



SAPIENZA
UNIVERSITÀ DI ROMA

Automated Floorplan Extraction and Refinement for Accessible Indoor Navigation

Department of Computer Science
Faculty of Information Engineering, Informatics, and Statistics
Master's Degree in Computer Science

Federico Raponi

ID number 1963339

Advisor

Prof. Emanuele Panizzi

A handwritten signature in black ink, appearing to read 'Emanuele Panizzi'.

Academic Year 2025/2026

Automated Floorplan Extraction and Refinement for Accessible Indoor Navigation

Master's Thesis. Sapienza University of Rome

© 2026 Federico Raponi. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: raponi.1963339@studenti.uniroma1.it

Abstract

Context The Navindoor project at Sapienza University of Rome is building an indoor navigation system for people with visual and motor disabilities. The system requires accurate digital maps of university buildings, and the source material available for them consists of printed emergency evacuation floor plans posted on corridor walls. Photographed with a smartphone, these plans are noisy, perspective-distorted, and cluttered with annotation text and symbols. Digitizing them by hand is slow enough that a single building can take days.

Objectives This thesis develops an automated pipeline that extracts structural elements from smartphone photographs of floor plans, converts them into clean polygonal annotations, and hands them to a web-based editor where a human operator reviews and enriches the result. The aim is to replace full manual tracing with a correction workflow. Verification by a human is not an optional refinement here: a user who follows spoken directions because they cannot see the building has no way to notice that the router sent them through a door that does not exist.

Methods I trained and compared two object detection architectures, YOLOv11-m (a convolutional network) and RT-DETR-L (a real-time detection transformer), on a dataset derived from CubiCasa5K. Building that dataset required a custom SVG parser that recovers absolute pixel coordinates through full transformation matrix accumulation, a mapping from CubiCasa5K’s eighty-odd vector classes down to the five Navindoor needs, and a dual-domain rasterization scheme that pairs each annotated plan with a clean and a scan-like rendering, doubling the effective dataset from roughly 5000 to roughly 10000 images at no annotation cost. Both models were trained at 1024×1024 resolution on a single RTX 3090 Ti under matched conditions, with augmentation adapted to the domain (no rotation, shear, or perspective, to preserve the orthogonal geometry of the annotations), and threefold oversampling of the rare classes. The detection output then passes through a geometric pipeline that merges overlapping boxes via unary union, segments rooms using a distance-transform-guided flood fill, simplifies polygons with class-adaptive Douglas-Peucker tolerances, and orthogonalizes edges to enforce architectural constraints. A decoupled OCR module built on PaddleOCR’s PP-OCRv4 Latin-script model extracts room codes from masked polygon crops. The complete system is deployed as a FastAPI backend with asynchronous GPU inference and a SvelteKit 5 editor offering SVG-based polygon manipulation, geometric snapping and semantic annotation tools.

Results On the CubiCasa5K validation split, RT-DETR-L reaches a mAP@50 of 0.894 against 0.870 for YOLOv11-m, with the margin nearly doubling on mAP@50-95, where it stands at 0.045 (0.764 against 0.719). The lead at IoU 0.5 comes almost entirely from door stoppers, an auxiliary class excluded from the exported annotations, where the transformer scores 0.948 against 0.735. On the classes that reach the output the two models are close, and YOLOv11-m stays ahead on staircase symbols (0.821 against 0.740). What separates the transformer is the set of properties

that matter once detections become polygons: it localizes more precisely, misses fewer objects outright (undetected walls fall from 14% to 9%, missed doors from 23% to 18%), its validation regression losses stay flat from epoch 50 to the end of training where YOLOv11-m's begin drifting upward after epoch 35, and its F1 remains near peak across confidence values from roughly 0.3 to 0.8 against a sharp maximum at 0.272 for the baseline. Testing on smartphone photographs of Sapienza floor plans, where no pixel-level ground truth exists, showed the same ranking informally: RT-DETR-L produced far fewer false walls from text blocks and evacuation arrows, and placed door boxes closer to the wall opening they belong to.

Conclusions A transformer-based detector, a geometry-aware post-processing chain, and an interactive human-in-the-loop editor together form a usable system for extracting floor plans from smartphone photographs. Global self-attention offers a measurable advantage on this spatially structured input, though not a uniform one, and the modular architecture lets the model, the post-processing, and the OCR engine be replaced independently as better options appear. Two limits define what the work does not yet establish. The controlled comparison runs on rendered residential plans rather than on the photographs the problem is defined around, and the reduction in operator effort, while apparent during development, was never formally measured. The thesis closes by identifying concrete directions for future research, self-supervised pretraining on unlabeled floor plans foremost among them, which could cut the labeled data required to deploy the system to new building types.

Contents

1	Introduction	1
1.1	Background: Indoor Navigation Systems and Digital Maps	1
1.2	The Challenge of Floor Plan Extraction	2
1.3	Proposed : a three stage deep learning pipeline	4
1.3.1	Model Selection: From CNN to Transformer	5
1.3.2	The Inference Pipeline	5
1.3.3	Human-in-the-Loop Refinement	6
1.4	Contributions	7
1.5	Thesis Outline	8
2	State of the Art and Related Work	9
2.1	Early Approaches: Established Rules and Heuristics	9
2.2	The Deep Learning Era	10
2.3	Object Detection Architectures: CNN vs. Transformer	11
2.4	Challenges of Emergency Floor Plans	12
2.5	Indoor Navigation and Map Representation	13
2.6	The Gap: raw input and a practical implementation	13
3	Deep Learning Methodology	14
3.1	Dataset Engineering	14
3.1.1	The CubiCasa5K Foundation	14
3.1.2	SVG Parsing and Coordinate Recovery	14
3.1.3	Class Mapping	15
3.1.4	Dual-Domain Rasterization	16
3.1.5	Format Conversion: COCO to YOLO Segmentation	17
3.1.6	Image Denoising Exploration	17
3.2	Model Architectures	18
3.2.1	YOLOv11-m	18
3.2.2	RT-DETR-L	19
3.3	Training Setup	20
3.3.1	Domain-Specific Data Augmentation	21
3.3.2	Handling Class Imbalance	22
3.4	Evaluation Metrics	22
4	Experimental Results	24

4.1	Experimental Setup	24
4.2	Global Training Trajectories	25
4.3	Mean Average Precision	26
4.4	Validation Loss and Training Stability	27
4.5	Per-Class Analysis	28
4.5.1	Precision-Recall and F1 Curves	29
4.5.2	Confusion Matrices	32
4.6	Qualitative Observations	33
5	System Architecture and Human-in-the-Loop Refinement	34
5.1	Backend Architecture	34
5.1.1	Application Lifecycle	34
5.1.2	Inference Pipeline	35
5.1.3	OCR Service	39
5.2	Frontend Architecture: The Interactive Editor	40
5.2.1	Core State Management	40
5.2.2	Rendering	41
5.2.3	Interaction Managers	41
5.2.4	Geometric Operations	42
5.2.5	Semantic Editing	42
5.2.6	Door-Room Linking	43
5.2.7	Undo/Redo	43
5.2.8	Import and Export	44
5.2.9	Authentication and Routing	44
5.3	Deployment	45
6	Conclusions and Future Work	46
6.1	Summary of Contributions	46
6.2	Limitations	48
6.3	Future Work	50
6.3.1	Self-Supervised Pretraining for Floor Plans	50
6.3.2	Geometric Loss Functions	50
6.3.3	Vectorization as an Alternative to Detection	51
6.3.4	End-to-End Routing Integration	51
	Bibliography	52

List of Figures

1.1	A typical emergency evacuation floor plan from a Sapienza building, photographed with a smartphone. Note the perspective distortion, uneven lighting, dimension lines, and text overlays that complicate automated extraction.	2
1.2	Visual comparison of input data quality. Left: A clean, high-quality floor plan image typical of the CubiCasa5K dataset. Right: A smartphone photograph of a printed emergency plan from the Navindoor dataset, exhibiting typical real-world artifacts like uneven lighting and perspective distortion.	3
1.3	Raw output of the detection model of a floor plan, before geometric post-processing and human review in the frontend editor.	4
1.4	Output of the backend inference pipeline overlaid on the original floor plan image. Extracted architectural elements are converted into structured polygons, showing merged wall structures in red, flood-filled navigable rooms and corridors in green, and door openings in blue and yellow.	6
1.5	The frontend editor, designed to reduce floor plan corrections to a single direct gesture per operation.	7
3.1	Left: vectorized rendering (<code>F1_vectorized.png</code>). Right: scanned-style rendering (<code>F1_scaled.png</code>). Both images share the same COCO annotations extracted from the SVG.	17
3.2	Architecture of the YOLO11 model, detailing the Backbone, Neck, and Head components along with C3k2, SPPF, and C2PSA blocks.	19
4.1	Validation recall (left) and precision (right) over the training run: 115 epochs for YOLOv11-m, 133 for RT-DETR-L.	25
4.2	mAP@50 (left) and mAP@50-95 (right) validation curves over the training run.	26
4.3	Training and validation metrics over 115 epochs for YOLOv11-m. Note the upward trend in validation box losses (<code>val/box_loss</code> and <code>val/df1_loss</code>) after epoch 35, indicating overfitting on the localization task.	27
4.4	Training and validation metrics over 133 epochs for RT-DETR-L. The validation losses maintain a stable plateau after epoch 50, demonstrating better generalization for bounding box regression.	28

4.5	Per-class precision-recall curves at $\text{IoU} = 0.5$ for YOLOv11-m. . . .	29
4.6	Per-class precision-recall curves at $\text{IoU} = 0.5$ for RT-DETR-L. . . .	30
4.7	Per-class F1-confidence curves for YOLOv11-m.	30
4.8	Per-class F1-confidence curves for RT-DETR-L.	31
4.9	Normalized confusion matrix on the validation split for YOLOv11-m. Columns represent ground truth; rows represent predictions.	32
4.10	Normalized confusion matrix on the validation split for RT-DETR-L. Columns represent ground truth; rows represent predictions.	32
5.1	High-level system architecture. The user uploads a floor plan photograph to the FastAPI backend, where it passes through preprocessing (CLAHE), RT-DETR-L inference, polygon post-processing, and COCO serialization. A decoupled OCR service extracts room codes from polygon crops. The SvelteKit frontend loads the COCO output, provides an interactive editor for review and correction, and exports enriched annotations to the Navindoor routing engine. All endpoints are protected by Keycloak OIDC authentication (dashed arrows). . .	35
5.2	CLAHE contrast normalization. Left: original BGR image with uneven corridor lighting. Right: after CLAHE on the L channel, wall edges are sharper and the dynamic range is compressed.	36
5.3	Effect of unary union merging on wall detections. Left: raw bounding boxes, many overlapping. Right: after merging, walls are contiguous polygons.	37
5.4	Flood-fill room segmentation pipeline. (a) The detector's bounding box provides a spatial prior. (b) Distance transform identifies the pixel furthest from walls as the seed point. (c) Flood fill produces a polygon that conforms to wall boundaries.	38
5.5	Orthogonalization of a polygon corner. (a) The raw edge angle recovered after simplification, here 84° , differs from the nearest cardinal direction by less than the 10° threshold. (b) The edge lines are rotated to exactly 0° and 90° and the vertex is recomputed as their intersection, producing an architecturally plausible right angle. . . .	39
5.6	Door-room linking visualization. Dashed curves connect doors to their target room centroids. The properties panel (right) shows editable fields for the selected element.	43
5.7	The main editor interface. Left panel: category legend with visibility toggles. Center: SVG canvas with the floor plan image and polygon overlay. Right panel: properties for the selected element.	44

List of Tables

3.1	Mapping from CubiCasa5K SVG classes to target COCO categories.	16
3.2	Training hyperparameters for YOLOv11-m and RT-DETR-L. Values that differ between models are shown in bold.	21
4.1	Per-class AP@50 on the validation split for the selected checkpoint. Difference is RT-DETR minus YOLO. The door stopper class (COCO ID 4) is excluded from the final pipeline output but is retained during training because forcing the model to distinguish the threshold line from the door arc and from nearby walls reduces door-wall confusion and improves the overall quality of door detections.	28

Chapter 1

Introduction

1.1 Background: Indoor Navigation Systems and Digital Maps

Outdoor localization is largely a solved problem: GPS provides a reliable position estimate across most outdoor environments and keeps correcting it as the user moves. Indoors, no equivalent signal exists, and that absence changes what the map itself has to do. Outdoors, a coarse or even outdated map is often good enough, since the constant position fix keeps the system on track regardless. Indoors, without that fix, the map has to carry far more of the burden: the finer and more accurate it is, the better a system can infer where the user stands and which route is actually viable, relying on room boundaries, door positions, and the connections between spaces as its only real source of information.

This burden is compounded by a separate problem: availability. Even where digital floor plans exist, they tend to sit in formats a navigation system cannot directly consume: scanned blueprints, CAD files with proprietary encoding, or occasionally paper documents pinned to a corridor wall. The *Navindoor* project at *La Sapienza University in Rome* addresses this gap: it builds an indoor navigation system aimed primarily at people with visual or mobility impairments. This requires detailed, machine-readable maps of university buildings: room coordinates, door positions and widths, staircase types, and the semantic labels a routing engine needs to judge whether a given path is wheelchair-accessible.

Getting those maps into the system is the actual bottleneck. Sapienza's buildings already have emergency evacuation floor plans posted throughout their corridors, and this is not a coincidence specific to the university: Italian workplace safety law requires any code-compliant building to post this kind of plan, floor by floor. The same source data is therefore available, in principle, in every building meeting the same regulatory requirements, not only in the ones this thesis was developed on. These sheets show walls, rooms, doors, staircases, and evacuation routes marked with arrows and text, and in practice they are the most accurate and up-to-date spatial representation of each building available. A smartphone photograph of one

of these sheets takes little effort to obtain, but converting that photograph into a structured set of polygons with room codes, door types, and accessibility flags is what this thesis addresses. Figure 1.1 shows a typical example.

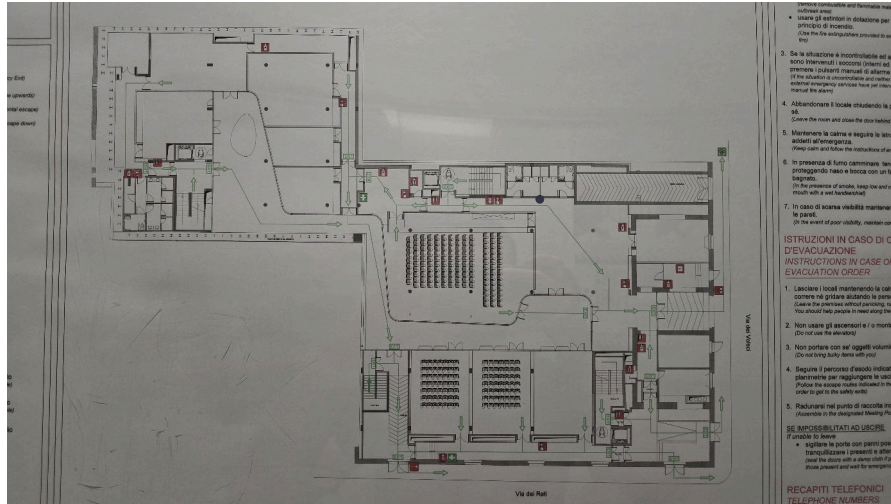


Figure 1.1 A typical emergency evacuation floor plan from a Sapienza building, photographed with a smartphone. Note the perspective distortion, uneven lighting, dimension lines, and text overlays that complicate automated extraction.

1.2 The Challenge of Floor Plan Extraction

A photograph of a floor plan is a hostile input for a computer vision pipeline. The paper is rarely flat against the wall, so perspective distortion stretches rectangles into arbitrary quadrilaterals. Indoor lighting, usually fluorescent tubes in a corridor, creates gradients across the image that confuse edge detectors. The plans themselves are dense with annotations that a human reader parses without effort and that an extraction system reads as pure noise: evacuation arrows crossing through rooms, dimension lines running parallel to walls, text labels of varying sizes and orientations, icons for fire extinguishers and emergency phones. Age adds its own damage: plans that have hung on a wall for years are often creased or faded, sometimes torn.

The traditional approach to floor plan digitization is manual, with a person tracing walls and rooms in a CAD tool or a GIS editor, a process slow enough that a single building with five floors and forty rooms per floor might take days⁽²⁴⁾, with results that depend heavily on the operator’s precision and attention. Automated methods have been explored since at least the early 2000s: Dosch et al.⁽⁵⁾ built a complete system for architectural drawing analysis based on hard-coded rules about line thickness and junction patterns, while Macé et al.⁽¹⁸⁾ used a Hough transform pipeline combined with graph-based spatial reasoning to detect rooms. Such systems perform well on clean, vector-quality drawings scanned under controlled conditions, a very different setting from the kind of photograph that comes out of a smartphone in a corridor.

The machine learning era brought end-to-end trainable systems that learn to ex-

tract structure from pixels rather than from hand-written rules. The CubiCasa5K dataset⁽¹³⁾, released in 2019, provided 5000 annotated floor plans with over 80 object categories represented as *SVG*¹ vector data, and it enabled the first generation of deep floor plan models; parts of it form the training foundation for this work as well. Its scope is residential, however, so its distribution of room types, wall thicknesses, and layout topologies reflects Finnish homes rather than Italian university buildings. A model trained exclusively on it will have never encountered a lecture hall or a laboratory bench layout, let alone a double-loaded corridor with offices lining both sides. Chapter 3 details how I adapted the dataset to the target domain.

