top row all decrease smoothly, showing the model is learning from the combined dataset without issues. The classification loss in particular drops sharply in the first 25 epochs, which is expected since the model is learning to distinguish nine classes from scratch on a larger dataset.
On the validation side, the losses follow the same downward trend but with some notable differences compared to Method 1. The segmentation loss shows a spike around epoch 50 before settling back down, and the classification loss takes longer

to stabilize. The precision and recall metrics start low, with some early drops as far as 0.4 and 0.5, before climbing to above 0.90. This slower start makes sense because the model is learning from a mixed dataset where synthetic and real images have different characteristics, and it takes more epochs for the model to find a balance between both. The mAP50 and mAP50-95 eventually settle around 0.99 and 0.95 respectively, but the path to get there is less smooth than Method 1's finetuning approach.

Figure 30: Normalized confusion matrix of Method 2 on the validation set.



The confusion matrix shows that the model correctly classifies all pieces, with no confusion between different piece classes. Most pieces score 1.00, with HotPink and Yellow slightly below at 0.98. However, the background false positive rates are noticeably higher than in Method 1. DarkBlue has the highest at 0.41, meaning the model frequently detects a DarkBlue piece where there is only background. HotPink, Orange, Yellow, and Board also show elevated false positive rates between 0.10 and 0.14. In Method 1, the highest background false positive was Orange at 0.33, while here DarkBlue reaches 0.41. These false positives would be filtered by the confidence threshold, but the higher rates suggest the merged model is less precise in distinguishing pieces from the background compared to the finetuned model.

7.3. Comparison

Both methods produced models with strong overall metrics, but when tested in practice the differences became clear. To compare them fairly, the same real photograph was fed into both models.

The finetuned model (Method 1) detected all eight pieces and the board correctly, with confidence scores ranging from 94.5% to 99.6%. Every detection was accurate and the board was found with 97.8% confidence. The merged model (Method 2) struggled with the same image: it detected the board twice, once at 45.3% and once at 25.6%, and DarkBlue was barely detected at 21.2% confidence. In the application, a duplicate board detection would cause problems for the corner detection and homography step, and a confidence that low risks being filtered out entirely.

The confusion matrices tell the same story. Method 1 has a maximum background false positive rate of 0.33 (Orange), while Method 2 reaches 0.41 (DarkBlue) with several other classes above 0.10. This means Method 2 more frequently detects pieces where there are none.

The likely explanation is that the finetuned model benefits from a two-stage learning process: it first built a solid understanding of all piece shapes from the large synthetic dataset, and then made small, targeted adjustments for real-world conditions. The merged model had to learn both at the same time, and the synthetic images may have dominated the training signal, making it harder for the model to fully adapt to real photographs.

For these reasons, Method 1 (pretrain and finetune) was chosen as the final model used in the application.

Figure 31: Method 1 (finetuned) detection on a real photograph. All pieces detected with high confidence.

