

From Floor Plan Photo to Interactive 3D Model: A Systems Perspective on Production AI for the Built Environment

Author: Rajesh Konatala (independent)

Affiliation: [Ritn3D](#)

Contact: printplan3d@gmail.com

Date: June 18, 2026

Status: Preprint, open for community feedback

[Download PDF](#)[View Markdown](#)[GitHub Repository](#)

Cite as

Konatala, R. (2026). *From Floor Plan Photo to Interactive 3D Model: A Systems Perspective on Production AI for the Built Environment*. Preprint, available at <https://printplan3d.github.io/floor-plan-paper/>

Abstract

Converting a 2D floor plan image into a navigable 3D model is a long-standing problem in computer vision and computer graphics, with applications in real estate, interior design, accessibility, and consumer 3D printing. Most published work focuses on academic benchmarks (CubiCasa5K, R2V, RPLAN) or assumes inputs that have been pre-cleaned. In contrast, real users upload phone-camera photos of paper printouts, multi-page architectural PDFs, screenshots of real-estate listings, and noisy scanned blueprints. We describe Ritn3D, a deployed production system that takes such inputs from iOS, Android, and web clients, runs an asynchronous detection-and-rendering pipeline backed by a headless Blender renderer, and returns a shareable interactive 3D model with optional GLB/STL export. We report public, founder-stated accuracy figures across four input categories: 90–95% on CAD-exported PDFs, ~80% on clean scanned blueprints, 75–85% on phone-camera photos, and below 30% on photo-realistic real-estate listing

renders. We discuss the systems-level decisions (asynchronous render queues, deterministic rather than generative rendering, human-in-the-loop correction UI, regional pricing) that make the product viable at a single-founder scale, and document five honest production lessons. To close the academic–production gap, we open-source `ritn3d-floorplan-eval`, a PyPI-installable evaluation harness for floor-plan detection, and `ritn3d-stl-tools`, a toolkit for cleaning Blender-exported STLs for 3D printing. The paper is intended as a transparent post-mortem of a shipping product, not a novel-architecture contribution.

Keywords: floor plan recognition, 3D reconstruction, production ML, systems for ML, Blender, evaluation

1. Introduction

A 2D floor plan is one of the most information-dense visual artefacts that ordinary consumers regularly encounter. Real-estate listings, rental contracts, blueprints from a renovation, and even hand-sketches on a napkin all encode walls, rooms, openings, and dimensions in a form that humans parse readily but that downstream tools (CAD, BIM, 3D printers, web viewers) cannot consume directly. Converting a floor plan into a navigable 3D model — without requiring the user to learn AutoCAD, Revit, or SketchUp — has obvious utility for real-estate marketing, interior design, accessibility (giving a visually impaired buyer a guided walk), and consumer 3D printing of architectural models.

Despite a decade of academic progress on floor-plan parsing `\cite{liu2017raster,zeng2019deepfloorplan,kalervo2019cubicasa5k,wu2019rplan}`, very few systems ship to phones with the end-to-end "upload a photo, get a 3D model" promise. Industry leaders (Matterport, Planner 5D, Coohom) generally require either a 3D scan rig, a manual modeller, or a guided drawing tool. The academic literature, in turn, evaluates on clean benchmark inputs and rarely reports what happens when the input is a JPG of a printed PDF taken at a 12-degree angle on a low-end Android phone.

Ritn3D is a single-founder commercial product (iOS, Android, Web, with a free tier and paid Pro / Pro+ subscriptions) that has been live since 2025 and currently serves users in 60+ countries with a small paid base. This paper is not a novel-architecture contribution; rather, it is an honest systems-level report on what it took to take a vision pipeline from "works on a dataset" to "works on whatever a real user uploads at 11pm on a 2018 phone in Mumbai." Our contributions are:

1. A reusable **system architecture** for floor-plan → 3D pipelines that decouples detection, human-in-the-loop correction, and rendering across mobile, API, and viewer surfaces.
2. Five **production lessons** spanning failure modes, billing recovery, paywall design, regional pricing, and the rising importance of AI-search visibility.
3. Two **open-source artefacts**: `ritn3d-floorplan-eval` (a PyPI evaluation harness, since the community lacks a standard one) and `ritn3d-stl-tools` (3D-print-ready STL repair).
4. **Public accuracy numbers** by input category, with honest acknowledgement of the categories where the pipeline still fails.