This quality gap becomes even sharper when comparing image sources directly: CubiCasa5K images are clean renders or high-quality scans, while Navindoor’s inputs are smartphone photographs of printed emergency plans, visible side by side in Figure 1.2.

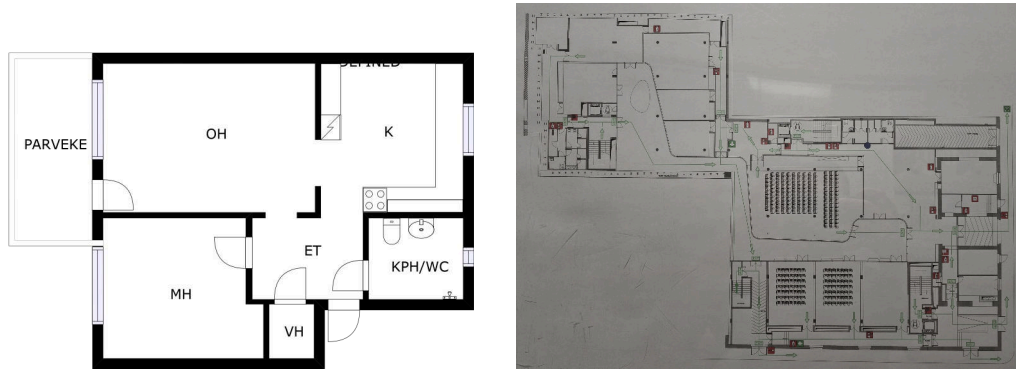


Figure 1.2 Visual comparison of input data quality. Left: A clean, high-quality floor plan image typical of the CubiCasa5K dataset. Right: A smartphone photograph of a printed emergency plan from the Navindoor dataset, exhibiting typical real-world artifacts like uneven lighting and perspective distortion.

¹SVG stands for Scalable Vector Graphics, a popular web-friendly image file format used to display two-dimensional graphics

1.3 Proposed : a three stage deep learning pipeline

The system I built has three main components: a deep learning model that detects structural elements (walls, rooms, doors, stairs, plus an auxiliary door-stopper class) from a photograph of a floor plan; a backend server that runs the model, applies geometric post-processing to the raw detections, and optionally performs optical character recognition to extract room codes; and a frontend web editor that lets a human operator review, correct, and enrich the automatic output before it enters the navigation database. Figure 1.3 shows what the model produces before any of that post-processing takes place.



Figure 1.3 Raw output of the detection model of a floor plan, before geometric post-processing and human review in the frontend editor.

The output of this pipeline is not simply a digital copy of the floor plan for someone to look at. It becomes the geometry the routing engine reasons over. A misplaced door changes which rooms the system treats as connected; get a wall's position wrong by a few centimeters and a corridor can flip from accessible to too narrow for a wheelchair; miss a staircase entirely, and the router will assume a connection between floors that does not exist. A sighted user can often catch a wrong turn and correct course on the spot. Someone following spoken turn-by-turn directions because they cannot see the building has no equivalent way to catch the error. That asymmetry, not just a general concern for accuracy, is why the editor is not an

optional polishing step: a human has to verify the map before it reaches someone who depends on it, rather than merely consults it, to get through a building.

1.3.1 Model Selection: From CNN to Transformer

I started with YOLOv11-m⁽¹⁶⁾, a convolutional neural network in the YOLO family, pre-trained on COCO and fine-tuned on a floor plan dataset derived from CubiCasa5K. Detection performance was reasonable, yet the model showed systematic failures on the kinds of clutter that define the problem: evacuation arrows got misclassified as thin walls, large rooms were split along dimension lines, and staircase symbols, visually distinct from the rectangular rooms the model was trained on, were frequently missed altogether. The core issue, I came to believe, was the limited receptive field of a convolutional backbone. To be precise, YOLOv11 is not purely convolutional: its C2PSA blocks apply a partial form of spatial self-attention on a subset of channels within the backbone. But that attention is local to a downsampled feature map, whereas floor plans are relational by nature across the full image: a wall counts as a wall because it separates two rooms, independent of local texture, and a door reads simply as an interruption in a wall segment. A transformer-based detector, whose object queries are matched to objects as a set and refined against multi-scale image features, should in principle capture these long-range dependencies more completely. The question is one of extent, not of presence: how much global context is enough to resolve the ambiguities that plague this domain.

I switched to RT-DETR-L⁽²⁸⁾, a real-time detection transformer developed by Baidu and built on a hybrid encoder (CNN backbone with a transformer neck) and a transformer decoder with denoising queries. The change in behavior was immediate and qualitative: text was no longer confused with walls, and rooms bisected by annotation lines stayed whole instead of being split apart. Staircases were the exception: they remained the hardest class for both models, and the benchmark comparison in Chapter 4 shows YOLOv11-m still ahead there.

The quantitative comparison in Chapter 4 evaluates both models on the CubiCasa5K validation split. Although this is not the target domain (smartphone photographs of emergency plans), it offers a controlled setting where pixel-perfect ground truth is available and every metric is reproducible. The qualitative observations on real Sapienza photographs in Section 4.6 then suggest that the relative ranking of the two architectures transfers to the target domain, even though per-pixel ground truth for those images does not exist.

1.3.2 The Inference Pipeline

Raw bounding boxes from a detector do not amount to a usable floor plan: walls come out as overlapping rectangular blocks, rooms are loose boxes that spill into neighboring halls, and a door’s orientation relative to the wall it belongs to is often ambiguous given only its bounding rectangle. The backend pipeline addresses these limitations through an initial preprocessing step followed by geometric post-processing.

Preprocessing normalizes the input image, correcting for resolution and applying

adaptive histogram equalization (CLAHE)⁽³⁰⁾ to handle the uneven lighting typical of photographs taken in indoor corridors. This ensures the detector receives input features that match its training distribution.

What comes out of the detector must then be converted into clean geometry for a routing engine. Overlapping boxes belonging to the same wall are merged into contiguous polygons, and noisy boundaries are simplified using polygon reduction algorithms such as Douglas-Peucker⁽⁶⁾ alongside orthogonal snapping. Rooms require a dedicated correction. Bounding boxes are rectangular by construction, but rooms in institutional buildings often have irregular layouts, and models trained on residential datasets inherit this rectangular prior. A distance-transform-seeded flood-fill step lets each room expand into the actual physical boundaries of the space. Finally, the structured elements are formatted into standardized polygon annotations⁽¹⁷⁾ ready to populate the navigation database. Figure 1.4 overlays the result on the original photograph.



Figure 1.4 Output of the backend inference pipeline overlaid on the original floor plan image. Extracted architectural elements are converted into structured polygons, showing merged wall structures in red, flood-filled navigable rooms and corridors in green, and door openings in blue and yellow.

1.3.3 Human-in-the-Loop Refinement

No automated pipeline produces perfect output, and on noisy smartphone photographs the automatic result is rarely accurate enough to use without review. Given how much a navigation error can cost someone who cannot see the building, this human-in-the-loop step carries more weight here than it would in a purely cosmetic mapping tool. General-purpose vector editors such as Adobe Illustrator or Inkscape can technically perform the same operations, but they expose the full complexity of a vector graphics tool: path nodes, bezier handles, layer management, none of which an operator correcting a floor plan actually needs. The frontend renders the detected polygons as an interactive overlay on the original floor plan image and

collapses each editing task down to the single gesture it requires: drag a vertex to reshape a wall, click two elements to connect a door to its room, type into a field to assign a room code. Operations that would take several steps and some familiarity with vector graphics conventions in a general-purpose tool happen in one, so an operator with no CAD or design background can correct a floor plan without first having to learn a drawing tool. Figure 1.5 shows the editor.

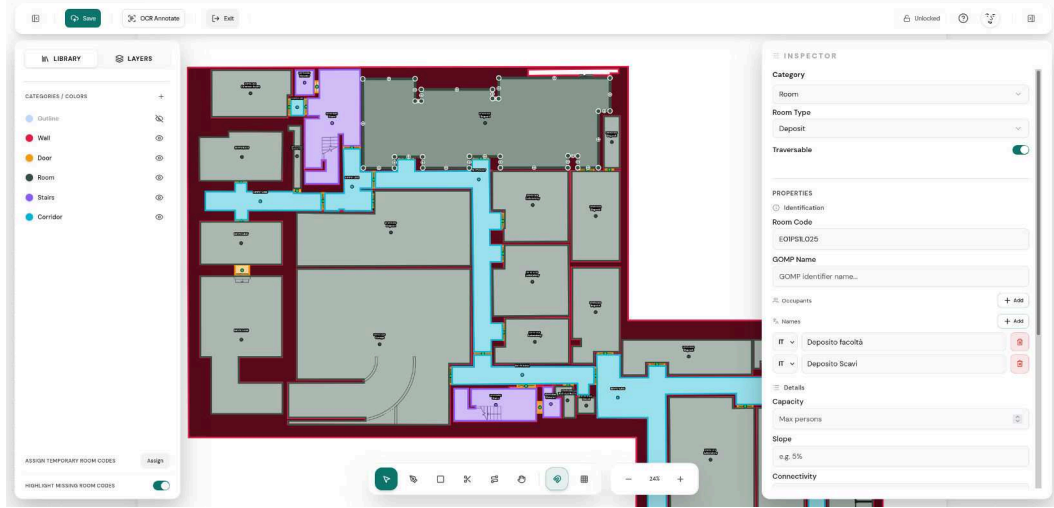


Figure 1.5 The frontend editor, designed to reduce floor plan corrections to a single direct gesture per operation.

Through this interface, an operator corrects the automatic detections and enriches them with room codes, names, accessibility flags, and door-to-room connections, turning the machine’s rough output into a navigation-grade map. The full editor system is described in Chapter 5.

1.4 Contributions

The contributions of this thesis, in summary, are:

1. A controlled comparative evaluation of YOLOv11-m and RT-DETR-L on the CubiCasa5K floor plan dataset, showing that the transformer’s advantage is not uniform: at IoU 0.5 YOLOv11-m is marginally ahead, on the classes that reach the output and clearly ahead on staircase symbols, yet the transformer wins on the measurements that govern deployment: localization accuracy, the fraction of objects missed entirely, and validation losses that stay flat instead of drifting upward as training continues. Qualitative testing on smartphone photographs of real emergency plans suggests that the relative ranking holds on the target input modality, where the transformer also avoids reading text and evacuation arrows as walls.
2. A geometric post-processing pipeline that converts raw bounding-box detections into clean, non-overlapping polygonal annotations through unary union

merging, flood-fill room segmentation guided by distance transform, class-adaptive Douglas-Peucker simplification, and orthogonalization, producing COCO-formatted output ready for ingestion by a navigation engine.

3. A decoupled optical character recognition service, using PaddleOCR’s PP-OCRv4 Latin-script model, which covers Italian text, that extracts room codes from cropped polygon regions through a five-stage pipeline of padding, polygon masking, cross-bleed removal, upscaling, and heuristic word ordering, integrated as an independent endpoint that can be called separately from the detection step.
4. A full-stack web application comprising a FastAPI backend with asynchronous model serving and OIDC authentication, and a SvelteKit 5 frontend editor with SVG-based polygon manipulation, geometric snapping, Boolean slicing and merging, undo/redo history, and semantic annotation for door-room linking and accessibility metadata.

1.5 Thesis Outline

Chapter 2 reviews the related work in floor plan analysis, from rule-based systems through the deep learning era, and identifies the specific research gap this thesis addresses: the extraction of structured spatial data from smartphone photographs of emergency evacuation plans. Chapter 3 describes the dataset engineering process, the model architectures, and the training methodology. Chapter 4 presents the experimental results, with a detailed quantitative comparison between YOLOv11-m and RT-DETR-L across all object classes. Chapter 5 covers the system architecture: the backend inference pipeline, the geometric post-processing chain, the OCR service, and the frontend interactive editor. Chapter 6 concludes with a summary of the work and directions for future research, including the potential for self-supervised pretraining on unlabeled floor plan images.

Chapter 2

State of the Art and Related Work

This chapter situates the present work within the broader literature on floor plan analysis. I trace the evolution from early rule-based systems to contemporary deep learning approaches and examine the datasets that have shaped the field, then identify the specific gap that this thesis fills: the extraction of structural elements from smartphone photographs of emergency evacuation floor plans, where the input quality is worse and the target semantics simpler than in most published work on the topic.

2.1 Early Approaches: Established Rules and Heuristics

The first generation of floor plan analysis systems, developed between roughly 2000 and 2015, relied on explicit geometric rules encoded by the programmer. Dosch et al.⁽⁵⁾ presented an end-to-end system for architectural drawing analysis that detected walls by their line thickness and identified junctions through *template matching*; from the graph of connected wall segments this produced, rooms could then be inferred as the graph’s *minimal cycles*. The system assumed clean, black-and-white vector scans at known resolution; it handled common architectural conventions like *hatching* and *dimension lines*¹ through dedicated filtering modules.

Macé et al.⁽¹⁸⁾ proposed a room detection system based on the *Hough transform*⁽⁸⁾. Lines were detected and clustered by orientation and proximity, then assembled into a wall skeleton, from which rooms were identified through graph-based analysis of the partitions the wall lines formed. The system achieved reasonable results on clean architectural drawings but was explicitly limited to orthogonal wall layouts, a constraint inherited from the line detection stage.

¹Hatching is the pattern of parallel or cross-hatched lines used to fill a cross-sectioned area in a technical drawing, typically marking it as solid material such as a wall cut in section. Dimension lines are the annotated lines, usually terminated by arrows, that indicate a measured distance between two points on the drawing.

These early systems share two limitations that make them unsuitable for the Navindoor use case. First, they require the input to be a high-quality scan or a vector file; the output quality degrades sharply with the noise, perspective distortion, and uneven illumination of a smartphone photograph. Second, their rule sets encode assumptions about drawing conventions that do not transfer across countries, decades, or even individual architects: the thickness of a wall line, the representation of a door swing, the symbology for stairs. A rule-based system needs a new set of heuristics for every source of plans, and Navindoor must ingest plans from buildings constructed between the 1930s and the 2010s, drawn by different architects to different standards.

2.2 The Deep Learning Era

The release of large annotated datasets and the maturation of convolutional neural networks shifted floor plan analysis from hand-crafted rules to data-driven models. The CubiCasa5K dataset⁽¹³⁾, published by Kalervo et al. in 2019, quickly became the landmark resource in the field, containing 5000 floor plan images with pixel-perfect SVG vector annotations covering over 80 object categories: walls, doors, windows, room polygons, and symbols for furniture, appliances, and fixtures. The images come in three visual styles (high-quality architectural, high-quality, and colorful), each rendered from the vector annotations, so there is zero annotation noise at pixel level. The dataset’s scale and annotation quality made it the de facto standard for floor plan deep learning, and it forms the training foundation for this work as well.

CubiCasa5K’s dataset is nonetheless residential in scope. Its floor plans come from Finnish apartment listings: small rooms, few corridors, wall thicknesses that follow residential building codes, and a narrow set of room types (kitchen, bedroom, bathroom, living room, sauna). Navindoor operates in university buildings, which have a different topology: long double-loaded corridors, auditoriums, laboratories, and irregularly shaped common areas. The gap between the training distribution and the target distribution is one of the problems I had to address, and I discuss the measures taken (dual-domain dataset construction, domain-specific augmentation) in Chapter 3.

The floor plan research community has produced a steady stream of models built on CubiCasa5K. Kalervo et al.’s own multi-task model used an hourglass network⁽¹⁹⁾ with a ResNet-152 backbone and uncertainty-weighted loss⁽¹⁴⁾ to jointly predict room segmentation, icon segmentation, and wall junction heatmaps. The output went through a post-processing stage that extracted polygons from the segmentation masks and resolved junctions through a combinatorial optimization over the wall graph.

More recent work has explored transformer-based architectures for floor plan understanding. MitUNet⁽²²⁾ combined a Mix Transformer encoder (MiT) with a UNet decoder for wall segmentation on CubiCasa5K, reporting improvements over pure CNN approaches in boundary accuracy. A different direction came from FGSS-Net⁽²⁰⁾, which introduced feature-guided attention modules that direct the network’s focus toward semantically ambiguous regions like door openings, where the visual boundary between two adjacent rooms is a thin gap rather than a solid line. Hakert

and Schönfelder⁽¹⁰⁾ took yet another approach, applying informed machine learning methods that incorporate geometric priors such as wall parallelism and orthogonality directly into the loss function to produce more architecturally plausible outputs.

2.3 Object Detection Architectures: CNN vs. Transformer

A central design choice in this thesis was whether to use a convolutional or a transformer-based object detector. The two families differ fundamentally in how they process spatial information.