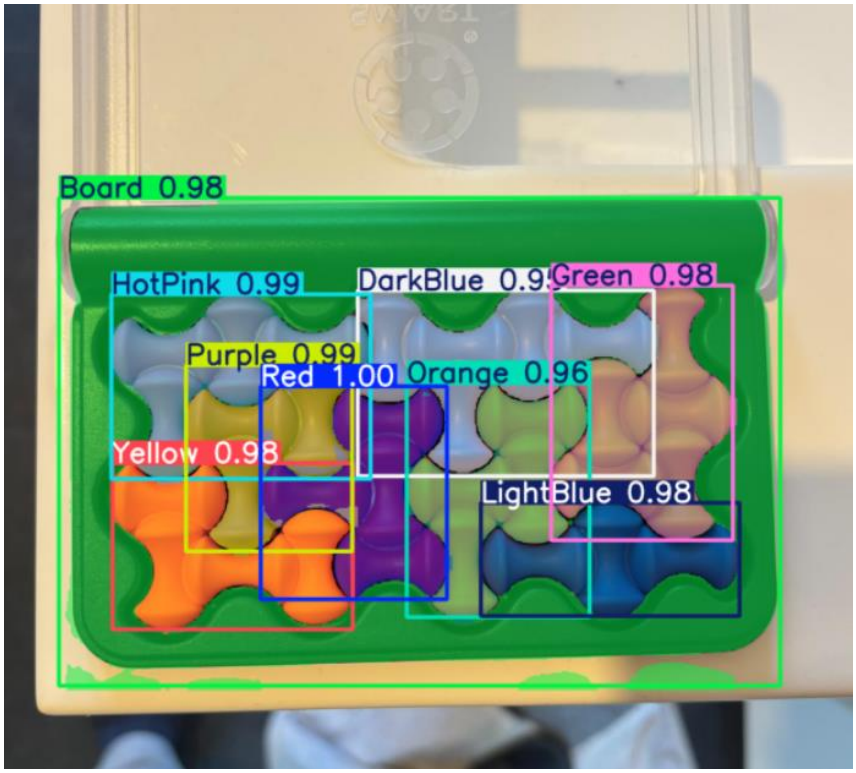
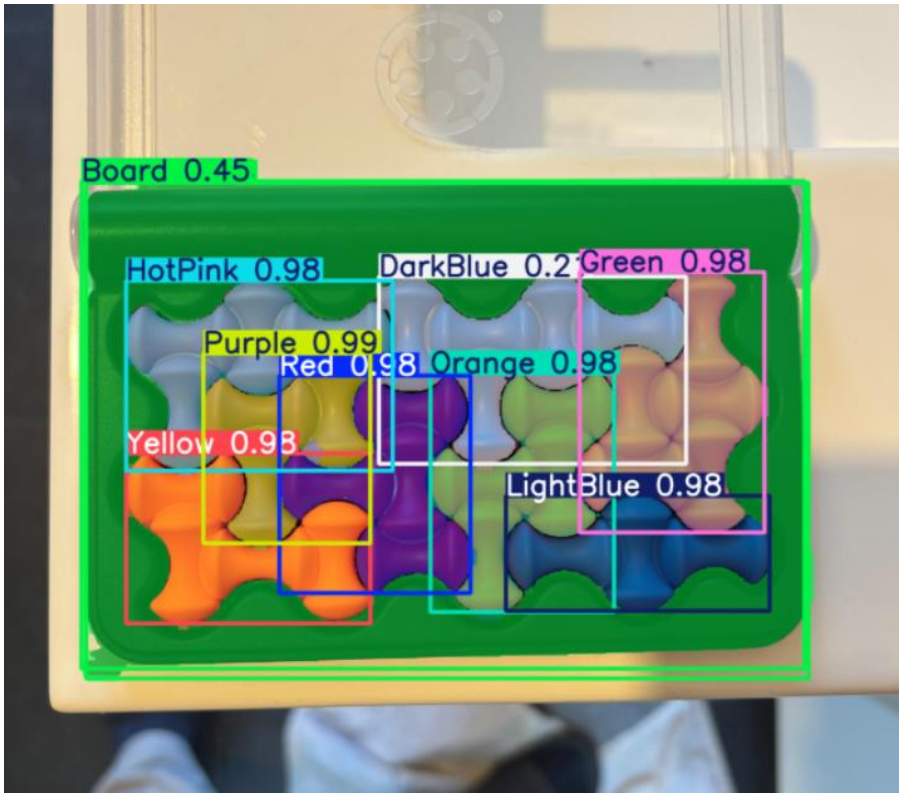


Figure 32: Method 2 (merged) detection on the same photograph. The board is detected twice and DarkBlue scores only 21.2%.



8. Additional Projects

Alongside the main IQ Waves project, two additional projects were worked on during the internship. These projects followed the same principles and techniques but applied them to different contexts.

8.1. IQ Stars

IQ Stars is another Smart NV puzzle where players place star-shaped pieces onto a board. While the concept is the same as IQ Waves, everything about the puzzle is different: the board, the grid, the piece shapes, and the cell layout. The same approach was followed to build the application: generate synthetic data in Blender, label real photos in Roboflow, train a YOLO segmentation model, and build a browser-based application with the same pipeline architecture.