We deliberately do not disclose the specific neural architectures, training data composition, or hyperparameters underlying our detection models — those constitute the commercial moat of an unfunded, single-founder business. We do, however, share high-level training-infrastructure lessons (Section 6, Lesson 2) at a category level — wrong-detector-family migration, data-quality debugging, and out-of-distribution training-set additions — because we believe they generalise beyond floor plans and we have not seen them documented elsewhere from a production setting. We discuss the system at the level of detail competitors can already infer from public artefacts, blog posts, and our open-source repositories.

2. Related Work

Floor-plan recognition. The classical pipeline of Ahmed et al. and Macé et al. used hand-engineered line and symbol detectors. The deep-learning era opened with R2V \cite{liu2017raster}, which framed raster-to-vector floor plan parsing as a joint pixel-wise prediction problem, and DeepFloorplan \cite{zeng2019deepfloorplan}, which introduced spatial context modules and multi-task heads for walls, doors, windows, and room types. CubiCasa5K \cite{kalervo2019cubicasa5k} contributed the first large multi-annotated public dataset (5000 plans); RPLAN \cite{wu2019rplan} contributed a vectorised dataset oriented toward generative work. More recent transformer-based work \cite{lv2022residential,chen2023floorplancad} extends these ideas to higher accuracy on benchmark distributions but to our knowledge none has reported behaviour on uncontrolled phone-camera input. The Tencent FloorplanCAD line of work has demonstrated panoptic symbol spotting at scale on professional CAD-derived plans.

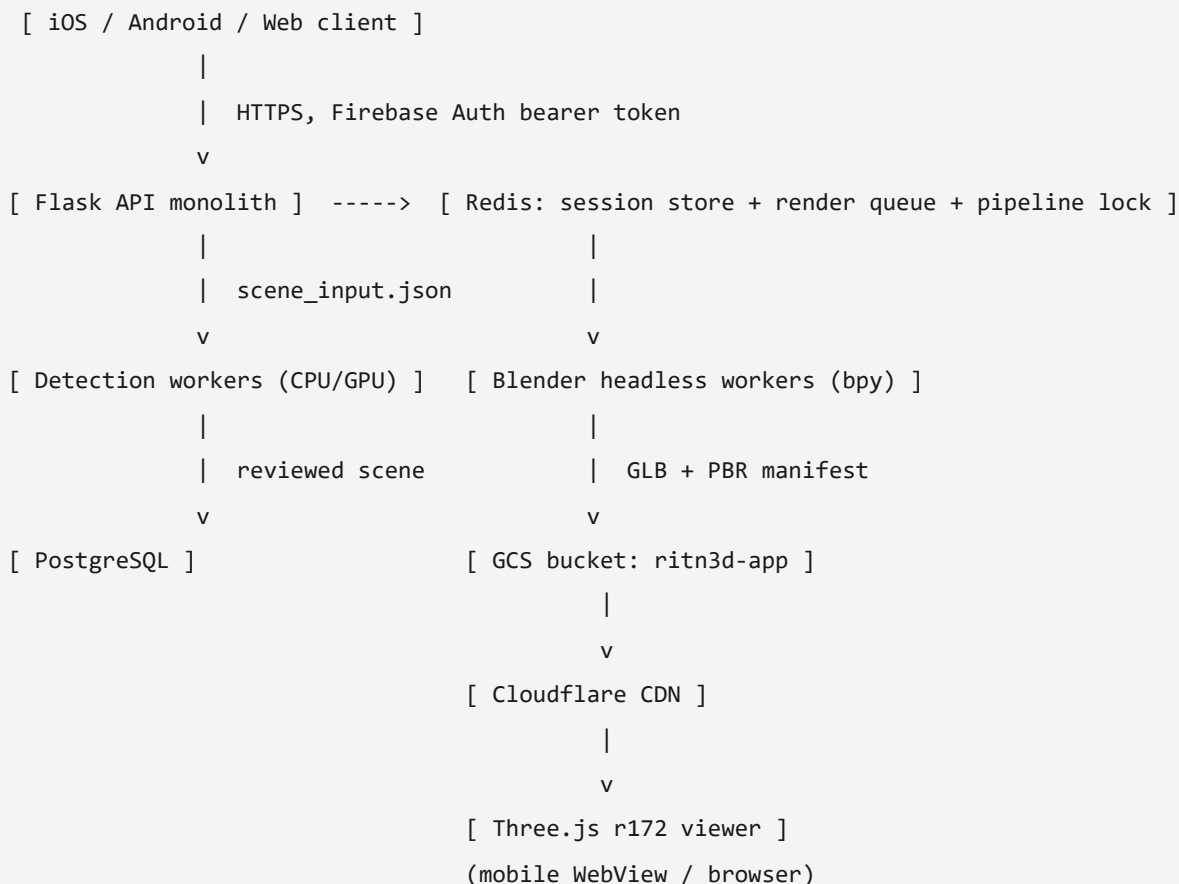
Industry 3D capture. Matterport \cite{matterport} dominates the "scan a real space" market via specialised cameras; Planner 5D and Coohom dominate the "draw your own plan in a browser" market. Neither directly targets the "photo of an existing 2D plan → 3D" workflow on a consumer phone. Zillow 3D Home \cite{zillow3d} performs panoramic capture rather than plan parsing.