Convolutional detectors, of which the YOLO series⁽¹⁶⁾ is a prominent example, build representations hierarchically: each layer in a CNN backbone aggregates information from a small spatial neighborhood, and as the network deepens, each pixel in the feature map comes to represent a progressively larger region of the input image. YOLOv11’s architecture follows the established pattern of CSPNet backbone, PAN-FPN neck, and a decoupled head that predicts classification, bounding box regression, and distribution focal loss separately. It is not purely convolutional, however: after the SPPF module, C2PSA blocks apply a partial form of spatial self-attention on a subset of channels. This is a limited relational mechanism, in both extent (a fraction of the channels, at a coarse resolution, inside the backbone) and role (refining features rather than establishing object-level relations). The dominant inductive bias remains convolutional: translation equivariance and local connectivity, a good fit for natural images where objects are compact and relatively independent of each other. On floor plans, however, a wall segment that spans half the image width has the same class label at every point along its length; a kernel whose effective context is local cannot determine whether the dark line it is processing is a wall, a dimension annotation, or the edge of a table, because that determination depends on context that may be hundreds of pixels away. The question separating the two detector families examined in this thesis is how much of that context each architecture makes available to a decision: YOLOv11’s limited, channel-split attention offers far less than the global, cross-scale attention of a detection transformer.

Transformer-based detectors address this limitation through self-attention. The DETR architecture⁽⁴⁾, introduced by Carion et al. in 2020, reformulated object detection as a set prediction problem: a fixed number of learned object queries attend to a feature map produced by a CNN backbone and a transformer encoder, and the output of cross-attention between queries and features is decoded directly into class labels and bounding boxes. DETR was slow to converge and struggled with small objects, problems that were successively addressed by Deformable DETR⁽²⁹⁾ and DINO⁽²⁷⁾, which introduced deformable attention and denoising training to stabilize convergence.

RT-DETR⁽²⁸⁾ is the most recent iteration in this line, designed specifically for real-time performance. It uses a hybrid encoder where a CNN backbone (HGNetv2) produces multi-scale feature maps that are processed by an intra-scale feature interaction module and a cross-scale feature fusion module (a form of CNN-transformer

hybrid neck) before being passed to a transformer decoder. The decoder uses denoising queries during training: a subset of the object queries is perturbed with noise and the model learns to reconstruct the original ground truth from the noisy version, an auxiliary task that accelerates convergence and improves performance, particularly on small objects. A query selection mechanism further prunes low-confidence queries before the final cross-attention stage, reducing both computation and false positives. RT-DETR was designed to match or exceed YOLO’s speed while surpassing it in accuracy, and its reported advantages on small and context-dependent objects made it a natural candidate for the floor plan detection task.

2.4 Challenges of Emergency Floor Plans

Most floor plan research, both in the computer vision and the indoor navigation communities, works with clean inputs: vector renders, high-resolution scans, or structured CAD exports. A photograph of a printed emergency plan introduces challenges that are qualitatively different from those addressed in the core floor plan literature.

The first is geometric degradation. Perspective projection turns orthogonal walls into arbitrary quadrilaterals, the paper is often creased or curved against the wall, and the camera is rarely held perfectly parallel to the plan, introducing keystone distortion that stretches one part of the image relative to another. Standard data augmentation techniques (random rotation, perspective warping) can simulate some of these effects during training, but they cannot fully cover the distribution of real-world degradations without also destroying the geometric structure that the model needs to learn.

The second is semantic noise. Emergency plans are annotated with evacuation routes: colored arrows that cut through rooms, large text blocks listing safety procedures, symbols for fire equipment, and dimension lines that run alongside walls. A model that has been trained primarily on clean vector renders will tend to classify these annotations as structural elements. The evacuation arrow becomes a thin diagonal wall; the dimension line becomes a room boundary. How well a model generalizes to this kind of noise depends on how it represents spatial context: if the model can attend to the entire floor plan at once and recognize that a dark line that crosses five rooms is likely an annotation rather than a wall, it has an advantage over a model that processes each local patch independently.

Hassaan et al.⁽¹²⁾ addressed a similar problem in their 2023 work on emergency floor plan digitization, where they combined color filtering and clustering to extract walls and trained two Faster R-CNN detectors to classify emergency symbols on photographs of printed plans. Their pipeline relies on the color scheme of the input plans as a preprocessing cue, which limits its applicability to plans with different color conventions. My approach avoids manual color assumptions entirely and relies on the model’s learned representations to distinguish structure from annotation.

2.5 Indoor Navigation and Map Representation

A complementary line of work, adjacent to floor plan analysis proper, addresses the problem of converting extracted spatial data into navigable representations. Tesseract⁽¹⁾, developed at Carnegie Mellon University, converts low-semantic floor plan images into navigable graph structures through a pipeline of spatial segmentation, text detection, and graph construction with pruning. The system was evaluated on real university buildings and achieved over 90% navigational completeness, meaning that a routing algorithm could find a path between any two reachable rooms in the generated graph. Tesseract’s approach shares with this thesis the goal of producing machine-readable spatial data from images, but its input assumptions are closer to architectural drawings than to smartphone photographs, and its output graph representation is coarser than the polygon-level annotations required by Navindoor’s routing engine.

2.6 The Gap: raw input and a practical implementation

The existing literature leaves a specific gap that this thesis addresses. No published work, to my knowledge, tests a transformer-based detector (specifically RT-DETR-L) on smartphone photographs of emergency evacuation floor plans and compares its performance against a convolutional baseline (specifically YOLOv11-m) on the same task with the same training data. The quantitative advantage of global self-attention for understanding noisy, cluttered floor plan images has been hypothesized, and studies with earlier DETR variants on clean architectural plans provide suggestive evidence, but direct evidence with modern real-time transformer detectors on the specific input modality that Navindoor requires is absent.

The second gap is practical rather than methodological: the integration of a deep learning detector with a geometric post-processing pipeline, an OCR module for room code extraction, and an interactive frontend editor into a single deployable system that produces navigation-grade maps. Published work on floor plan analysis typically evaluates detection or segmentation accuracy in isolation, using COCO metrics on clean test splits. A deployed navigation system needs more than accuracy metrics. It needs a pipeline reliable enough for uncured real-world inputs and fast enough for interactive use, with a workflow that lets a human operator efficiently correct whatever errors remain. The system described in Chapter 5 is the engineering response to that requirement.

Chapter 3

Deep Learning Methodology

This chapter describes the dataset engineering process, the model architectures, and the training methodology I used to build the floor plan object detector. The end product is a model that takes a photograph of a floor plan as input and outputs class-labeled bounding boxes for five categories: walls, rooms, doors, door stoppers, and stairs.

3.1 Dataset Engineering

3.1.1 The CubiCasa5K Foundation

All training data derives from the CubiCasa5K dataset⁽¹³⁾, which contains 5000 floor plan images with per-pixel SVG vector annotations. The dataset ships with three visual styles per floor plan: `high_quality`, `high_quality_architectural`, and `colorful`. Each sample is a directory containing a raster rendering of the floor plan (`F1_scaled.png`) and an SVG file (`model.svg`) with the vector annotations. Multi-floor buildings include additional images (`F2_scaled.png`, etc.) with corresponding floor-level SVG groups.

The dataset comes with predefined train/val/test splits in three text files listing folder IDs. I validated these splits for data leakage before proceeding: I checked for duplicate folder IDs within each split and for overlapping IDs between splits. No leaks were found. The split distribution is approximately 80% train, 10% validation, 10% test, consistent with standard machine learning practice.

3.1.2 SVG Parsing and Coordinate Recovery

The SVG annotations encode geometry through a combination of XML elements (`<polygon>`, `<path>`, `<rect>`, `<line>`) and hierarchical transformation matrices. Recovering the absolute pixel coordinates for each annotation requires traversing the XML tree upward from each element, accumulating the product of all `transform` attributes along the path to the root. The SVG specification supports five transform types and their compositions.

I implemented a full SVG transform parser in Python. For each leaf element, the parser walks from the element to the root, collecting all `transform` attributes, then multiplies the matrices in the correct order (root to leaf, as specified by the SVG standard) to obtain the cumulative transformation matrix $M = (a, b, c, d, e, f)$. Each vertex (x, y) of the primitive is then transformed as:

$$x' = a \cdot x + c \cdot y + e, \quad y' = b \cdot x + d \cdot y + f \quad (3.1)$$

Path elements, which use SVG's `d` attribute with commands like `M` (move), `L` (line), `C` (cubic Bézier), and `Q` (quadratic Bézier), require additional handling. Bézier curves are tessellated into polyline segments by sampling at $t \in \{0.0, 0.1, \dots, 1.0\}$, applying the standard Bézier formulas:

$$B_{\text{quad}}(t) = (1-t)^2 P_0 + 2(1-t)t P_1 + t^2 P_2 \quad (3.2)$$

$$B_{\text{cubic}}(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad (3.3)$$

The resulting polyline vertices are then run through the transformation matrix as above. Area is computed from the transformed polygon using the Shoelace formula:

$$A = \frac{1}{2} \left| \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right| \quad (3.4)$$

where $(x_n, y_n) = (x_0, y_0)$ closes the polygon. The arithmetical centroid is computed as the average of all vertices:

$$(c_x, c_y) = \left(\frac{1}{n} \sum_{i=0}^{n-1} x_i, \frac{1}{n} \sum_{i=0}^{n-1} y_i \right) \quad (3.5)$$

3.1.3 Class Mapping

CubiCasa5K defines over 80 SVG class names covering architectural elements (Wall, Space, Door, Window), room types (Kitchen, Bedroom, LivingRoom), and furniture (Sofa, Bathtub, Oven). For the Navindoor use case, only five classes are relevant. I mapped the SVG classes to COCO category IDs as shown in Table 3.1.

Table 3.1 Mapping from CubiCasa5K SVG classes to target COCO categories.

COCO ID	Name	Relevant SVG Classes
1	Wall	Wall
2	Door	Door, Door-Open, Door-Shower
3	Room	Space, Kitchen, Bedroom, LivingRoom, and 40+ room subtypes
4	Door Stopper	DoorStopper (threshold lines within door elements)
5	Stairs	Stairs and Elevators

Door stoppers are extracted by inspecting child elements of each Door node: a child whose ID or class contains the substring **threshold** is classified as a door stopper. All other descendants of the Door node that represent geometry (polygons, paths, rects) are classified as the door itself. This heuristic correctly separates the door arc from its threshold line across the full CubiCasa5K dataset.

3.1.4 Dual-Domain Rasterization

A model trained exclusively on clean vector renderings will struggle with the photographic inputs it encounters at inference time. To close this gap, I constructed a dataset that pairs each vector-annotated floor plan with two raster domains:

Vectorized domain The `F1_vectorized.png` renderings: clean, high-contrast, black-on-white line drawings with uniform stroke widths.

Scaled domain The `F1_scaled.png` images: raster versions that approximate what a scanned or photographed floor plan looks like, with varying line widths, anti-aliasing artifacts, and paper-like backgrounds.

Both images for each sample share the same COCO annotations, since they are renders of the same underlying SVG. This doubles the effective dataset size from approximately 5000 to approximately 10000 training images without introducing additional labeling cost. The scaled domain gives the model a coarse proxy for photographic inputs; the vectorized domain keeps it anchored to a clean representation of the underlying geometry. The split assignment described in Section 3.1.1 is performed at the folder level, before the two domain variants are generated, so both variants of a given floor plan always fall in the same split; the doubling therefore does not introduce cross-split leakage between train, validation, and test.

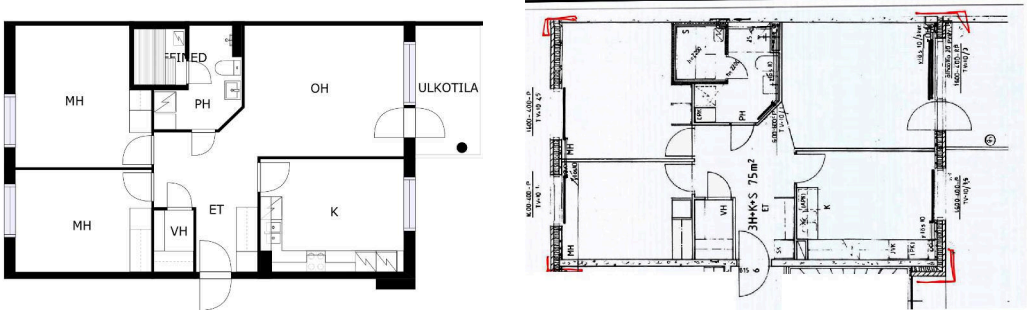


Figure 3.1 Left: vectorized rendering (F1_vectorized.png). Right: scanned-style rendering (F1_scaled.png). Both images share the same COCO annotations extracted from the SVG.

3.1.5 Format Conversion: COCO to YOLO Segmentation

The SVG-to-COCO conversion produces standard COCO segmentation JSON files with polygon coordinates in absolute pixel units. Object detection frameworks in the YOLO ecosystem expect a different format: class-normalized polygons, where each polygon vertex (x, y) is divided by the image width W and height H to map coordinates into the $[0, 1]$ range:

$$x_{\text{norm}} = \frac{x}{W}, \quad y_{\text{norm}} = \frac{y}{H} \quad (3.6)$$

The normalized polygon is written to a text file, one line per annotation, with the format:

```
<class_id> <x1> <y1> <x2> <y2> ... <xn> <yn>
```

where the class ID is a zero-indexed integer matching the order in the `dataset.yaml` configuration file. In my setup the mapping is: 0 = wall, 1 = door, 2 = room, 3 = door_stopper, 4 = stairs.

3.1.6 Image Denoising Exploration

As an auxiliary experiment, I trained an image-to-image denoising model to restore clean floor plans from synthetically degraded versions. The motivation was to see whether a preprocessing step could close the domain gap between training images and real smartphone photographs before the object detector saw the input. The denoiser uses a UNet architecture with a ResNet34 encoder pre-trained on ImageNet, operating on three-channel RGB images. The loss function combines L1 pixel-wise loss with a structural dissimilarity term:

$$\mathcal{L}_{\text{denoise}} = \mathcal{L}_{\text{L1}} + 0.2 \cdot (1 - \text{SSIM}(I_{\text{pred}}, I_{\text{target}})) \quad (3.7)$$

where SSIM is the structural similarity index⁽²⁶⁾. Degradation is applied on-the-fly during training, with a random composition of flash/lighting gradients, Gaussian

noise, blur, brightness/contrast shifts, and gamma distortion. Each sample sees a unique degradation pattern in every epoch.

In practice, RT-DETR-L proved capable of absorbing most of the photographic noise without a dedicated denoising step. The denoiser remains an architectural option for future deployments where input quality is particularly poor, but it is not part of the main inference pipeline.

3.2 Model Architectures

I trained and compared two object detectors: YOLOv11-m, representing the convolutional family, and RT-DETR-L, representing the transformer family. Both were initialized from the standard pre-trained checkpoints distributed by Ultralytics, which are COCO-pretrained for both models, and fine-tuned on the floor plan dataset.

3.2.1 YOLOv11-m

YOLOv11⁽¹⁶⁾ follows the architectural pattern of its predecessors in the YOLO series. The backbone is a CSPNet (Cross Stage Partial Network) that splits feature maps and processes them through parallel convolutional branches, merging them at the output. A Spatial Pyramid Pooling–Fast (SPPF) module at the end of the backbone aggregates multi-scale context, and is followed by C2PSA (Convolutional block with Parallel Spatial Attention) blocks. The C2PSA block deserves attention in the context of this thesis: part of its channels are processed by a parallel spatial attention (PSA) module that applies multi-head self-attention over spatial positions, while the remaining channels bypass it. YOLOv11 is therefore not purely convolutional—it contains a limited form of self-attention, operating on a subset of channels of a downsampled feature map within the backbone. The neck uses a PAN-FPN (Path Aggregation Network with Feature Pyramid Network) structure that propagates semantic information from deep layers upward and spatial detail from shallow layers downward through lateral connections. The detection head is decoupled: classification and bounding box regression are predicted by separate convolutional branches, and a Distribution Focal Loss (DFL) branch predicts the distribution of each bounding box edge rather than regressing it directly.

The YOLO loss function is a weighted sum of three components:

$$\mathcal{L}_{\text{YOLO}} = \lambda_{\text{box}} \cdot \mathcal{L}_{\text{CIoU}} + \lambda_{\text{cls}} \cdot \mathcal{L}_{\text{BCE}} + \lambda_{\text{dfl}} \cdot \mathcal{L}_{\text{DFL}} \quad (3.8)$$

$$\mathcal{L}_{\text{CIoU}} = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}_{gt})}{c^2} + \alpha v \quad (3.9)$$

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] \quad (3.10)$$

$$\mathcal{L}_{\text{DFL}} = -\frac{1}{N} \sum_{i=1}^N [(y_{i+1} - y) \log(S_i) + (y - y_i) \log(S_{i+1})] \quad (3.11)$$

CIoU (Complete IoU) extends the standard IoU loss with a center-point distance penalty and an aspect ratio consistency term, where ρ is the Euclidean distance between box centers, c is the diagonal length of the smallest enclosing box, and αv is the aspect ratio term. DFL discretizes the continuous regression target into n bins and trains the model to predict a distribution over these bins, focusing the loss on the two bins adjacent to the ground truth value. Figure 3.2 summarizes the resulting architecture.

