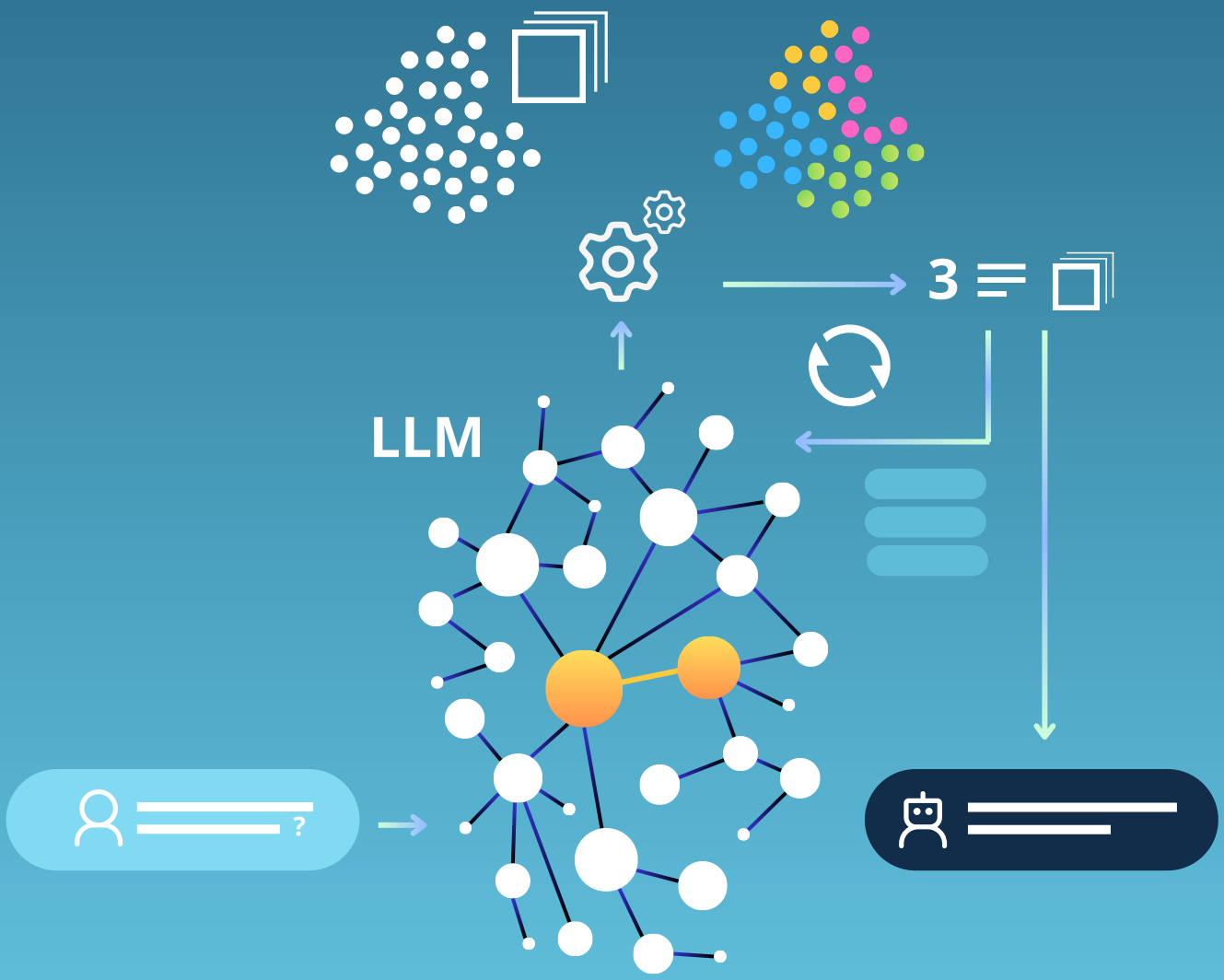
SpatiaLLM

Bridging the Gap Between Natural Language and 3D Scans

Synthesis Project (GEOIT1501)

Mark van der Meer - 6301975
Hongyu Ye - 6286240
Segher ter Braak - 4909682
Julia Pille - 5303117
Neelabh Singh - 6052045



TU Delft

Scanplan
pointclouds to data

Abstract

Recent advances in large language models (LLMs) have expanded natural language reasoning and multimodal understanding but remain limited in grounding with 3D spatial environments. This project addresses that gap by developing a system that enables natural language interaction with indoor spatial data derived from light detection and ranging (LiDAR) point clouds and panoramic imagery provided by the client: ScanPlan. The system processes spatial data through a pipeline that includes room segmentation, geometric analysis, and object clustering. A structured query language lite (SQLite) database stores the structured information, which an AI agent queries using a reasoning framework that translates natural language into actionable commands. The system supports multimodal input, allowing users to interact via text or by selecting objects in 2D panoramas, which are then mapped to 3D point clouds using segment anything model 2 (SAM2). The interface combines a chat function with 2D and 3D viewers, making spatial data accessible to non-experts. While the prototype successfully answers a range of spatial and semantic queries, challenges remain in scaling room segmentation and handling complex multi-room relationships. The project demonstrates a step towards making rich 3D building data queryable through intuitive, language-based interaction.

KEYWORDS: Large Language Models, 3D spatial reasoning, LiDAR point clouds, multimodal interaction, natural language interface.

Acknowledgements

We first and foremost thank Dr. Shayan Nikoohemat from ScanPlan, our client. Shayan's domain expertise in 3D reconstruction from point clouds, together with ScanPlan's perspective on Scan-to-BIM and digital twins, grounded this project from day one. We are grateful for the data and platform access, weekly guidance, and technical support that enables us to test ideas on realistic datasets. Communication was fast and clear; questions were answered quickly, problems were addressed pragmatically, and we consistently felt we had the freedom to explore alternatives and make our own design choices. This guidance kept us focused on what matters in practice: traceability, usability (for non-experts), and evidence-backed answers drawn from panoramic images and point clouds.

We are also grateful to our academic supervisor, Dr. Liangliang Nan, for his steady guidance and consistently clear, fast, and constructive feedback. Liangliang ensured the project ran smoothly, coordinated logistics, and served as our point of contact with Shayan from the beginning. Although he did not steer the technical implementation directly, he was always available for thoughtful ideas, practical advice, and help when we were stuck. He struck a valuable balance: giving us the freedom to explore and learn while providing timely nudges that kept the project moving in the right direction.

We would like to thank the course instructors as well for proposing an interesting and relevant topic and for creating the conditions that allowed us to iterate purposefully.

Finally, we did our very best to create a system that translates the facets of the spatial world into data that is understandable, accessible, and workable for a wide range of users. We had fun working on this project, and we hope you enjoy reading about it.

Sincerely,
Mark, Segher, Julia, Neelabh and Hongyu.

Delft, October 2025

Acronyms

AABB	axis-aligned bounding box
ACI	agent-computer interface
AEC	architecture, engineering and construction
AF	advancing front
AI	artificial intelligence
API	application programming interface
BIC	Bayesian information criterion
BIM	building information modeling
BLP	binary linear program
CGAL	Computational Geometry Algorithms Library
CLI	command-line interface
COCO	common objects in context
CSV	comma-separated values
DB	database
FEC	fast Euclidean clustering
GMM	Gaussian mixture model
GPT	generative pre-trained transformer
GUI	graphical user interface
HTTP	Hypertext Transfer Protocol
ID	identity documentation
JSON	JavaScript Object Notation
k-d tree	k-dimensional tree
***k*-NN**	*k*-nearest neighbors
LiDAR	light detection and ranging
LLM	large language model
MST	minimum spanning tree
NLP	natural language processing
PCL	Point Cloud Library
PDE	partial differential equation
PID	Project Initiation Document
PLY	polygon file format
PNG	Portable Network Graphics
RAG	retrieval-augmented generation
RANSAC	random sample consensus
RGB	red, green, and blue
S3DIS	Stanford Large-Scale 3D Indoor Spaces Dataset
SAM2	segment anything model 2
SAM	segment anything model
SQLite	structured query language lite
UOBB	up-oriented bounding box
URL	uniform resource locator
WebGL	Web Graphics Library
YOLOv11	you only look once, version 11

Contents

1 Introduction

Recent advances in large language models (LLMs) have significantly improved capabilities in communication, reasoning, and 2D image understanding (Hong et al., 2023b). However, a critical gap remains: these models are largely ungrounded in 3D environments, limiting their ability to reason about the spatial relationships that are crucial for real-world tasks like counting objects in a room or measuring areas (Yamada et al., 2024). The next step for advanced artificial intelligence (AI) is to combine images, 3D scenes, and geolocation data to achieve genuine spatial understanding. (Hong et al., 2023b). This challenge is particularly relevant for the architecture, engineering and construction (AEC) sector and in real estate. In these fields, stakeholders—often non-technical users—require quick, reliable insights from rich 3D building data for critical tasks such as renovation planning, cost estimation, and maintenance.

The industry standard for capturing these data is Scan-to-BIM. Building information modeling (BIM) workflows typically use light detection and ranging (LiDAR) to generate detailed, high-density point clouds. These point clouds can be classified into building elements such as walls, floors, and windows. When combined with panoramic images from the scanner and segmentation models, this data feeds high-fidelity digital twins that represent physical spaces. Despite the richness of this information, extracting answers through querying is time-consuming and needs manual processing. This creates a significant accessibility barrier for non-experts and hinders broader applications like faster decision-making and more scalable asset management.

Bridging the gap between data-rich 3D scans and accessible querying is important. Without it, organizations risk underutilizing their expensive digital models, leading to slower project timelines and a higher likelihood of errors. Successfully addressing this gap would democratize advanced spatial data, unlocking benefits such as improved collaboration, cost savings, more efficient facility management, and more effective renovation planning.

This project is conducted in collaboration with ScanPlan, a company based in Zwolle that specializes in storing and classifying point clouds. ScanPlan has developed an application for storing, sharing, and segmenting point clouds, effectively connecting raw scan data to BIM workflows to keep digital models updated with accurate as-built information. Their goal is to shorten the entire Scan-to-BIM process, making digital twins more accurate and 3D assets more efficient to use. Their clients frequently ask practical, answerable questions based on the existing data, such as, "*How much will it cost to paint all the walls in the house?*" Currently, answering these queries relies on slow, costly, and difficult-to-scale manual inspections. ScanPlan's vision is to make their digital twins queryable using plain language, allowing non-experts to receive a traceable, evidence-backed answer drawn directly from panoramic images and point cloud data.

Consequently, the goal of this project is to develop a chatbot that understands natural-language queries and retrieves relevant spatial and visual evidence, thereby bridging human language with spatial-visual reasoning. Specifically, we aim to connect an LLM to a classified point cloud and panoramic images via a spatial AI layer. This will enable users to ask questions and receive correct, auditable results, with the system seamlessly handling multiple input types, including text, image, and location data.

To address the core problem of LLMs' lack of spatial reasoning, this project focuses on four key objectives:

- **Integrate available spatial LLMs with classified input data (images and LiDAR scans).** This foundational step is necessary to move beyond theoretical reasoning and enable the model to analyze real-world 3D environments with multiple objects and complex spatial relationships.
- **Enable the LLM to convert natural language queries into actionable data queries.** This is crucial for ensuring the system is accessible and intuitive, allowing users to interact with complex data using plain language.

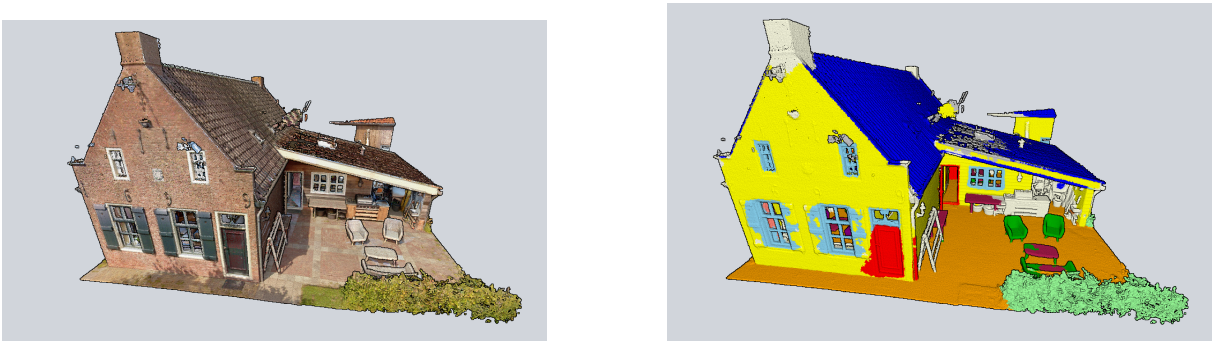
- **Interpret user prompts to extract relevant information from classified point clouds and segmented images.** Even with structured data, the system must intelligently identify what is pertinent to a user's specific question to reduce manual interpretation and deliver accurate, context-specific responses.
- **Support multimodal input handling,** including text, images, and geolocation data, to provide flexibility needed for comprehensive reasoning in complex real-world environments.

The purpose of this report is to provide an authoritative handover that defines the problem, documents methods and results, and outlines next steps for assessment and reuse. The report is structured as follows: Chapter 2 details the methodology and pipeline; Chapter 3 presents the results; Chapter 4 discusses the findings; Chapter 5 offers the final conclusion; and finally, Chapter 6 proposes directions for future work.

2 Methodology

This chapter details the data and pipeline behind our system. This project is based on point clouds recorded with 3D laser scanning. The original .e57 files contain point cloud (Figure 2.1b), the scanner poses, and the corresponding panoramic images. Through their online portal, ScanPlan provides a processing pipeline that segments these point clouds into 30 predefined classes (Figure 2.1a, see Appendix A.1 for all classes). Building on these labeled data, we cluster class-specific points into objects, generate meshes, and organize the results per room and floor. These meshes support consistent geometric properties (e.g., area, volume) and relationships across multiple objects (e.g., wall-opening, door-room). All derived attributes are persisted in a database that the chatbot queries to return traceable, correct answers. User questions are issued through a graphical interface that can also be used to extract visual features from 2D images. The questions asked by the user are presented in a graphical interface, which has the ability to extract visual features from 2D images.

The chapter mirrors the project's end-to-end flow: room segmentation, geometric functions, preprocessing pipeline, database, AI agent, 2D to 3D, and finally the 3D viewer—moving from raw scans to structured 3D understanding to reliable answers grounded in both geometry and imagery.



(a) Data not segmented

(b) Data segmented

Figure 2.1: House dataset

2.1 Room Segmentation

In order for the next chapters to have any effect, a clear distinction between objects and rooms has to take place. Therefore, we created a room segmentation algorithm that takes the point cloud of a single floor of a building and then generates a 2D floor plan per floor. The next subchapters elaborate more on which steps are taken to generate a final room reconstruction.

2.1.1 Height Measurements

The room segmentation generates 2D floor plans per floor. In order to generate the floor plans, the different floors have to be defined. A similar idea to the room segmentation can be used to define the heights, that is, to project all 3D points onto the z-axis and count the number of points per height value. The counting of points is done via a histogram, where all points are put into small bins. Based on the bins, a plot is generated, and the different peaks are shown in a plot. To generate strong peaks, a mask is created that states that the current peak should be bigger than half of the maximum peak height. In Figure 2.2 the resulting plot and peaks are shown. What can be seen is that there are two peaks very close to each other. Since a lot of points do exist at the different floor heights, it is harder to distinguish them. To solve this, we go over the peaks from lowest to highest and set a range of one

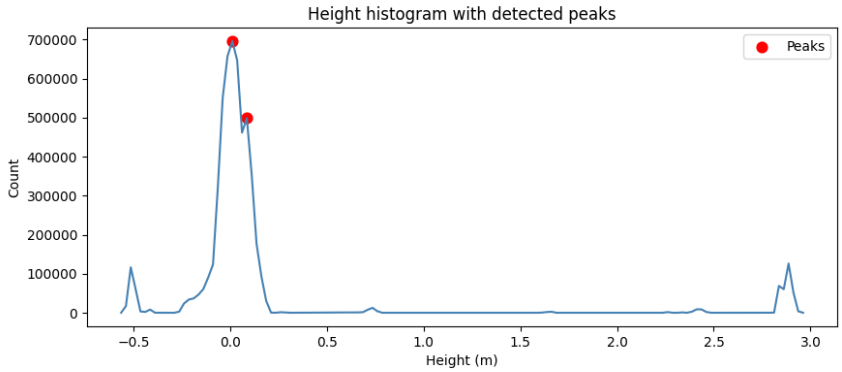


Figure 2.2: Heights for the ground floor (house dataset)

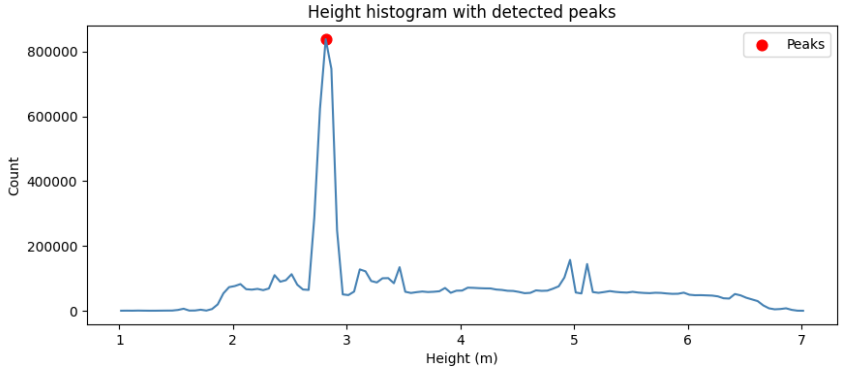


Figure 2.3: Heights for the upper floor (house dataset)

meter. Within this meter all other peaks are taken and then compared to find the highest peak. This highest peak is then used as the floor height.

To get the accurate height measurements, a distinction between the ground floor and upper floors has to be made. For the ground floor, only one peak can be used, since there is only one ground floor, and for upper floors, multiple floors can be used. To generate both heights, different input datasets are used. For the ground floor only the points in the dataset with label `floor` are used, while for the upper floors only points with the label `ceiling` are used. These input datasets are generated in the preprocessing steps of the pipeline, which can be seen in figure 2.7.

2.1.2 Floor Plan

Generating the floor plan involves four steps: first is projecting the 3D points to a 2D grid, second is assigning labels to each room to generate a 2D floor plan, third is expanding the walls of the rooms, and lastly is reprojecting the 2D grid onto the 3D points. Once the final steps have taken place, all the points have been assigned to at least one room.

Before the 3D points can be projected, it is good to mention which 3D points will be projected onto a 2D grid. The results of the height measurements are used to make a slice through a point cloud consisting of all the structural elements, like ceilings, floors, walls, beams, and columns. This slice is done a few centimeters lower than the floor and a few centimeters higher than the ceiling. Slicing the building this way results in a point cloud of each floor, mostly consisting of the walls. Lastly, the floor point clouds are sliced at 20 centimeters above the floor and beneath the ceiling. This last slice leads to a dataset that only includes the wall. Figure 4.1 displays the heights of the height measurements and different slices.

Figure 2.4: Height information of ground floor (house dataset)

It is important that the last slice only includes all the walls and not any of the objects or floor and ceiling, as this will not lead to a correct floor plan. These points that have been generated are downsampled and then projected onto a 2D grid. In Figure 2.5 the left image shows the projection of the points onto the grid with the different colors being the different labels, where the dark blue color indicates the walls. An intermediate step is added that excludes all 'rooms' that are smaller than one square meter in order to remove the noisy rooms; this can be seen in the middle figure. The third step is to expand all the walls of the rooms. This creates the floor plan shown in the rightmost grid of Figure 2.5. The orange and yellow colors indicate that the points in the 2D grid are assigned to multiple labels. The goal is to create a point cloud per room, but for rooms sharing a wall, it is important that in the final point cloud of both rooms the same wall is included. Once the final wall expansion 2D grid is created, all 3D points are taken and are assigned the correct room labels to them. By doing this, the point clouds of different rooms can be generated.

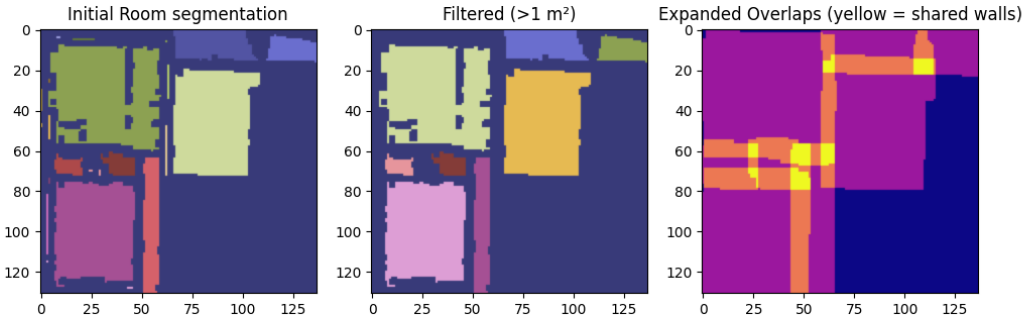


Figure 2.5: Floor plan of ground floor (house dataset)

This structure to generate separate point cloud files for the shells of the rooms can also be followed for all the different objects in the house. A point cloud of the whole house is broken down into only a point cloud with an object label. This object label point cloud is split into multiple floors and then split into multiple rooms. All these steps result in a unique folder per room with the shell and all the objects it contains.

2.1.3 PolyFit Reconstruction

PolyFit (Nan and Wonka, 2017) targets piecewise-planar man-made objects such as architectural elements, indoor walls, and furniture with prominent planes and sharp edges. The core consists of three steps:

1. **Candidate-face generation:** random sample consensus (RANSAC) extracts and iteratively merges plane segments. The cropped planes are then intersected pairwise to produce many candidate faces, while preserving face–edge adjacency needed for manifold/watertight constraints.
2. **Face selection:** For each candidate face f_i, introduce a binary variable $x_i \in \{0,1\}$ and minimize a weighted sum of energies (data fidelity, model complexity, point coverage) under watertight-manifold constraints:

$$\min_X \ \lambda_f E_f + \lambda_m E_m + \lambda_c E_c \quad \text{s.t.} \quad \sum_{j \in N(e)} x_j \in \{0,2\}, \ \ x_i \in \{0,1\}.$$

   This enforces that each edge is either unused or shared by exactly two faces, yielding a closed 2-manifold polygonal model solved as a binary linear program (BLP).
3. **Data term:** Consistency between faces and points is evaluated via confidence-weighted neighborhood support:

$$E_f = 1 - \frac{1}{|P|} \sum_i x_i \, \text{support}(f_i).$$

   The other two terms penalize sharp corners (encouraging coplanar merging) and use an α-shape notion to regulate extrapolation.