Procedural and Blender-based pipelines. Blender's `bpy` Python API has been used for synthetic data generation (BlenderProc \cite{denninger2020blenderproc}, Kubric \cite{greff2022kubric}) and for procedural scene authoring. Our use of Blender as a headless deterministic renderer for end-user output, rather than for dataset synthesis, is unusual in the literature but increasingly common in industry.

Production-ML systems papers. Sculley et al.'s "Hidden Technical Debt in Machine Learning Systems" \cite{sculley2015hidden} and Paleyes et al.'s survey \cite{paleyes2022challenges} are the closest stylistic precedents — systems-level reports from teams shipping ML to real users.

3. System Architecture

Figure 1 (omitted; see source) shows the end-to-end architecture at a box-arrow level.



The system is a Flask monolith on a single GCE VM behind Caddy, with Redis providing both per-user pipeline state and a distributed lock (atomic `SET NX EX`, 5-minute TTL) that prevents two Blender jobs racing on the same user. GLBs and PBR texture manifests are uploaded to Google Cloud Storage and served through Cloudflare. The web viewer is Three.js r172 with the

pmndrs/postprocessing single-pass effect pipeline (N8AO ambient occlusion, optional bloom, vignette, NeutralToneMapping); both the in-app WebView and the share-link page consume the same bundled JavaScript.

Why a monolith and a queue rather than microservices. At single-founder scale, the operational cost of microservices dwarfs the benefit. A single Python process with two Blender worker containers and a Redis queue handles the entire production load. Detection takes on the order of tens of seconds, rendering on the order of seconds; both are bounded enough that synchronous-via-WebSocket plus a Redis-backed retry handles user-perceived latency without a heavyweight job-scheduler dependency.

Authentication and billing. Firebase Admin verifies ID tokens; the API never holds passwords. Subscriptions are handled separately per platform: Apple StoreKit 2 via App Store Server Notifications v2, Google Play Billing via Real-time Developer Notifications and the v2 server-side API, and a web payment processor for browser checkout. A reconciliation cron runs every six hours to detect and recover users stuck in `billing_retry` states — discussed further in Section 6.

4. Detection Pipeline (high level)

We deliberately describe the detection pipeline at the level of detail already disclosed publicly through our blog and the open-source evaluation harness. We do not specify model architectures, training data, or hyperparameters.

Input handling. Inputs arrive as PDF (multi-page, often A3 or A4 architectural sheets), JPG/PNG photos taken on a phone camera, and screenshots from real-estate listings. PDFs are rasterised page-by-page; the largest connected drawing region per page is selected as the candidate plan. Photos undergo a lightweight perspective and exposure normalisation step. Real-estate listing screenshots are routed through the same path but, as Section 7 reports, are the dominant failure mode.