8.1.1. Dataset

The synthetic dataset was generated in Blender using the same automated approach as IQ Waves. The Blender script rendered the puzzle with around 100 different background images, each time varying the position of the sun and the lighting intensity to simulate different real-world conditions. The piece placements were randomized, with pieces randomly added or removed from the board to create a variety of configurations. This produced a total of 1580 synthetic images, each with automatically generated segmentation annotations.

Figure 33: Examples of synthetic IQ Stars images generated in Blender with varying backgrounds and lighting.

For the real data, 200 images were taken with a camera and labeled in Roboflow using instance segmentation. Both pixel-level and polygon annotation tools were used to accurately capture the star shapes of the pieces. The same workflow as IQ Waves was followed: upload, annotate, and export in YOLO format.

Figure 34: Examples of real IQ Stars photos used for training.

8.1.2. Training Results

The same training approach as IQ Waves was used: pretrain on synthetic data, then finetune on real images. The synthetic dataset of 1580 images was split 80/20 into training and validation. For finetuning, 167 real images were used for training and 33 for validation, with the first 16 layers frozen to preserve the knowledge from the synthetic stage.
The finetuned model achieved a mask mAP50 of 0.98 and a mask mAP50-95 of 0.94 overall, with precision at 0.965 and recall at 0.97. Training stopped early at epoch 67 with the best results at epoch 47. All classes performed well, with Red and Yellow scoring the highest at mAP50-95 of 0.995 and 0.984 respectively. Purple had the

lowest precision at 0.875, and Blue had the lowest mAP50 at 0.935, but both are still reliable enough for the application to function correctly. The Board class was detected with perfect recall of 1.00, which is important since the entire pipeline depends on finding the board first.

Figure 35: Per-class results of the finetuned IQ Stars model on the validation set.

Class	Images	Instances	Box(P	R	mAP50	mAP50-95)	Mask(P	R	mAP50	mAP50-95):
all	33	138	0.965	0.97	0.98	0.979	0.965	0.97	0.98	0.94
Blue	15	16	0.975	0.938	0.935	0.935	0.975	0.938	0.935	0.908
Board	31	31	0.952	1	0.98	0.976	0.952	1	0.98	0.951
Green	19	19	1	0.963	0.995	0.99	1	0.963	0.995	0.942
Orange	13	13	1	0.947	0.995	0.995	1	0.947	0.995	0.925
Pink	17	17	0.986	0.941	0.99	0.99	0.986	0.941	0.99	0.931
Purple	15	15	0.875	1	0.954	0.954	0.875	1	0.954	0.881
Red	14	14	0.933	1	0.995	0.995	0.933	1	0.995	0.995
Yellow	13	13	1	0.972	0.995	0.995	1	0.972	0.995	0.984

Figure 36: IQ Stars model detection on a real photograph, showing all pieces and the board detected with high confidence scores.

8.1.3. Application

The IQ Stars application follows the same architecture as IQ Waves: a fully browser-based React app with ONNX Runtime running the model locally. The user opens the camera, lines up the board with a star-shaped silhouette guide, and takes a photo. The pipeline processes the image through the same steps: letterboxing, YOLO inference, mask decoding, board corner detection, perspective correction, and grid mapping.

The key difference is in the grid. Instead of a rectangular 4×8 grid with hourglass cells, IQ Stars uses a hexagonal layout with four rows of alternating lengths: 7, 6, 7, and 6, totaling 26 cells. The grid mapping uses circular regions around each star socket instead of the hourglass polygons used in IQ Waves. The overlap threshold is also lower at 40% compared to 55% for IQ Waves, since the circular regions leave more space between cells.

The solver was adapted for the hexagonal grid, where pieces can move in six directions instead of four, and each piece has up to six rotational orientations based on 60-degree rotations instead of the 90-degree rotations used in IQ Waves. The puzzle contains 7 pieces instead of 8.
The application was given a night sky theme with twinkling stars and a crescent moon, matching the IQ Stars brand identity. The buttons and layout follow the same structure as IQ Waves, with Scan, Hint, Solve, and Solvable as the main actions.