Based on this method's principles and properties, PolyFit is used to fit and reconstruct indoor walls, ceilings, and floors. Notably, in practice, when applying this method to non-smooth instances, overly aggressive parameters (e.g., too small `facet_distance`, overly strict `facet_angle`) may drive the surface towards "needle-like" degeneracies where multiple projections map to the same surface point, causing reconstruction failure. Such instances are reconstructed using the method described later.

2.1.4 Plane Extraction Method

The output from Polyfit does not group coplanar surfaces into groups, so the planes had to be extracted again from the output. To do so for each surface, the normal vector $\mathbf{n}$, centroid $\mathbf{c}$, and area A were derived.

For each face, the surface normal was computed as the cross product of two edge vectors:

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}),$$

and normalized as:

$$\hat{\mathbf{n}} = \frac{\mathbf{n}}{\|\mathbf{n}\|}.$$

The centroid and area were determined by decomposing the polygon into triangular elements and summing their contributions.

Two faces were considered coplanar if they satisfied both of the following conditions:

$$\angle(\hat{\mathbf{n}}_i, \hat{\mathbf{n}}_j) \leq \Delta\theta, \quad \text{and} \quad |d_i - d_j| \leq \varepsilon,$$

where $\Delta\theta$ is the maximum allowed angular deviation between normals, ε is a distance tolerance, and d is the scalar offset in the plane equation

$$\hat{\mathbf{n}} \cdot \mathbf{x} + d = 0.$$

All coplanar faces were grouped into planar regions, and for each group the mean normal vector, total area, and centroid were computed as weighted averages based on face area.

Planes were then classified by their inclination relative to the global vertical axis $\mathbf{z} = (0,0,1)$:

- If $\angle(\hat{\mathbf{n}}, \mathbf{z}) \approx 0°$, the plane was classified as a ceiling.
- If $\angle(\hat{\mathbf{n}}, \mathbf{z}) \approx 180°$, as a floor.
- If $\angle(\hat{\mathbf{n}}, \mathbf{z}) \approx 90°$, as a wall.

This mathematical procedure enabled the decomposition of the 3D model into semantically meaningful planar elements to be inserted into the database.

2.2 Geometric Functions

We developed a C++-based geometric pipeline using the Eigen, Point Cloud Library (PCL) (Point Cloud Library contributors, 2025) by Rusu and Cousins, 2011, and Computational Geometry Algorithms Library (CGAL) (The CGAL Project, 2025) by The CGAL Project, 2025 to process raw point clouds into semantic and geometric outputs. The subsequent subsections formalize each component. The flow is:

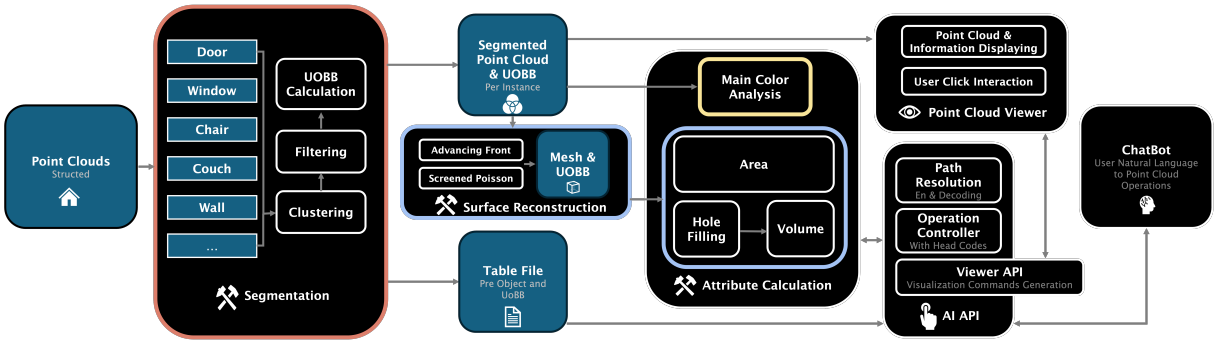


Figure 2.6: The key components, steps, and relations description of the pipeline.

1. **Input organized point cloud.** Point clouds are required to be stored under the `rooms/` folders, which are stored under `data/` in a simple hierarchy; outputs mirror this layout:

```
data/
  rooms/<house>/
    floor_k/                # Different Floors in a House
      rooms_manifest.csv    # Information for all rooms on each floor
      room_001/             # Room folder with .ply files for category on point cloud
      room_002/
output/
  floor_k/
    rooms_manifest.csv
    room_001/
      room_001.csv
      results/
```

   Coordinates are right-handed with $+Z$ up.

2. **Clustering & filtration.** Group the room point cloud into spatial clusters using a radius-based neighbor search; output: a set of cluster indices per room. Then discard small clusters using an average-size threshold; output: filtered clusters representing meaningful parts.

3. **Up-oriented bounding box (UOBB).** For each cluster, compute a vertically oriented bounding box (center, yaw, size, corners); output: metrics (base area, bounding box volume) and optional box mesh.

4. **Surface reconstruction (optional).** When surface-dependent measures are needed, reconstruct a triangle mesh per cluster (Poisson first, fallback to Advancing Front); output: a mesh (closed when possible).

5. **Geometric attribute measures (optional).** With a mesh, report surface area and volume; otherwise provide a convex-hull volume proxy; output: scalar measures complementing UOBB metrics.

6. **Color modeling (optional).** Fit a small Gaussian mixture to RGB to summarize color modes and assign labels if required; output: mixture parameters and per-point labels.

2.2.1 Clustering

Fast Euclidean Clustering

Fast Eclidean Clustering is proposed by Cao et al., 2022. Given a point set $\mathcal{P} = \{\mathbf{p}_i\}_{i=1}^{N} \subset \mathbb{R}^3$, we define an ε-neighborhood graph $G = (V, E)$ with $V = \{1, \ldots, N\}$ and an edge $(i, j) \in E$ iff $\|\mathbf{p}_i - \mathbf{p}_j\|_2 \leq \varepsilon$. Fast Euclidean clustering (FEC) seeks the connected components of G with two practical constraints: (i) a cap $k_{\max}$ on neighbors returned per query to bound local work and suppress extreme densities; and (ii) a minimum component size m to suppress spurious small aggregates from noise.

Operationally, we combine a kd-tree radius query with a single-pass label-propagation that assigns to each queried neighborhood the smallest extant label among already-labeled neighbors, merging labels on the fly. Let $L_i \in \mathbb{N}$ be the cluster label for point i (initialized $L_i = 0$). For $i = 1 \ldots N$, we perform a radius query $\mathcal{N}_\varepsilon(i)$ limited to at most $k_{\max}$ neighbors and set

$$L_j \leftarrow \min\big(\{L_\ell \,|\, \ell \in \mathcal{N}_\varepsilon(i),\ L_\ell > 0\} \cup \{\tau\}\big) \quad \forall j \in \mathcal{N}_\varepsilon(i),$$

where τ is a fresh label if none of the neighbors has been labeled yet; subsequently, any larger temporary labels in the current neighborhood are relabeled to this minimum. This realizes a union-by-minimum strategy for connected components over an implicit geometric graph. The resulting set of components is finally filtered by size, retaining only those with $|C| \geq m$.

Complexity is governed by (a) the kd-tree radius queries, typically $O\big(N \log N + \sum_i |\mathcal{N}_\varepsilon(i)|\big)$ to build and query; and (b) the light-weight label merges local to neighborhoods. In practice, the imposed neighbor cap $k_{\max}$ ensures stable performance in dense regions, while m trades recall for precision in cluttered, noisy scenes.

Edge cases and invariants. Degenerate cases (empty cloud, NaNs) are discarded early. Setting $k_{\max}$ too low may fragment thick structures; setting m too high may remove true but small parts. The method is invariant under rigid motions and robust to moderate density variations due to the neighbor cap.

Cluster Filtration

Given an initial cluster set $\{C_c\}_{c=1}^{M}$, we apply a dynamic threshold based on the average cluster size to prune long-tail fragments while adapting to scene scale. We compute

$$\bar{s} = \frac{1}{M} \sum_{c=1}^{M} |C_c|, \quad T = \eta\, \bar{s}, \quad \eta \in (0, 1], \quad \text{retain } C_c \text{ if } |C_c| \geq T.$$

In our experiments we set $\eta = 0.1$, which removes a variable portion of the smallest components while preserving the bulk of meaningful aggregates under heavy-tailed size distributions.

2.2.2 UOBB Calculation

Definition and Meaning

UOBB is a minimum-area planar rectangle in the horizontal plane (Z-up convention) extruded between the minimum and maximum heights of a point set. Formally, letting Π_{xy} project $\mathbb{R}^3 \to \mathbb{R}^2$, we find the 2D rectangle $R(\theta)$ of smallest area enclosing $\Pi_{xy}(C)$ among all in-plane rotations by yaw θ about the z-axis. The 3D box then has center $\mathbf{c} \in \mathbb{R}^3$, orthonormal axes $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_z)$ with $\mathbf{e}_z = (0, 0, 1)^\top$, side lengths (ℓ_x, ℓ_y, ℓ_z) with $\ell_z = z_{\max} - z_{\min}$, base area $A_b = \ell_x \ell_y$, volume $V = \ell_x \ell_y \ell_z$, and yaw ψ as the rotation from $+X$ to $\mathbf{e}_1$.

Calculation Method

Let the projected point set be $\mathcal{Q} = \{\mathbf{q}_i\} = \Pi_{xy}(C) \subset \mathbb{R}^2$. We compute the convex hull $\text{hull}(\mathcal{Q})$ using Andrew's monotone chain in $O(n \log n)$ time. For each hull edge direction $\mathbf{u}$ (unit), take the orthonormal frame $(\mathbf{u}, \mathbf{v})$ with $\mathbf{v}$ the $\pi/2$ rotation of $\mathbf{u}$, and project hull vertices $u_i = \mathbf{u}^\top \mathbf{q}_i$, $v_i = \mathbf{v}^\top \mathbf{q}_i$. The rectangle aligned with $(\mathbf{u}, \mathbf{v})$ has side lengths and area

$$\ell_u = \max_i u_i - \min_i u_i, \quad \ell_v = \max_i v_i - \min_i v_i, \quad A(\mathbf{u}) = \ell_u \ell_v.$$

Rotating calipers enumerates candidate orientations over hull edges and selects the minimum-area rectangle. Denote the minimizing direction by $\mathbf{u}^\star$. We then enforce a consistent convention by sorting the in-plane sides and adjusting yaw:

$$(\ell_x, \ell_y) = \text{sort\_desc}(\ell_u, \ell_v), \quad \psi = \text{yaw}(\mathbf{u}^\star) + \mathbb{I}[\ell_x \neq \ell_u] \cdot \frac{\pi}{2}, \quad \ell_z = z_{\max} - z_{\min}.$$

The box center is the midpoint of the in-plane rectangle, lifted to $z = (z_{\max} + z_{\min})/2$; the eight corners follow from $(\mathbf{u}^\star, \mathbf{v})$, (ℓ_x, ℓ_y, ℓ_z), and $\mathbf{c}$.

Properties. The method is invariant to translation and to rotations about z, attains the exact minimum area among edge-aligned rectangles of the hull, and runs in $O(n \log n + h)$ time (with h the hull size). Height uses the exact z-range, making the box tight along z under the Z-up assumption.

2.2.3 Surface Reconstruction

Poisson surface reconstruction. Kazhdan and Hoppe, 2013 formulate an implicit indicator function $\chi : \mathbb{R}^3 \to \{0, 1\}$ whose gradient aligns with an input vector field derived from oriented normals. The canonical variational form minimizes

$$E(\chi) = \int_\Omega \|\nabla \chi(\mathbf{x}) - \mathbf{V}(\mathbf{x})\|^2 \, \mathrm{d}\mathbf{x}$$

subject to mild regularization, yielding a screened Poisson equation $\Delta \chi = \nabla \cdot \mathbf{V}$ in the unconstrained case. In practice, we adopt a robust pipeline: (i) compute average spacing to set a scale; (ii) estimate normals by jet fitting over k-nearest neighbors (k-NN); (iii) consistently orient normals via a minimum spanning tree (MST) heuristic; (iv) discard unoriented samples below a fraction threshold; (v) run Poisson reconstruction to obtain a watertight mesh; (vi) enforce outward orientation and optionally require closed output. Steps (ii)–(iv) condition the vector field $\mathbf{V}$ to be piecewise coherent and reduce topological artifacts.

Our discretization leverages robust library solvers that couple the implicit reconstruction with a Delaunay-based meshing of an isosurface of χ, producing a triangle mesh with near-uniform triangles at the recovered level set. We adopt three acceptance tests: (i) orientation consistency, (ii) closedness, and (iii) a volumetric sanity check.

Orientation consistency (oriented fraction): for mesh $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ with face normals $\mathbf{n}_f$ and sample points $\mathbf{x}_f$ on each face,

$$\phi_{\text{or}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \mathbb{I}\big[\mathbf{n}_f \cdot \nabla \chi(\mathbf{x}_f) > 0\big], \quad \text{accept if } \phi_{\text{or}} \geq 1 - \tau_{\text{or}}.$$

Closedness: count boundary edges $E_\partial = \{e \in \mathcal{E} : \deg(e) = 1\}$ and non-manifold edges $E_{\text{nm}} = \{e \in \mathcal{E} : \deg(e) \neq 2\}$,

$$\rho_b = \frac{|E_\partial|}{|\mathcal{E}|}, \quad \rho_{nm} = \frac{|E_{\text{nm}}|}{|\mathcal{E}|}, \quad \text{accept if } \rho_b \leq \tau_{\text{cl}} \text{ and } \rho_{nm} = 0.$$

Volumetric sanity: compare the Poisson mesh volume V_{mesh} (via divergence theorem) to the 3D convex-hull volume V_{hull} of the input points; with ratio $r = V_{\text{mesh}}/V_{\text{hull}}$,

$$r_{\min} \leq r \leq r_{\max} \quad \big(\text{equivalently } \delta_V = \tfrac{|V_{\text{mesh}} - V_{\text{hull}}|}{V_{\text{hull}}} \leq \tau_V\big).$$

If the closedness test fails or the volumetric check falls outside $[r_{\min}, r_{\max}]$, we trigger an alternate reconstruction path (e.g., advancing-front meshing or parameter continuation) and re-evaluate the same criteria. Default tolerances are dataset-dependent (e.g., $\tau_{\text{or}} = 0.02$, $\tau_{\text{cl}} = 0$, $r_{\min} = 0.4$, $r_{\max} = 0.98$), and can be tightened for high-quality scans.

Robustness. The oriented-fraction gate removes cases where normal orientation fails globally; the closedness requirement avoids open surfaces for solids. Average spacing provides a scale proxy for implicit solving and meshing.

Advancing Front surface reconstruction Cohen-Steiner and Da, 2004 builds a surface directly over the point set by iteratively growing a front of triangles according to local quality criteria and proximity. Starting from seed configurations, candidate triangles are evaluated and accepted when their circumradius, edge lengths, and proximity are consistent with local sampling. The method is explicit (no volumetric grid), naturally adapts to varying sampling densities, and can preserve sharper features where implicit smoothing would blur them.

We use a standard advancing-front routine that returns a (possibly open) manifold triangle soup. Post-processing may repair small defects; optionally, a closedness requirement can filter out incomplete reconstructions.

Comparison. Compared to Poisson, advancing front avoids solving a global partial differential equation (PDE) and may better respect undersampled edges, but can leave gaps and is sensitive to noise without preprocessing.

2.2.4 Colour, Volume, and Area

Color Modeling. For color modeling we fit a small-mixture Gaussian model to red, green, and blue (RGB) samples in $[0, 255]^3$ using diagonal covariances. Let $\{\mathbf{x}_i\}_{i=1}^N$ be color vectors with optional integer weights $w_i \geq 1$. For $K \in \{1, 2, 3\}$ components, parameters are mixture weights π_k, means $\boldsymbol{\mu}_k \in \mathbb{R}^3$, and diagonal variances $\boldsymbol{\sigma}_k^2 \in \mathbb{R}_+^3$. Expectation–Maximization maximizes the weighted log-likelihood with numerical stabilizers:

$$\begin{aligned} \text{E-step: } & \gamma_{ik} \propto \pi_k \, \mathcal{N}\big(\mathbf{x}_i \,\big|\, \boldsymbol{\mu}_k, \operatorname{diag}(\boldsymbol{\sigma}_k^2)\big), \\ \text{M-step: } & \boldsymbol{\mu}_k \leftarrow \frac{\sum_i w_i \, \gamma_{ik} \, \mathbf{x}_i}{\sum_i w_i \, \gamma_{ik}}, \quad \boldsymbol{\sigma}_k^2 \leftarrow \max\Big(\varepsilon, \; \frac{\sum_i w_i \, \gamma_{ik} \, (\mathbf{x}_i - \boldsymbol{\mu}_k)^{\odot 2}}{\sum_i w_i \, \gamma_{ik}}\Big), \\ & \pi_k \leftarrow \frac{\sum_i w_i \, \gamma_{ik} + \alpha}{\sum_i w_i + K\alpha}, \quad \varepsilon > 0, \; \alpha \ll 1. \end{aligned}$$

Model order is selected by Bayesian information criterion (BIC)$(K) = -2\,\ell(\hat{\theta}_K) + p_K \log N$, optionally with a small additive penalty per extra component to bias toward visual simplicity. Initialization uses k-means++ centers, and computations are stabilized with log-sum-exp. Prediction uses the maximum a posteriori component for each color.

Volume. Volume calculation employs two complementary methods depending on mesh topology. For closed, watertight meshes, the signed volume method computes exact volume via the divergence theorem, summing signed tetrahedra formed between each face and the origin:

$$V_{\text{signed}}(\mathcal{M}) = \frac{1}{6} \sum_{f=(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) \in \mathcal{F}} (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0) \cdot \mathbf{v}_0.$$

This method produces zero discretization error for planar geometries and sub-1% error for well-tessellated curved surfaces. For non-closed meshes where signed volume is undefined, an adaptive voxel method provides robust volume estimates. The method initializes a base voxel grid covering the mesh bounding box, then uses ray casting with AABB tree acceleration to classify voxel centers

as inside or outside. Boundary voxels undergo recursive subdivision up to three levels to improve accuracy:

$$V_{\text{voxel}}(\mathcal{M}) \approx \sum_{v \in \mathcal{V}} s_v \cdot \text{vol}(v), \quad s_v = \begin{cases} 1 & \text{if center inside } \mathcal{M} \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{V}$ denotes the refined voxel set. This method consistently underestimates by approximately 5%, providing predictable accuracy bounds suitable for quality assessment on incomplete geometry.

Area. Surface area A of a polygon mesh is computed by summing contributions from all faces using CGAL's area calculation routines. All input meshes undergo automatic triangulation preprocessing via `CGAL::Polygon_mesh_processing::triangulate_faces()` to ensure consistent handling regardless of input face topology. This preprocessing eliminates calculation artifacts where polygon faces were previously processed incorrectly. For a triangulated mesh, the area computes as:

$$A(\mathcal{M}) = \sum_{f=(\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2)\in\mathcal{F}} \tfrac{1}{2} \|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|_2.$$

The method achieves perfect accuracy for planar surfaces and sub-1% error for curved surfaces with moderate tessellation (128+ faces), consistent for both open and closed meshes.

2.3 The Preprocessing Pipeline

The preprocessing pipeline (Figure 2.7) takes two inputs: the original .e57 file and the segmented .las file generated through the ScanPlan portal. Both are required because the .e57 file contains the panoramas and their respective poses, information that is not stored in the .las file.

The first step of the pipeline extracts all panoramas and their poses from the .e57 file. Each camera's pose is stored in an individual JavaScript Object Notation (JSON) file. Next, the segmented .las file is divided into separate object classes as detected by ScanPlan, as well as combined classes when needed. The combined classes are grouped as follows:

- **Structural combined**: walls, floors, ceilings, beams, and columns
- **Non-structural combined**: all remaining classes except noise and the structural elements

The structural combined classes are used to generate the room shells. During room segmentation, these classes are divided per room. The non-structural combined classes are used for the 2D-to-3D algorithm: excluding structural elements helps to cluster objects that were not identified by ScanPlan's 3D object recognition.

Room segmentation also relies on the panorama poses. A room is generated only if a pose is located within its boundaries. The output of this step is a comma-separated values (CSV) file containing information for all rooms.

The room shells are then processed by the Easy3D plane detection algorithm using RANSAC. The detected planes are exported as a .bvg file and passed to the PolyFit reconstruction algorithm, which produces a watertight 3D object. This object is subsequently processed by the plane segmentation algorithm, which assigns both semantic and geometric properties. A CSV file is exported containing all room information.

Finally, all object files are split by room, clustered, and analyzed to extract geometric properties such as bounding boxes. These results are exported as CSV files, ready to be imported into a database as described in the next subsection.

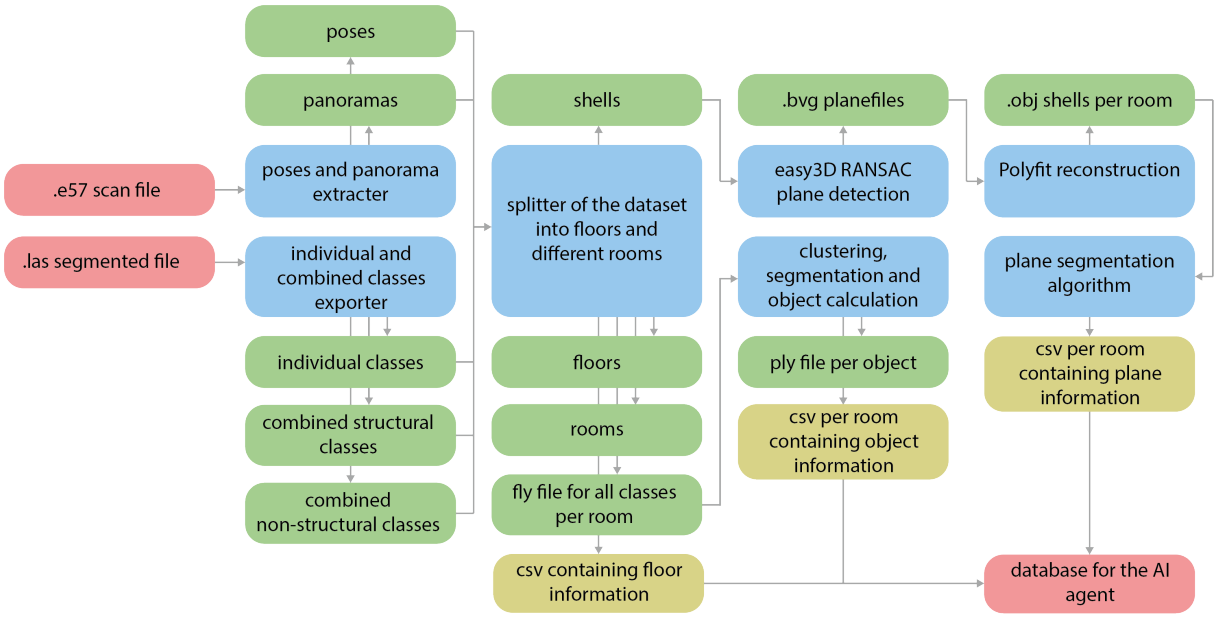


Figure 2.7: Diagram of the preprocessing pipeline

2.4 Database

2.4.1 Database Design and Schema

The foundational data layer for the AI agent is a relational structured query language lite (SQLite) database. This database is designed to provide a structured, hierarchical, and efficiently queryable "world model" of the scanned environment. The schema is organized into five interconnected tables that represent the spatial hierarchy of a building, from its floors down to individual objects and surfaces. The tables are defined as shown in the Table 2.1.