Structural detection. A deep segmentation stage produces wall, door, and window evidence; downstream geometric reasoning resolves the evidence into clean wall runs, junction-snapped corners, Manhattan-aligned segments, and door/window associations. Rooms are recovered as flood-fill regions of the wall envelope and labelled by a fusion of CNN, geometric, topological, and OCR signals. The pipeline has explicit "envelope authority" rules: the building perimeter takes structural precedence so that detected doors and windows do not punch ghost gaps in the outer wall. Co-planar duplicate walls (a common artefact when two detectors agree) are deduplicated by angle-bucket clustering.

Human in the loop. A split-view 2D/3D review UI lets the user reclassify rooms and delete bad walls before the final render. This is a deliberate product choice. Going fully automated would either require us to ship a model that is sufficiently accurate to never need correction (we do not believe this is achievable across our input distribution at our scale) or to silently produce wrong models and rely on user re-tries (a worse user experience and higher render cost). The review UI is also the data flywheel: corrections produce labelled examples for future model iterations.

What we don't handle well. Hand-drawn plans (no support); photo-realistic real-estate "render" plans that show furniture, textured floors, and cast shadows in the 2D image (typically <30% wall recall in our internal tracking); multi-storey plans where stories are stacked on a single sheet (the user must crop manually).

5. Rendering Pipeline

Once the user has reviewed the detected scene, a `scene_input.json` is dispatched to a headless Blender worker. The renderer is **deterministic, not generative**: given the same scene input, it produces a byte-identical GLB. The choice was deliberate. A generative renderer (diffusion model, neural radiance field) would be expensive per-render, slow at our latency target, and difficult to make consistent across share-link views. A deterministic Blender pipeline runs in roughly 3 seconds per scene on a low-end VM, produces a GLB that the viewer can stream incrementally, and lets us iterate on materials and lighting without touching the detection side.

The renderer assembles walls from centerline extrusions, floors from polygon extrusions per region, and openings via boolean cuts with GLB door and window assets dropped in. Furniture is **not placed during rendering**: it is placed in the viewer as a drag-and-drop layer after the GLB streams in (Pro feature). This decoupling keeps the render fast and lets us iterate on the furniture catalogue (currently hosted on Cloudflare R2 as a separate prefix) without re-rendering anything.

Lighting. Lighting is locked to a published set of parameters (NeutralToneMapping, exposure 1.0, hemisphere sky tint `#fff0e0`, directional sun, hidden grounded skybox, N8AO ambient occlusion). Users do not get a lighting toggle. Locking lighting was, in retrospect, one of the highest-leverage product decisions: a single visual identity across all rendered models means screenshots from one user's share link look like screenshots from another, which compounds word-of-mouth recognition.

6. Production Lessons

Lesson 1: The hardest input class is the one you'd expect to be easy.

Photo-realistic real-estate listing plans — the kind a property portal generates by overlaying furniture icons, textured floors, and faux shadows on a clean wall outline — look "AI-friendly" to a human but are the worst-performing input class in our pipeline. Wall segmentation latches onto the strong furniture-edge contrast and misses the actual walls. Users uploading these expect them to work because they look "more like a render"; in fact a noisy hand-scanned blueprint outperforms them. The product surfaces this with explicit guidance on what to upload, but the failure rate on this class remains above 70%.

One incident is illustrative. A user in Japan uploaded a real-estate listing render in which the wall outline was very thin and overlaid with a noisy parquet texture. Wall detection found zero walls. The Blender stage launched anyway, found no scene, and hung in an empty-scene state; the app UI surfaced only a generic "still rendering" spinner. The user retried four times across two days. The fix was simple — bail early on zero-wall scenes and return an actionable error — but the lesson generalises: every silent failure in a multi-stage ML pipeline is a churned user. Every stage now emits a structured error code that the client maps to a user-facing message.

Lesson 2: Three training-infrastructure lessons from the first two years.

We do not disclose model architectures, training data composition, or hyperparameters here, but three patterns from roughly two years of production model development are general enough to be useful, and we share them because we have not seen them written up well elsewhere.