Figure 37: Grid mapping on IQ Stars, showing the circular cell regions and cell coordinates on a perspective-corrected board.

Figure 38: The IQ Stars application showing a fully solved board with all 7 pieces placed.

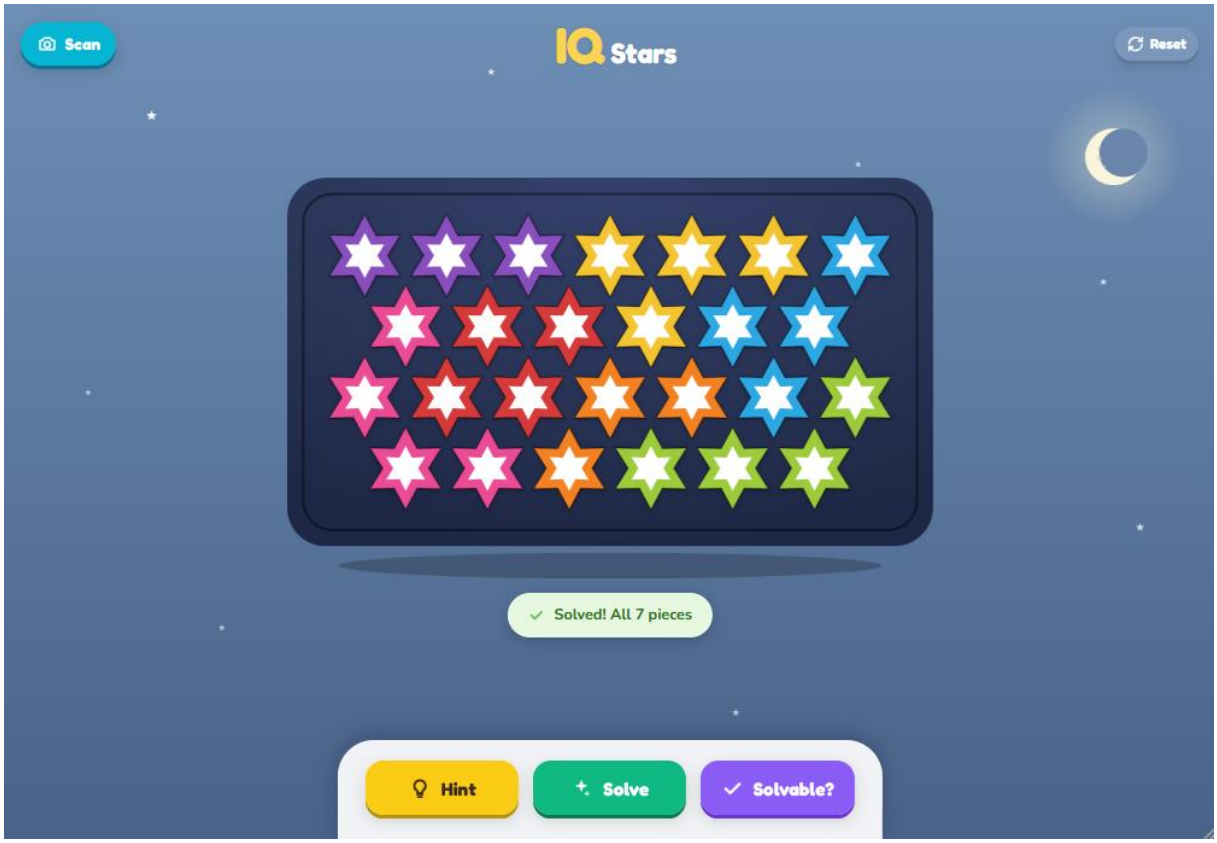


Figure 39: The IQ Stars camera modal with the star-shaped silhouette guide on a mobile device.

8.2. AR Game

During the internship, Smart NV was also developing an augmented reality game where physical objects interact with a cardboard scene. Unlike IQ Waves and IQ Stars, this was a team project involving multiple people. Each person had a specific responsibility: one handled the AR integration, another focused on training the model, and the main contribution on this side was labeling the dataset.