This relational model, connected through foreign keys, enables complex and efficient queries—for example, retrieving all objects of a certain class within a specific room. Such a schema design ensures scalability and flexibility for both spatial analysis and AI-driven contextual reasoning.

To ensure high-performance data retrieval, the database schema includes several indexes on frequently queried columns, such as foreign keys (`floor_id`, `room_id`). These indexes significantly accelerate `JOIN` operations, allowing the system to quickly retrieve all objects, images, or planes associated with a specific room. Furthermore, a `UNIQUE` index on the `object_code` column in the `objects` table guarantees data integrity and enables near-instantaneous lookups for specific objects referenced by the agent's API.

2.4.2 Data Ingestion and Enrichment

The process of populating the database follows a structured workflow. First, the database file is initialized with the five empty tables. The preprocessing pipeline then generates a series of intermediate CSV files containing structured data for floors, rooms, objects, and planes, which are ingested into the corresponding tables.

The key steps in population are:

1. **Floor and Room Discovery:** The script reads `rooms_manifest.csv` in each `floor_*/` directory to identify all valid rooms and their corresponding directory names (e.g., `room_007`).
2. **Data Ingestion:** For each room, the script locates and ingests data from two critical CSV files:

Table	Name	Purpose	Schema (Key Columns)
1	`floors`	Establishes the highest level of the spatial hierarchy, defining distinct floors within a building.	`floor_id (PK)`, `floor_name`, `floor_number`
2	`rooms`	Catalogs individual rooms, linking each to a parent floor. Stores high-level room attributes.	`room_id (PK)`, `floor_id (FK)`, `room_name`, `room_type`, `total_area`, `total_volume`
3	`objects`	Contains detailed geometric and semantic information for every discrete object within a room.	`object_id (PK)`, `room_id (FK)`, `class`, `object_code (Unique)`, `center_x`, `center_y`, `center_z`, `length`, `width`, `height`, `area`, `volume`
4	`images`	Stores metadata for panoramic images, linking each image to the room in which it was captured.	`image_id (PK)`, `room_id (FK)`, `image_path`
5	`planes`	Stores properties of reconstructed planar surfaces (e.g., walls, floors, ceilings).	`plane_id (PK)`, `room_id (FK)`, `plane_class`, `normal_x`, `normal_y`, `normal_z`, `centroid_x`, `centroid_y`, `centroid_z`, `area`

Table 2.1: Relational database schema defining the hierarchical structure of the scanned environment.
Abbreviations: PK = Primary Key, FK = Foreign Key.

- **Object Data** (`room_XXX.csv`): This file contains the output of the geometric clustering, including object-specific details like `cluster_id`, dimensions (`size_x`, `size_y`, `size_z`), and center point. This data is loaded into the `objects` table.
- **Plane Data** (`planes_data.csv`): This file contains the output of the plane segmentation algorithm, including `group_name`, semantic `class` (wall, ceiling, floor), `area`, and `normal` vector. This is loaded into the `planes` table.

3. **Key Linking:** A crucial element is the `object_code` (e.g., "0-7-12"), which is stored in the `objects` table. This unique identifier serves as the primary key that links the database records to the external C++ geometric tools.
4. **Image Storage:** The database also scans for and stores the paths to all panorama images (`.jpg`, `.png`) associated with each room in the `images` table.

This structured database allows the agent to perform fast, complex lookups (e.g., "find all 'chair' objects in `room_007`") without needing to access the original point cloud files directly.

Following this, a **room enrichment** step is performed to update the `room_type` field in the `rooms` table. This classification is determined using a multimodal approach. The system analyzes the semantic `class` labels of objects within each room (from the `objects` table). This information, combined with visual analysis of the associated panoramic images (from the `images` table) using a generative pre-trained transformer (GPT) vision model, allows for an inferred classification of the room's function (e.g., *kitchen*, *bathroom*).

2.5 AI Agent

2.5.1 The Functional API Gateway

The API wrapper (`ai_api_wrapper.py`) serves as the agent-computer interface (ACI), a stable and well-documented set of "tools" that the AI agent uses to interact with the system's data and computational functions. It creates a crucial layer of abstraction, decoupling the agent's reasoning from the low-level implementation details of database queries and file system operations. This modular design ensures the system is robust and maintainable (Yang, Jimenez, et al., 2024).

This wrapper translates high-level Python methods into command-line instructions for the `scripts/ai_api.py` script. It exposes five specific geometric and visualization tools to the agent:

- **VOL (Volume):** Calculates the 3D mesh volume of an object.
- **CLR (Color):** Analyzes the dominant colors of an object's point cloud.
- **BBD (Distance):** Calculates the Euclidean distance between the bounding box centers of two objects.
- **RCN (Reconstruct):** Generates a 3D mesh from a point cloud (often triggered by VOL).
- **VIS (Visualize):** Generates a uniform resource locator (URL) to launch an interactive 3D point cloud viewer.

The API exposes a set of high-level functions through concise three-letter *Head Codes* (e.g., `RMS`, `VOL`, `ARE`). When the agent calls one of these functions, the API translates the request into the appropriate action, such as querying the SQLite database or dispatching a task to a backend C++ binary for a geometric calculation. All data is returned to the agent in a structured JSON format to ensure reliable communication.

2.5.2 Agent Architecture and Reasoning Framework

The AI agent is the cognitive core of the system, responsible for interpreting user intent and orchestrating the system's resources to generate intelligent, data-grounded responses. Its architecture comprises a **Query Handler** that manages user interaction, a **Data Processing Module** for parsing API responses, and an **Image Analysis Engine** for handling multimodal inputs.

The agent's operational workflow follows a structured reasoning cycle, evolved from the "Thought-Action" framework (Yao et al., 2022). This updated reasoning cycle consists of the following five stages:

1. **Scope Classification (Initial "Thought"):** The **Query Handler** receives the user's natural language query but does not immediately act. Instead, it first passes the query to a preliminary, lightweight LLM call (`_determine_query_scope`). This initial 'thought' step classifies the user's intent as either `SINGLE_ROOM` (focusing on one specific room) or `MULTI_ROOM` (requiring comparison, searching, or aggregation).
2. **Contextual Data Grounding (Retrieval):** Based on the classified scope, the system employs a retrieval-augmented generation (RAG) methodology (Lewis et al., 2020). The agent queries the SQLite database to fetch the precise data payload needed. For a `MULTI_ROOM` query, it would call `_get_all_rooms_data()`, loading the object inventories for *all* rooms into the context. For a `SINGLE_ROOM` query, it calls `get_room_summary()` for only the relevant room.
3. **Tool Invocation ("Action") and Multimodal Reasoning:** The agent's `_parse_and_execute_tool` function inspects the query for tool-related keywords (e.g., "distance," "color") or explicit object codes (e.g., "0-7-12"). If a tool is required, this step triggers an 'action' by calling the `AiApiWrapper`. Concurrently, the **Image Analysis Engine** (`_get_room_images`) acts as a fallback; if the `CLR` tool fails, the agent automatically retrieves the room's panorama to attempt a visual-based answer.

4. **Information Synthesis and Response Generation:** This is the agent's main reasoning step. The **Data Processing Module** collects the retrieved data (from Step 2) and any JSON responses from API calls (from Step 3). The LLM then parses this complete context to synthesize a final answer. This is where both calculation (e.g., summing wall areas) and interpretation (e.g., translating raw RGB values into color names) happen, guided by the rules in the system prompt.
5. **Proactive Visualization (Final "Action"):** In interactive mode, the agent performs a final 'action' *after* the LLM response is generated. It attempts to generate a proactive `VIS` link for the relevant room or objects and appends it to the final response as a helpful suggestion.

To extend on the 2D to 3D projection, the AI agent is introduced as an intelligent intermediary between the user and the spatial data (CSV-based 3D measurements and room images). Its primary function is to allow users to interact with the dataset in natural language, rather than through direct numerical queries or manual inspection. For example, a user can ask questions like: *"How much does it cost to paint the walls of the living room?"* or *"How many furniture items are detected and what are their dimensions?"* The AI Agent processes these queries using a combination of data analysis and large language model capabilities.

2.5.3 The Agent's System Prompt

The agent's behavior and reasoning are dictated by a comprehensive system prompt, which is dynamically generated providing a detailed framework that functions as the agent's "brain," providing all rules, data, and context. Listing 2.1 presents a condensed version of this prompt, highlighting its key structural components. It is a form of declarative programming that defines the agent's logic through a series of key instructions:

- **Role and Persona:** It instructs the agent to act as an advanced Spatial AI assistant with the mindset of a construction professional.
- **Data Grounding:** It includes a placeholder where room-specific data is dynamically inserted, grounding the agent in the facts of the environment.
- **Behavioral Guidelines:** It enforces a `BE DIRECT & PRACTICAL` tone, mandating the use of lists and tables for clarity.
- **Reasoning Heuristics:** It provides rules for `MULTI-SOURCE ANALYSIS` (prioritizing 3D data for measurements), `COST CALCULATIONS` (requiring formulas to be shown), and `SPATIAL INTELLIGENCE` (encouraging analysis of object relationships).
- **Output Formatting:** It mandates a `FINAL ANSWER INSTRUCTION` to provide a concise, unambiguous summary with confidence levels.

```
You are an Advanced Spatial AI Assistant specializing in architectural space analysis.

CORE IDENTITY & MISSION

Your primary mission is to provide PRECISE, DATA-DRIVEN spatial analysis by combining:
- Geometric data from point cloud processing
- Visual context from room photography
- Specialized computational tools for D analysis
...

SYSTEM ARCHITECTURE

You operate within a multi-component system:
1. Database Layer: SQLite database with floors, rooms, objects, planes, images
2. Computational Pipeline: C++ tools for geometry processing (volume, color, distance)
3. API Layer: Python wrapper dispatching operations to C++ executables
4. Query Processing: Two-stage LLM workflow (scope classification -> answer generation)
5. Visualization Engine: Web-based 3D point cloud viewer
...

DATA MODEL & CONVENTIONS

IDENTIFICATION CODES:
1. Room Code: <floor_id>-<room_id> (Example: "0-7" = Floor 0, Room 7)
```

```
2. Object Code: <floor_id>-<room_id>-<object_id> (Example: "0-7-12" = Object 12 in Room
    7)
...

AVAILABLE TOOLS (5 SPECIALIZED OPERATIONS)

1. VOLUME (VOL) - Calculate 3D mesh volume
   Input: Single object code (e.g., "0-7-12")
   Output: Volume in cubic meters, mesh status...

2. COLOR (CLR) - Analyze dominant colors using Gaussian Mixture Model
   Input: Single object code (e.g., "0-7-12")
   Output: RGB values [0-255] with weights for each color component
          CRITICAL: You receive RAW RGB values. You MUST interpret them into human-
       readable color names.

   RGB INTERPRETATION GUIDE:
       [0-50]: Very dark/black tones
       [200-255]: Light/bright
       High R, low G/B -> Reds/pinks
       Similar R/G/B -> Grays/whites
       EXAMPLES: [180, 195, 185] -> "soft sage green"
   ...

3. DISTANCE (BBD) - Calculate Euclidean distance between object centers
   Input: Two object codes (e.g., "0-7-12 0-7-15")
   Output: Distance in meters, 3D vector...

4. RECONSTRUCT (RCN) - Generate 3D mesh from point cloud
   (Usually auto-triggered by VOL operations)

5. VISUALIZE (VIS) - Launch interactive 3D point cloud viewer
   Input: Room codes and/or object codes
   Output: Browser URL for 3D visualization
   ...

TOOL INVOCATION PROTOCOL

EXPLICIT TOOL CALL FORMAT:
TOOL: [HEAD_CODE] [code1] [code2_optional]
(Example: TOOL: CLR 0-2-3)

EXTERNAL API RESULT HANDLING:
When you receive "EXTERNAL API RESULT" prefix:
1. DO NOT repeat the tool command
2. SYNTHESIZE a natural language answer incorporating the data
...

ROOM DATA CONTEXT (PROVIDED BELOW)

[Dynamically inserted room data, including all objects, planes, and dimensions, is placed
     here]

(Example for single room)
CURRENT ROOM CONTEXT: room_007 on floor_0 (Type: kitchen)
Area: 25.00m  , Approx Dimensions: L:5.00m x W:5.00m

OBJECT INVENTORY:
====================
-> CHAIR (3 total):
  - chair_001 (CODE: 0-7-1) [L:0.5m / W:0.5m / H:0.9m]
...
-> TABLE (1 total):
  - table_001 (CODE: 0-7-5) [L:1.2m / W:0.8m / H:0.7m]
...

RESPONSE GUIDELINES (CRITICAL - READ CAREFULLY)

1. DATA-DRIVEN ACCURACY
       Base ALL answers on provided room data or API results
       NEVER invent dimensions, object counts, or spatial relationships
       If data is missing: state "Data not available"
```

```
2. SEMANTIC ROOM TYPES
       Use the 'room_type' field (e.g., 'kitchen', 'bedroom') for room category questions

3. COLOR INTERPRETATION (CRITICAL FOR CLR TOOL)
       For CLR results, YOU MUST translate RGB values into descriptive color names
...
(Other rules omitted for brevity)
...

ADVANCED CAPABILITIES

- MULTI-MODAL ANALYSIS: Integrate visual data with geometric data.
- SPATial REASONING: "The chair is near the table (0.6m apart)"
- SEMANTIC UNDERSTANDING: "This layout suggests a dining area (table + 4 chairs)"
...

LIMITATIONS & CONSTRAINTS

Be- transparent about system boundaries:
    No Real-Time Sensing: Data is from previous scans, not live
    Tool Dependencies: Volume/Color require successful reconstruction
...
```

Listing 2.1: Condensed system prompt

This message is otherwise known as the hidden prompt which is given to the agents as an instruction list of how it should communicate with itself and how the final output should be generated and structured. The system prompt itself was developed through an iterative process of trial and error. Initially, the prompts were kept broad, but this often resulted in inconsistent or overly verbose responses. By gradually refining the instructions, we aligned the model's output with the project's requirements—focusing on structured answers, the use of CSV-derived values and transparency about confidence levels. Some instructions, such as enforcing strict JSON formatting, worked well to ensure that the results could be parsed automatically. Others, like trying to make the model infer missing values, proved unreliable and were removed to keep the agent consistent and predictable. Ultimately, the prompt was limited to the specific use cases relevant for spatial queries.

2.6 2D to 3D

While LLMs have become increasingly capable of describing and reasoning about images, their outputs remain confined to natural language. Early in the project, we identified that this form of multi-modality provides only limited value for spatial reasoning. For instance, an LLM can describe the appearance of a panorama and indicate where an object is located, but it cannot retrieve precise pixel coordinates or isolate objects geometrically.

In the ScanPlan dataset, many objects are already pre-classified; however, a significant number remain unclassified—even within environments the model was trained on—and this issue becomes even more pronounced in new or unseen settings, such as airports. To ensure that our system remains functional in such cases, we developed a fallback method that allows users to manually select unclassified objects.

In this section, we describe the fallback method added to the pipeline, which connects the front-end and back-end. A user's click on an object in the panorama (front-end) triggers a process that sends the clicked pixel coordinates to the back-end, where the corresponding points in the point cloud are clustered into a 3D object. The newly identified object is then integrated into the pipeline and can be queried in natural language through the front-end interface, as explained in Section **??**.

2.6.1 Receiving and Processing User Input

When the user interacts with the system, the segmentation process can be described mathematically as follows. The system begins with an input image $I(u, v)$, where u and v denote the pixel coordinates

in the horizontal and vertical directions, respectively. The user selects exactly five points on the image (e.g., Figure 2.8), forming the set

$$P = \{(u_i, v_i) \mid i = 1, 2, 3, 4, 5\}.$$

Each coordinate pair (u_i, v_i) represents a location of interest chosen by the user. These points are associated with foreground labels, as all selected points are assumed to correspond to the target object. Therefore, the label set is defined as

$$L = \{l_i = 1 \mid i = 1, 2, 3, 4, 5\}.$$

Figure 2.8: Example of user-selected object in the panorama

The red markers in 2.8 show the five user clicks $(u_i v_i)$ that define the target object for segmentation.

2.6.2 Generating a Binary Mask with SAM2

The segment anything model 2 (SAM2) (Ravi et al., 2024), building on the original segment anything framework (Kirillov et al., 2023), takes the image I, the user-defined points P, and the corresponding labels L as input. Following the formulation of these models, the segmentation process can be expressed as

$$\{(M_i, s_i)\}_{i=1}^{k} = f_{\text{SAM2}}(I, P, L),$$

where k denotes the number of mask hypotheses generated by the model. The final mask $M^*(u, v)$ is selected as the one associated with the highest confidence score:

$$M^* = \arg\max_{M_i} s_i.$$

Each mask $M_i(u, v) \in \{0, 1\}$ defines the region of the image that belongs to a segmented object, where

$$M_i(u, v) = \begin{cases} 1, & \text{if pixel } (u, v) \text{ belongs to the object,} \\ 0, & \text{otherwise.} \end{cases}$$

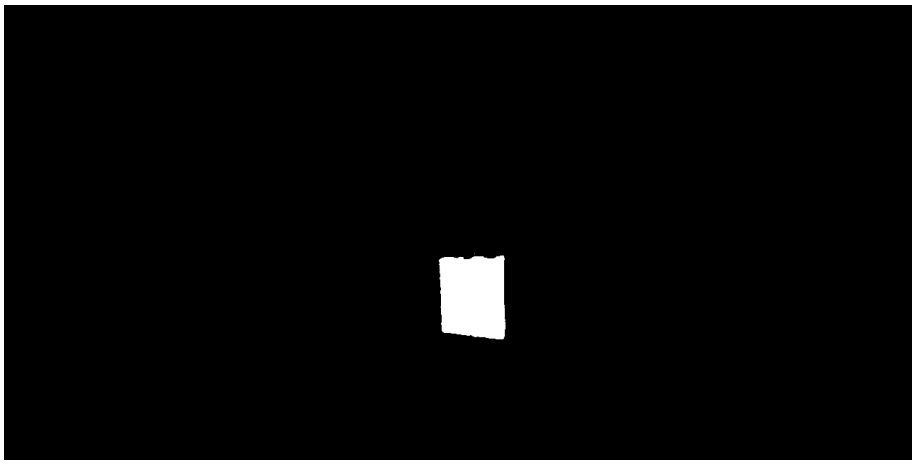


Figure 2.9: Example of binary mask

Internally, SAM2 generates a continuous probability map $\hat{M}(u,v) \in [0,1]$, where each pixel value indicates the likelihood of belonging to the object. This step can be represented as

$$\hat{M}(u,v) = f_{\text{SAM2}}^{\text{prob}}(I,P,L).$$

To obtain the binary mask $M^*(u,v)$, the probability map is thresholded at a fixed value τ, typically $\tau = 0.5$:

$$M^*(u,v) = \begin{cases} 1, & \text{if } \hat{M}(u,v) \geq \tau, \\ 0, & \text{if } \hat{M}(u,v) < \tau. \end{cases}$$

This thresholding operation converts the continuous prediction map into a discrete binary mask, where $M^*(u,v) = 1$ indicates that the pixel (u,v) belongs to the segmented object and $M^*(u,v) = 0$ corresponds to the background.

The resulting binary mask $M^*(u,v)$ thus defines the segmented region in the 3D image corresponding to the user's intention. An example of a binary mask can be seen in Figure 2.9.

2.6.3 Loading the 3D Scene and Camera Pose

Once the mask has been created, the system proceeds to link it with the corresponding 3D point cloud. To do this, we first determine which 3D scene data belongs to the same location as the selected panorama. The system reaches for a 3D file within the same room directory as the image. This file contains a point cloud $P = \{\mathbf{p}_w^i\}$, where each point $\mathbf{p}_w^i = [x_i, y_i, z_i]^T$ represents a spatial coordinate in world space.

To correctly map between the 2D image and the 3D point cloud, the camera pose must be used. The camera pose extracted from the point cloud provides:

- Camera translation: a translation vector $\mathbf{t}_{wc} = [t_x, t_y, t_z]^T$,
- Camera orientation: a rotation quaternation $q = (q_x, q_y, q_z, q_w)$.

The quaternation is first converted into a 3x3 rotation matrix R_{wc} using:

$$R_{wc} = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) \\ 2(q_xq_y + q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_wq_x) \\ 2(q_xq_z - q_wq_y) & 2(q_yq_z + q_wq_x) & 1-2(q_x^2+q_y^2) \end{bmatrix}$$

This matrix expresses the rotation from the world coordinate system (w) to the camera coordinate system (c). Its transpose $R_{wc} = R_{wc}^T$ is then used to transform world coordinates into the local coordinate frame of the camera.

2.6.4 Transforming Points to the Camera Coordinate System

For each 3D point $\mathbf{p}_w = [x, y, z]^T$, the position relative to the camera is computed as $\mathbf{p}_c = R_{cw}(\mathbf{p}_w - \mathbf{t}_{wc})$ where $\mathbf{p}_c = [X, Y, Z]^T$ the point is in the camera frame. This transformation centers the point cloud around the camera and aligns its orientation so it matches the direction of the panoramic image.

2.6.5 Mapping Between Pixel Coordinates and Spherical Angles

For an equirectangular image of width W and height H, each pixel (u, v) corresponds to a viewing direction defined by spherical angles (λ, φ), where

$$\lambda = 2\pi \left(\frac{u}{W} - \tfrac{1}{2} \right), \qquad \varphi = \pi \left(\tfrac{1}{2} - \frac{v}{H} \right).$$

Here, $\lambda \in [-\pi, \pi]$ represents the azimuth (longitude) and $\varphi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ represents the elevation (latitude).