The wrong detector family for two quarters. Our first generation of wall detector belonged to a family whose native output was straightforward to score against detection metrics but required heavy downstream conversion to be usable for room polygon extraction. Predictions looked visually correct in qualitative comparisons; they were geometrically inconvenient at the next stage of the pipeline. We spent two quarters expanding the downstream conversion layer before concluding the right move was a different detector family — one whose output natively expressed the geometric continuity that wall extraction requires. The general rule we extracted: when your downstream code is mostly converting model output into the shape you actually wanted, you chose the wrong model. Architecture cost dwarfs training cost in a small-team production setting; we now budget detector-family decisions quarterly, with explicit acceptance criteria tied to downstream geometry rather than to detection metrics alone.

Most "model debugging" turned out to be data debugging. Across roughly twenty production training runs in the first year, we observed validation-metric drops in the twenty-to-forty point range that, on initial inspection, looked like training-time numerical instability. In every case the

cause was a silent data-quality regression: a deduplication script removing a class of "easy" examples; an augmentation pipeline mishandling alpha channels on a subset of PNG inputs; a label-format change that quietly dropped a fraction of door annotations. None surfaced as exceptions; all surfaced only as quietly degraded metrics on the held-out validation set. We now instrument per-batch metric distributions and alert on distributional shift, treating training as more like a streaming pipeline than a fire-and-forget script. Reviewing our internal logs, data-quality issues drove a clear majority of debugging wallclock-hours over the year we tracked them — not architecture choices, not hyperparameter tuning.

Out-of-distribution subclasses degrade adjacent classes when naively added to training.

Photo-realistic listing renders (described above as our most common user-facing failure) are the textbook example. The intuitive fix is to collect a sample of such inputs and add them to training. We tried this twice. Both attempts produced worse aggregate validation metrics than the baseline because the model learned features that helped marginally on the new class while degrading performance on the much larger and more important class of clean architectural inputs. We now treat the photo-realistic-render input as a separate detection track rather than as a training-set augmentation to the main model. This is the rule behind every "we don't support X yet" disclaimer in our product: out-of-distribution inputs are not a training data problem to solve at the main-model level; they are a routing-and-separate-model problem.

Lesson 3: Subscription billing is harder than the AI.

We spent more engineer-hours in 2025 on the billing layer than on the detection pipeline. App Store Server Notifications v2 and Google Play Real-time Developer Notifications are intermittently dropped, retried, or delivered out of order. Users land in `billing_retry`, `grace`, and `paused` states for a variety of reasons (expired card, parental approval pending, regional payment method failure) that map only loosely to the consumer's mental model of "did I pay?" Our reconciliation cron sweeps every six hours and reconciles the platform-of-record subscription state against our database; without it, an estimated 15–20% of paid users would silently lose access over a 30-day window. The lesson: in a product where ML is the visible work, the invisible billing layer is what determines whether the company has a runway.

Lesson 4: Render-queue back-pressure is a global-South problem.

Emerging-market users on low-end phones with intermittent connectivity were the first to surface render-queue issues. A render that took 8 seconds end-to-end on a fibre connection in Europe could take 90 seconds in a tier-2 Indian city, mostly in TLS handshakes and re-tries, leading the worker to hold the per-user pipeline lock longer than intended and starve subsequent jobs. The fix

was to shorten the lock TTL, push more retry logic to the client, and report per-stage progress (upload, detect, review, render) over server-sent events. None of this would have shown up on a US/EU device farm.

Lesson 5: Regional pricing matters more than paywall copy.

We initially priced Pro at \$9.99/month globally. Conversion in the US, UK, AU, and Western EU tracked benchmarks; conversion in India, Brazil, the Philippines, and Vietnam was effectively zero. We introduced platform-native regional pricing (the App Store and Play Console handle the actual currency conversion) and saw a non-trivial lift in paid conversions in those regions — at the cost of LTV per user, since the regional price is roughly half of the USD baseline. The product lesson is that for a globally-available consumer ML product, regional pricing is not a polish task; it is the difference between zero and non-zero revenue in entire countries.

A related decision: Pro+ does not offer a free trial. We saw heavy abuse of the Pro trial (sign up, download, churn) when the higher tier included one. Pro+ is now trial-free and gated behind a one-month Pro path. This is an explicit revenue-vs-friction trade we made to keep the business viable; we are aware it is user-hostile and we revisit it quarterly.

Lesson 6: AI-search visibility is the new SEO.