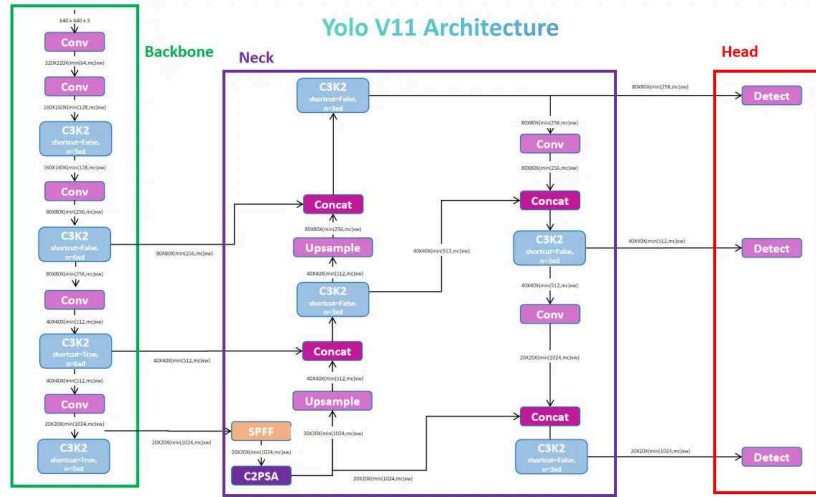


Figure 3.2 Architecture of the YOLO11 model, detailing the Backbone, Neck, and Head components along with C3k2, SPPF, and C2PSA blocks.

3.2.2 RT-DETR-L

RT-DETR⁽²⁸⁾ is a real-time detection transformer that combines a CNN backbone with a transformer encoder-decoder architecture. The backbone is HGNetv2, an efficient hierarchical CNN that produces feature maps at strides 8, 16, and 32. These multi-scale features are processed by a hybrid encoder: an intra-scale feature interaction module (AIFI), which applies self-attention within the coarsest feature map only (S_5 , at 1/32 of the input resolution), followed by a cross-scale feature fusion module (CCFF), which merges features across scales through a series of convolutional fusion blocks.

The decoder operates on a fixed set of learned object queries. During training, denoising queries are used: a portion of the ground truth bounding boxes has noise added to both their class labels and coordinates, and these noisy queries are fed to the decoder alongside clean queries. The model must reconstruct the original ground truth from the noisy input, which acts as an auxiliary reconstruction task that stabilizes training and accelerates convergence, particularly on small objects.

The RT-DETR loss combines classification and regression terms:

$$\mathcal{L}_{\text{RT-DETR}} = \lambda_{\text{cls}} \cdot \mathcal{L}_{\text{VFL}} + \lambda_{\text{L1}} \cdot \mathcal{L}_{\text{L1}} + \lambda_{\text{GIoU}} \cdot \mathcal{L}_{\text{GIoU}} \quad (3.12)$$

$$\mathcal{L}_{\text{VFL}}(p, q) = \begin{cases} -q(q \log(p) + (1 - q) \log(1 - p)) & \text{if } q > 0 \\ -\alpha p^\gamma \log(1 - p) & \text{if } q = 0 \end{cases} \quad (3.13)$$

$$\mathcal{L}_{\text{L1}} = \frac{1}{N} \sum_{i=1}^N |b_i - b_{gt,i}| \quad (3.14)$$

$$\mathcal{L}_{\text{GIoU}} = 1 - \text{IoU} + \frac{|C \setminus (B \cup B_{gt})|}{|C|} \quad (3.15)$$

Varifocal Loss (VFL) is an asymmetric variant of focal loss where positive samples (the assigned ground-truth boxes) receive the full binary cross-entropy loss weighted by the IoU score q between the predicted and ground truth box, while negative samples receive a reduced penalty modulated by a focusing parameter γ . This encourages the model to produce high-confidence predictions for well-matched boxes while suppressing low-quality predictions that happen to have a high IoU with some ground truth. GIoU (Generalized IoU) adds a penalty proportional to the area of the smallest enclosing box C that is not covered by the union of the predicted box B and the ground truth box B_{gt} , making the loss sensitive to the spatial relationship between non-overlapping boxes.

3.3 Training Setup

Both models were trained under identical hardware and data conditions on a single NVIDIA RTX 3090 Ti (24 GB VRAM): same training set, same image resolution, same optimizer family, and, unlike an initial plan that considered Objects365 weights for RT-DETR, the same pretraining dataset: the standard Ultralytics checkpoints are COCO-pretrained for both models, as noted in Section 3.2. Table 3.2 summarizes the hyperparameter configuration.

Table 3.2 Training hyperparameters for YOLOv11-m and RT-DETR-L. Values that differ between models are shown in bold.

Parameter	YOLOv11-m	RT-DETR-L
Epochs	150	150
Epochs completed	115	133
Image size	1024×1024	1024×1024
Batch size	6	6
Optimizer	AdamW	AdamW
Initial learning rate	1×10^{-3}	1×10^{-4}
Final learning rate factor	0.01	0.01
Weight decay	5×10^{-4}	5×10^{-4}
LR schedule	Cosine annealing	Cosine annealing
Warmup epochs	0	3
Warmup momentum	—	0.8
Precision	Mixed (AMP)	Mixed (AMP)
Seed	42	42
Deterministic	True	True

The learning rate schedule follows common practice for transformer decoders, which tend to train more stably with a lower initial learning rate and a warmup phase, followed by cosine annealing down to $lr_0 \cdot lr_f$.

Mixed-precision training (AMP) was enabled for both models to reduce memory consumption and increase throughput. For RT-DETR, this required disabling AMP for certain operations where the reduced precision led to NaN values, but the Ultralytics framework handles this automatically.

3.3.1 Domain-Specific Data Augmentation

Standard object detection augmentation pipelines assume natural images where rotation, shear, and perspective warping are physically plausible. Floor plans violate these assumptions. Walls meet at right angles, door openings sit flush against the walls that contain them, and a room is simply the space enclosed by its surrounding walls. Rotating a floor plan by 45 degrees teaches the model to expect diagonal walls, which do not exist in the target domain.

I configured the augmentation pipeline as follows:

- **Horizontal flip:** 50% probability. Vertical flip is disabled because the distinction between a wall and a flipped wall is meaningless, but the spatial relationship between rooms and their labels is directional in multi-floor buildings (stair symbols point upward).
- **Rotation:** 0%. Rotated walls are out-of-distribution.
- **Shear and perspective:** 0%. Both introduce non-orthogonal geometry.
- **Translation:** up to 10% of the image dimension. Small shifts are acceptable.

- **Scaling:** up to $\pm 30\%$. Wall thicknesses vary across drawings; scale augmentation provides robustness to variations in line weight.
- **HSV shifts:** $h = \pm 0.015$, $s = \pm 0.3$, $v = \pm 0.3$. Saturation and brightness variations simulate different paper colors and lighting conditions. The hue shift is kept minimal because wall color is semantically neutral in floor plans.
- **Mosaic:** 50% probability, disabled after epoch 15. Mosaic augmentation, which stitches four training images into a single composite input, exposes the model to more diverse spatial configurations early in training. Disabling it for the final 15 epochs allows the model to fine-tune on full-resolution, single-image inputs, matching the inference condition.
- **Mixup and erasing:** 0%. Overlapping two floor plans creates ambiguous geometry; random erasing can delete door openings.

3.3.2 Handling Class Imbalance

The CubiCasa5K class distribution is heavily skewed. Walls and rooms appear in virtually every image. Doors appear in most images but with far fewer instances per image. Door stoppers and stairs are sparse: many images contain no staircase at all, and door stoppers are thin threshold lines that are sometimes absent even from images that contain doors.

To prevent the model from ignoring rare classes, I oversampled them during training data preparation: every image that contains at least one instance of a door stopper or staircase is duplicated so that it appears three times in total in the training set, both the image file and its label file. This provides three times as many exposures to the rare classes without altering the distribution of wall and room instances within those images.

Checkpoint selection and early stopping use the default Ultralytics fitness for both models, $0.1 \cdot \text{mAP}@50 + 0.9 \cdot \text{mAP}@50-95$, so the two detectors are selected by the same criterion.

3.4 Evaluation Metrics

I evaluate detection performance using the standard COCO metrics.

Intersection over Union (IoU) quantifies the spatial overlap between a predicted bounding box B_p and a ground truth box B_{gt} :

$$\text{IoU}(B_p, B_{gt}) = \frac{|B_p \cap B_{gt}|}{|B_p \cup B_{gt}|} \quad (3.16)$$

IoU depends only on the area enclosed, not on the shape of the boundary that encloses it. A polygon with dozens of jagged vertices along a wall edge scores the same as a clean straight line, as long as the two enclose roughly the same area against the ground truth; the metric has no way to penalize a boundary that is geometrically noisy but happens to sit inside the right region.

A prediction is considered a true positive if its IoU with a ground truth box of the same class exceeds a threshold, typically 0.5. A prediction that does not exceed the IoU threshold with any unmatched ground truth box is a false positive; a ground truth box that is not matched by any prediction is a false negative.

Precision and **Recall** are computed from the set of all predictions across all images, sorted by confidence:

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.17)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.18)$$

Average Precision (AP) summarizes the precision-recall curve by computing the area under it, interpolated at 101 recall points from 0 to 1. mAP@50 reports the mean AP across all classes at an IoU threshold of 0.5. mAP@50-95 averages mAP over ten IoU thresholds from 0.5 to 0.95 in steps of 0.05, providing a more demanding metric that rewards precise localization.

Confusion matrices are normalized per ground truth class: each entry (i, j) is the fraction of ground truth instances of class i that were predicted as class j . Diagonal entries represent recall per class; off-diagonal entries reveal systematic confusions between classes.

Chapter 4

Experimental Results

This chapter compares YOLOv11-m and RT-DETR-L on the floor plan detection task using a shared experimental setup: the same data, image resolution, batch size, and optimizer, with both models evaluated on an identical validation split. Overall, the transformer-based detector comes out ahead, and the margin is largest on classes whose correct label depends on their position within the surrounding floor plan rather than on their appearance alone.

These numbers come from the CubiCasa5K validation split, chosen for its pixel-perfect annotations and reproducibility. It is, however, a proxy domain: CubiCasa5K contains renderings of residential floor plans, and the actual target application is smartphone photographs of university emergency evacuation plans. The quantitative results settle which architecture wins under controlled conditions; the qualitative observations that close the chapter then check, without ground truth, whether that ranking still holds on real Sapienza photographs, the target input modality.

4.1 Experimental Setup

Training and evaluation both ran on a single NVIDIA RTX 3090 Ti with 24 GB of VRAM. The schedule allowed up to 150 epochs at 1024×1024 resolution with batch size 6, with early stopping in place: YOLOv11-m stopped at epoch 115, RT-DETR-L at epoch 133. All results reported in this chapter refer to the checkpoint selected by the default Ultralytics fitness for each model (*best.pt*), evaluated on the validation split; the training curves in Figures 4.1 to 4.4 are the per-epoch validations logged during training. Under these settings, a YOLOv11-m epoch takes about 11.4 minutes against 20.4 minutes for RT-DETR-L, putting total training time at roughly 21.8 hours for YOLOv11-m and 45.2 hours for RT-DETR-L. RT-DETR’s extra cost per image comes largely from its decoder, which processes more query embeddings per forward pass because of denoising queries: during training, a subset of the ground truth boxes is deliberately corrupted with noise and passed to the decoder alongside the normal object queries, and the decoder has to recover the clean boxes from that noisy input. This adds auxiliary queries to train against without requiring any extra step at inference time.

Both models were trained through the Ultralytics framework⁽²⁵⁾, which exposes a common training and evaluation API for YOLO and RT-DETR alike. It took care of mixed-precision training, gradient accumulation, multi-worker data loading, and per-epoch metric logging, and validated on the full validation split at the end of every epoch.

4.2 Global Training Trajectories

Figure 4.1 plots validation recall and precision for both models across the full training run, and the shapes of the two curves already say something about how differently the two architectures learn.

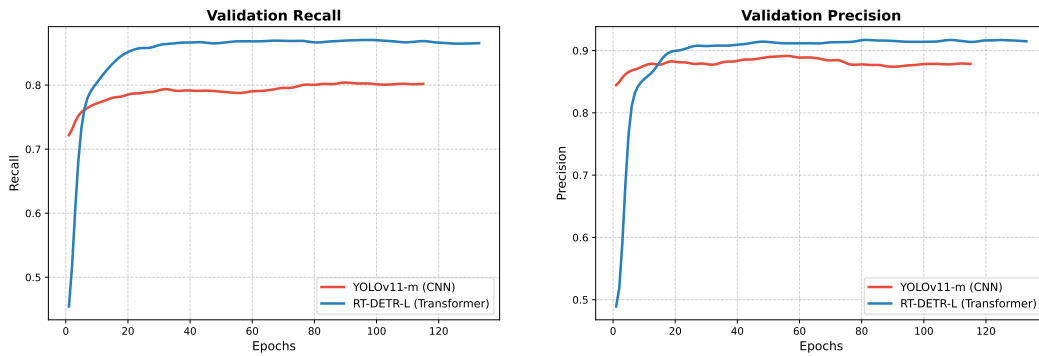


Figure 4.1 Validation recall (left) and precision (right) over the training run: 115 epochs for YOLOv11-m, 133 for RT-DETR-L.

YOLOv11-m has little ground to make up. By the end of epoch 1, recall already stands at 0.701 and precision at 0.833, a sign of how much of the dense prediction head’s job is already done before training even really begins. From there, gains slow down: recall climbs to about 0.795 by epoch 30, then edges up through minor oscillations to a final 0.802; precision settles near 0.878. Past roughly epoch 45, almost nothing changes, and the remaining two-thirds of the run amount to noise around a fixed point.

RT-DETR-L starts from a much weaker place: recall of 0.308 and precision of 0.565 after epoch 1. It does not stay there for long. Recall overtakes YOLOv11-m’s between epochs 5 and 6, and by epoch 30 it is already at 0.864, a few thousandths short of the eventual 0.866. Precision climbs the same way, from 0.565 up to a plateau around 0.915. Both curves cross above YOLOv11-m’s by roughly epoch 35 and never fall back below it.

The weak start is a direct cost of how RT-DETR’s decoder works: its object queries are learned embeddings with no built-in notion of where or what to look for, and they need several epochs to specialize toward particular positions, scales, and categories. YOLOv11 is anchor-free, but its dense head skips that step all the same: every location of the feature grid predicts a box relative to its own position, which acts as a spatial prior from the first epoch. The different learning rate schedules of the two models also contribute to the gap in the first epochs. None of this matters much for

the final model, though; what counts is where each curve ends up, not how fast it got there, and RT-DETR’s ceiling is simply higher.

4.3 Mean Average Precision

Figure 4.2 tracks mAP@50 and mAP@50-95 across training. Because each metric collapses precision and recall into a single number, they serve as the main basis for comparing the two detectors.

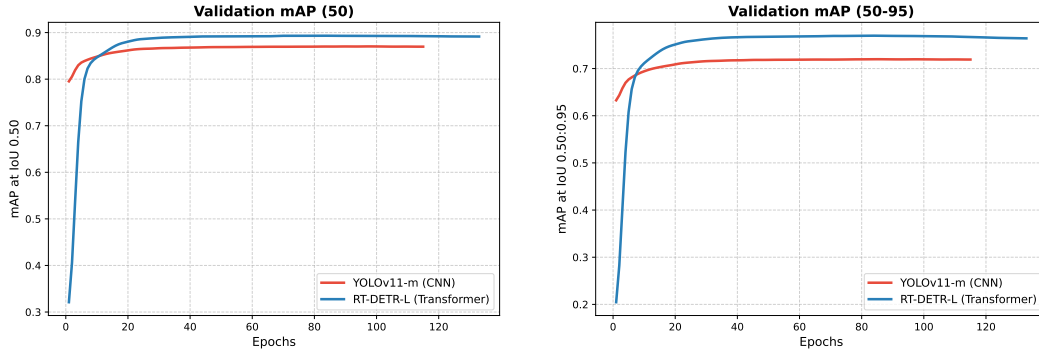


Figure 4.2 mAP@50 (left) and mAP@50-95 (right) validation curves over the training run.

At the selected checkpoint, RT-DETR-L reaches a mAP@50 of 0.894 and a mAP@50-95 of 0.764, against 0.870 and 0.719 for YOLOv11-m. The gap is +0.024 at mAP@50 against +0.045 at mAP@50-95, so it nearly doubles once localization accuracy is folded into the metric. That pattern fits what the two architectures actually do at the box-regression stage: RT-DETR’s decoder refines each box over several layers, with every query sampling multi-scale features around its current estimate, whereas YOLO’s head predicts the coordinates in a single step from the features at one grid location. Iterative refinement should mainly help localization rather than detection in general, which is consistent with the asymmetry the numbers show.

The crossover between the two curves happens earlier on the stricter metric than on the looser one: RT-DETR-L overtakes YOLOv11-m on mAP@50-95 by epoch 7, but needs until epoch 12 to do so on mAP@50, consistent with a decoder whose main advantage is localization rather than raw detection. The underlying trajectories explain why. RT-DETR-L opens at a mAP@50 of just 0.193, well below YOLOv11-m’s starting 0.771, closes that distance by epoch 12, and reaches its own plateau by roughly epoch 35, after which it oscillates in a narrow band for the remaining hundred or so epochs. YOLOv11-m, meanwhile, needs until about epoch 45 to settle into its own plateau of 0.868–0.870. Once both models have converged, the gap between them barely moves, which points to something built into the two architectures rather than to one model simply having trained for longer.