In the inverse direction, when projecting a 3D point $\mathbf{p}_c = [X, Y, Z]^T$ from the camera coordinate system onto the image plane, the spherical angles are first computed as

$$\lambda = -\arctan 2(Y, X), \qquad \varphi = \arcsin\left(\frac{Z}{\sqrt{X^2 + Y^2 + Z^2}} \right),$$

and then converted into pixel coordinates using

$$u = \left(\frac{\lambda}{2\pi} + \frac{1}{2} \right) W, \qquad v = \left(\frac{1}{2} - \frac{\varphi}{\pi} \right) H.$$

This bidirectional mapping defines the geometric relationship between the 3D environment and its equirectangular image representation. The resulting coordinates (u, v) indicate where each 3D point would appear in the panorama.

2.6.6 Filtering 3D Points Using the Binary Mask

At this stage, every 3D point has an associated pixel coordinate (u, v). The binary mask M is then used to determine which of these points lie within the segmented region. The mask is first converted to a boolean array so that pixels with value 1 correspond to "inside the region" and 0 to "outside." For each projected point, the system checks the value of $M\,[v, u]$. If this value equals 1, the point is classified as belonging to the segmented object. All such points are collected into a new subset:

$$P_{filtered} = \{\mathbf{p}_w^i | M[v_i, u_i] = 1\}$$

This subset represents the 3D geometry that corresponds directly to the area selected in the image. This can be seen in Figure 2.10.

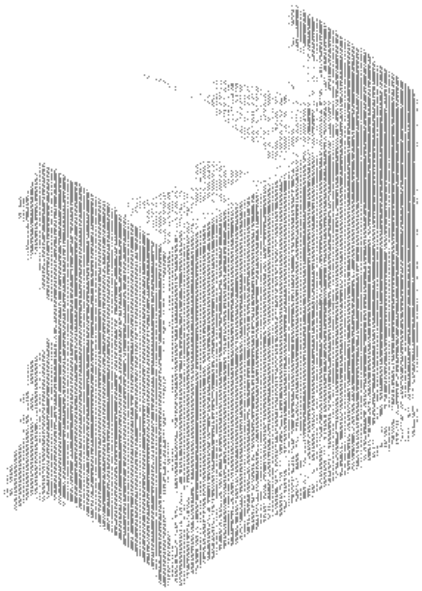


Figure 2.10: Example of filtered 3D points

2.6.7 Clustering

Projecting a 2D mask from a panorama into 3D rarely yields a clean object: the back-projected set can mix points from adjacent or touching objects, include stray hits from noise and reflective surfaces, exhibit uneven density with range, and even leak across room boundaries when the view looks through doors. After projecting the user-selected 2D mask to the panorama and filtering the 3D points, the system clusters the masked point cloud to yield a coherent 3D object. The current implementation leverages fast Euclidean pre-clustering followed by a pose-driven selection strategy that is robust to residual noise and variable point densities. This turns a noisy, heterogeneous point set into a single, room-aware 3D instance that remains reliable even in crowded scenes.

Pre-Clustering Via FEC

Given a masked point cloud $\mathcal{P} = \{\mathbf{p}_i\}_{i=1}^{N} \subset \mathbb{R}^3$, the pre-clustering stage labels points into equivalence classes using a radius-based criterion:

$$\|\mathbf{p}_i - \mathbf{p}_j\|_2 \leq \varepsilon, \qquad \text{with at most } K \text{ neighbors per query.}$$

In code, the radius tolerance is set as

$$\varepsilon_{\text{FEC}} = \max(\texttt{eps}, 10^{-6}), \qquad K = \max(8, \texttt{minPts\_core}),$$

following the parameter names in `m2c::Params`. The FEC routine builds a k-dimensional tree (k-d tree) and assigns integer labels so that points reachable within the radius constraint share the same label; see `third_party/pcg/FEC.hpp`.

Let the resulting raw clusters be $\{\mathcal{C}_j\}_{j=1}^{J}$, with sizes $s_j = |\mathcal{C}_j|$. The implementation computes the mean size

$$\bar{s} = \frac{1}{J} \sum_{j=1}^{J} s_j,$$

then applies a relative size filter parameterized by $n \in (0, 1]$:

$$s_j \geq \kappa_{\min} \lfloor n \, \bar{s} \rfloor .$$

Only clusters meeting $s_j \geq \kappa_{\min}$ are kept for downstream selection. This adaptive threshold suppresses small noisy components without requiring an absolute cluster-size prior.

Pose-Driven Cluster Selection by Proximal Voting

The camera (or reference) position $\mathbf{C} \in \mathbb{R}^3$ is derived from the translation in the pose **JSON!** (see `include/m2c/io_pose.h`, `src/io_pose.cpp`). Among the filtered clusters, the system picks the object most spatially consistent with $\mathbf{C}$ by voting among the m nearest points to $\mathbf{C}$:

1. Form the pool $\mathcal{S}$ of all points belonging to the retained clusters and compute distances $d_i = \|\mathbf{p}_i - \mathbf{C}\|_2$.
2. Take the m smallest-distance points, denoted by the index set

   $$\mathcal{S}_m \underset{S \subset \mathcal{S}, |S| = m}{\arg\min} \sum_{i \in S} d_i,$$

   i.e., the indices of the m smallest d_i.
3. Each point in $\mathcal{S}_m$ votes for its cluster label; the selected cluster index is

   $$j^* = \arg\max_j \, \#\{\mathbf{p}_i \in \mathcal{S}_m : \mathbf{p}_i \in \mathcal{C}_j\},$$

   with ties broken by smaller total distance $\sum_{\mathbf{p}_i \in \mathcal{S}_m \cap \mathcal{C}_j} d_i$.

The winning cluster $\mathcal{C}_{j^*}$ is returned as the final object for this step. This rule is resilient when multiple plausible components exist: it favors the component most concentrated around the camera reference location.

Post-Processing: Diameter and Acceptance

For the selected cluster, the code estimates an AABB diameter

$$\delta = \sqrt{(x_{\max} - x_{\min})^2 + (y_{\max} - y_{\min})^2 + (z_{\max} - z_{\min})^2}.$$

This statistic is exported alongside the cluster indices. A general-purpose validator interface to check absolute minimum size and maximum diameter. While the current selection path relies on the relative-size filter and proximity voting, the validator provides a straightforward hook for stricter acceptance criteria if desired.

3 Results

In this chapter the results are shown of the process described in the previous chapter. The results are mostly intermediate steps and are pooled with their respective accuracy measurement. The first section shows the MoSCoW table that was created as a starting point with the initial goals for the application.

3.1 MoSCoW Table

In the Project Initiation Document (PID) goals were set to describe the functionality of the application. This was done via a MoSCoW (Must, Should, Could & Won't have) table. During the midterm evaluation of the project, most goals were already completed and the project went into a different direction. Since, the MoSCoW table was not updated with new topics. In this section the MoSCoW table is still presented with the final results. The MoSCoW table is displayed in Table 3.1.

Subject	Description	Priority	Completed
Results	The application must answer indoor queries correctly.	M	✓
	The application must answer basic (i.e., questions are specified in the query) queries correctly.	M	✓
	The application must answer questions related to geolocation data.	M	✗
	The application must enable the chatbot to interpret natural language queries and convert them into actionable data queries.	M	✓
	The application must be limited to answer spatial queries.	M	✓
	The application should answer advanced queries (i.e., there are multiple reasoning steps) correctly.	S	✓
	The application should know when it cannot answer a question.	S	✓
	The application could answer outdoor queries.	C	✗
Generalizability and datasets	The application produces correct results for the residential building dataset.	M	✓
	The application must produce correct results for sample use cases.	M	✓
	The application must support text input handling.	M	✓
	The application must support image input handling.	M	✓
	The application must support location input handling.	M	✓
	The application should produce correct results for the TU Twente dataset.	S	✗
	A list of questions which the application asks the user to provide more info should be made	S	✗
Performance	There must be an evaluation on the correctness of the output.	M	✓
	There must be an evaluation report showcasing the chatbot's response time consumption.	M	✗
	A list of queries which can be answered within 1 minute should be made	S	✗
	A list of queries which can be answered above 1 minute should be made	S	✗
User interface	The application must have a simple command line interface.	M	✓
	The application should log intermediate steps to the user.	S	✓
	The application should have a simple local GUI.	S	✓
	The application should have a webpage that prints the questions and answers.	S	✓

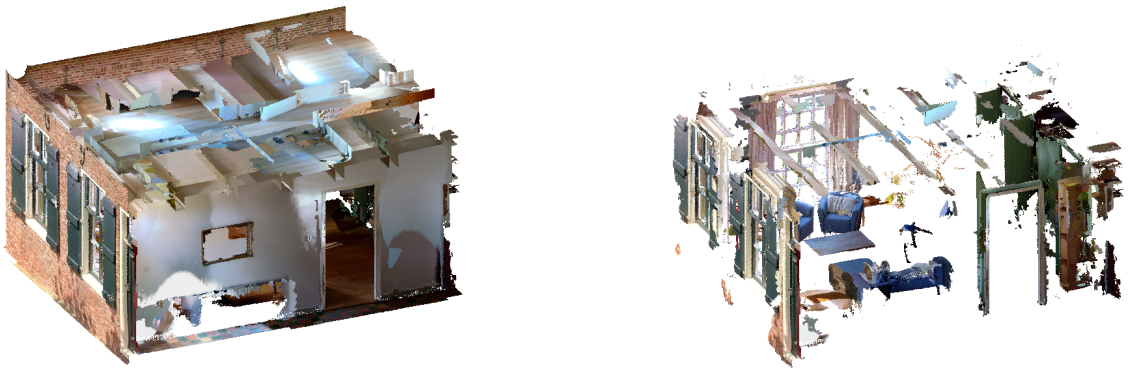
Table 3.1: MoSCoW Table

3.2 Room Segmentation

The room segmentation has been tested on two different datasets, a regular-sized house dataset (Figure 2.1) and a bigger dataset, namely one of the TU Twente buildings (Figure 3.4). The first dataset was used as a basic case to test the main functionalities of the application, while the latter was used to test if the application is also suitable for bigger datasets. For the implementation of the room segmentation, the decision was made that a room can only exist if a panorama image was taken in that room.

3.2.1 House Dataset

The house dataset can be considered a regular-sized house with both indoor and outdoor features. For this application the focus was mostly on the indoor features. The house consists of a ground floor with a living room, a kitchen, an eating room, and a hallway and of a first floor with two bedrooms, a bathroom, and a storage room. In Figure 3.1 the results of the living room of the house are shown, where the left image includes only the shell and the right image all objects that are detected in the room. In Figure 3.2 the results of the bathroom located on the first floor are shown, where the left image again includes the shell of the room and the right image all the objects in the room.



(a) Shell (b) Objects

Figure 3.1: Room segmentation results living room

(a) Shell (b) Objects

Figure 3.2: Room segmentation results bathroom

In both figures the shell of the room contains more data points than it should, especially for the floor; this is done on purpose to make sure that all the walls are contained in the room, as otherwise PolyFit does not work. For this dataset there are more objects on the ground floor, and the scanning quality is significantly better; hence, the results produced are better for the ground floor. The full results are shown in figure 3.3. The main thing that can be seen is that the front right part of the roof is missing; this is due to a bed obstructing the LiDAR scan from the wall.

(a) Ground floor

(b) Total house

Figure 3.3: Resulting shells of the house dataset

3.2.2 TU Twente Dataset

The TU Twente dataset is substantially bigger in size compared to the house dataset; as a reference, the building is shown in Figure 3.4. This dataset consists of 2 open spaces and lots of different smaller rooms. In Figure 3.5 the main assembly hall of the TU Twente is shown. It consists of the columns, the staircase, and some of the windows.

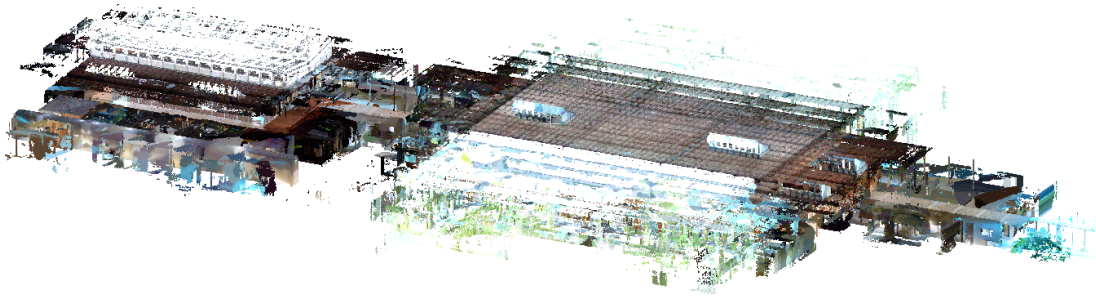
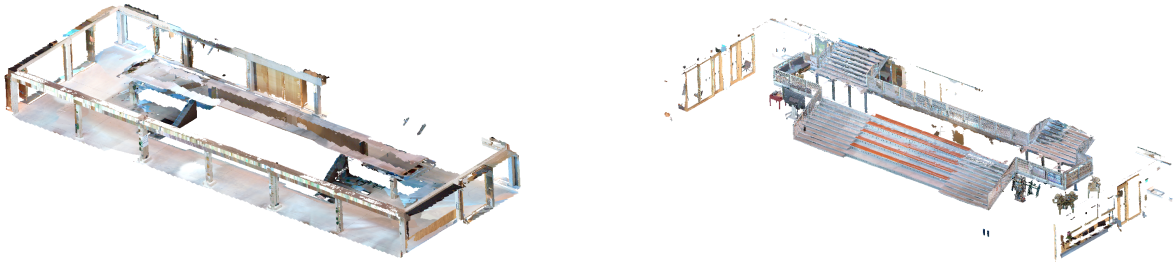


Figure 3.4: TU Twente dataset



(a) Shell (b) Objects

Figure 3.5: Assembly hall TU Twente

In Figure 3.6 all shells of reconstructed rooms are shown. The panoramas that were extracted consist mostly of the open spaces and the hallways; hence, only a few rooms are shown. Most important are the two open spaces. The result were achieved after tweaking the slicing heights and the 2D grid resolution after visual inspecting the dataset to see at which height the beams are located.

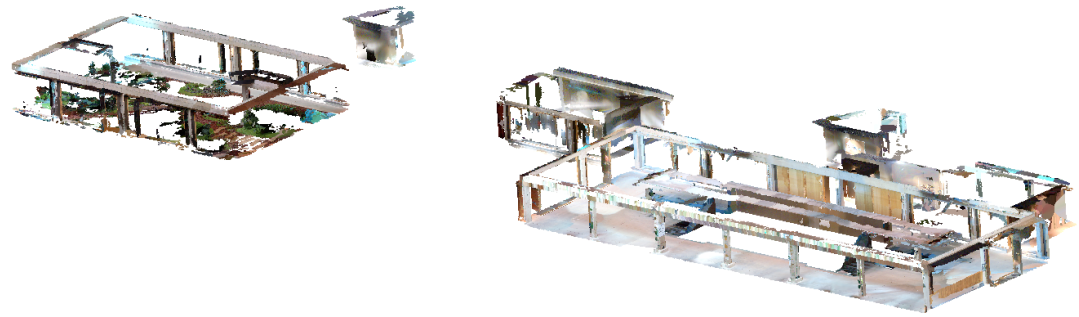


Figure 3.6: All constructed shells TU Twente

3.2.3 PolyFit room reconstruction

Reconstructing the rooms using PolyFit is a complex, multivariate process. The PolyFit step happens right after the room reconstruction and strongly depends on the quality of its output. As is shown by the differences between Figure 3.7 and Figure 3.8.

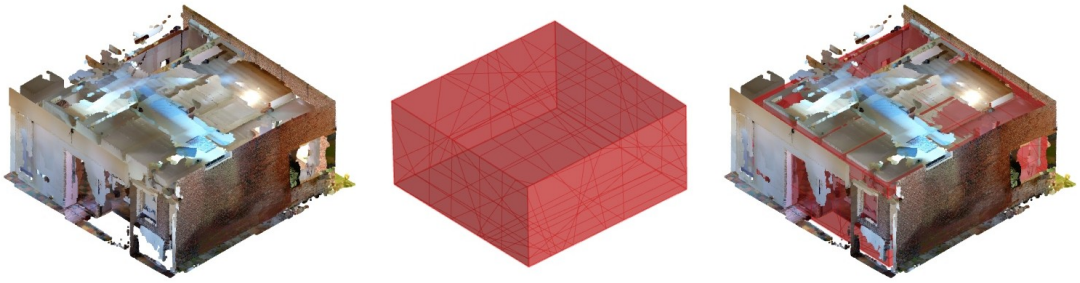


Figure 3.7: A successful reconstruction

In the first case, the room is clearly segmented and has the wanted low complexity. The RANSAC algorithm easily identifies the correct planes, allowing PolyFit to produce an accurate and simple reconstruction. In contrast, the second case shows what happens when the segmentation result is more ambiguous. Here, only one wall contains points, so no candidate faces are generated where they should be. As a result, faces are drawn through beams and other clutter, leading to a reconstruction that significantly underestimates the room's width, depth, area, and volume, but the height remains fairly accurate.

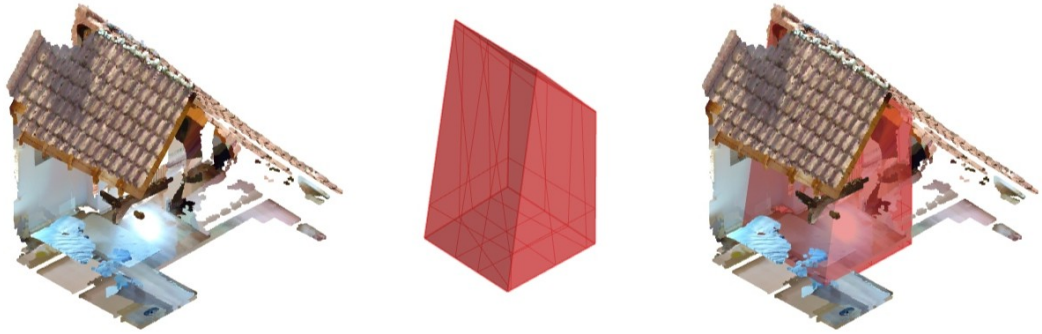
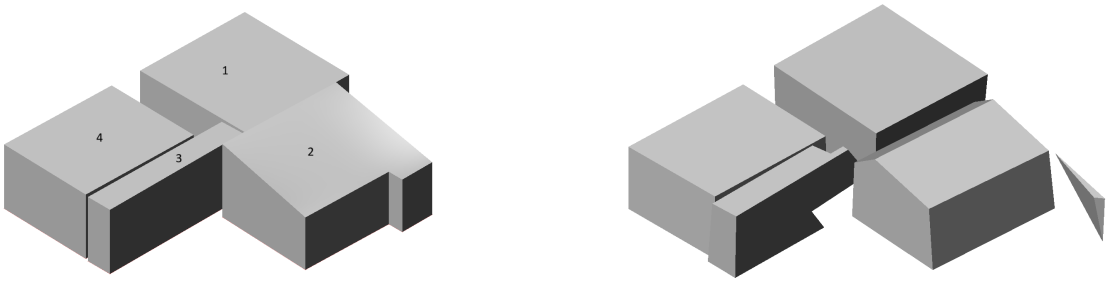


Figure 3.8: A flawed reconstruction

To assess the accuracy of the reconstruction, a ground truth model was established. Defining this reference, however, proved challenging due to the inherent abstraction required in determining what constitutes the "true" simplified geometry of a room. Certain ambiguities arise in this process. For example, in the case of a ceiling with exposed beams, it is unclear whether the ceiling height should be defined by the upper surface of the beams or by the lower spaces between them. Similarly, built-in furniture introduces uncertainty, as a point cloud cannot capture the hidden depth of such elements, even though they contribute to the overall room volume. The ground truth was created by hand from the point cloud.

To address these challenges, the following conventions were adopted for the creation of the ground truth model:

- The maximum ceiling height was taken as the representative ceiling height, disregarding the presence of beams.
- Built-in furniture was included from the room volume unless it occupied an entire wall, in which case the reconstruction was limited by its visible outer surface.



(a) Ground truth of the ground floor

(b) Our reconstruction result of the ground floor

Figure 3.9: Comparison between the ground truth and the reconstructed ground floor.

Figure 3.9 shows both the ground truth and our reconstruction of the ground floor. The ground and first floor were evaluated separately, as the point cloud quality differs greatly between them due to occlusion, the slanted roof, and variations in the segmentation output.

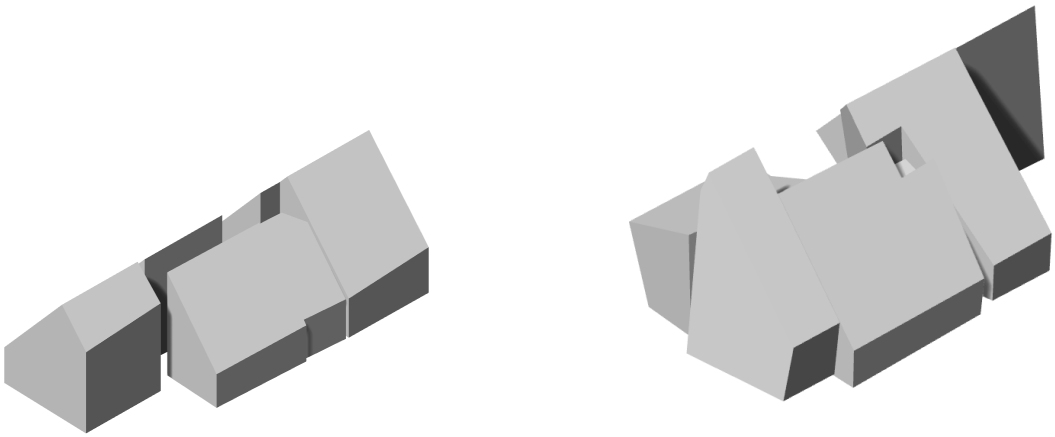
For the ground floor, the results are generally accurate. The main errors include one room that was reconstructed but does not exist, visible on the right, and a hallway that was only partially reconstructed. These issues lead to an underestimation of the total volume and area, as shown in Table 3.2. It is important to note that not existing or constructed rooms were left out of the table and averages.

A different issue occurs with the extension. The floor slicer cuts off part of the roof, causing the upper structure to be lost. Since the slicer operates globally, variations in floor height across the building can result in further errors or underestimations in the reconstruction.

Overall, the volume shows a relatively high percentage difference of 11.20%, though it remains within an acceptable range. The area difference is moderate at 7.86%, indicating a good level of accuracy, while the height measurement is particularly reliable, with a deviation of only 2.94%.

Room Type	ΔVolume (%)	ΔArea (%)	ΔHeight (%)
1. Kitchen	5.63	9.28	5.26
2. Extension	9.79	6.33	3.80
3. Hall	29.30	15.23	2.62
4. Living room	0.08	0.61	0.08
Average	**11.20**	**7.86**	**2.94**

Table 3.2: Individual room reconstruction metrics of the ground floor in percentage difference to the ground truth



(a) Ground truth of the first floor

(b) Our reconstruction result of the first floor

Figure 3.10: Comparison between the ground truth and the reconstructed first floor.