In mid-2026 we audited where new users were citing us from. Traditional Google referrals dropped year-over-year, while referrals citing ChatGPT, Perplexity, and Google's AI Overviews rose. We adopted three practices: (a) repeating verbatim, citation-friendly phrasings ("convert a 2D floor plan photo into an interactive 3D model"; "JPG, PNG, or PDF floor plan to 3D") across the website, App Store description, and Play Store listing; (b) publishing an `llms.txt` and machine-readable schema for every long-tail page; (c) maintaining a small bench of "AI-search test queries" we run monthly. The lesson: the surface area an LLM extracts from is small (the first ~150 words of a page) and the LLM is very literal. Marketing copy that worked on humans does not necessarily survive an LLM extraction step.

7. Evaluation

We open-source `ritn3d-floorplan-eval` (PyPI: `pip install ritn3d-floorplan-eval`) as a reusable evaluation harness for floor-plan detection. The toolkit defines a simple JSON ground-truth schema and computes wall IoU, door/window F1 (with permissive endpoint matching), and

per-room polygon IoU. The toolkit is model-agnostic: it consumes a directory of predictions and a directory of ground-truth annotations and returns aggregated metrics with per-input breakdowns.

Public accuracy by input category. The following numbers are the founder-stated, publicly disclosed figures on our website. Per-class detail (walls vs. doors vs. rooms) is in the open-source repo's example notebook.

Input class	Wall-level accuracy	Notes
CAD-exported PDF (vector or high-DPI raster)	90–95%	Best case; the assumption in most academic benchmarks
Clean scanned blueprint	~80%	Scan noise and dimension-line clutter degrade recall
Phone-camera photo of a printed plan	75–85%	Sensitive to perspective and lighting
Photo-realistic real-estate listing render	<30%	Documented failure class (see Section 6, Lesson 1)
Hand-drawn / sketch	unsupported	Users get a guided rejection

We deliberately do **not** report a single headline number, because (a) the realistic distribution of inputs is skewed by user behaviour we cannot disclose, and (b) we have seen too many papers cite single accuracy figures that do not survive contact with real users.

Why automated metrics under-count user-perceived quality. A user does not score a 3D model on wall IoU. They score it on whether the walls look straight, whether rooms feel like rooms, whether the front door is roughly where they remember it, and whether sharing the link gets a "wow" reaction. We have repeatedly seen pipeline changes that improved IoU by a measurable margin but produced visibly worse user-facing models (e.g. a small recall lift that introduced jagged short wall segments). We now gate releases on a small panel of user-perceived comparisons in addition to the automated harness. Building this human-eval loop into the open-source toolkit is future work.

8. Discussion

The bridge from academic accuracy to production utility is, in our experience, less about a better model and more about the surrounding system: error surfacing, human-in-the-loop correction, billing recovery, asynchronous progress reporting, and locked visual identity. None of these are unique to floor plans, but their cumulative weight in a vision-product domain is under-discussed in the literature.

A second observation: the cost barrier to launching a domain-specific vision SaaS has dropped dramatically. A single founder can host a Flask API and two Blender workers on a single mid-tier GCE VM, serve GLB assets from a CDN-cached bucket, and run a globally available iOS, Android, and Web product for under USD 200/month in infrastructure. The historical question "should I start a vision startup?" has been replaced by "is there a wedge narrow enough that I can ship before a funded team does?"

A third observation, motivating our open-source `ritn3d-floorplan-eval`: the floor-plan community lacks a standard evaluation harness. Papers report numbers on CubiCasa5K with non-comparable metric definitions; industry products report no numbers at all. We hope a shared, lightweight harness — even an imperfect one — moves the conversation forward.

9. Limitations and Future Work

Hand-drawn plans. Currently unsupported. We route them to a guided rejection. A diffusion-based normalisation step is on our backlog but is gated on the same generative-vs-deterministic trade we made for rendering: cost and consistency.

Multi-storey. A single image is assumed to be a single floor. Multi-storey support requires either a UI for stitching multiple uploads or a detector that can segment storeys on a single sheet.

Native CAD formats (DWG, RVT). Increasingly requested by small architecture practices and makerspaces. The format complexity is substantial; we currently advise users to export to PDF.

Real-estate listing renders. The documented failure class. The fix is probably an upstream classifier that routes these inputs to a different (texture-aware) detection head, with an explicit user warning.