4.4 Validation Loss and Training Stability

Aggregate metrics like mAP compress detection quality into a single number, at the cost of saying nothing about how each model is generalizing from one epoch to the next. The validation loss curves fill that gap. Figures 4.3 and 4.4 give the full set of training and validation metrics, component losses included, for YOLOv11-m and RT-DETR-L respectively.

YOLOv11-m’s validation losses show a clear turning point: both the bounding box regression loss (`val/box_loss`) and the distribution focal loss (`val/df1_loss`) bottom out around epoch 35 in Figure 4.3, then climb steadily for the rest of the 115-epoch run. That climb is a textbook sign of mild overfitting on the localization task. Overall mAP does not reflect it, since a predicted box can drift somewhat and still clear the Intersection over Union (IoU) threshold needed to count as a true positive, but the model’s actual precision in placing boxes on validation data keeps eroding past that point.

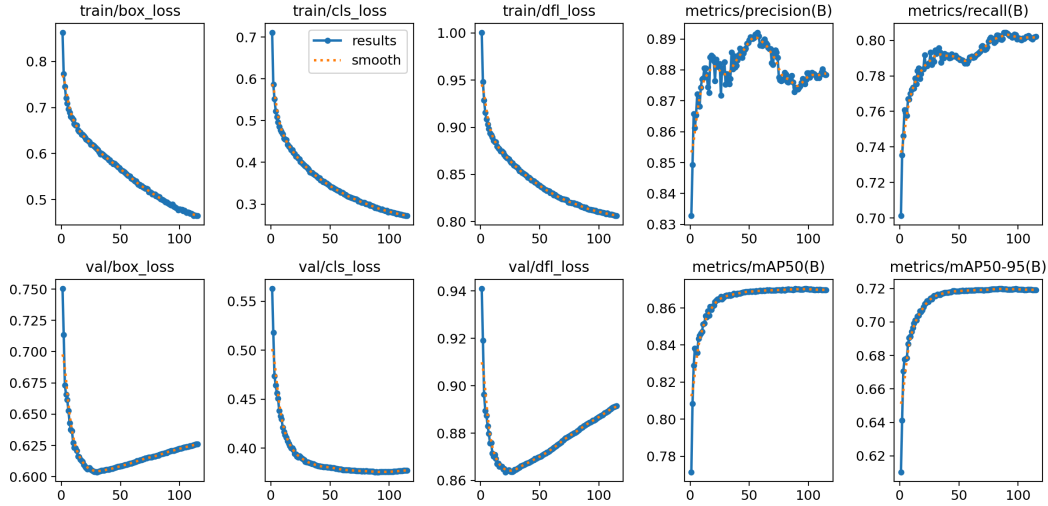


Figure 4.3 Training and validation metrics over 115 epochs for YOLOv11-m. Note the upward trend in validation box losses (`val/box_loss` and `val/df1_loss`) after epoch 35, indicating overfitting on the localization task.

RT-DETR-L’s validation losses tell a different story. Both `val/giou_loss` and `val/l1_loss`, shown in Figure 4.4, fall sharply early on, flatten out around epoch 50, and stay there with no upward drift all the way to epoch 133. Nothing in these curves suggests the model is starting to memorize exact box coordinates from the training set the way YOLOv11-m’s losses do. One plausible reason is that attending over the whole image gives the decoder less incentive to latch onto specific local patterns than a convolutional head has. Whatever the exact mechanism, the practical effect is that RT-DETR-L keeps its localization sharp on unseen data instead of degrading the way the CNN-based detector does, consistent with the wider mAP@50-95 margin noted earlier and a good sign for the polygon-generation step that follows.

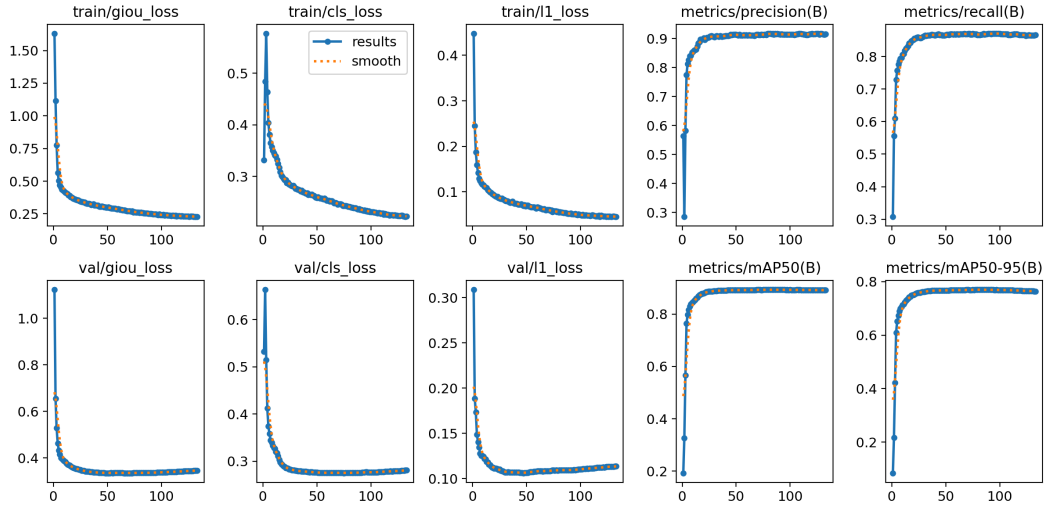


Figure 4.4 Training and validation metrics over 133 epochs for RT-DETR-L. The validation losses maintain a stable plateau after epoch 50, demonstrating better generalization for bounding box regression.

4.5 Per-Class Analysis

None of this shows up in an aggregate score. A model can look fine on overall mAP while still missing staircases or confusing structural boundaries, and either failure is a real problem for navigation. Table 4.1 breaks AP@50 down by class for both models on the validation split, computed from the same precision-recall evaluation.

Table 4.1 Per-class AP@50 on the validation split for the selected checkpoint. Difference is RT-DETR minus YOLO. The door stopper class (COCO ID 4) is excluded from the final pipeline output but is retained during training because forcing the model to distinguish the threshold line from the door arc and from nearby walls reduces door-wall confusion and improves the overall quality of door detections.

Class	YOLOv11-m	RT-DETR-L	Difference
Wall	0.884	0.889	+0.005
Door	0.956	0.949	-0.007
Room	0.954	0.940	-0.014
Door Stopper	0.735	0.948	+0.213
Stairs	0.821	0.740	-0.081
All	0.870	0.894	+0.024

The breakdown shows the two architectures behaving differently depending on how common the class is. On the three dominant structural classes, walls, doors, and rooms, both models score near or above 0.90, which mostly reflects how much training data those categories have. YOLOv11-m actually edges out RT-DETR-L on doors (0.956 vs. 0.949) and rooms (0.954 vs. 0.940), by a small margin in both cases.

What decides the overall mAP@50 lead is the door stopper class, where RT-DETR-L jumps ahead by +0.213 AP.

Minority classes like stairs and door stoppers are where the two mechanisms pull apart. YOLO’s convolutional features are tied to a small receptive field, and on this dataset that turns out to help on stairs: the model fits closely to whatever graphical pattern for stairs is most common in the training data, which is enough to win on this particular split even though stair symbols vary a lot across real floor plans, from parallel step lines to diagonal arrows to segmented polygons. A receptive field that size is prone to fitting on recurring line densities, treating them as a reliable local cue rather than learning what a staircase actually looks like in general.

Door stoppers flip the outcome. A door stopper here is a thin threshold line sitting right against a door arc, and telling the two apart from a small patch of image is hard. RT-DETR-L’s decoder instead reads the surrounding walls and room geometry before deciding, which is enough to separate a threshold line from a wall boundary in cases where a purely local view would not be. That context-driven approach costs some accuracy on stairs, where standardized layouts reward pattern-fitting over reasoning about context, but it is what drives RT-DETR-L’s clear win on door stoppers and, more speculatively, points to better generalization on floor plans the model has not seen before.

4.5.1 Precision-Recall and F1 Curves

Figure 4.5 and Figure 4.6 show the per-class precision-recall curves at IoU = 0.5 for YOLOv11-m and RT-DETR-L, respectively. In both plots, the area under each curve corresponds to the Average Precision (AP) of that specific class. Figure 4.7 and Figure 4.8 then show the corresponding F1-confidence curves, marking the confidence threshold that best balances precision and recall for each class.

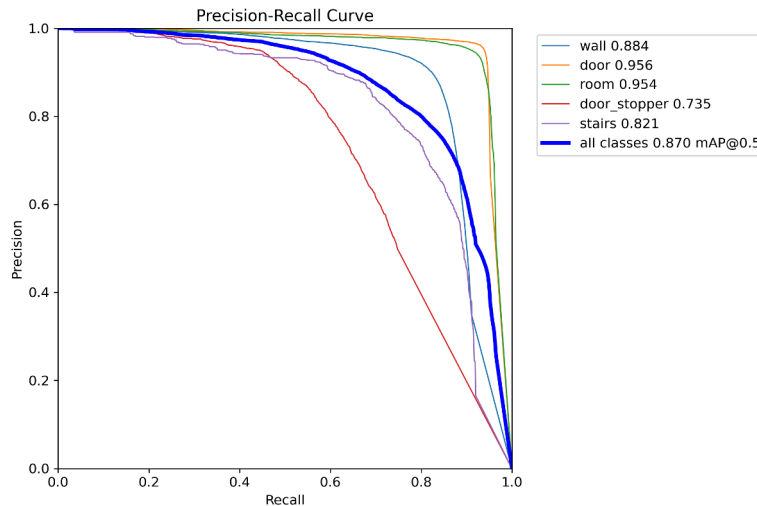


Figure 4.5 Per-class precision-recall curves at IoU = 0.5 for YOLOv11-m.

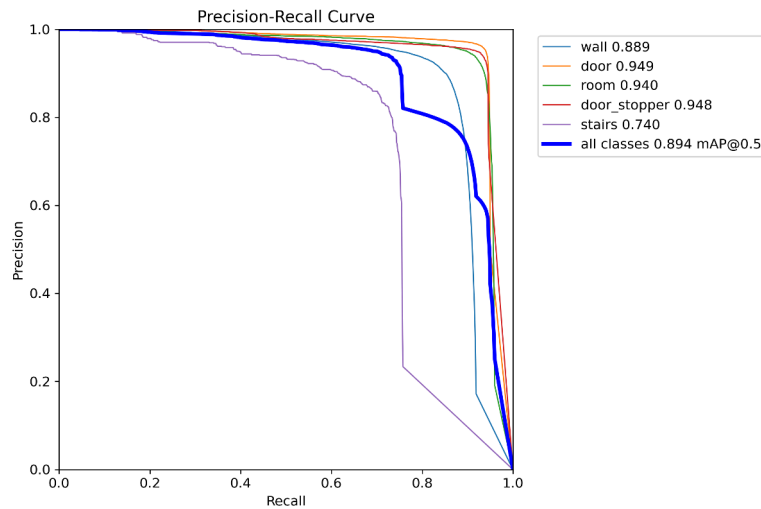


Figure 4.6 Per-class precision-recall curves at $\text{IoU} = 0.5$ for RT-DETR-L.

For YOLOv11-m, the door stopper curve falls off sharply past roughly 0.5 recall, meaning every additional detection beyond that point comes at the cost of a lot of false positives. Stairs hold up a little better on recall, with precision eroding steadily once recall passes 0.6.

RT-DETR-L’s door stopper curve stays high across a much wider recall range, close to what the major structural classes achieve. Stairs are the harder case here too: precision falls sharply around 0.75 recall, but it holds above 0.80 right up to that point, which points to a set of high-confidence detections that eventually runs out, rather than a steady leak of false positives across the whole range.

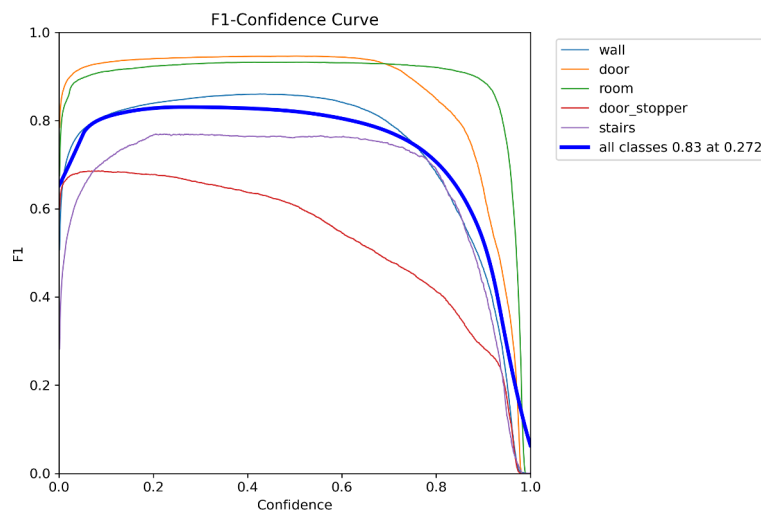


Figure 4.7 Per-class F1-confidence curves for YOLOv11-m.

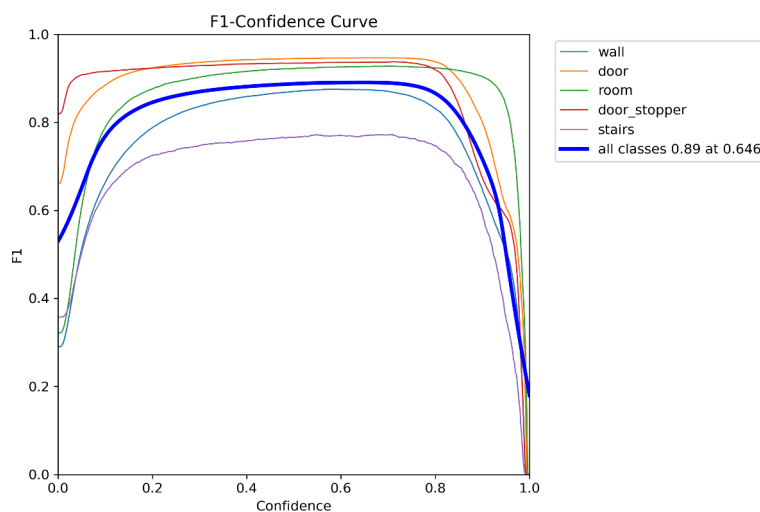


Figure 4.8 Per-class F1-confidence curves for RT-DETR-L.

The F1 curves back this up. Peak F1 on stairs comes out about the same for both models, around 0.77. On door stoppers, RT-DETR-L’s peak of 0.94 dwarfs YOLOv11-m’s 0.69. The bigger difference is in how each peak is reached. YOLOv11-m hits its global maximum F1 at a confidence threshold of just 0.272 and falls off quickly on either side of it, whereas RT-DETR-L peaks at 0.89 around a confidence of 0.646 and stays high across the whole 0.3-to-0.8 range. That wide plateau means the exact threshold chosen at inference time matters far less for RT-DETR-L, one less parameter to get wrong at deployment.

4.5.2 Confusion Matrices

Figure 4.9 and Figure 4.10 show the normalized confusion matrices on the validation split for YOLOv11-m and RT-DETR-L, respectively, with ground truth along the columns and the model’s predictions along the rows. The shaded diagonal gives each class’s recall; everything off the diagonal is a systematic misclassification.

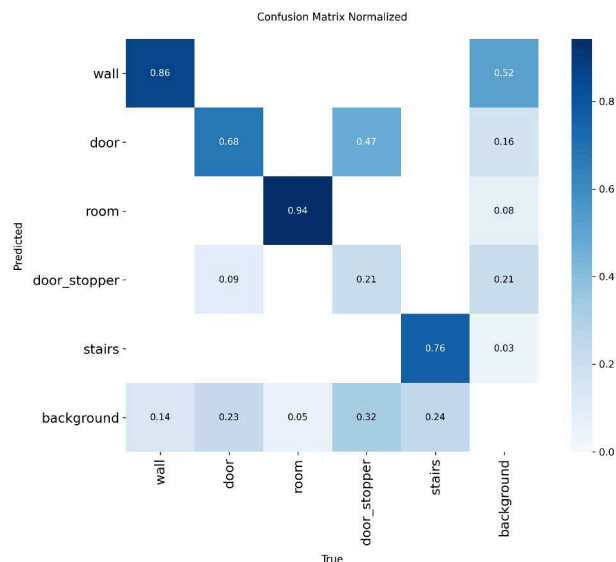


Figure 4.9 Normalized confusion matrix on the validation split for YOLOv11-m. Columns represent ground truth; rows represent predictions.