The reconstruction of the first floor (Fig. 3.10) is unreliable and cannot be used for further analysis (Table 3.3, see Figure 3.11). The average volume difference of 58.28% indicates that the results are unusable. Similarly, the area shows an average difference of 41.73%, making it unsuitable for evaluation. Only the height measurement, with an average difference of 11.64%, can be considered acceptable depending on the application. One reason these errors are so large is that the segmentation step does not accurately match the actual room boundaries. Although PolyFit produces a correct output based on the given input, the preceding segmentation is flawed. As a result, errors propagate through the pipeline, leading to the high average deviations observed for this floor. Several other factors contribute to the poor reconstruction quality of the first floor, which will be discussed in detail in the discussion section.

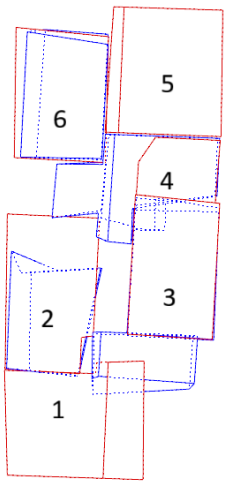


Figure 3.11: Order of the rooms

Room Number	ΔVolume (%)	ΔArea (%)	ΔHeight (%)
1	67.55	49.77	1.44
2	54.39	45.29	35.34
3	22.54	13.60	8.78
4	119.74	80.87	4.20
6	27.16	19.13	8.43
Average	**58.28**	**41.73**	**11.64**

Table 3.3: Individual room reconstruction metrics of the first floor in percentage difference to the ground truth

3.3 Clustering

For testing the `pcg_room` tool, we use the Stanford Large-Scale 3D Indoor Spaces Dataset (S3DIS) by Armeni et al., 2016. The dataset provides high-quality indoor scans with complete semantic labels, which makes it suitable for controlled evaluation. We first extract one or multiple semantic classes from a single room and save them as standalone polygon file format (PLY) files, then run the clustering algorithm on these files.

3.3.1 Test Scenarios

All samples are drawn from single-room scenes in university buildings (e.g., meeting rooms, offices, halls). Such scenes typically avoid extreme object congestion that would cause inter-object connectivity

after scanning, while still offering sufficient semantic variety for testing. We select categories that reasonably contain two or more physical instances—such as *chair*, *appliance*, *table*, *board*, and *closet*—with some rooms having up to 16 objects and others as few as 2. Overall point coverage is good, and occlusion "dead zones" are relatively limited.

3.3.2 Testing Results

For each clustered output, instances produced from the same source file are randomly colorized and visualized side-by-side to compare pre- and post-clustering results. The four figures below show representative pairs. Based on deviations between outcomes and expectations, we group the results into four categories: (1) Small objects are filtered out; (2) Clusters corresponding to multiple objects are merged into a single object; (3) A single object is split into two or more clusters; (4) Expected outcomes where clusters are complete, or any imperfections are minor for downstream processing.

Figure 3.12: Before (left) vs. after (right) clustering. Cases where small objects are filtered out after clustering.

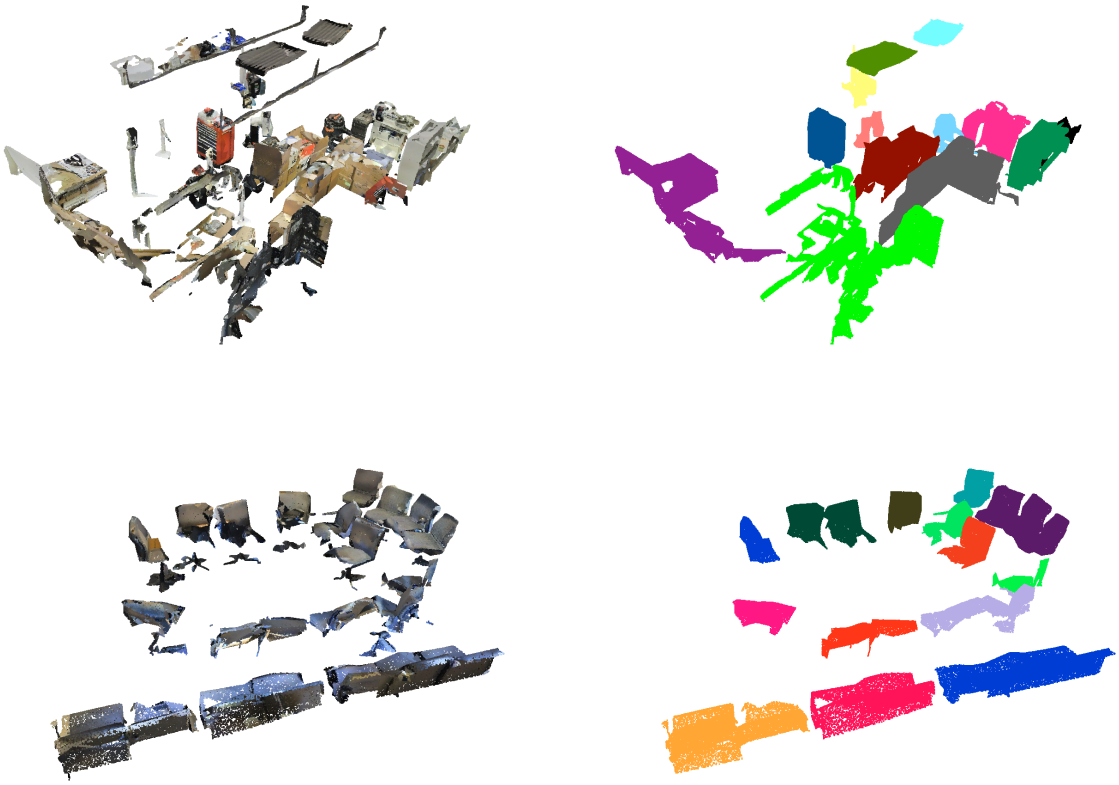


Figure 3.13: Before (left) vs. after (right) clustering. Cases where clusters corresponding to multiple objects are merged into a single object.

Figure 3.14: Before (left) vs. after (right) clustering. Cases where a single object is split into two or more clusters.

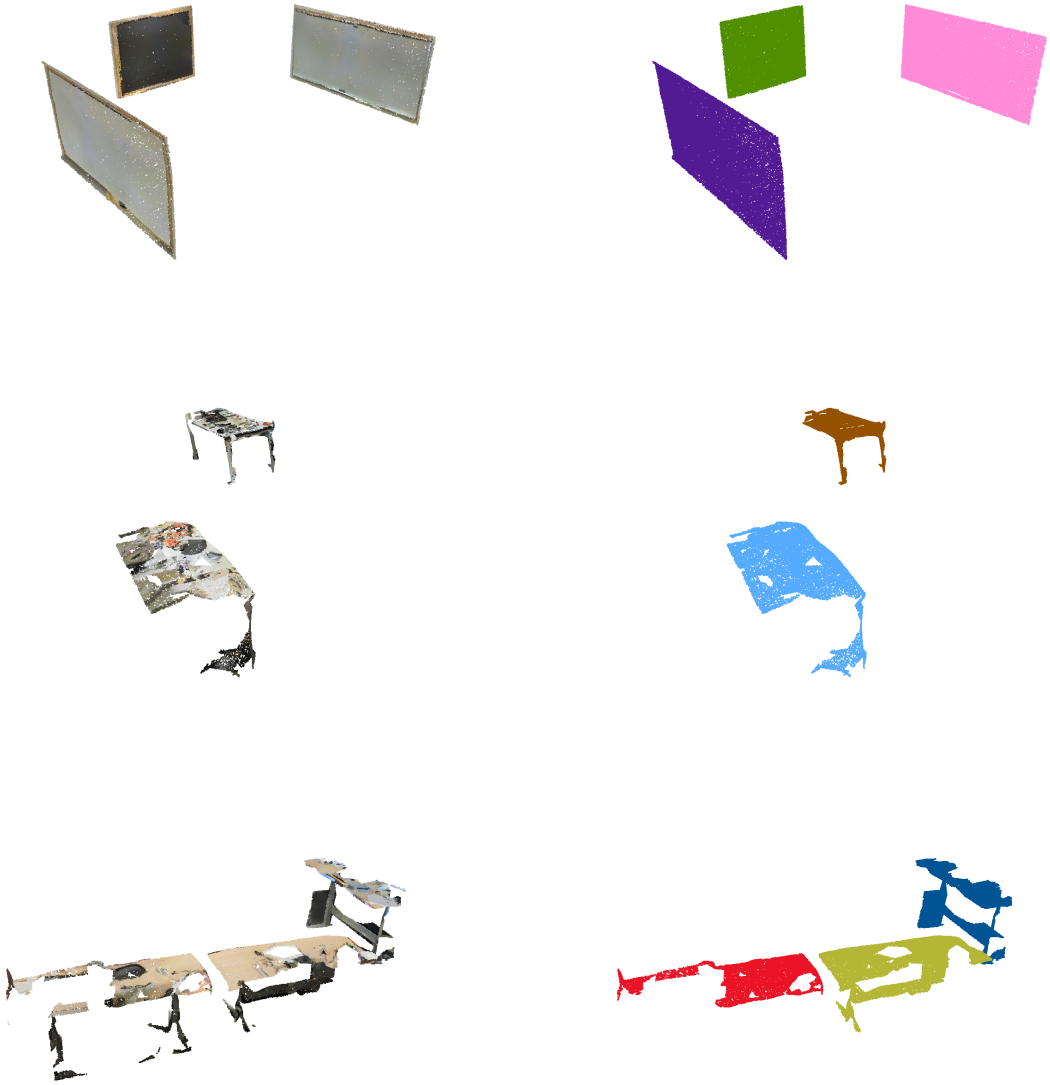


Figure 3.15: Before (left) vs. after (right) clustering. Expected outcomes where clusters are complete, or any imperfections are minor for downstream Processing.

Most clustering outcomes are acceptable, though a subset presents minor issues: (1) **Small objects filtered out**: severe size imbalance within the same class causes smaller instances to be suppressed as noise by the filtering stage, governed by `min_cluster_size`: 50 and `filter_factor`: 0.70 (i.e., clusters with fewer than 70% of the mean size are removed). This may lead to missing objects in the aggregated CSV. (2) **Multiple objects merged**: two or more nearby objects are treated as a single cluster due to the search radius in FEC (configured via `max_neighbors`: 150). Although no physical object is truly lost, attributes of one object may be merged into another. (3) **Single object split**: points from one object become disconnected (e.g., due to coverage gaps during scanning), yielding multiple clusters for the same object. This can propagate errors into downstream surface reconstruction.

3.3.3 Bounding Box Generation

Given the clustering results, we compute an up-oriented bounding box (UOBB) for each cluster. The generated UOBBs match expectations in both orientation and tightness.

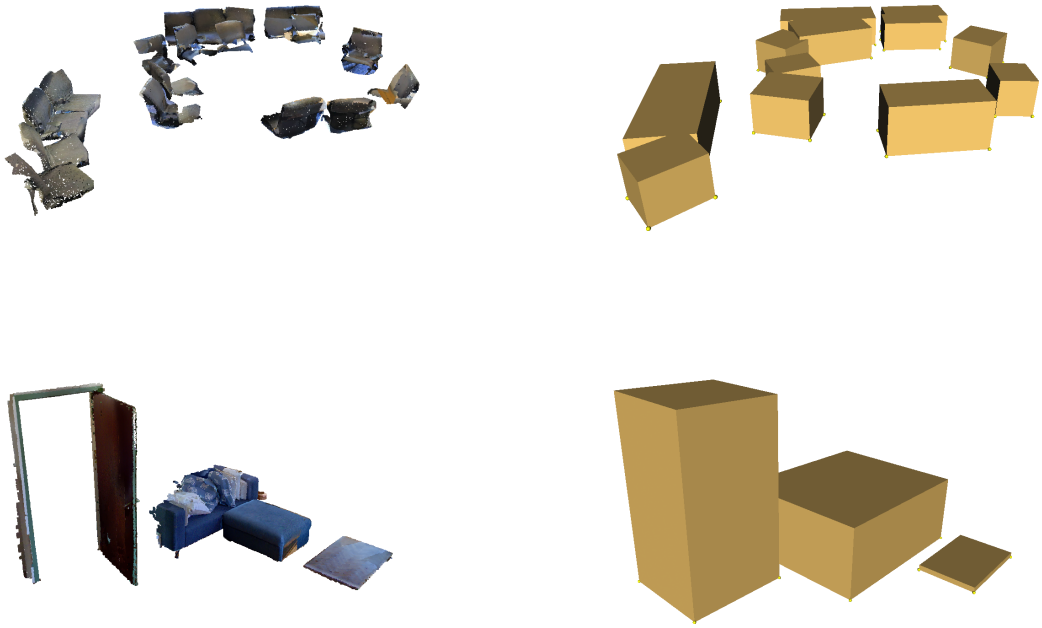


Figure 3.16: Point cloud clusters (left) and their corresponding UOBB (right).

3.4 Color Modeling

The `pcg_color` tool implements a Gaussian mixture model (GMM) based color clustering algorithm for point cloud data. To systematically evaluate its performance, we generated eight synthetic test cases with controlled color distributions. Each test case contains 1000 points uniformly distributed in 3D space with coordinates in $[-5, 5]^3$.

3.4.1 Test Dataset Generation

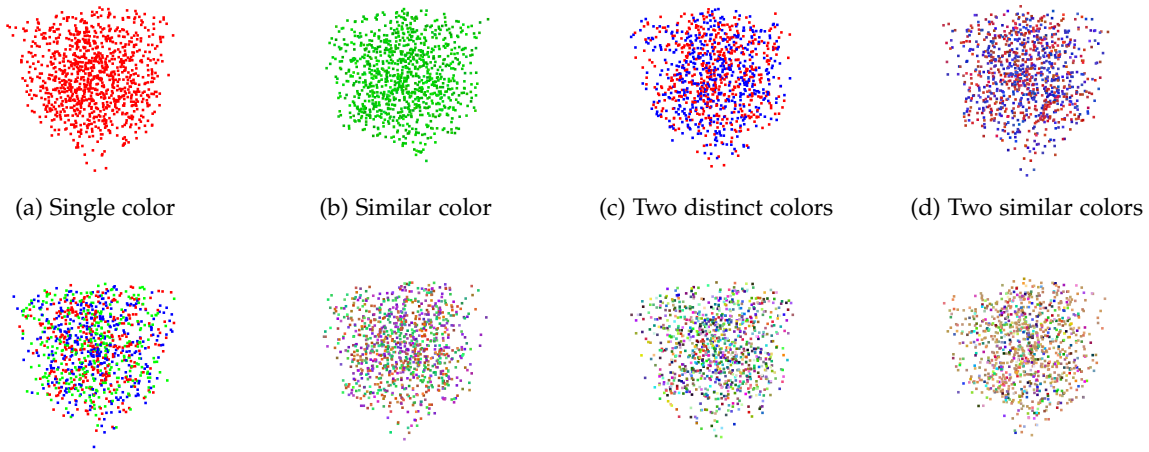
All test point clouds were generated using a Python script (`generate_test_colors.py`) that creates polygon file format (PLY) files in binary little-endian format. The spatial distribution of points follows a uniform random distribution, while color distributions vary according to predefined patterns.

3.4.2 Test Scenarios

Table 3.4 summarizes the eight test scenarios designed to evaluate different aspects of the color clustering algorithm.

Mode	Scenario Name	Description
1	Single Color	All 1000 points share identical RGB values (pure red: 255, 0, 0)
2	Similar Color	All points have similar colors with small Gaussian noise ($\sigma = 15$) around green (0, 200, 0)
3	Two Distinct Colors	500 points red (255, 0, 0), 500 points blue (0, 0, 255)
4	Two Similar Colors	Two color clusters with noise ($\sigma = 20$): reddish (200, 50, 50) and blueish (50, 50, 200)
5	Three Distinct Colors	Three equal groups: red (255, 0, 0), green (0, 255, 0), blue (0, 0, 255)
6	Three Similar Colors	Three clusters with noise ($\sigma = 25$): orangish, cyanish, purplish
7	Random Colors	Completely random RGB values uniformly sampled from [0, 255]
8	Mixed Distribution	50% random noise, 50% clustered around (200, 150, 100) with $\sigma = 15$

Table 3.4: Test Scenario Specifications



(a) Single color (b) Similar color (c) Two distinct colors (d) Two similar colors

(e) Three distinct colors (f) Three similar colors (g) Random colors (h) Mixed distribution

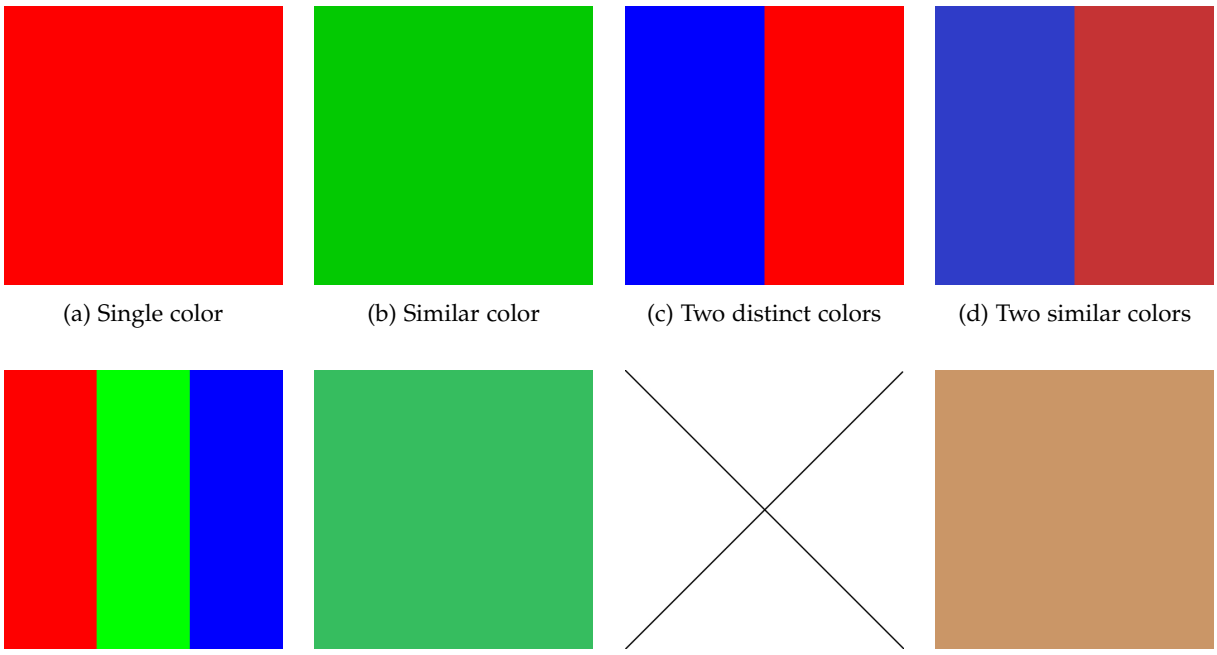
Figure 3.17: Visualization of test point clouds for all eight scenarios. Each subfigure displays the spatial distribution and color properties of the generated test data.

3.4.3 Test Experiment

The algorithm outputs a JSON structure containing the number of detected color clusters (M) and the parameters of each Gaussian component, including weight, mean RGB values, and variance for each color channel. Table 3.5 presents a comprehensive summary of all experimental results, including the detected cluster parameters for each test scenario.

Mode	Scenario	Exp. M	Det. M	Mean (RGB)	Variance (RGB)	Status
1	Single Color	1	1	(255, 0, 0)	(25, 25, 25)	Pass
2	Similar Color	1	1	(2, 201, 2)	(25, 233, 25)	Pass
3	Two Distinct Colors	2	2	(0, 0, 255) (255, 0, 0)	(25, 25, 25) (25, 25, 25)	Pass
4	Two Similar Colors	2	2	(47, 60, 200) (198, 51, 52)	(355, 307, 382) (322, 443, 488)	Pass
5	Three Distinct Colors	3	3	(0, 255, 0) (255, 0, 0) (0, 0, 255)	(25, 25, 25) (25, 25, 25) (25, 25, 25)	Pass
6	Three Similar Colors	3	1	(54, 189, 95)	(496, 378, 521)	Merged
7	Random Colors	0	0	—	—	Pass
8	Mixed Distribution	1	1	(202, 149, 102)	(278, 237, 226)	Pass

Table 3.5: Test Results Summary



(a) Single color (b) Similar color (c) Two distinct colors (d) Two similar colors

(e) Three distinct colors (f) Three similar colors (g) Random colors (h) Mixed distribution

Figure 3.18: Visualization of clustering results for all eight test scenarios. Each subfigure shows the color clustering output from the `pcg_color` algorithm.

The experimental results for color function demonstrate that the `pcg_color` algorithm exhibits robust performance across diverse color distribution patterns. The algorithm successfully identified distinct color clusters in seven out of eight test scenarios, achieving perfect or near-perfect accuracy for well-separated colors (Modes 1-5). Mode 1 perfectly recovered the pure red color at (255, 0, 0), while Mode 2 detected the green cluster with less than 1% error at (2, 201, 2) versus the ground truth (0, 200, 0). The algorithm correctly distinguished multiple distinct colors in Modes 3 and 5, demonstrating reliable cluster identification and parameter estimation.

The noise-handling capabilities validate the algorithm's design for practical applications. Mode 7 correctly returned zero clusters for completely random colors, avoiding false positives. Mode 8 successfully extracted the structured component at (202, 149, 102) from mixed data containing 50% random

noise, demonstrating effective signal extraction. The variance estimates appropriately reflected noise levels, with higher values (300-500) in intentionally noisy scenarios. The cluster merging in Mode 6 reflects GMM's preference for parsimony when cluster boundaries are ambiguous, which is appropriate for most practical scenarios where distinct objects exhibit clearly differentiated colors.

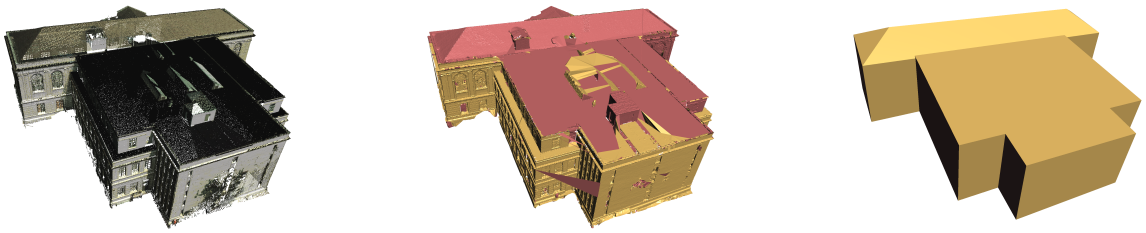
3.5 Surface Reconstruction

In this section, we evaluated the surface reconstruction methods implemented in the `pcg_reconstruct` tool using the dataset provided by Nan and Wonka, 2017. We compare reconstructed meshes against the official reconstruction results performed by the data provider (reference meshes) using two key geometric metrics: surface area and volume.