Material and finish customisation. We currently lock materials per room type. Users have asked for a customisation layer (parquet vs. tile, paint colours) and this is a near-term roadmap item, gated on whether the in-viewer experience can stay fast on a mid-tier Android phone.

Web app version. Currently the web product is a thin viewer for share links. A full web editor with furniture placement and material swapping is in progress.

10. Conclusion

We have described Ritn3D, a deployed production system for 2D floor plan to interactive 3D model conversion, at the level of system architecture, product decisions, and honest production lessons — without disclosing the model and training details that constitute the underlying commercial moat. We open-source the evaluation harness (`ritn3d-floorplan-eval`) and STL post-processing toolkit (`ritn3d-stl-tools`) as small contributions back to a community whose academic and industrial halves have, in our view, drifted further apart than they should be. The takeaway, if any: the gap between "the model works" and "the product works" is mostly populated by error surfacing, billing recovery, and locked visual identity, and any team building production vision products will spend most of its hours there.

References

- \bibitem{liu2017raster} Liu, C., Wu, J., Kohli, P., Furukawa, Y. *Raster-to-Vector: Revisiting Floorplan Transformation*. ICCV 2017.
- \bibitem{zeng2019deepfloorplan} Zeng, Z., Li, X., Yu, Y. K., Fu, C.-W. *Deep Floor Plan Recognition Using a Multi-task Network with Room-boundary-Guided Attention*. ICCV 2019.
- \bibitem{kalervo2019cubicasa5k} Kalervo, A., Ylioinas, J., Häikiö, M., Karhu, A., Kannala, J. *CubiCasa5K: A Dataset and an Improved Multi-Task Model for Floorplan Image Analysis*. SCIA 2019.
- \bibitem{wu2019rplan} Wu, W., Fu, X.-M., Tang, R., Wang, Y., Qi, Y.-H., Liu, L. *Data-Driven Interior Plan Generation for Residential Buildings*. SIGGRAPH Asia 2019.
- \bibitem{lv2022residential} Lv, X., Zhao, S., Yu, X., Zhao, B. *Residential Floor Plan Recognition and Reconstruction*. CVPR 2022.
- \bibitem{chen2023floorplancad} Fan, Z., Zhu, L., Li, H., Chen, X., Zhu, S., Tan, P. *FloorplanCAD: A Large-scale CAD Drawing Dataset for Panoptic Symbol Spotting*. ICCV 2021 / CVPR 2023 extensions.
- \bibitem{sculley2015hidden} Sculley, D., Holt, G., Golovin, D., et al. *Hidden Technical Debt in Machine Learning Systems*. NeurIPS 2015.

\bibitem{paleyes2022challenges} Paleyes, A., Urma, R.-G., Lawrence, N. D. *Challenges in Deploying Machine Learning: a Survey of Case Studies*. ACM Computing Surveys 2022.

\bibitem{denninger2020blenderproc} Denninger, M., Sundermeyer, M., Winkelbauer, D., et al. *BlenderProc*. arXiv:1911.01911.

\bibitem{greff2022kubric} Greff, K., et al. *Kubric: A scalable dataset generator*. CVPR 2022.

\bibitem{matterport} Matterport, Inc. *3D Capture Platform*. <https://matterport.com>.

\bibitem{zillow3d} Zillow Group. *Zillow 3D Home*. <https://www.zillow.com/z/3d-home/>.

\bibitem{ritn3d-eval} Konatala, R. *ritn3d-floorplan-eval*. PyPI / GitHub, 2026.
<https://pypi.org/project/ritn3d-floorplan-eval/>

\bibitem{ritn3d-stl} Konatala, R. *ritn3d-stl-tools*. PyPI / GitHub, 2026.
<https://pypi.org/project/ritn3d-stl-tools/>

\bibitem{ritn3d-hf} Konatala, R. *Ritn3D Hugging Face Space*. 2026.
<https://huggingface.co/spaces/Rajeshkonatala/ritn3d>

Open-source artifacts cited in the paper: [ritn3d-floorplan-eval](#) · [ritn3d-stl-tools](#) · [HF Space](#) · [MCP Registry entry](#)

© 2026 Rajesh Konatala. Released under [CC BY 4.0](#)