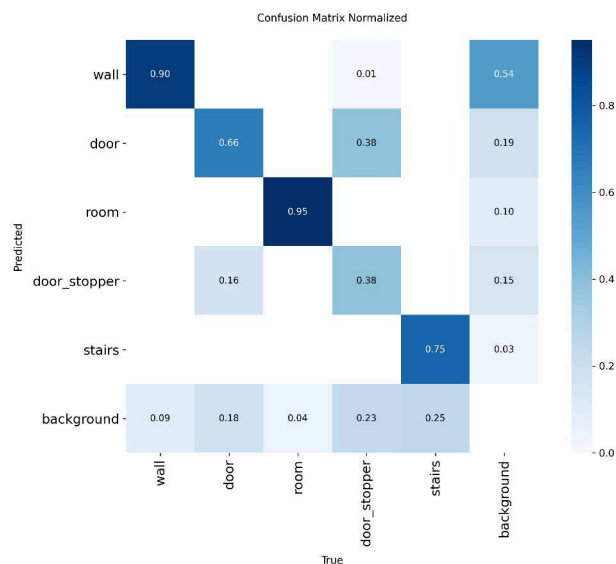


Figure 4.10 Normalized confusion matrix on the validation split for RT-DETR-L. Columns represent ground truth; rows represent predictions.

Two patterns stand out for YOLOv11-m. The worst is a mix-up between doors and door stoppers, with 47% of ground truth door stoppers labeled as doors. That is not surprising given what a door stopper actually is here: a thin threshold line sitting right at the base of the door arc, tightly coupled to a feature the model already has to detect, and a purely local view of the image struggles to tell the two apart. The bottom row of the matrix shows a second, separate problem: outright misses. 32% of door stoppers and 24% of staircases get missed entirely and fall into the background class.

RT-DETR-L cuts that door-stopper confusion from 47% down to 38%. It also misses fewer objects overall: undetected walls fall from 14% to 9% and missed doors from 23% to 18%, with door stoppers improving most, from 32% down to 23%. Stairs are the one exception, where the miss rate actually creeps up slightly, to 25%, matching the drop in precision and recall on that class noted earlier.

4.6 Qualitative Observations

Testing both models on real smartphone photographs of Sapienza floor plans surfaced a few qualitative differences that the metrics above do not capture. These observations are necessarily informal, since there is no pixel-level ground truth for the photographs. They were still enough to settle the choice of RT-DETR-L for the final pipeline.

The clearest difference showed up on text. These floor plans carry a lot of it, room numbers, safety instructions, printed dimensions, scattered across the page. YOLOv11-m regularly mistook blocks of this text for thin wall segments, drawing false walls straight through rooms. RT-DETR-L essentially never made that mistake. The likely reason is that a small dark region against white paper, sitting inside a room’s own polygon, only reads as text once it is seen in relation to the rest of the floor plan; a detector working from local patches has no way to make that connection, which is presumably why YOLOv11-m gets it wrong.

A related issue came from evacuation arrows, the thick dark lines that run from room to room pointing toward an exit. YOLOv11-m often read one of these as a room boundary and split a single room into two separate polygons; RT-DETR-L was much less prone to this. One explanation is that an arrow’s continuity across several rooms only becomes visible to a model that looks at the whole floor plan at once, and that continuity is exactly what marks it as an annotation rather than a wall.

Door detection showed a smaller, steadier gap. YOLOv11-m’s door boxes were sometimes offset from the actual wall line, as though the model had found the right general area without pinning down the exact opening. RT-DETR-L’s boxes sat closer to the true wall gap, which carried through into cleaner polygons after post-processing. This is the mAP@50-95 gap from earlier showing up in a concrete, visible way, in the shape of the generated polygon itself.

Chapter 5

System Architecture and Human-in-the-Loop Refinement

This chapter describes the full-stack system that turns a trained model into a usable tool for creating navigation-grade floor plan maps. At its core is a FastAPI backend that runs the model and applies geometric post-processing, with a decoupled OCR service reading room codes from the polygon regions. Their combined output lands in a SvelteKit frontend editor, where a human operator corrects the geometry and fills in the semantic detail the model cannot supply. Figure 5.1 shows how the three parts fit together.

5.1 Backend Architecture

The backend is a Python web application built with FastAPI⁽²³⁾ and served by Uvicorn, an ASGI server¹. It exposes two endpoints: one runs the full inference pipeline on an uploaded image, the other extracts room codes from polygon regions using optical character recognition. Both are protected by OIDC authentication via Keycloak, with RS256-signed JWTs validated against a JWKS endpoint cached at startup.

5.1.1 Application Lifecycle

Heavy resources load once at application startup and remain in memory for the lifetime of the process. Ultralytics pulls the model weights, several hundred megabytes of them, into GPU memory, and the OIDC provider fetches the JWKS from Keycloak and caches it locally. PaddleOCR is the exception: its detection and recognition models sit behind a thread-safe lazy singleton and are not loaded until the first OCR request arrives, after which they persist like everything else.

¹Designed to succeed the synchronous WSGI standard, the Asynchronous Server Gateway Interface (ASGI) provides a protocol for asynchronous communication between Python web servers and applications. This non-blocking architecture allows systems to handle long-lived connections, such as WebSockets, alongside standard HTTP requests.

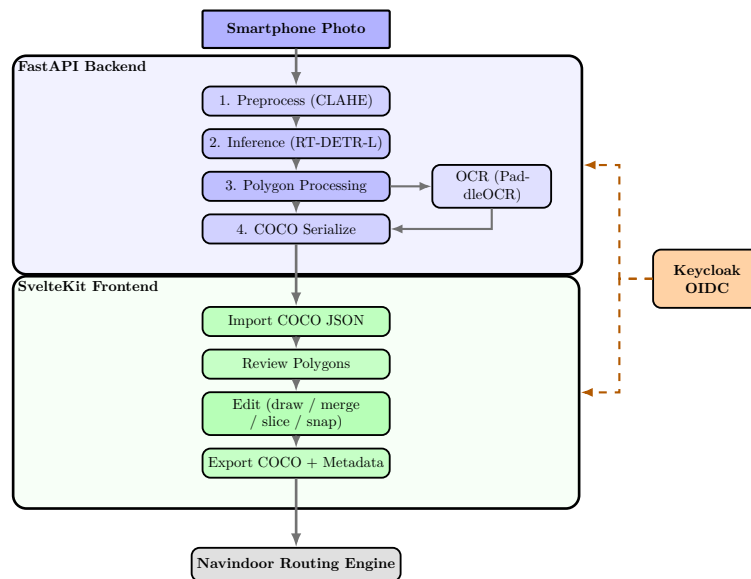


Figure 5.1 High-level system architecture. The user uploads a floor plan photograph to the FastAPI backend, where it passes through preprocessing (CLAHE), RT-DETR-L inference, polygon post-processing, and COCO serialization. A decoupled OCR service extracts room codes from polygon crops. The SvelteKit frontend loads the COCO output, provides an interactive editor for review and correction, and exports enriched annotations to the Navindoor routing engine. All endpoints are protected by Keycloak OIDC authentication (dashed arrows).

The startup sequence is orchestrated by a FastAPI `lifespan` context manager. If any of the three resources fails to initialize, the application refuses to start rather than accepting requests it cannot serve. A `/ready` endpoint reports whether the model has been loaded, providing a health check that a container orchestrator can use to decide when to route traffic to the instance.

5.1.2 Inference Pipeline

The inference pipeline decomposes the processing of a single uploaded image into four sequential steps, each implemented as a class conforming to a `PipelineStep` interface. Every step exposes a single `execute(ctx)` method that receives and returns a `PipelineContext` dataclass. The context object holds all intermediate state: the raw file bytes, the decoded image, the raw model output, the post-processed polygon groups, and the final COCO JSON.

Step 1: Preprocessing

OpenCV decodes the uploaded image from its compressed format (JPEG or PNG) into a BGR NumPy array. If the image's pixel count exceeds 4096×4096 , it is resized with aspect ratio preservation to fit within this ceiling, which prevents out-of-memory errors on the GPU. Downscaling uses area interpolation (`cv2.INTER_AREA`).

Contrast normalization follows. The image moves from BGR into the CIELAB color space, where the L channel carries perceptual luminance independently of the a and

b chrominance channels, and Contrast Limited Adaptive Histogram Equalization (CLAHE)⁽³⁰⁾ is applied to that channel alone with a clip limit of 2.0 and a tile grid of 8×8 before the image is converted back to BGR. On a photographed floor plan the effect is to compress the dynamic range of uneven indoor lighting without softening the sharp luminance transitions at wall edges, which leaves those boundaries more distinct for the detector. Figure 5.2 shows the effect on a corridor photograph.

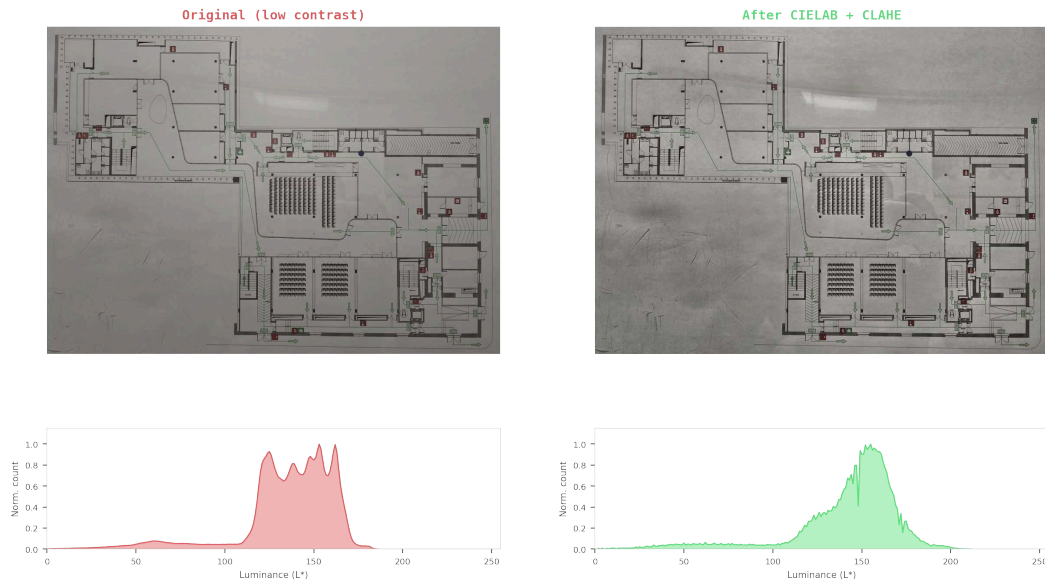


Figure 5.2 CLAHE contrast normalization. Left: original BGR image with uneven corridor lighting. Right: after CLAHE on the L channel, wall edges are sharper and the dynamic range is compressed.

Step 2: Inference

`model.predict()` takes the preprocessed image and internally resizes it to 1024×1024 with letterboxing, normalizes pixel values to $[0, 1]$, and converts it to a PyTorch tensor of shape $(1, 3, 1024, 1024)$ in channels-first layout. A forward pass then returns a **Results** object holding bounding boxes, confidence scores, and class labels for every detected object.

The confidence and IoU thresholds are configurable per request. The default confidence threshold of 0.25 is deliberately low: I prefer the pipeline to produce false positives that the human editor can delete over false negatives that leave rooms undetected. RT-DETR-L is trained with one-to-one matching and needs no non-maximum suppression, so only the confidence threshold applies; overlapping boxes of the same class are merged later by the unary union step.

Asynchronous Execution

Model inference is a blocking operation that holds the GPU. To prevent a single slow inference from blocking all other requests, the `analyze_image` method runs in

a thread pool via FastAPI's `run_in_threadpool`. Additional requests queue and are served in FIFO order.

Step 3: Polygon Processing

Raw bounding boxes from the detector are still a long way from a usable floor plan, and this step applies five transformations to turn them into clean, non-overlapping polygon annotations.

Bounding box to polygon. Each detected bounding box (x_1, y_1, x_2, y_2) is converted to a Shapely⁽⁹⁾ rectangle polygon. Boxes below the confidence threshold are discarded.

Unary union merging. Overlapping polygons of the same class are merged with Shapely's `unary_union` operation. Walls are the main beneficiary: a single long wall segment is often detected as several overlapping bounding boxes, each covering a portion of the wall's extent. The unary union collapses these partial detections into a single contiguous polygon. Invalid geometries (self-intersections, zero-area rings) are repaired with a zero-distance `buffer(0)` before merging. Figure 5.3 shows a wall before and after the union.

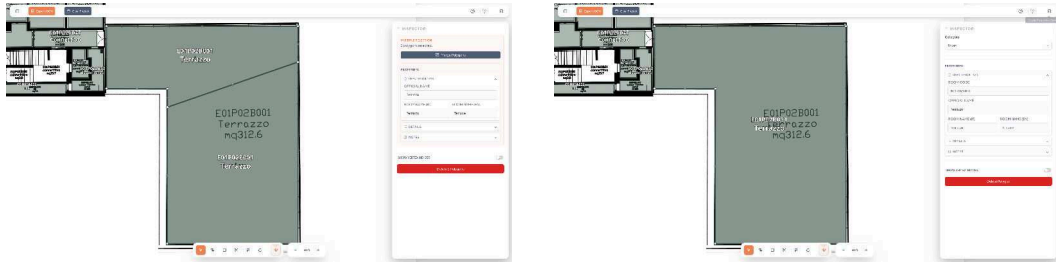


Figure 5.3 Effect of unary union merging on wall detections. Left: raw bounding boxes, many overlapping. Right: after merging, walls are contiguous polygons.

Flood-fill room segmentation. Room detections from the model are loose bounding boxes that include portions of adjacent corridors and neighboring rooms. To produce tight room polygons that conform to wall boundaries, I use a flood-fill approach guided by distance transform.

The obstacle mask is built by rasterizing all wall and door bounding boxes into a binary image, then applying morphological dilation with a 3×3 rectangular kernel for two iterations. The dilation seals small gaps at wall junctions where the detector may have left a one-pixel opening that would otherwise leak the flood fill into adjacent spaces.

For each detected room box, a safe seed point is identified using the distance transform of the inverted obstacle mask within that box. The seed is the pixel furthest from any wall, computed as:

$$D(p) = \min_{w \in \text{obstacles}} \|p - w\|_2 \quad (5.1)$$

and the seed is $\arg \max_p D(p)$. Using this seed, a flood-fill operation expands outward until it hits either the obstacle mask or the boundaries of regions already filled by previous rooms. The flood-fill uses a fixed marker value in a separate mask image (initialized to ones, set to zero where obstacles exist) and operates on the grayscale version of the original image with zero tolerance on both the low and high difference thresholds, meaning the fill stops only at explicit boundaries.

The filled region is extracted via `cv2.findContours` (external contours only), and the largest contour by area is taken as the room boundary. The contour is simplified with `cv2.approxPolyDP` using an epsilon value of 0.002 times the contour perimeter:

$$\varepsilon = 0.002 \cdot P(\text{contour}) \quad (5.2)$$

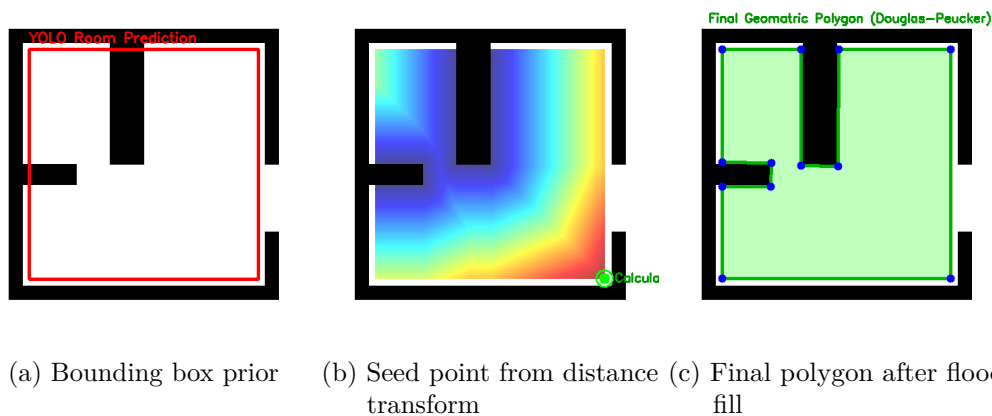


Figure 5.4 Flood-fill room segmentation pipeline. (a) The detector’s bounding box provides a spatial prior. (b) Distance transform identifies the pixel furthest from walls as the seed point. (c) Flood fill produces a polygon that conforms to wall boundaries.

This two-stage approach, detection for spatial priors and classical flood-fill for boundary conformity, avoids the wavy, pixelated boundaries that a pure segmentation model like Mask R-CNN produces, whose output is tied to the pixel grid it was predicted on. Deriving the boundary instead from a fill that stops at the walls themselves gives a cleaner geometry to start with and leaves less work for the simplification stage.

Simplification and orthogonalization. Each polygon is simplified using Shapely’s `simplify` with per-class tolerances. Walls use $\varepsilon = 5.0$ px (aggressive), doors $\varepsilon = 1.0$ (conservative), rooms $\varepsilon = 3.0$. The `preserve_topology` flag prevents self-intersections.

After simplification, an orthogonalization step snaps near-right-angle edges to exact horizontals and verticals. For each pair of consecutive vertices, the edge angle is computed; if the angle differs from 0° , 90° , 180° , or 270° by less than 10° , the edge is snapped to the nearest cardinal direction by adjusting the line equation. Vertex positions are recomputed as the intersections of the snapped edge lines, producing

clean orthogonal geometry that matches architectural expectations. Figure 5.5 illustrates the snap on a single corner.