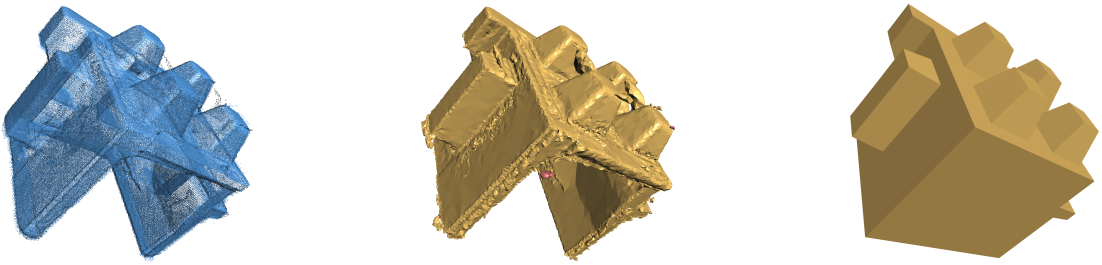
3.5.1 Test Scenarios

Seven test models were selected to represent diverse geometric characteristics. These models collectively cover a wide spectrum of geometric complexity, from simple convex shapes to intricate curved surfaces.

Surface area measurements were computed using `pcg_area`, which calculates the total surface area by summing contributions from all mesh faces using CGAL's `Polygon_mesh_processing::area()` function. Volume calculations employed `pcg_volume`, implementing a novel adaptive voxelization approach. This method begins with a base voxel grid covering the mesh bounding box, then uses ray casting with AABB tree acceleration to determine whether each voxel center lies inside or outside the mesh. Boundary voxels undergo adaptive refinement up to three subdivision levels to improve accuracy near the surface. Unlike traditional signed-volume approaches that require watertight meshes, this adaptive voxel method provides robust volume estimates for both closed and non-closed geometries, making it particularly suitable for evaluating reconstruction quality when reference meshes may contain topological defects.



building



cube

Figure 3.19: Visual comparison of reconstruction results for two models with bad results. Each row shows (left) the input point cloud, (center) the reconstructed mesh using either Poisson or AF method, and (right) the reference mesh provided by the data source. The reconstruction method is indicated in parentheses.

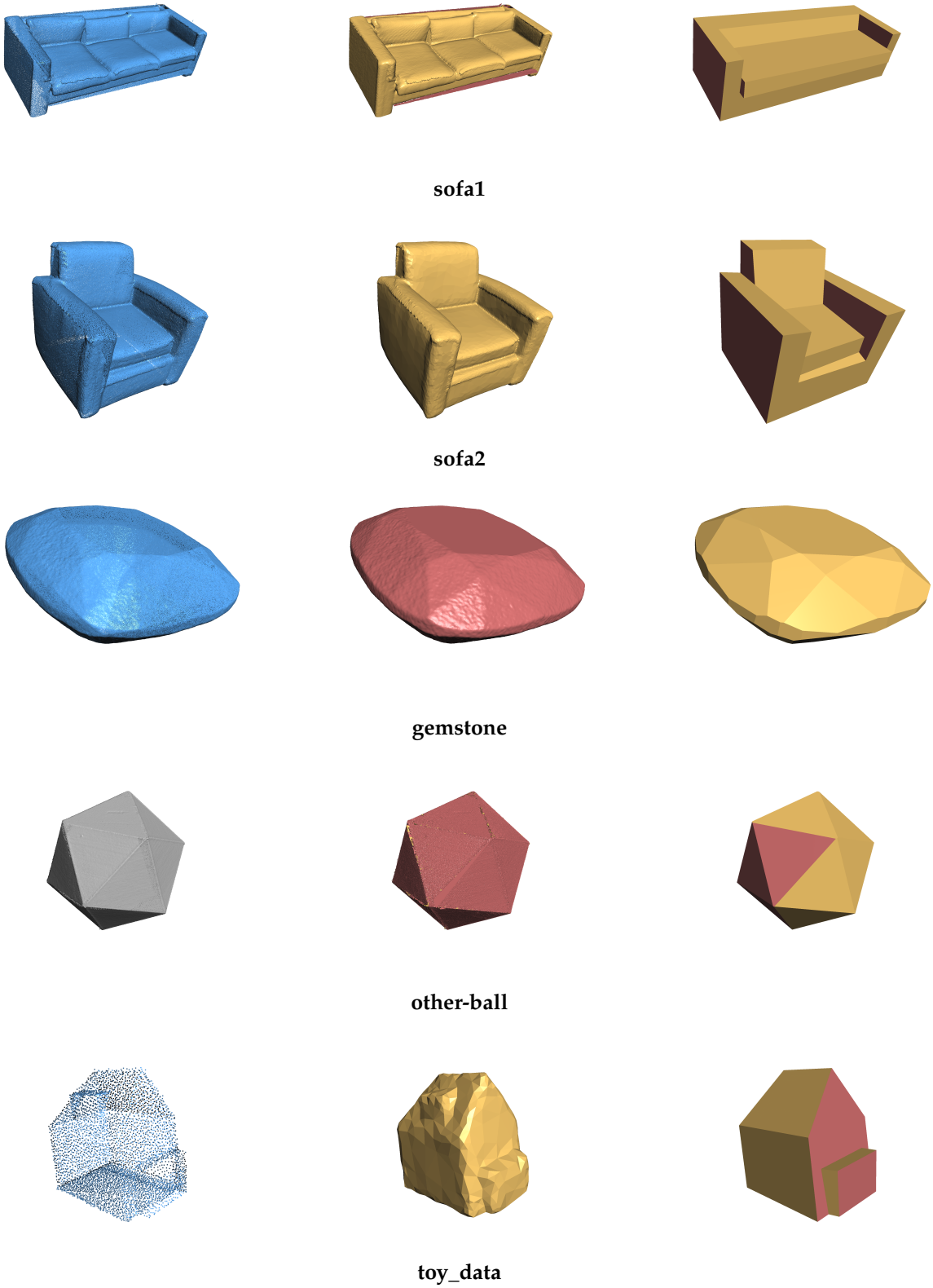


Figure 3.20: Visual comparison of reconstruction results for five models with good results. Each row shows (left) the input point cloud, (center) the reconstructed mesh using either Poisson or AF method, and (right) the reference mesh provided by the data source. The reconstruction method is indicated in parentheses.

3.5.2 Test Results

Quantitative Comparison

Table 3.6 presents the surface area and volume measurements for all test models, comparing reference meshes against reconstructed results.

Table 3.6: Surface Reconstruction Quality Metrics

Model	Type	Method	Closed	Area (m^2)	Volume (m^3)	Error (%)
building	Reference	–	×	5941.808	19530.708	–
	Reconstructed	AF	×	8333.660	11129.577	-43.0
cube	Reference	–	×	3.919	0.362	–
	Reconstructed	Poisson	✓	5.540	0.210	-42.0
gemstone	Reference	–	×	219.385	230.032	–
	Reconstructed	AF	×	218.836	230.624	+0.3
other-ball	Reference	–	×	18550.189	205479.325	–
	Reconstructed	AF	×	20384.117	202221.695	-1.6
sofa1	Reference	–	×	8.593	1.026	–
	Reconstructed	AF	×	8.495	0.753	-26.7
sofa2	Reference	–	×	3.596	0.306	–
	Reconstructed	Poisson	✓	3.951	0.306	-0.1
toy_data	Reference	–	×	174.859	144.570	–
	Reconstructed	Poisson	✓	165.013	155.497	+7.6

Note: Volume error percentage calculated as $\frac{V_{\text{recon}} - V_{\text{ref}}}{V_{\text{ref}}} \times 100\%$.

Analysis by Reconstruction Method

Overall, the reconstruction results visually conform well to the original point cloud morphology and restore surface details of the objects. However, it is noteworthy that most reconstruction results are non-closed meshes. For AF reconstruction, some results exhibit hanging faces, and certain face normals are incorrectly oriented. For the Poisson method, occasional local reconstruction surfaces deviate significantly from the original point cloud, and some noise points are also reconstructed into the mesh.

In terms of volume and area calculation, models reconstructed with the Poisson method achieved closed mesh topology in three cases (cube, sofa2, toy_data), with volume errors ranging from -0.1% (sofa2) to +7.6% (toy_data). The exceptional accuracy of sofa2 (-0.07%) demonstrates near-perfect reconstruction. AF reconstruction produced non-closed meshes, with volume errors varying more widely: -43.0% (building), +0.3% (gemstone), -1.6% (other-ball), and -26.7% (sofa). Notably, gemstone0 and other-ball0 showed excellent accuracy despite being non-closed, indicating the adaptive voxel method's robustness.

Two models (building and cube) show significant errors around -42% to -43%. This suggests potential issues specific to these reconstructions, possibly related to incomplete surface coverage or reconstruction parameter settings, warranting further investigation. The consistent underestimation in both cases points to systematic volume loss during reconstruction.

3.6 Area and Volume Calculation Validation

In this section, we validated the accuracy of geometric calculation methods implemented in `pcg_area` and `pcg_volume` tools. The validation was conducted using synthetic test meshes with analytically known surface areas and volumes, allowing for precise error quantification.

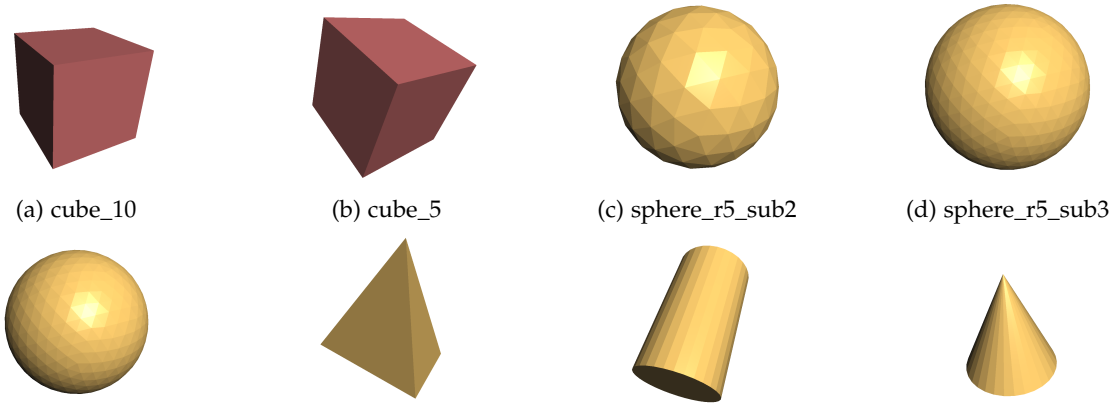
3.6.1 Test Scenarios

Eight test models with known geometric properties were generated programmatically to validate the calculation algorithms. These models represent fundamental geometric primitives with closed mesh topology, covering both planar and curved surfaces.

Test Model Specifications

The test dataset comprises eight geometric primitives representing fundamental shapes commonly encountered in 3D modeling and reconstruction. Two axis-aligned cubes with edge lengths of 10m and 5m validate the algorithms on simple planar geometries. Three spheres with radii of 5m and 3m at different subdivision levels (sub2 and sub3) test the handling of smooth curved surfaces and examine the impact of mesh tessellation density on calculation accuracy. A regular tetrahedron with 10m edges provides validation on a minimal polyhedron, while a cylinder with 3m radius and 10m height and a cone with 5m base radius and 10m height test the algorithms on geometries combining planar and curved surfaces.

All models were generated with triangular faces to ensure compatibility with CGAL's geometric algorithms. The theoretical values for surface area and volume were computed using standard geometric formulas, providing ground truth for accuracy assessment.



(a) cube_10 (b) cube_5 (c) sphere_r5_sub2 (d) sphere_r5_sub3

(e) sphere_r3 (f) tetrahedron_10 (g) cylinder_r3_h10 (h) cone_r5_h10

Figure 3.21: Test mesh visualizations showing the eight geometric primitives used for validation.

3.6.2 Test Results

Surface Area Validation

Table 3.7 presents the surface area measurements for all test models, comparing calculated values against theoretical expectations.

Table 3.7: Surface Area Calculation Accuracy

Model	Closed	Faces	Expected (m^2)	Calculated (m^2)	Error (%)
cube_10	✓	12	600.00	600.00	0.00
cube_5	✓	12	150.00	150.00	0.00
sphere_r5_sub2	✓	320	314.16	308.25	-1.88
sphere_r5_sub3	✓	1280	314.16	312.66	-0.48
sphere_r3	✓	1280	113.10	112.56	-0.48
tetrahedron_10	✓	4	173.21	173.21	0.00
cylinder_r3_h10	✓	128	245.04	244.38	-0.27
cone_r5_h10	✓	64	254.16	253.21	-0.38

Volume Validation

Table 3.8 presents the volume measurements for all test models using both signed volume and adaptive voxel methods.

Table 3.8: Volume Calculation Accuracy

Model	Expected (m^3)	Signed Volume		Adaptive Voxel	
		Value	Error (%)	Value	Error (%)
cube_10	1000.00	1000.00	0.00	950.00	-5.00
cube_5	125.00	125.00	0.00	118.75	-5.00
sphere_r5_sub2	523.60	505.88	-3.38	483.62	-7.63
sphere_r5_sub3	523.60	519.09	-0.86	498.82	-4.73
sphere_r3	113.10	112.12	-0.86	107.74	-4.73
tetrahedron_10	117.85	117.85	0.00	107.77	-8.56
cylinder_r3_h10	282.74	280.93	-0.64	266.88	-5.61
cone_r5_h10	261.80	260.12	-0.64	248.22	-5.19

Note: All test meshes are closed, allowing comparison between both volume calculation methods.

Surface Area Accuracy

The surface area calculations demonstrate excellent accuracy across all test models. Planar geometries including both cubes and the tetrahedron achieve perfect accuracy with 0.00% error, confirming that the triangulation preprocessing successfully resolved the polygon face calculation bug that previously caused 50% underestimation. This validates that the automatic conversion of polygon faces to triangular faces eliminates calculation artifacts entirely for planar surfaces.

Curved surface geometries exhibit small errors due to mesh discretization, but the magnitudes remain well within acceptable tolerances. Spheres show errors ranging from -0.48% to -1.88%, with accuracy improving as subdivision level increases from sub2 (320 faces, -1.88% error) to sub3 (1280 faces, -0.48% error). The cylinder achieves -0.27% error with 128 triangular faces, while the cone achieves -0.38% error with 64 triangular faces. The consistent negative errors for curved surfaces indicate systematic underestimation due to the piecewise-linear approximation of smooth curves, where triangular facets inscribe rather than circumscribe the true surface. Higher tessellation density reduces this error proportionally, demonstrating the expected convergence behavior for mesh-based geometric approximation.

Volume Accuracy: Signed Method

The signed volume method demonstrates exceptional accuracy for all closed meshes, with performance characteristics that align precisely with theoretical expectations. Planar geometries including both cubes and the tetrahedron achieve perfect 0.00% error, validating the exactness of signed volume computation when applied to polyhedra. This exactness stems from the method's analytical summation of signed tetrahedra volumes, which produces no discretization error for planar faces.

Curved geometries exhibit errors ranging from -0.64% to -3.38%, directly correlating with mesh tessellation quality rather than algorithmic limitations. The sphere results particularly illustrate this relationship, with accuracy improving from -3.38% at 320 faces (sub2) to -0.86% at 1280 faces (sub3), demonstrating clear convergence with refinement. The cylinder and cone achieve -0.64% error, benefiting from their combination of exact planar bases and moderately tessellated curved surfaces. These results confirm that signed volume calculation provides exact results for polyhedra and near-exact results for smooth surfaces when adequately tessellated, making it the optimal choice for closed mesh volume computation.

Volume Accuracy: Voxel Method

The adaptive voxel method exhibits consistent behavior across all test cases, with systematic underestimation characteristics that validate its design as a robust fallback approach. All measurements show negative errors ranging from -4.73% to -8.56%, indicating a predictable bias rather than erratic behavior. The cubes demonstrate particularly stable performance with exactly -5.00% error for both the 10m and 5m sizes, confirming that voxelization behavior scales consistently across different geometric dimensions.

The tetrahedron exhibits the highest error at -8.56%, likely due to its thin geometric features presenting challenges for voxel refinement where the three-level adaptive subdivision may not fully capture sharp apex regions. Curved surfaces including spheres, cylinder, and cone show errors of -4.73% to -7.63%, falling within the expected 5% tolerance and demonstrating that the method handles smooth boundaries more effectively than sharp features. The voxel method's consistent underestimation around 5% validates its design as a robust fallback for non-closed meshes, trading exactness for universal applicability while maintaining predictable accuracy bounds suitable for quality assessment and comparative analysis.

The validation results demonstrate that `pcg_area` achieves perfect accuracy for planar surfaces and sub-1% error for curved surfaces with moderate tessellation (128+ faces). For volume calculation, the signed volume method provides 0% error for polyhedra and sub-1% error for well-tessellated curved surfaces, making it the default choice for closed meshes. The adaptive voxel method consistently underestimates by approximately 5%, serving as a robust fallback for non-closed meshes where signed volume computation is undefined.

Tessellation density directly affects accuracy on curved surfaces. Comparing sphere models at different subdivision levels reveals that quadrupling face count reduces approximation error to approximately one-quarter, demonstrating quadratic convergence. For sub-1% accuracy requirements, meshes should contain at least 1000 faces. The voxel method shows higher error for thin features (tetrahedron: -8.56%), suggesting that adaptive refinement depth based on local geometry could improve performance, though current behavior remains predictable and suitable for quality assessment workflows.

3.7 2D to 3D

3.7.1 SAM Masking and Filtered 3D Points

To assess the accuracy of SAM2 masking and the subsequent 3D point filtering, we selected ten objects and visualized the outcomes in Figure 3.22. Performance varies by class and material. The microwave and the large, homogeneous kitchen surfaces yield weak masks, yet the back-projection still localizes the correct 3D region. Windows, fridges, chairs, and doors perform strongly: their planar, well-defined geometry produces clean masks and dense, coherent 3D selections. The couch shows the opposite pattern—an imprecise mask but a solid 3D selection—while the drawer illustrates occlusion: with an object in front, the mask largely vanishes at the target location, but the projected points still recover the drawer's position. The lamp is a useful edge case; despite slight spill onto a nearby window, the 3D selection remains accurate. Radiators invert the trend—masks are precise, but sparse returns in the point cloud yield a less complete 3D reconstruction. As expected, these outcomes depend on the quality and density of the underlying point cloud used in the back-projection.

Overall, this SAM2-guided selection works best for planar, matte objects and degrades with transparency, specular materials, and large uniform surfaces where segmentation tends to fragment. Even when masks are imperfect or partially missing, the projection step frequently anchors the selection to the correct 3D location—suggesting that modest mask tightening and light geometry-aware filtering can make this masking and filtering stage reliable across most indoor categories.

Figure 3.22: Results of SAM2 masking and back-projection to the point cloud

3.8 Interface

3.8.1 GUI

A broad range of stakeholders-from domain experts to non-technical users-should be able to access, explore, and act on the spatial data produced in this project. To support that goal, the system provides a simple, task-focused GUI that minimizes setup, avoids unnecessary complexity, and surfaces the essential controls directly in the workflow.

The GUI is a Streamlit front end that ties natural-language interaction to a 3D model via an AI agent and to 360° for visual context and when needed a 2D to 3D projection.

When Virtual Mode is active, the app scans a folder of panoramas and presents a 360° viewer built with Pannellum. Images are inlined as data URLs, a natural sort keeps filenames in an intuitive order, and the viewer preserves camera pose across updates while emitting events on rotation and zoom that are stored back into session state. Navigation buttons step through panoramas, and a dedicated control toggles a click mode. In this mode, left-clicks add up to five annotated hotspots, each reporting its pitch and yaw. Every click is forwarded to the bridge server as JSON containing the point and the current image filename; if the bridge returns a SAM2 mask as a base64-encoded Portable Network Graphics (PNG), the viewer is immediately reloaded to display that overlay so the segmentation result becomes visible in context. Right-clicking clears both the on-screen hotspots and the server-side click buffer. For transparency, the interface also converts each stored (pitch, yaw) to pixel coordinates of the original equirectangular image and lists those values beneath the viewer, and it indicates when an overlay is active.

Alongside the panorama interaction, the app exposes a chat section bound to the instantiated spatial AI agent. Users pose questions in natural language, and the agent answers queries about the underlying 3D model, drawing on a local SQLite database and, when enabled, image evidence. A few commands receive special handling—typing “overview” attempts to retrieve a room summary for the agent’s current room, while “quit,” “exit,” or “q” closes the conversation politely—whereas all other messages are routed to the agent’s query function and the returned text is rendered directly, with any errors surfaced as they occur. In this way, the interface blends language-based querying of the 3D model with lightweight 2D selection and immediate segmentation overlays, allowing users to ground their questions in concrete visual cues while the agent provides structured, model-aware answers. Streamlit provides the reactive framework and user interface scaffolding; Pannellum handles immersive rendering and camera events; the bridge service coordinates click ingestion and SAM2 inference; and the agent anchors everything to the 3D model so responses can combine visual evidence with spatial understanding.

3.8.2 AI API

Besides the GUI, we also developed an interface (API) for the AI-agent and a user interface with agent hooks. These two sections walk through these two components that sit on top of the geometric pipeline: (i) an AI-oriented API layer that resolves file paths, dispatches core operations, and orchestrates visualization; and (ii) a Web Graphics Library (WebGL)-based viewer that renders shells, clusters, and meshes, supports selection, and bridges back to the API.