The model needed to detect two types of objects: the game scene itself and the individual colored objects placed around it. A dataset of around 3000 synthetic images and 500 real images was built for this purpose. The labeling was done in

Roboflow using instance segmentation, with a mix of pixel-level and polygon annotations depending on the object. The model was trained using YOLO with the same workflow, though the use case is fundamentally different since this project is about AR interaction rather than puzzle solving.

9. Conclusion

What started as a research question about whether computer vision could replace a printed solution booklet has resulted in a fully working, browser-based application that detects physical puzzle pieces from a camera photo and provides real-time solving assistance. The application runs entirely in the browser, with no backend, no installation, and no internet connection required after the initial load. A child can scan their puzzle, receive progressive hints with voice guidance, and solve the puzzle at their own pace.

The most important takeaway from this project is that the quality of the data drives everything. A well-trained model makes the rest of the pipeline straightforward. The synthetic data generated in Blender provided the volume the model needed to learn the general shapes, and the real data labeled in Roboflow gave it the real-world variation to perform reliably in practice. The two-stage training approach, pretraining on synthetic data and then finetuning on real images, proved to be the most effective method, outperforming the merged dataset approach in both precision and real-world testing.

The computer vision pipeline that was built, from image preprocessing through YOLO inference, mask decoding, perspective correction, and grid mapping, works reliably and runs fast enough in the browser thanks to ONNX Runtime and WebAssembly. The solver logic handles hint generation, solvability checking, and piece removal suggestions, giving the user a complete assistance experience. The same approach was successfully applied to both IQ Waves and IQ Stars, two different puzzles with completely different boards, grids, and piece shapes, confirming that the architecture is reusable across different Smart NV puzzles.

Looking ahead, one area that could improve the workflow significantly is automating the annotation of real images. Currently, labeling real photos in Roboflow is done manually, which is the most time-consuming part of building a new dataset. Finding a way to use the trained model itself to pre-annotate new images and only correct the mistakes manually would speed up the process of supporting additional puzzles in the future.

Overall, this project was a challenging but rewarding experience. Computer vision is a field where small details, such as lighting, camera angle, or the shape of a grid cell, can make or break the entire pipeline. Working through those challenges and arriving at a solution that works reliably in practice made this one of the most fulfilling projects to work on.

USE OF AI TOOLS

Throughout the project, AI tools were used as supporting tools to improve efficiency, clarity, and quality, while all analysis, decisions, and final validation remained my own responsibility.

Brainstorming and research. Claude (claude.ai) was used to support brainstorming and research, exploring possible approaches, comparing options, and clarifying unfamiliar concepts. The ideas it produced were treated as starting points rather than final answers: each was evaluated against the actual project requirements, scope, and constraints before being acted upon.

Development. Claude Code was used during development as a coding assistant, helping with code snippets, repetitive logic, and implementation suggestions. None of the generated code was accepted blindly. I guided the tool, reviewed every output, debugged it where it went wrong, and adjusted or rewrote it where needed. All AI-assisted code was tested and verified before being included in the project, and the final responsibility for its correctness, suitability, and maintainability remained with me.

Documentation. AI was also used to support the writing of this documentation, correcting grammar and spelling, improving sentence flow, and making explanations clearer. The content itself, including the technical decisions and results described, reflects my own work; AI was used only to refine how it was presented.

Reflection. This experience also showed me the limits of relying on AI. While AI is useful for getting started and moving faster, I found that real experience, and the guidance of experienced developers, often led to better decisions than AI brainstorming alone. AI tools should therefore be used as assistance, not as a replacement for understanding, judgement, and validation, and they should not be relied on for everything.

Reference list

- Ultralytics. YOLO26 Documentation. https://docs.ultralytics.com/models/yolo26
- Roboflow. (2025). Roboflow Documentation. https://docs.roboflow.com
- ONNX Runtime. (2025). What is ONNX Runtime (ORT)? [Video]. YouTube. https://www.youtube.com/watch?v=M4o4YRVba4o