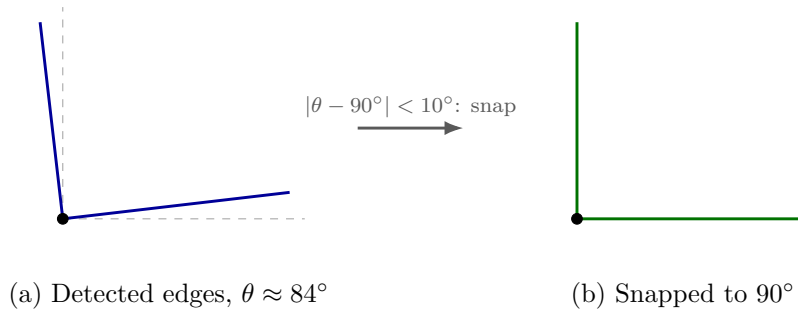


Figure 5.5 Orthogonalization of a polygon corner. (a) The raw edge angle recovered after simplification, here 84° , differs from the nearest cardinal direction by less than the 10° threshold. (b) The edge lines are rotated to exactly 0° and 90° and the vertex is recomputed as their intersection, producing an architecturally plausible right angle.

Morphological closing for stairs. Staircase symbols contain parallel step lines separated by white gaps, and the detector reads those gaps as separations, producing fragmented bounding boxes. A morphological closing operation with a buffer distance of 20 pixels (applied as a positive buffer followed by a negative buffer) fills the gaps and merges the fragments into a single polygon representing the staircase footprint.

Step 4: COCO Serialization

The final polygon groups are serialized to COCO segmentation format. For each polygon, exterior and interior rings are extracted as flat coordinate arrays. The door stopper class is excluded from the output, since the stopper is an annotation artifact. Each annotation includes the segmentation rings, a bounding box, the polygon area, a confidence score of 1.0 (since post-processing does not produce confidence estimates), and a visual centroid computed via Shapely’s `polylabel` algorithm, which finds the point inside a polygon furthest from its boundary.

5.1.3 OCR Service

Room codes (e.g., “RM001P01L001”) are printed as text inside the room polygons on the floor plan. Extracting them automatically reduces the manual labeling burden on the human editor. The OCR service is exposed as a separate endpoint, `/ocr-annotate`, which receives the original image and the COCO JSON annotations produced by the inference pipeline, and returns the extracted room code for each room polygon.

I use PaddleOCR⁽²¹⁾, specifically the PP-OCRv4 pipeline, the successor of the system described by Du et al.⁽⁷⁾. The recognition model is the Latin-script model, which covers Italian text; PP-OCRv4 does not ship a language-specific model for Italian as such. The engine is initialized lazily and shared across requests via a singleton protected by a `threading.Lock`. Batch inference is used to process multiple room crops in a single forward pass.

The per-polygon extraction pipeline has five stages:

1. **Bounding and padding.** For each room polygon, the combined bounding rectangle of all rings is computed with 5-pixel padding. If the crop would extend beyond the image boundaries, it is clamped to the image dimensions.
2. **Binary masking.** A mask is created where pixels inside the exterior ring are set to 255 and pixels inside interior rings (holes) are set to 0. The original image crop is bitwise-ANDed with this mask and composited onto a white background, ensuring that text from neighboring rooms, which may fall inside the bounding rectangle but outside the polygon, does not leak into the OCR input.
3. **Cross-bleed deletion.** An inverse mask is applied to the white background, blanking any residual dark pixels at the boundary between the extracted room and the white fill.
4. **Bicubic upscaling.** If either dimension of the crop is below 60 pixels, the entire crop is upscaled by a factor of 2 using bicubic interpolation. Small text on a floor plan, particularly room codes in 8-point font on an A3 sheet photographed at moderate distance, can be as small as 15 to 20 pixels tall. Without upscaling, PaddleOCR’s text detection module often fails to locate these tiny text regions.
5. **Top-to-bottom sorting.** PaddleOCR returns detected text regions in arbitrary order. I sort them by their vertical midpoint so that multi-line room labels (room code on line 1, room name on line 2) are read in the correct sequence. The first word is assigned as the room code; the second word, if present, becomes the room name and official name fields in the annotation.

5.2 Frontend Architecture: The Interactive Editor

On the client side, the editor is a single-page application built with SvelteKit 5⁽¹¹⁾, using the Runes reactivity system (`$state`, `$derived`, `$effect`) introduced in Svelte 5. It presents the automatically extracted floor plan as an interactive SVG overlay on the original image, with tools for correcting the geometry and attaching semantic labels to it, plus an export path for the finished result.

5.2.1 Core State Management

The central abstraction is the `EditorStore`, a facade class that aggregates four state slices:

- **DataState:** the ground truth. Polygons (each with id, category, rings, room metadata), categories, and the current image.
- **ViewState:** the viewport. Zoom level, pan offset, snap toggle, grid visibility, and panel visibility.
- **InteractionState:** the current session. Active tool mode (pointer, draw, draw_rect, pan, slice, link_doors), selected polygon IDs, clipboard, drawing

points for in-progress shapes, and building/floor identifiers.

- **HistoryState:** undo/redo. A linear history stack storing deep clones of the polygon list at each save point, plus a future stack for redo.

The **EditorStore** exposes approximately 70 getter/setter pairs and 20 action methods. The actions delegate to specialized modules for drawing, selection, merging, import/export, clipboard operations, and undo/redo history. Splitting the actions this way keeps each module small enough to test on its own, and components still have a single object to import.

5.2.2 Rendering

The floor plan is rendered as SVG rather than as a canvas bitmap. SVG elements (primarily `<polygon>`, `<polyline>`, and `<circle>`) are native DOM nodes that the browser compositor handles efficiently, and they receive CSS hover and click events directly, which removes the need to implement hit-testing by projecting mouse coordinates onto a flat pixel buffer.

The viewport is controlled by a CSS `transform` applied to the SVG group element:

```
transform: translate(panX px, panY px) scale(zoom)
```

Zoom is implemented via mouse wheel events, clamped to the range `[0.1, 5.0]`. Pan is implemented via mouse drag when the active tool is set to **pan**. For all other tools, mouse drag performs tool-specific actions (select, draw, move vertex) rather than panning.

A background grid is rendered as thin horizontal and vertical lines whose spacing adapts to the zoom level, implemented as a `<pattern>` with an SVG `<rect>`. Vertex indices, when enabled, are rendered as small numbered circles at each polygon vertex, displayed only when the zoom level is high enough for the numbers to be legible.

5.2.3 Interaction Managers

Two manager classes handle all user interaction on the canvas.

DrawManager orchestrates the active drawing session. When the user selects the polygon draw tool and clicks on the canvas, each click adds a vertex to the current drawing path. The manager provides live preview: a dashed line from the last placed vertex to the current cursor position updates on every mouse move. Snapping is integrated into the drawing flow: when the user's cursor comes within a threshold distance (15 pixels divided by the current zoom level) of an existing vertex or edge, the preview point snaps to that reference.

If the user clicks near the first vertex of the polygon (within the snap threshold), the manager closes the polygon automatically and calls `finishDraw()`, which creates a new polygon in the data store and triggers a state save for undo/redo. The rectangle draw mode uses a drag-out interaction: the user clicks to set one corner, drags to set the opposite corner, and releases to commit a four-vertex polygon.

ManipulationManager handles interactions with existing polygons. It supports four operations:

- **Vertex dragging:** clicking and dragging an individual vertex moves it to a new position, with visual snap guides appearing when the vertex aligns horizontally or vertically with any other vertex in the scene.
- **Midpoint insertion:** clicking on the midpoint handle of a polygon edge splits that edge, inserting a new vertex at the click position.
- **Edge dragging:** clicking and dragging an entire edge translates both its endpoints by the same offset, preserving the edge length and orientation.
- **Polygon translation:** dragging a selected polygon moves all its vertices by the cursor delta, with the option to snap the polygon's bounds to alignment guides.

All manipulation operations are wrapped in `saveState()` calls so they are fully undoable.

5.2.4 Geometric Operations

Snapping is implemented in the `snap.ts` utility. For a given cursor position and a set of reference points (typically the vertices of all visible polygons), the utility computes the nearest reference point within the snap threshold on each axis independently. If the cursor is within the threshold on the x-axis, the x-coordinate snaps to the reference x; similarly for y. This axis-independent snapping is essential for maintaining the orthogonal geometry of floor plans.

Slicing splits a polygon along a user-drawn cut line. The operation uses the polygon-clipping library, which implements the Greiner-Hormann algorithm for Boolean operations on polygons. The cut line is first expanded into a thin rectangle. Intersecting that rectangle with the target polygon serves as a validity check, since a usable cut has to cross the polygon boundary at least twice. Subtracting the rectangle from the polygon then yields the resulting fragments, which replace the original in the data store.

Merging combines two or more selected polygons into one. It uses Shapely's `unary_union` equivalent on the frontend: the union of the polygon rings is computed via polygon-clipping's `union` operation, and the result replaces the selected polygons. This is typically used when the detector has split a room that should be a single polygon.

5.2.5 Semantic Editing

A right-side properties panel is activated when a polygon is selected. The fields displayed depend on the polygon's category:

- **Rooms:** room code, room name (in Italian and English), official name, room type (classroom, laboratory, office, bathroom, corridor), capacity, accessibility notes.

- **Doors:** door number, door type (standard, sliding, revolving, automatic), width, accessibility flag, and a target room code field that links the door to the room it opens into.
- **Stairs:** stair type (straight, L-shaped, U-shaped, spiral), room codes for the upper and lower floors connected by the staircase.

Room codes are rendered as text at the polygon’s visual centroid, computed as the center of mass if it falls inside the polygon, or via polylabel (the pole of inaccessibility) for concave shapes like L-shaped corridors.

5.2.6 Door-Room Linking

A specific interaction mode, `link_doors`, lets the operator visually connect doors to rooms. The operator selects a room and one or more doors, then triggers the link action. The link is stored as a `target_room_code` field on the door polygon, and an animated SVG line is rendered from each linked door to its room centroid, drawn as a dashed path that follows a cubic Bézier curve rather than a straight line to avoid overlapping with other annotations. Figure 5.6 shows the links rendered over a floor plan.

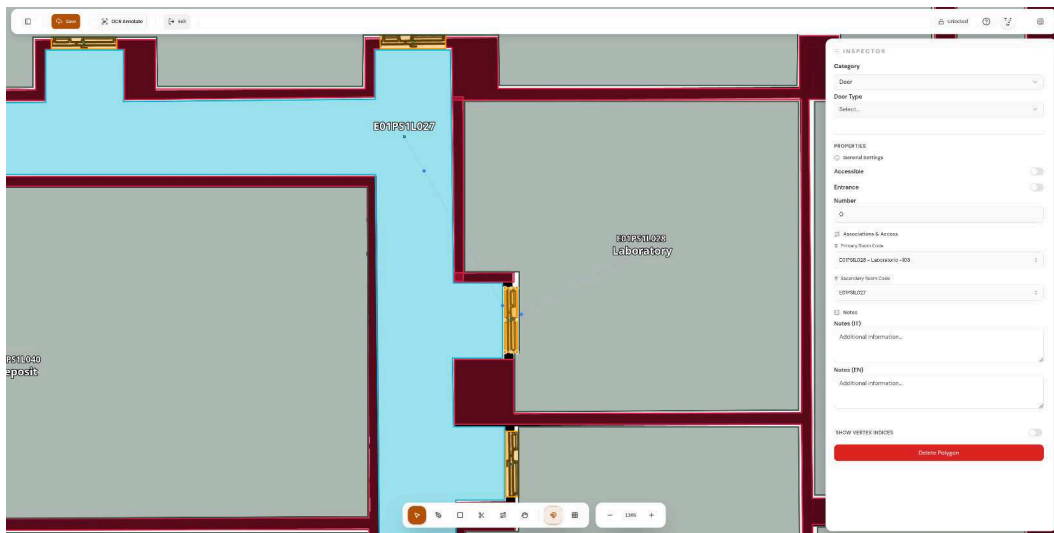


Figure 5.6 Door-room linking visualization. Dashed curves connect doors to their target room centroids. The properties panel (right) shows editable fields for the selected element.

5.2.7 Undo/Redo

The history system captures a deep clone of the polygon list (and, for semantic edits, the relevant metadata fields) before every state-changing operation. The clone is stored as a JSON-serializable snapshot and pushed onto a linear history stack. Undoing pops the current state onto the future stack and restores the previous snapshot; redoing does the reverse. The history is capped at 50 entries to bound memory usage. Operations that would push beyond the cap discard the oldest entry.

5.2.8 Import and Export

The editor can import predictions from two sources: directly from the backend API response (COCO JSON), or from a previously saved local file. On import, the COCO segmentation arrays are parsed into ring structures (arrays of $\{x, y\}$ point objects), and each annotation becomes a `PolygonData` entry with default empty metadata fields.

Export produces a COCO-formatted JSON string that includes all annotations, enriched with the metadata fields added during editing (room codes, door types, accessibility flags). The export is compatible with the backend's OCR endpoint, which supports a round trip: detection first, OCR over its output, then review and correction of the combined result, with the final enriched annotations exported for ingestion by the Navindoor routing engine. Figure 5.7 shows the complete editor interface.



Figure 5.7 The main editor interface. Left panel: category legend with visibility toggles. Center: SVG canvas with the floor plan image and polygon overlay. Right panel: properties for the selected element.

5.2.9 Authentication and Routing

The frontend uses `Auth.js`⁽³⁾ with the `Keycloak`⁽¹⁵⁾ provider for OIDC authentication. A server-side hook redirects unauthenticated users to a login page. The access token is attached to API requests via an `Authorization` header and validated by the backend's OIDC provider. In development mode, a debug token can be used to bypass authentication.

The application supports creating new extractions from scratch and editing previously saved floor plans. Floor plans are organized hierarchically: entities (organizations or campuses) contain buildings, and buildings contain floor plans, each identified by a floor number. The backend API for CRUD operations on entities, buildings, and floor plans is separate from the AI inference endpoints and manages the persistent

storage of finalized maps.

5.3 Deployment

Both the backend and frontend are containerized with Docker using multi-stage builds. The backend Docker image uses a `python:3.12-slim-bookworm` base and pre-downloads PaddleOCR models during the build phase to avoid runtime delays. The frontend uses the SvelteKit Node adapter and is built as a standalone Node.js server. The CI/CD pipeline², defined in GitLab CI, runs a Ruff⁽²⁾ linter and the tests (pytest) on every commit, builds Docker images on the main branch.

²Continuous Integration and Continuous Delivery: the practice of automatically building, testing and releasing every change to the codebase through an automated pipeline.

Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

This thesis set out to solve a practical problem: turning a smartphone photograph of an emergency evacuation floor plan into a structured digital map that an indoor navigation system can route on. The problem sits at the intersection of computer vision, computational geometry, and web engineering, and the solution I built spans all three.

Chapter 2 identified two gaps in the existing literature. The first is methodological: no published work compares a modern real-time detection transformer against a convolutional baseline on this input modality, with the same data and the same training conditions. The second is practical: published work evaluates detection or segmentation accuracy in isolation, whereas a deployed navigation system needs a pipeline that runs end to end and a workflow for correcting whatever the model gets wrong. The four contributions below address the first gap partially, in the sense explained shortly, and the second one in full.

The first contribution is the dataset engineering and the comparative evaluation of detection architectures that it enabled. Building the training set required a full SVG transform parser that recovers absolute pixel coordinates by accumulating transformation matrices from each leaf element to the root, a mapping from CubiCasa5K’s eighty-odd vector classes down to the five that Navindoor needs, and a dual-domain rasterization scheme that pairs each vector-annotated plan with a clean and a scan-like rendering, doubling the effective dataset from roughly 5000 to roughly 10000 images at no annotation cost. Rare classes were oversampled threefold. On the resulting dataset I trained YOLOv11-m and RT-DETR-L under matched conditions and compared them on detection accuracy, per-class behavior, and qualitative output on real smartphone photographs. RT-DETR-L reaches a mAP@50 of 0.894 against 0.870 for YOLOv11-m, and the margin nearly doubles on mAP@50-95, where it stands at 0.045 (0.764 against 0.719). The stricter metric is the one that rewards precise box placement, so the shape of the gap already indicates that the transformer’s advantage lies in localization rather than in finding more objects.

That headline figure needs one qualification, and it is worth stating plainly. Almost all of the mAP@50 advantage comes from a single auxiliary class. On door stoppers RT-DETR-L scores 0.948 against 0.735, a difference of 0.213 that dwarfs every other per-class gap in the table, and door stoppers are excluded from the exported annotations: the class is trained only because forcing the model to separate the threshold line from the door arc improves door detection. On the classes that actually reach the output, the two models are close at IoU 0.5, and YOLOv11-m is marginally ahead on doors (0.956 against 0.949) and rooms (0.954 against 0.940), with a wider lead on stairs (0.821 against 0.740).