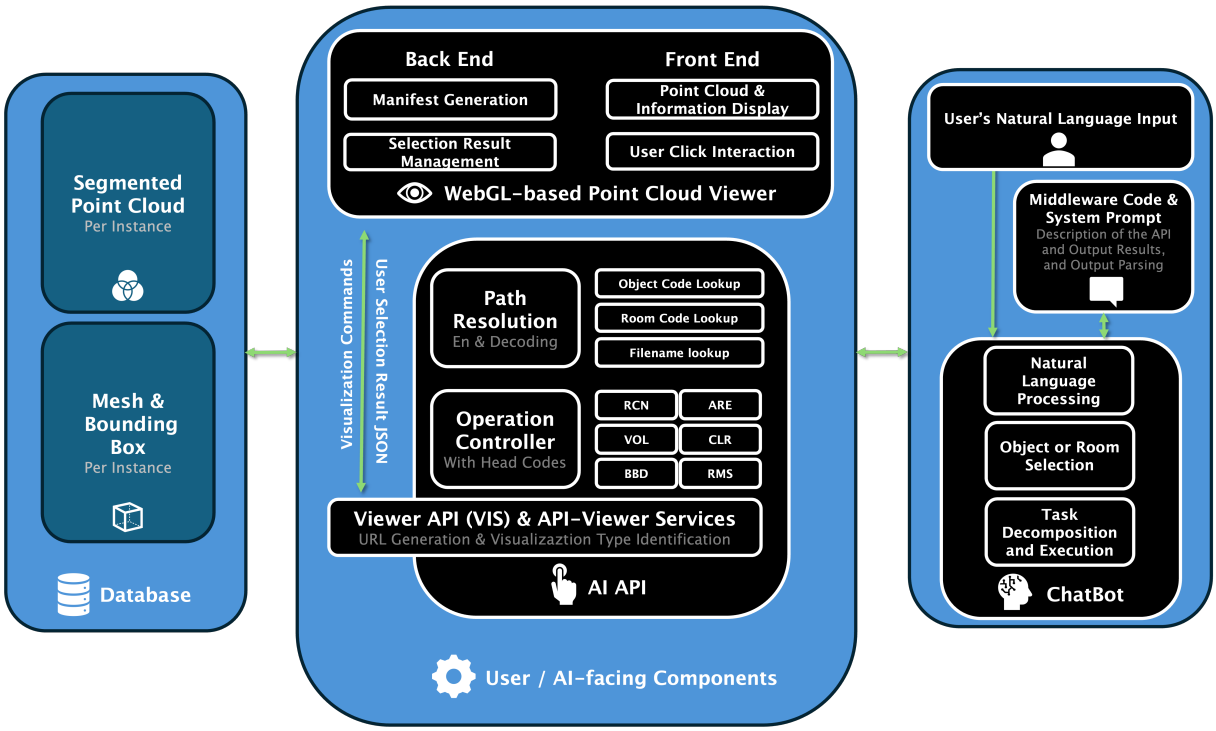


Figure 3.23: End-to-End interaction between the AI API, geometry pipeline, artifact store, manifest builder, and the WebGL viewer, including the Hypertext Transfer Protocol (HTTP) bridge for resolution and selection feedback.

The AI API (`scripts/ai_api.py`) exposes a small set of robust, composable primitives for path resolution and operation dispatch over the project's output tree. It is designed to be used programmatically (as a Python module) or via a minimal command-line interface (CLI) for agent workflows. Responses prefer JSON where possible to ease downstream parsing.

The AI API communicates with the AI agent through middleware code and follows the prompts provided to guide API usage.

Core Responsibilities

- **Path indexing and resolution.** A one-time scan of `output/builds` lookups by filename, object code, and room code; later queries resolve clusters, UOBBs, meshes, CSVs, and room shells.
- **Head codes (3-letter op codes).** Concise verbs dispatch to pipeline operations: reconstruct mesh (RCN), compute volume (VOL), surface area (ARE), analyze color (CLR), bounding-box distance (BBD), room manifest summary (RMS), and visualization orchestration (VIS).
- **Visualization orchestration.** Bridges to the web viewer by preparing downsampled assets, generating a manifest, and optionally booting dev servers; returns a ready-to-open URL.

Path Resolution

The API normalizes several common references:

- **Object code** `<floor>-<room>-<object>` (e.g., `0-7-12`) maps to all assets for that object: cluster PLYs, UOBB PLYs, and reconstructed meshes (if present).
- **Room code** `<floor>-<room>` (e.g., `0-7`) maps to the room CSV, room-level shell, and room-level shell UOBB, plus the filtered clusters directory if available.
- **Filename lookup** finds arbitrary files under `output/` (used when a user references a bare PLY name).

Head Codes

Operations are exposed through compact 3-letter head codes, each resolving inputs and executing the corresponding pipeline binary as needed. All return structured values or machine-parseable text.

- **RCN** (reconstruct): given an object code or cluster filename, reconstruct a mesh and return its path.
- **VOL** (volume): compute the volume of a mesh; if missing and allowed, reconstruct first; returns path, volume, and closedness flag.
- **ARE** (area): compute mesh surface area with the same auto-reconstruct behavior; returns path, area, and closedness flag.
- **CLR** (color): analyze dominant color components for a cluster or mesh; returns JSON if emitted by the backend.
- **BBD** (bounding box distance): compute distance/vector between two UOBB centers for a pair of object codes.
- **RMS** (room manifest summary): enumerate rooms across `rooms_manifest.csv` files and optionally prepare a multi-room visualization.
- **VIS** (visualization): prepare viewer inputs and optionally start dev servers; returns a URL with manifest.

Dispatch, Execution, and JSON-first Outputs

Each operation resolves to exactly one file target (with heuristics when multiple candidates exist), then invokes the appropriate binary (e.g., `pcg_reconstruct`, `pcg_volume`, `pcg_area`, `pcg_bbox`). When available, the API prefers JSON emitted by the binaries; otherwise it falls back to legacy text parsing with guarded extractors.

Visualization Orchestration (VIS)

The `VIS` operation prepares data and launches the viewer in one motion: it reads defaults from `data/configs/default.yaml`, resolves shells and clusters via the path index, down-samples as requested, generates a unified manifest, and, if enabled, starts the dev servers. The return value includes the viewer URL.

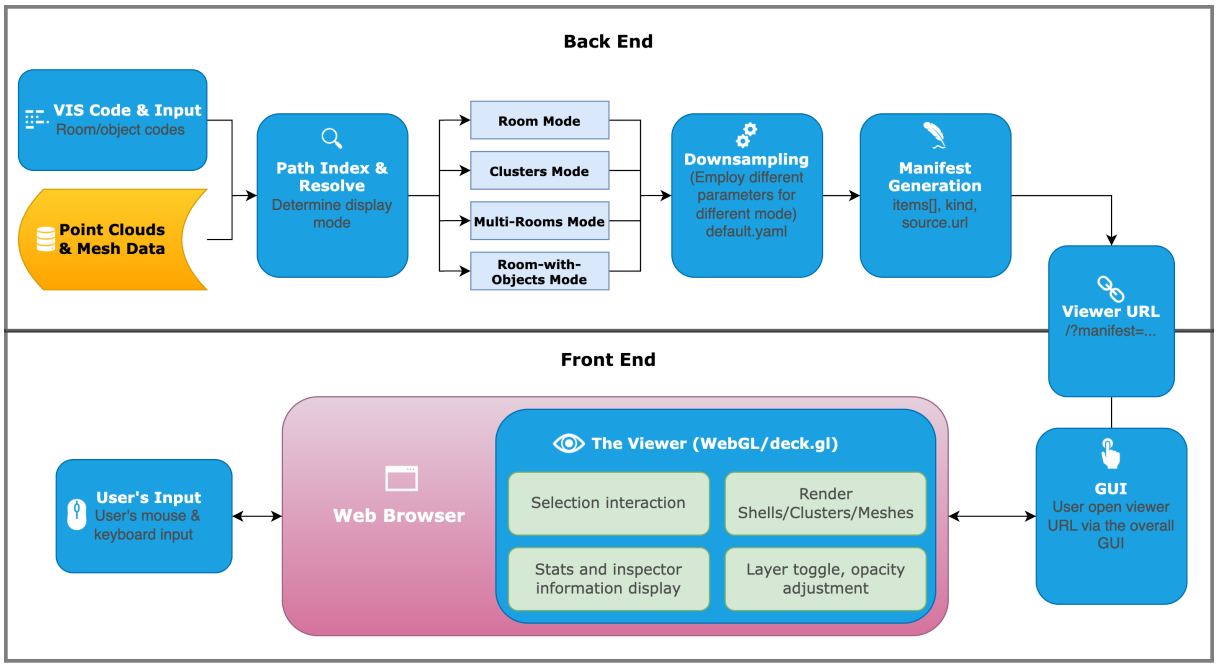


Figure 3.24: The VIS pipeline prepares viewer inputs by resolving paths, downsampling assets, generating a unified manifest, and returning a URL to launch the WebGL viewer.

Environment, Health, and Climate-line interface

A lightweight CLI provides discovery and diagnostics. JSON output can be enabled via config; resolvers and ops are exposed as subcommands.

Error Handling

All head-code entry points validate inputs (e.g., exactly one of object code or filename), produce clear errors when executables are missing, and guard text parsing with JSON-first fallbacks. For multi-candidate assets the selector prefers canonical locations (e.g., `filtered_clusters/` for clusters and `recon/` for meshes).

3.8.3 Point Cloud Viewer

The viewer (`web/pointcloud-viewer`) is a React application using deck.gl vis.gl contributors, 2025 for rendering. It consumes a unified manifest and renders point clouds and meshes with grouping, toggles, and selection (refer to Wang, 2019 for a system overview). A small HTTP API provides path confirmations and file downloads for interaction.

What the Viewer Shows

- **Shells and clusters:** room-level shells (optionally gray) and per-object clusters.
- **Meshes:** UOBBs or reconstructed meshes overlayed with configurable opacity.
- **Layer panel:** per-item visibility toggles, point size, and mesh opacity.
- **Inspector:** selection list with names, roles, object codes, and clicked-point attributes when available.

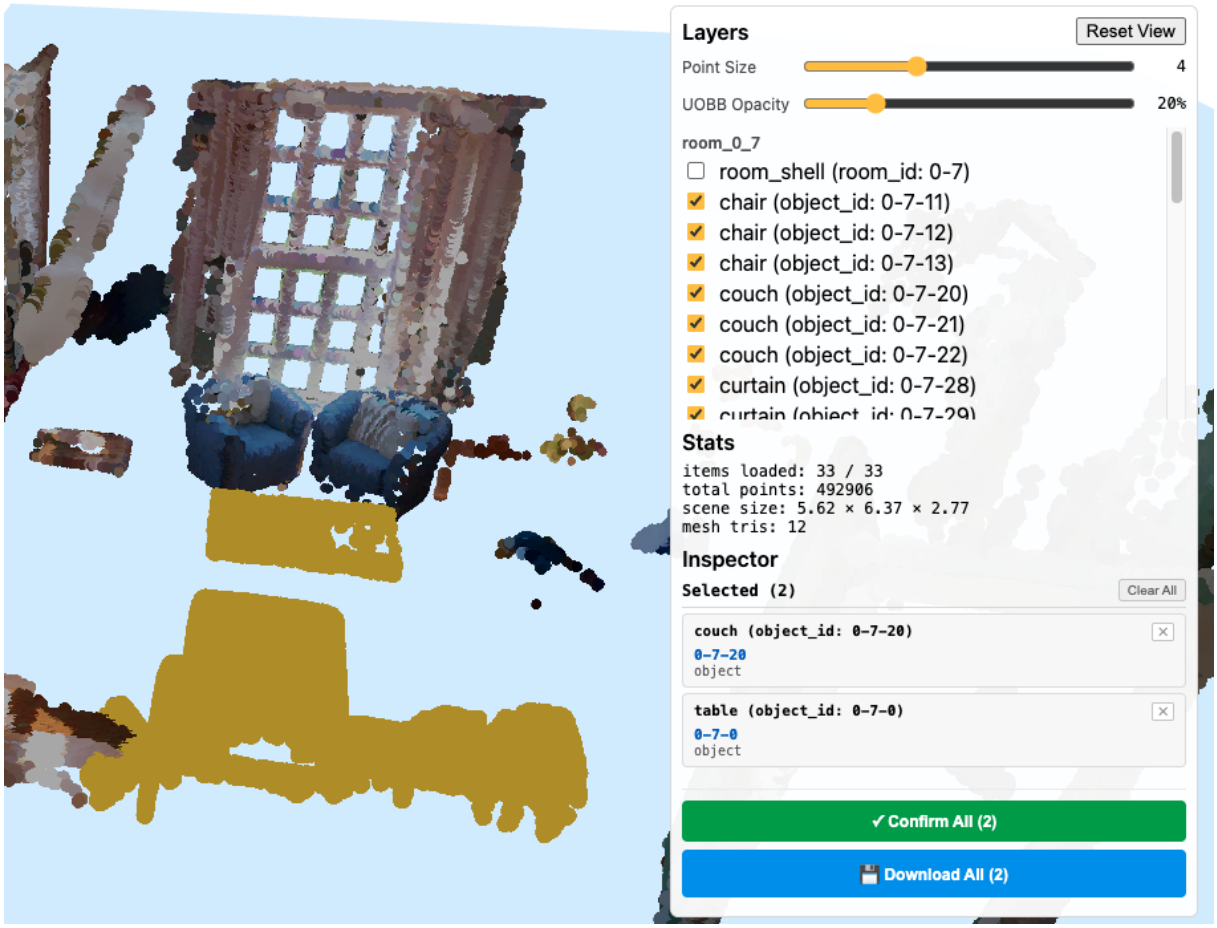


Figure 3.25: The WebGL viewer showing shells, clusters, and meshes with layer controls, selection highlighting, and an inspector panel exposing object codes and sources

Manifest and Loading

The viewer expects a unified manifest with an `items` array representing shells, clusters, or meshes. Each item has an `id`, `name`, `kind` (`pointcloud` or `mesh`), a `source.url`, and optional `group`, `style`, and `filters`. Typical usage passes a manifest via query parameter: `/?manifest=room_007.json`.

```
{
  "version": 1,
  "items": [
    { "id": "shell-room_007", "name": "Room 007 shell", "kind": "pointcloud",
      "role": "shell", "source": { "url": "/data/room_007/shell.ply" },
      "style": { "colorMode": "constant", "color": [180,180,180] } },
    { "id": "cluster-0-7-12_couch_cluster", "name": "couch (object_id: 0-7-12)",
      "kind": "pointcloud", "role": "object",
      "source": { "url": "/data/room_007/clusters/0-7-12_couch_cluster.ply" },
      "group": "room_007" }
  ]
}
```

Interaction and Selection

Clicking any point selects the entire point cloud it belongs to; selected clouds are highlighted in bright yellow. Multiple selections are supported (toggle by re-clicking an item). The Inspector shows: name, role, object code (parsed from display text or URL), and attributes of the clicked point such as `point_id`

or `label` when present. A batch action confirms selections against the backend and optionally posts them to the API server.

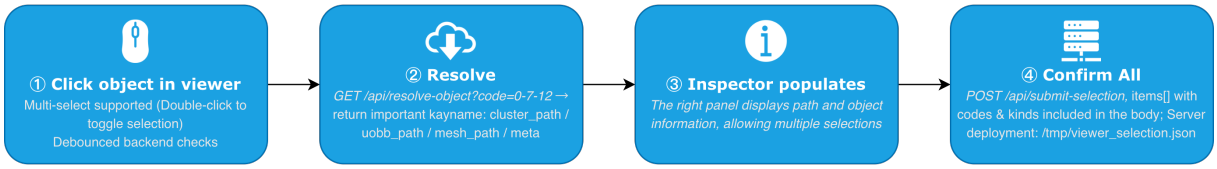


Figure 3.26: Selection Workflow—Click to select, resolve via the API, populate the inspector, and submit all selections back to the backend as structured JSON.

Backend Integration

The viewer communicates with the API server for path confirmations and downloads:

- **Resolve object**: `GET /api/resolve-object?code=object_code` → returns cluster, UOBB, mesh paths and CSV metadata.
- **Resolve room**: `GET /api/resolve-room?code=room_code` → returns shell paths, CSV, and cluster directory.
- **Submit selection**: `POST /api/submit-selection` with an array of items to emit real-time logs and persist a `/tmp/viewer_selection.json` summary for agents.
- **Download**: `GET /api/download-file?path=file` streams source files with simple in-repo path sandboxing.

Visualization Modes and Preparation

Four modes define how data is prepared for the viewer: *room* (shell + all clusters), *clusters* (selected objects), *multi-rooms* (multiple shells only), and *room-with-objects* (one shell with selected objects). The preparation script resolves paths via the same AI API, downsamples shells/clusters (ratio and optional voxel), creates the manifest, and prints the manifest URL. The AI API can trigger this end-to-end via `VIS`.

Performance and Ergonomics

Large rooms benefit from aggressive shell downsampling and optional color stripping; point size and mesh opacity are adjustable at runtime. The app logs bounds, item counts, and first-sample diagnostics to the console to aid troubleshooting.

3.8.4 Putting it Together

A typical session: resolve object codes into source files; reconstruct meshes as needed; measure area/volume; then prepare a focused visualization to inspect results and confirm object identities. Selections made in the viewer feed back to the backend as structured JSON, ready to seed subsequent operations (e.g., batch volume computation over confirmed items).

3.9 Accuracy

3.9.1 Geometric Functions

The accuracy of geometric-function results largely depends on the intrinsic quality of the point-cloud data, especially point-cloud scanning quality, class-label quality, and point-cloud segmentation quality.

Point-Cloud Scanning Quality Incomplete coverage (holes/shadows) tears open reconstructed surfaces. Non-uniform sampling (very dense in some areas, sparse in others) skews normals and plane fitting: sparse regions flatten or drift with noise, while dense regions get ripples, inflating the area. Noise and outliers enlarge UOBB and introduce spikes in AF Reconstruction/Poisson, yielding artificially high area and unstable volume. If multi-scan registration is biased, walls "double," planes split, and aggregated attributes become distorted.

Class-Label Quality Class labels determine which reconstruction strategy and parameters you choose. Mislabeling "clutters" as "walls/ceilings" may bypass a object reconstruction in geometry pipeline to room reconstruction, which blurs edges and alters both area and volume. Conversely, treating thin-shell objects (sofas/curtains) as planar can over-simplify thickness and underestimate volume. Poor representativeness (taxonomy misaligned with the scene or heavy tail classes merged away) biases class-level summaries (e.g., "total wall area," "furniture volume"). Low consistency (the same object labeled differently across samples) makes priors and parameters (thickness, closure policy, hole tolerance) conflict across batches, enlarging variance in geometric metrics.

Point-Cloud Segmentation Quality Semantic/instance segmentation is the "entry file" that decides which points are fed to geometry. Over-segmentation splits one object into many pieces, causing UOBB, area, and volume to be double-counted. Under-segmentation/leakage welds neighboring objects together, creating cross-object patches (e.g., wall plus cabinet filled as one surface) and overestimating area/volume. Imprecise boundaries also mislead hole filling: door openings may be filled as noise, inflating room volume. For Poisson, such errors cause normal estimation neighborhoods to cross classes, inducing flips and self-intersections. In short, the closer segmentation aligns with real object boundaries, the closer reconstruction and downstream metrics are to ground truth and the more stable they become.

3.9.2 AI Agent

Multiple user queries have been formed to give an indication of the performance of the application. The user queries and complete test results are shown in the following Table 3.9. The different user queries cover different aspect which the application should be able to handle, going from basic to more advanced.

The results demonstrate that the agent successfully integrates its core components—scope classification, database retrieval, tool execution, and response synthesis—to answer a wide range of spatial and semantic questions.

Table 3.9: Analysis of Agent Test Query Results

Test Query	Capability Tested	Agent Response (Summarized)	Analysis & Confidence
Category 1: Explicit Tool Calls (ACI Validation)			

Continued on next page...

Table 3.9 – continued from previous page

Test Query	Capability Tested	Agent Response (Summarized)	Analysis & Confidence
"What is the dominant color of the couch 0-3-0?"	Explicit `CLR` Tool	"The couch has two dominant colors. Primary: medium-dark gray (57%)..."	**Success.** Agent ran `CLR` and interpreted RGB values. **Confidence: High** (API data).
"What is the distance between chair 0-2-4 and shell 0-2-3?"	Explicit `BBD` Tool	"Distance: 24.11 meters. Vector: (0.355, -24.093, 0.708)."	**Success.** Correct `BBD` execution with explicit codes. **Confidence: High** (API data).
"Show me the couch 0-3-0"	Explicit `VIS` Tool	"I've prepared a 3D visualization of the couch..."	**Success.** Agent identified `VIS` tool and parsed object code.
Category 2: NLP-Driven Tool Calls (Single-Room)			
"What is the distance between the chair and the door in the kitchen?"	Semantic Room ID + NLP `BBD`	"Distance between chair (0-2-4) and door (0-2-7): 0.93 meters."	**Success.** Identified kitchen, used NLP for objects, ran `BBD`. **Confidence: High** (API data).
"What is the volume of the table in room 4 floor 1?"	Semantic Room ID + NLP `VOL`	"Both tables have volume 0.000m³ (unclosed mesh)."	**Success.** Found 2 tables, ran `VOL`, reported error correctly. **Confidence: High** (API data).
"Visualize the chairs in room 2"	NLP `VIS` + Fallback	"Visualizing kitchen (room 0-2) with all objects."	**Partial Success.** NLP failed for chairs, fell back to room visualization.
Category 3: Database Reasoning (Multi-Room Scope)			
"How many chairs are in the kitchen?"	Semantic DB Query (Count)	"Kitchen (room_002): 1 chair total."	**Success.** Correct scope, identified kitchen, counted from DB. **Confidence: High** (DB data).
"List the tables in the bedroom."	Semantic DB Query (List)	"Bedroom (room_004): 2 tables (1-4-1, 1-4-2)."	**Success.** Identified bedroom, listed matching tables. **Confidence: High** (DB data).
"which room has the highest width of the chair"	Multi-Room DB Aggregation	"Room 003 (hallway): Chair 0-3-7, Width = 0.71m."	**Success.** Multi-room scope, find-maximum aggregation. **Confidence: High** (DB data).

Continued on next page...

Table 3.9 – continued from previous page

Test Query	Capability Tested	Agent Response (Summarized)	Analysis & Confidence
"Which room has the largest floor area?"	Multi-Room DB Aggregation	"Room_002 (kitchen): Floor Area: 58.93m²"	**Success.** Queried planes table, compared floor areas. **Confidence: High** (DB data).
"List all rooms with floor area > 20m²"	Multi-Room DB Filter	"6 rooms meet criteria: [list of rooms]"	**Success.** Applied numerical filter across all rooms. **Confidence: High** (DB data).

3.9.3 Discussion on Accuracy and Confidence

The accuracy of these answers is fundamentally tied to the data source. The agent is designed to be transparent about its reasoning process.

- **High Confidence:** Whenever the agent pulls values directly from the **database** (such as pre-calculated **areas**, object **dimensions**, or object **counts**) or from a successful **real-time tool call** (like `BBD` for distance, `CLR` for color, or `VOL` for volume), the response is considered to have high confidence. These numbers come from verified data sources or calculations.
- **Low Confidence:** In cases where the agent might have to make an estimation beyond what is explicitly stored or calculated (e.g., interpreting complex spatial relationships not covered by a tool), it would mark the response with low confidence.