The case for RT-DETR-L therefore rests on the properties that matter once detections have to become polygons, and there the evidence is consistent across several independent measurements. Localization is more accurate, which is what the mAP@50-95 margin quantifies. Fewer objects are missed outright: the fraction classified as background falls from 14% to 9% for walls, from 23% to 18% for doors, and from 32% to 23% for door stoppers. Training is also better behaved, with the RT-DETR validation regression losses flat from epoch 50 through to epoch 133, while YOLOv11-m’s box and DFL losses bottom out around epoch 35 and climb for the remaining 80 epochs of its run. A fourth difference shows up at deployment time: RT-DETR-L holds close to its peak F1 across confidence values from roughly 0.3 to 0.8, whereas YOLOv11-m peaks sharply at 0.272 and degrades on either side, which makes the inference threshold far less delicate to choose. On real Sapienza photographs the same ranking held: RT-DETR-L avoided most of the systematic errors that affected the convolutional baseline, namely text blocks read as thin wall segments, evacuation arrows read as room boundaries, and door boxes offset from the wall line they belong to. The cost is training time, 45.2 hours against 21.8 on a single RTX 3090 Ti, and a much weaker start, with a first-epoch recall of 0.308 against 0.701. Neither matters for a model trained once and then deployed for inference, and in any case RT-DETR-L is ahead on recall by epoch 6 and on both mAP metrics by epoch 12.

One auxiliary experiment belongs here as a negative result. I trained an image-to-image denoiser, a UNet with an ImageNet-pretrained ResNet34 encoder supervised by a combined L1 and SSIM objective, on the hypothesis that restoring degraded photographs before detection would narrow the domain gap. RT-DETR-L turned out to absorb most of the photographic noise on its own, and the denoiser earns no place in the deployed pipeline. It remains available for deployments where input quality is worse than anything encountered here.

The second contribution is the geometric post-processing pipeline that converts raw bounding box detections into clean polygonal annotations. Detections first become Shapely rectangles, overlapping boxes of the same class are collapsed by unary union, and room boxes are replaced by a distance-transform-guided flood fill that conforms to wall boundaries rather than to the rectangular prior the detector inherits from a residential training set. What comes out is then simplified with class-adaptive Douglas-Peucker tolerances and orthogonalized to enforce architectural constraints, and staircase fragments are rejoined by morphological closing. The result is a COCO-formatted JSON document with non-overlapping polygons, ready for ingestion by a

routing engine or for review by a human operator.

The third contribution is the decoupled OCR module. Using PaddleOCR’s PP-OCRv4 Latin-script recognition model, which covers Italian text, the service extracts room codes from cropped polygon regions through a five-stage pipeline that masks out neighboring rooms, upscales small text, and sorts detected words by vertical position. The module is architecturally independent from the detection pipeline: it can be called separately, allowing a workflow where the operator runs detection, reviews the geometric output, and then runs OCR selectively on the polygons that need labeling.

The fourth contribution is the full-stack web application. Its backend, built with FastAPI, serves the model asynchronously with GPU concurrency control, OIDC authentication, and a modular pipeline architecture that makes individual processing stages replaceable. On the client side, a SvelteKit 5 editor presents the result as an interactive SVG overlay, with polygon drawing, vertex and edge manipulation, geometric snapping, Boolean slicing and merging, undo/redo history, room code assignment, and visual door-room linking. Taken together, these tools let a human operator turn an approximately correct automatic extraction into a navigation-grade map by correcting it rather than by redrawing it, which is the step the rest of the pipeline exists to make cheap. The editor is not a convenience feature. As Chapter 1 argues, a user who follows spoken directions because they cannot see the building has no way to notice that the router sent them through a door that does not exist, so somebody has to verify the map before it reaches them. How much time that verification actually costs is not something this thesis measures: no formal timing of operator correction was conducted, so the effort reduction claimed in Chapter 1 rests on informal experience during development.

Against the methodological gap from Chapter 2, then, the answer this thesis provides is partial by construction. The controlled comparison runs on CubiCasa5K, not on the smartphone photographs the gap was defined around, because those photographs have no pixel-level ground truth to score against. What the target modality contributes is qualitative evidence that the ranking survives the domain shift. That is enough to justify the engineering decision to deploy RT-DETR-L, and not enough to close the question the way a labeled photographic benchmark would.

6.2 Limitations

The work has several limitations that define the scope of its applicability and suggest directions for improvement.

Domain gap between evaluation and application. The quantitative results in Chapter 4 are measured on a CubiCasa5K split that shares the visual style and data distribution of the training set. The intended application, smartphone photographs of emergency evacuation plans, is a different domain with noise, perspective distortion, and annotation clutter that CubiCasa5K does not capture. The dual-domain rasterization narrows the gap by exposing the model to a scan-like rendering alongside the clean one. A rendering that approximates a scan is still not a photograph

taken at an angle under fluorescent light. Section 4.6 suggests that the relative advantage of RT-DETR-L over YOLOv11-m transfers to the target domain, though real photographs lack pixel-level ground truth, so the absolute performance on Sapienza plans remains unmeasured. A small set of manually annotated smartphone photographs, even 20 to 30 images, would provide a direct estimate of the model’s performance on the actual deployment input and is a necessary next step.

Single-run training without variance estimates. Both models were trained once, from a fixed seed. The comparison in Chapter 4 reports a mAP@50 gap of 0.024 and a mAP@50-95 gap of 0.045 between RT-DETR-L and YOLOv11-m; without multiple runs at different random seeds, though, the variance of these gaps is unknown. Two observations make the ranking credible even so. The gap holds on both metrics and, after both models converge, stays essentially constant for the rest of the run rather than drifting, and it is corroborated by measurements that do not derive from mAP at all, namely the background miss rates in the confusion matrices and the shape of the validation loss curves. Training three to five runs per model would still be needed to attach confidence intervals to the numbers. That was not feasible here: a single pair of runs already costs 21.8 plus 45.2 hours on one RTX 3090 Ti, so three to five repetitions would amount to roughly 200 to 335 hours of GPU time.

Stairs are the weakest class of the deployed model. Choosing RT-DETR-L means accepting a regression on staircases specifically. Its AP@50 on stairs is 0.740 against 0.821 for YOLOv11-m, it is the one class where the background miss rate does not improve (25% against 24%), and its precision on stairs collapses past roughly 0.75 recall. This is the result that resisted the most effort: images containing stairs were oversampled threefold, yet the class still trails. Part of the explanation is the heterogeneity of the symbols, which range from parallel step lines to diagonal arrows to segmented polygons, and which a locally fitted convolutional detector reproduces more closely on a split drawn from the same dataset. Part of it may be the class definition itself, since the mapping in Chapter 3 folds elevators into the same category as stairs, so a single label spans two visually unrelated symbol families.

Dataset domain is residential. The training data is CubiCasa5K, a dataset of Finnish residential floor plans. The model has never seen a lecture hall, a laboratory, or a split-level corridor. It performs well on the structural elements that generalize across building types, reaching 0.889 AP@50 on walls, 0.949 on doors, and 0.940 on rooms. What it learned alongside them is the residential symbol vocabulary. University drawings use conventions that fall outside it: elevator shafts marked with a cross pattern, ramps indicated by diagonal hatching, security doors carrying specialized icons. Extending the training set with labeled examples from the target domain would close this gap, at the cost of the annotation effort this project was designed to avoid.

Curved walls are not handled. Architectural plans with non-orthogonal elements, common in older Sapienza buildings, are simplified to polygonal approximations by the orthogonalization step, which snaps any edge lying within 10° of a cardinal direction and may therefore produce boundaries that diverge visibly from the true

curved wall. Disabling orthogonalization preserves the curve but at the cost of less architecturally clean output for the majority of plans where walls are orthogonal. A more sophisticated approach would apply orthogonalization selectively based on a local curvature estimate.

Perspective distortion is not corrected. Chapter 1 lists it as the first challenge of the problem, yet the pipeline does not rectify it: preprocessing is limited to resizing and CLAHE, and perspective augmentation was excluded during training. Mild distortion is absorbed by the detector and partly undone by orthogonalization, but edges tilted beyond the 10° threshold are left as they are. A rectification step, even a semi-manual one where the operator marks the four corners of the sheet in the editor, would address this directly.

6.3 Future Work

6.3.1 Self-Supervised Pretraining for Floor Plans

The PhD proposal that extends this work identifies self-supervised learning as a promising direction for reducing the dependence on labeled floor plan data. The idea is to pretrain a vision transformer on a large corpus of unlabeled floor plan images using masked image modeling (MAE): random patches of the input image are masked out, and the model learns to reconstruct the missing regions from the visible context. Floor plans suit this approach well. Most of the image is white background, so the informative signal is sparse and concentrated in the strokes; those strokes are highly structured, with walls forming continuous lines and rooms enclosed as bounded regions; and the whole drawing obeys geometric constraints that a reconstruction objective can exploit, since edges are predominantly orthogonal and walls predominantly parallel. The appeal is also economic: labeled floor plans are scarce, and the dual-domain trick used here doubles the dataset only because both renderings inherit the same vector annotations, which does not generalize to plans that arrive as photographs with no annotations at all.

The hypothesis is that an MAE-pretrained encoder would develop attention patterns that track wall continuity and room boundaries without being explicitly taught to do so. These representations could then be fine-tuned on a small number of labeled examples, potentially matching or exceeding the performance of a fully supervised model trained on the full CubiCasa5K dataset. If that held, the practical consequence would be a shorter path to each new deployment: extending Navindoor to a new building type or a new drawing convention would require a handful of annotated examples rather than a new labeled dataset.

6.3.2 Geometric Loss Functions

A related direction is incorporating geometric priors directly into the training objective. Current detection losses (L1, GIoU, Varifocal Loss) treat all pixels equally. A geometrically informed loss could penalize predictions that violate architectural constraints: walls that are not straight, doors misaligned with the wall they sit in, or

corners that depart from a right angle where the rest of the plan is orthogonal. Hakert and Schönfelder⁽¹⁰⁾ have shown that adding parallelism and orthogonality terms to the loss produces more architecturally plausible segmentation output. Adapting this idea to an object detection framework would require defining differentiable geometric losses that operate on the predicted bounding boxes rather than on pixel-level masks. It would also move a constraint that is currently applied after the fact, in the orthogonalization step of the post-processing pipeline, into the model itself, where it could shape the predictions instead of repairing them.

6.3.3 Vectorization as an Alternative to Detection

The detection-plus-flood-fill approach works well when walls are thick, dark lines on a light background. On some floor plans, wall representation is thinner and the gap at door openings is only a few pixels wide. An alternative approach is direct vectorization: treat the floor plan as a line drawing and apply skeletonization to extract the centerlines of walls, then reconstruct the wall polygons by buffering the skeleton. This approach has the advantage of producing topologically connected wall graphs (every wall segment is linked to its neighbors at junctions), which the detection approach can miss when walls are fragmented. It also revisits an idea the early literature relied on, since the rule-based systems of the 2000s reconstructed rooms from exactly this kind of wall graph, and it inherits the weakness that made those systems unsuitable here: sensitivity to noise. A skeletonization algorithm on a noisy photograph will produce spurious branches for every dark region that is not actually a wall. A hybrid approach that uses the detector’s wall proposals to constrain the skeletonization, pruning branches that do not correspond to a detected wall, could combine the topological correctness of skeletonization with the noise robustness of learned detection.

6.3.4 End-to-End Routing Integration

The final step in the pipeline, not addressed in this thesis, is the conversion of the enriched COCO annotations into a format consumable by the Navindoor routing engine. The routing engine needs a graph representation where nodes are rooms and edges are doorways, with associated costs for distance, accessibility, and user preference. Much of the material for that graph is already produced by the editor, since the door-room linking mode records a `target_room_code` on each door polygon, which is precisely the room-to-room adjacency a routing graph is built from. Building this conversion layer would also make the system measurable end to end, in the way Tesseract⁽¹⁾ reports navigational completeness over the graphs it generates: the fraction of room pairs a router can actually connect is the metric that matters for the people Navindoor is built for, and it is the one number this thesis cannot yet report. Establishing it, together with the latency from photograph to route, is the necessary step before deployment on the Sapienza campus.

Bibliography

- [1] Yaqoob Ansari, Ammar Karkour, Eduardo Feo Flushing, and Khaled A. Harras. Tesseract: Unfolding navigable graph representations from low-semantic floor plans. In *Proceedings of the 33rd ACM International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '25, page 569–579, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400720864. doi: 10.1145/3748636.3762771. URL <https://doi.org/10.1145/3748636.3762771>.
- [2] Astral Software Inc. Ruff: An extremely fast Python linter and code formatter. <https://docs.astral.sh/ruff/>, 2026.
- [3] Auth.js Contributors. Auth.js. <https://authjs.dev/>, 2023.
- [4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers, 2020. URL <https://arxiv.org/abs/2005.12872>.
- [5] Philippe Dosch, Karl Tombre, Christian Ah-Soon, and Gérald Masini. A complete system for the analysis of architectural drawings. *IJDAR*, 3:102–116, 12 2000. doi: 10.1007/PL00010901.
- [6] David H. Douglas and Thomas K. Peucker. *Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature*, chapter 2, pages 15–28. John Wiley & Sons, Ltd, 2011. ISBN 9780470669488. doi: <https://doi.org/10.1002/9780470669488.ch2>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470669488.ch2>.
- [7] Yuning Du, Chenxia Li, Ruoyu Guo, Xiaoting Yin, Weiwei Liu, Jun Zhou, Yifan Bai, Zilin Yu, Yehua Yang, Qingqing Dang, and Haoshuang Wang. Pp-ocr: A practical ultra lightweight ocr system, 2020. URL <https://arxiv.org/abs/2009.09941>.
- [8] Maarten Fokkinga. The hough transform. *Journal of Functional Programming*, 21:129–133, 02 2011. doi: 10.1017/S0956796810000341.
- [9] Sean Gillies et al. Shapely: Manipulation and analysis of geometric objects. <https://github.com/shapely/shapely>, 2007.
- [10] Alexander Hakert and Phillip Schönfelder. Informed machine learning methods for instance segmentation of architectural floor plans. 09 2022.

- [11] Rich Harris et al. Svelte. <https://svelte.dev/>, 2016.
- [12] Mohab Hassaan, Philip Alexander Ott, Ann-Kristin Dugstad, Miguel A. Vega Torres, and André Borrmann. Emergency floor plan digitization using machine learning. *Sensors*, 23(19), 2023. ISSN 1424-8220. doi: 10.3390/s23198344. URL <https://www.mdpi.com/1424-8220/23/19/8344>.
- [13] Ahti Kalervo, Juha Ylioinas, Markus Häikiö, Antti Karhu, and Juho Kannala. Cubicasa5k: A dataset and an improved multi-task model for floorplan image analysis, 2019. URL <https://arxiv.org/abs/1904.01920>.
- [14] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics, 2018. URL <https://arxiv.org/abs/1705.07115>.
- [15] Keycloak Project (Red Hat). Keycloak: Open source identity and access management. <https://www.keycloak.org/>, 2016.
- [16] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements, 2024. URL <https://arxiv.org/abs/2410.17725>.
- [17] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. URL <https://arxiv.org/abs/1405.0312>.
- [18] Sébastien Macé, Hervé Locteau, Ernest Valveny, and Salvatore Tabbone. A system to detect rooms in architectural floor plan images. pages 167–174, 06 2010. doi: 10.1145/1815330.1815352.
- [19] Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation, 2016. URL <https://arxiv.org/abs/1603.06937>.
- [20] Hugo Norrby, Gabriel Färm, Kevin Hernandez-Diaz, and Fernando Alonso-Fernandez. Fgssnet: Feature-guided semantic segmentation of real world floor-plans, 2025. URL <https://arxiv.org/abs/2507.10343>.
- [21] PaddlePaddle Contributors. PaddleOCR: Awesome multilingual OCR toolkit based on PaddlePaddle. <https://github.com/PaddlePaddle/PaddleOCR>, 2023. PP-OCRv4 documentation.
- [22] Dmitriy Parashchuk, Alexey Kaspshitskiy, and Yuriy Karyakin. Enhancing floor plan recognition: A hybrid mix-transformer and u-net approach for precise wall segmentation, 2026. URL <https://arxiv.org/abs/2512.02413>.
- [23] Sebastián Ramírez. FastAPI. <https://fastapi.tiangolo.com/>, 2018.
- [24] Smplrspace. Floor plan digitization services. <https://www.smlrspace.com/platform/digitization-services#digitization-services>, 2026.
- [25] Ultralytics. Ultralytics YOLO format documentation. <https://docs.ultralytics.com/datasets/detect/>, 2024.

- [26] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861.
- [27] Hao Zhang, Feng Li, Shilong Liu, Lei Zhang, Hang Su, Jun Zhu, Lionel M. Ni, and Heung-Yeung Shum. Dino: Detr with improved denoising anchor boxes for end-to-end object detection, 2022. URL <https://arxiv.org/abs/2203.03605>.
- [28] Yian Zhao, Wenyu Lv, Shangliang Xu, Jinman Wei, Guanzhong Wang, Qingqing Dang, Yi Liu, and Jie Chen. Detrs beat yolos on real-time object detection, 2024. URL <https://arxiv.org/abs/2304.08069>.
- [29] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection, 2021. URL <https://arxiv.org/abs/2010.04159>.
- [30] Karel Zuiderveld. *Contrast limited adaptive histogram equalization*, page 474–485. Academic Press Professional, Inc., USA, 1994. ISBN 0123361559.