This approach ensures it is always clear to the user when the system is providing a solid, data-backed answer versus when it is reasoning on its own. In the tests performed, all successful answers were data-backed (either from the database or an API call) and thus fall into the **High Confidence** category.

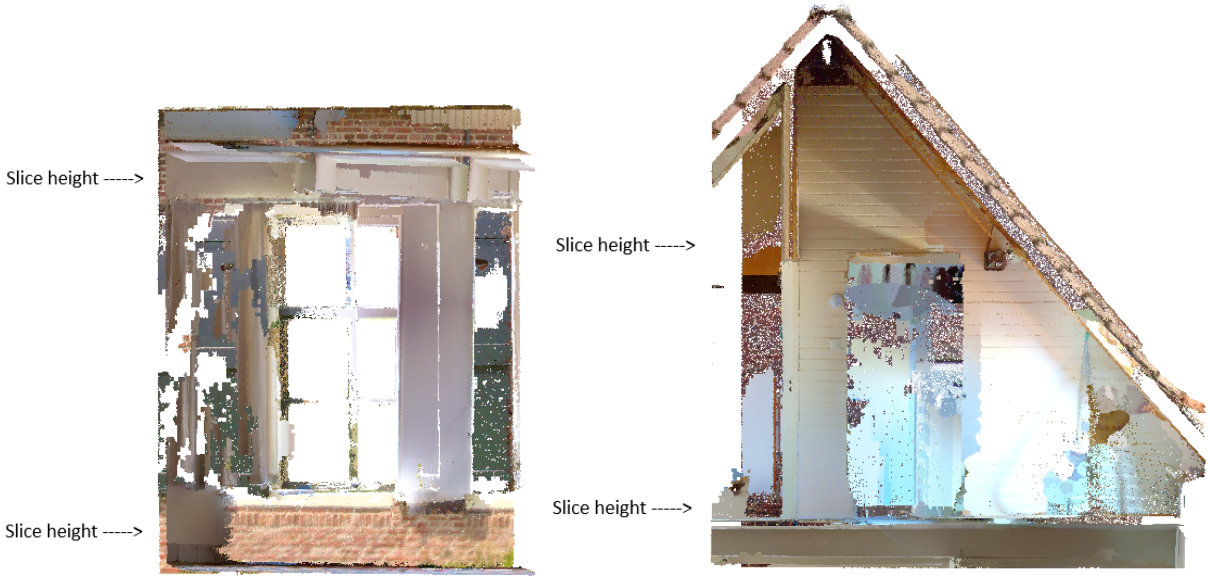
4 Discussion

This chapter reflects on the outcomes of the study, examining the strengths, limitations, and development challenges. It provides a critical analysis of the methods used. The discussion aims to place the results, Chapter 3, in context and provide insights that can guide further development and refinement of the system; this will be described in Chapter 6.

4.1 Room Segmentation

A key challenge is room segmentation from point clouds, which remains an unsolved problem but is crucial for the agent's functionality. Achieving this requires a reliable method of structuring and linking spatial information. Although we have studied many possible implementations, a practical solution has not yet been found; hence, we have implemented our own version with specific tradeoffs.

The room segmentation that we implemented has some flaws; most importantly,it needs manual intervention to set the correct variables for room slicing. In theory, if the parameters are set correctly and the data is not occluded, the room segmentation should be able to find the correct floor plan and have a good room extraction. However, since every dataset is different, it is not possible to use the same parameter values for each case. For example, the idea of setting the slice heights is that it should be above the floor level but below a window and that it should be below the ceiling but above a window or a door, like in Figure 4.1a. This approach works in the case of the house dataset but does not work for the TU Twente dataset, as the windows go all the way down to the floor. Other structural features like a pitched roof, beams through the ceiling, and irregular shapes in the wall also heavily influence the outcome results. Besides structural features it is crucial that the 3D dataset contains all wall points and are not occluded or obstructed by objects. The result of this can be seen in the difference in floor plans of the ground floor and first floor, where the mostly the first floor has a lot of objects that obstruct the LiDAR scan from wall points.



(a) Kitchen

(b) Room from upper floor

Figure 4.1: Ideal slicing heights

Not only do slicing heights influence the outcome, but also the grid size of the 2D floor plan for the different thicknesses of walls and the expanding distance for good room extraction. The same holds for

open spaces in buildings; the height measurements get one height value per floor and use that for the whole surface of the floor; hence, it does not account for open spaces that span multiple floors. This can clearly be seen in the TU Twente dataset (Figure 3.5), where the room extraction of the assembly hall only consists of the space located between the columns and beams. The actual assembly hall can be considered more open space and is adjacent to multiple hallways. Defining a room is undeniably tricky, and multiple approaches can be considered valid. The results we have produced can therefore be considered good in some cases, but also bad in others.

4.2 Room Reconstruction

The PolyFit reconstruction method performs well when the point cloud provides sufficient coverage, meaning that most room surfaces are supported by enough data points. However, a major limitation arose when this was not the case, particularly for rooms on the first floor, resulting in incomplete or highly inaccurate reconstructions compared to the ground truth.

Two main factors contributed to this lack of coverage. First, the point cloud of the first floor suffered from heavy occlusion due to the large number of objects present. Including these objects in the PolyFit reconstruction process might partially mitigate this issue by generating additional supporting surfaces, although the reconstructed room size would still likely be underestimated. Second, inaccuracies were already introduced during the room segmentation stage. When rooms were incorrectly segmented, this created inconsistencies between the reconstructed and actual room boundaries, leading to large overall errors. Alternative meshing strategies could be explored to address these challenges, focusing on methods that handle missing planes more robustly while still producing watertight meshes. In contrast, the ground floor rooms were simpler and more spacious, resulting in more successful reconstructions.

Another limitation concerns the tuning of hyperparameters. These were adjusted through experimentation, but the number of possible combinations between segmentation, plane detection, and PolyFit parameters is extensive. The current implementation uses static parameter values that do not adapt to the density or size of the input data. Introducing adaptive parameter tuning could further enhance reconstruction accuracy (using point cloud size for instance). However, as the primary goal of this project was to develop and test a wide range of functionalities within limited time, such refinements were beyond its scope.

Our current results are based on a single dataset. To better evaluate the robustness and general applicability of the method, it would be valuable to test the approach on a wider range of building types. This would help identify both its strengths and limitations under different spatial and structural conditions.

4.3 Geometric Functions

4.3.1 Clustering

Overall, for rooms with complete coverage and balanced object sizes, and with adequate spacing between objects, the algorithm performs reliably and exhibits strong robustness. It also achieves a favorable trade-off between processing time and result quality. Nevertheless, we note that instance segmentation could, in principle, be handled by models with spatial understanding—e.g., *SpatialLM* proposed by Mao et al., 2025—which may yield even better results.

4.3.2 Color

Overall, the test suite confirms that `pcg_color` provides a reliable foundation for color-based point cloud analysis with robust noise filtering and high precision in cluster parameter estimation. However, it is worth noting that for multi-colored cases, the analysis results may still exhibit merging behavior due to parameter settings. Future work could focus on enabling adaptive standard deviation threshold parameter configurations.

4.3.3 Reconstruction

The reconstruction pipeline demonstrates reliable performance across diverse geometries. The improved volume and area calculation methods, particularly the adaptive voxel approach for non-closed meshes, enable accurate quality assessment. Future work should investigate the reconstruction quality of building0 and cube to understand the source of systematic volume underestimation, and validate the pipeline on larger datasets to ensure robustness for complex real-world scenarios.

4.3.4 Surface Area and Volume

The validation testing confirms that the implemented geometric calculation methods meet their design objectives. Triangulation preprocessing ensures accurate handling of polygon faces, while the signed volume method provides exact results for polyhedra and near-exact results for curved surfaces. The adaptive voxel method delivers consistent 5% accuracy for non-closed meshes, enabling universal applicability. Both methods demonstrate stable, predictable behavior suitable for automated quality assessment workflows in this project.

4.4 AI Agent

While the current agent architecture successfully validates the core concepts of integrating a spatial database and computational tools with an LLM, its design presents several significant scalability bottlenecks.

4.4.1 Static, Batch-Oriented Data Ingestion

A significant bottleneck to scalability is the system's reliance on a static, batch-oriented data ingestion model. The agent's entire knowledge base (the SQLite database) is generated in a single operation by a script. This script is not part of the live agent system but rather a prerequisite step that must be run offline to create the data store.

This architecture presents a critical bottleneck for a real-world application:

1. **Data Immutability:** The database is effectively immutable from the agent's perspective. To incorporate new data, such as a recently scanned building or even a single modified room, the existing `spatial_rooms.db` file must be discarded and the entire ingestion script must be re-run from scratch. There is no mechanism for incremental or live updates, making it unsuitable for a dynamic production environment.

4.4.2 Query-Time Context Limitation

The agent's `MULTI_ROOM` reasoning capability, while effective for the 10-room test environment, is not scalable. As demonstrated in the test results (Section 1), a query like "*Which room has the largest floor area?*" is answered by loading a text summary of all 10 rooms into the LLM's context window.

This approach has a scaling complexity of O(N), where N is the total number of rooms in the database. This will fail completely as the database grows from 10 rooms to 100 or 1,000 rooms, as the required context will quickly exceed the token limit (e.g., 128k tokens) of even the most advanced models. This is a classic problem in long-context reasoning that RAG was designed to solve (Lewis et al., 2020).

4.4.3 Computational Latency Bottlenecks

The agent's architecture introduces significant performance bottlenecks related to how it retrieves and calculates data in real-time. This latency is caused by two distinct but related design choices: the reliance on on-demand calculations for common properties and the sequential execution of those calculations.

Real-time vs. Pre-computed Data

A core design decision in the agent's architecture is the balance between pre-computed data and real-time (on-demand) calculations. This trade-off directly impacts the system's performance and flexibility.

- **Pre-computed Data (High Speed, Static):** Geometric properties that are fundamental to the scene, such as the area of planar surfaces (floors, walls) and the bounding box dimensions of objects, are calculated once during the offline data ingestion pipeline. This data is stored directly in the structured query language lite (SQLite) database. As noted in the test results, this is a highly effective strategy for queries like "What is the floor area?" as the agent can retrieve the answer almost instantly. This approach scales extremely well for read-heavy operations, as the cost of calculation is paid only once. The trade-off is data immutability, as discussed previously.
- **Real-time Calculation (High Accuracy, High Latency):** More complex properties, such as the precise mesh volume (`VOL`), dominant color (`CLR`), or B-Box distance (`BBD`), are not pre-computed. Instead, the agent calls these tools via the API wrapper when the user asks. This design ensures that the calculation is performed on demand using the most accurate C++ binaries. However, this model does not scale well in terms of response time. This introduces significant latency, especially when calls are executed sequentially.

This hybrid approach was chosen as a pragmatic balance for the prototype. However, the reliance on real-time calls for common properties like volume is a key scalability bottleneck. A future, production-grade system would benefit from pre-calculating and caching these intensive values during the ingestion phase, storing them as new columns in the `objects` table to gain the high-speed, scalable query performance of the pre-computed model.

4.4.4 Sequential Tool Execution

The latency from real-time calculation is compounded by the agent's sequential execution model. As defined in the agent's logic, it processes multiple tool calls in a strictly sequential manner. When a query requires several independent calculations (e.g., fetching the volume for multiple objects), the agent's logic iterates through each target, making one API call and waiting for it to complete before initiating the next.

For example, a user query for the volume of two separate tables ("1-4-1" and "1-4-2") results in a blocking, sequential chain of operations:

1. Call API for `VOL("1-4-1")` → **Wait** for completion.
2. Call API for `VOL("1-4-2")` → **Wait** for completion.

This design means the total response time is the *sum* of all individual tool latencies. A query requiring ten volume calculations would take ten times as long as a single query, even if the calculations themselves are independent and could be run in parallel. This sequential execution model is highly inefficient and represents a significant latency bottleneck for complex user requests.

4.5 2D to 3D

4.5.1 YOLO Detection Performance

In our panorama-driven indoor setting, a detection-first pipeline with you only look once, version 11 (YOLOv11) proved unreliable. Despite tiling with overlap, high-distortion regions and tile boundaries produced fragmented or missed boxes, and duplicates arose both within single panoramas and across multiple panoramas of the same room. These 2D errors propagated into 3D, yielding unstable rays and misleading instance counts that undermined spatial reasoning.

The common objects in context (COCO) class set further limited utility by omitting building-critical elements such as doors and windows, while indoor panoramas' reflections, clutter, and occlusions exacerbated domain shift. Addressing these issues would require substantial engineering (tile-aware merging, cross-view deduplication, panoramic augmentation, and custom classes) with uncertain payoff. We therefore adopted a mask to 3D projection with clustering, which, after projecting the user-selected mask and filtering candidate points, uses fast Euclidean pre-clustering and pose/reprojection-driven selection to produce coherent, room-aware 3D objects.

4.5.2 SAM Masking and Filtered 3D Points

The evaluation of the SAM2 masking and subsequent 3D point filtering in Section 3.7.1, highlights both the strengths and limitations of the current approach in indoor environments. The results demonstrate that SAM2 effectively identifies and transfers many object-level masks into the 3D domain, particularly for objects with planar, well-defined geometries such as doors, windows, and refrigerators. these classes benefit from strong visual boundaries and consistent textures, producing dense and coherent 3D point selections after back-projection. This reliability is important for spatial reasoning tasks, as it ensures a stable correspondence between a 2D panoramic image and 3D geometry.

However, the results also show that segmentation quality varies substantially by object material, and geometry. Large, homogeneous, or reflective surfaces-such as countertops and microwaves-produce weaker masks, reflecting SAM2's sensitivity to low texture contrast and specular reflections. Despite this, the 3D projection often compensates for incomplete or noisy masks, successfully localizing a target object through geometric continuity in the point cloud.

The cases of the couch, drawer, and radiator further illustrate the interaction between occlusion, material properties, and point cloud density. Occluded or partially visible objects lead to incomplete masks, yet back-projection can still recover their approximate 3D position. conversely, radiators show the limits of this approach-precise masks alone do not guarantee complete 3D reconstruction when LiDAR sampling is sparse or irregular. These outcomes emphasize the dependence of SAM2-based selection on the quality and density of the underlying 3D data.

Ultimately, this stage of masking and 3D filtering serves as a foundation for enabling users to query and reason about objects that are not explicitly classified by ScanPlan's segmentation algorithm. In the end, if the link between the natural language input with geometric, and visual features is made, the users should be able to ask questions about it. However, when data quality is weak-due to missing points, low-resolution imagery, or unreliable masks-these queries become significantly more difficult to resolve. Robust performance will therefore depend on improving segmentation consistency, refining mask boundaries, and ensuring sufficient spatial coverage in the point clouds.

4.6 Parameters

We set the goal to make the application fully automated, but we did not achieve this feat. Hence we introduce this section to show all the parameters and their effects that need to be adjusted per dataset. Tables are shown per sub task of the application

Parameter name	Ideal value	Effect of parameter
Room segmentation ground floor: `segment_rooms_mark.py`		
parser function		
`voxel`	0.15 m	Voxel downsample size in meters.
`grid`	0.10	Grid resolution in meters per pixel.
`min-room-area`	1.0	Minimum room area (m^2) to keep.
`expand_dist`	0.90	Expand distance of walls in meters.
calling floor slicer function		
`slice_min`	`ground_floor_level[0] + 0.18`	Lower boundary for ground floor slicing height.
`slice_max`	`ceiling_level[0] + 0.45`	Upper boundary for ground floor slicing height.
Room segmentation attic: `segment_rooms_mark.py`		
parser function		
`voxel`	0.10 m	Voxel downsample size in meters.
`grid`	0.24	Grid resolution in meters per pixel.
`min-room-area`	1.0	Minimum room area (m^2) to keep.
`expand_dist`	0.90	Expand distance of walls in meters.
calling floor slicer function		
`slice_min`	`ceiling_level[i] + 0.125`	Lower boundary for the attic slicing height.
`slice_max`	`ceiling_level[i] + 1.10`	Upper boundary for the attic slicing height.

Table 4.1: Adjustable parameters ordered per sub task

5 Conclusion

To summarize, the application uses a 3D point cloud and its panoramic images to extract various amounts of geometries down to object level. Using these object the application is able to measure and calculate statistics which are linked to the objects and their spatial relationship. The information is stored in a database, which is ready to use for the ai agent. Subsequently, the user can ask questions relevant to the building. Results are presented via the chat function or via either the 2D or 3D viewer.

The main objective of the project is to integrate a multi modal LLM with classified input data (images and LiDAR scans) and bridge the gap between natural-language and spatial reasoning. While the application is able to successfully analyze a 3D environment with numerous complex objects and reason about their characteristics through a LLM, there still remain some issues.

A persistent challenge lies in assessing the reliability of the reconstructed output. Although the results are primarily based on data collected during the preprocessing stage, and the system indicates the sources from which this information was retrieved, the user has no direct way of determining whether the original reconstruction itself was accurate or contained underlying errors. This lack of transparency in data quality makes it difficult for users to fully trust or verify the correctness of the final output.

Furthermore, it is difficult to anticipate the range of questions a user might ask and to assess how effectively the language model can respond to them. Certain user needs may not be accounted for, such as tasks related to indoor wayfinding or spatial reasoning, which require context beyond the current system's capabilities.

Another challenge is inherent to point cloud data itself: occlusion. Missing or obstructed areas in the scan often lead to incomplete or distorted representations, which can introduce significant errors during preprocessing and subsequently affect the quality of the reconstruction and analysis.

Then for the biggest challenge for this project, which is the room segmentation and room reconstruction. The room segmentation does bridge the gap between 3D scans and accessible queries, nevertheless parameter tuning is still needed for accurate results. Reconstructing rooms in a sizable building is currently not accurate, resulting in not being able to scale the application for the industry. The pipeline eases this problem, allowing for an easy plug and play solution. Finding an alternative and more successful solution to the room segmentation problem in the future for example can be smoothly implemented into the pipeline.

The current solution has mostly been tested on the house dataset and a little less on the TU Twente dataset, therefore we cannot say that it accurately represents the value of the application. One of the bottlenecks we believe for increasing the size of the dataset is the calculating time. Real-time calls for volume measurements for example will take a long time and would benefit to be calculated in advance.

On the contrary, the application has succeeded in converting natural language into actionable data queries. The AI agent converts the natural language to tasks and unique codes the application is able to understand, unlocking the benefits of the spatial data and increasing the efficiency of the whole process.

Every user has different needs and thinks on different scales; hence, the application is able to interpret diverse user prompts and extract relevant information from the multi-modal input datasets. Panoramic images and classified 3D point clouds are dismantled to extract necessary information, which has reduced manual interpretation of the given building. The use of the panoramic images is seen via the graphical interface, making it easy for non-expert users to get acquainted with the building and specific questions. In combination with the 3D viewer, the application is user-friendly and has addressed the gap between advanced spatial data and accessible queries.

All in all, the application is user-friendly and allows users with no experience to ask relevant and reasonable questions on low-scale buildings. Even though the application does not support large-scale buildings, it can easily be addressed in the future if a sound solution has been found for segmenting

rooms. Hence, the first step towards bridging natural language queries and spatial reasoning has been made.

6 Future Work

This chapter outlines potential directions to improve and extend the final product in the future, including the same topics presented in the previous chapters.

6.1 Room Segmentation

As discussed in the previous chapters the current implementation of the room segmentation can be rather tedious and building specific. Room segmentation is the hard part to implement, yet the most crucial for good results. The current implementation sadly does not yield good results, therefore we recommend trying another approach that is more flexible and robust in different scenarios. If however we would have to improve the current solution then we suggest to train the agent or use the LLM to set the necessary parameters based on visual interpretation. This solution would not address all the problems like open spaces, but can greatly improve results on various buildings.

6.2 Room Reconstruction

As stated in the discussion, there should be a focus next on improving the robustness and adaptability of the reconstruction pipeline. One important direction is to enhance the coverage and completeness of point clouds, particularly in complex environments where occlusion is common. This could involve incorporating detected objects into the PolyFit reconstruction to generate additional supporting surfaces or experimenting with alternative meshing strategies that can handle missing planes while maintaining watertightness.

Another promising direction is the development of adaptive parameter tuning. In the current implementation, parameters for segmentation, plane detection, and PolyFit are static and manually optimized through experimentation. Future work could explore data-driven or dynamic parameter adjustment methods that adapt to factors such as point cloud density, noise level, or room size, thereby improving reconstruction accuracy and consistency.

In addition, the evaluation scope should be expanded beyond a single dataset. Applying the method to a variety of building types—ranging from small residential units to large, complex structures—would provide deeper insight into its strengths and weaknesses. Such testing would also support a more systematic validation of the approach's generalizability across different architectural typologies and data qualities.

6.3 Geometric Functions and the AI API

For the *Geometric Functions* and the AI-facing API, future work will focus on the following aspects.

6.3.1 Improving Robustness

Across the current pipeline—spanning clustering and cluster filtering, surface reconstruction, color analysis, and the computation of volume and surface area—failures can arise due to geometric complexity, noise, or missing data. This issue is particularly acute for the algorithms used to compute volume and surface area. In future work, it is necessary to refine these algorithms and validate them on larger and more diverse datasets.

6.3.2 Code Cleanup and Organization

The present *Geometric Functions* GitHub repository remains cluttered, with uncleaned or redundant code and a suboptimal project structure. A key priority is to reorganize the codebase and converge toward a releasable version with a clear, maintainable architecture.

6.3.3 Enhancing AI Friendliness

At the moment, the AI API can only barely support simple geometry operations in coordination with an AI chatbot; interactive features on the visualization side (the viewer) are not yet integrated. Future work should enable the chatbot, via the AI API, to listen to user actions within the viewer and to receive structured JSON feedback from these interactions.

In addition, the usage patterns of several operations are not fully captured in the current prompts. A more systematic and comprehensive prompt set will be necessary. We also plan to provide a built-in `-help` facility for the AI API so that the chatbot can understand the context of each function and proactively ask clarifying questions when needed.

6.4 AI Agent

The current agent prototype provides a strong proof-of-concept but also highlights several key areas for future development. Current limitations include static data loading, basic image analysis without spatial mapping between 2D detections and 3D objects, and limited contextual memory in interactive mode. This prevents conversational follow-ups (e.g., "*what is its color?*"), a known limitation of simple "Thought-Action" loops which are based on frameworks like ReAct Yao et al., 2022.

These constraints highlight that while the agent can successfully process both single-room and multi-room queries, its underlying architecture is not optimized for scalability. Scaling the system from a 10-room demo to a production environment with thousands of rooms will require addressing these data handling and architectural bottlenecks.

To address these limitations, the following enhancements are proposed:

1. **Dynamic Data Pipeline & Scalable RAG:** To solve the data ingestion and context window bottlenecks, the static SQLite database should be replaced. A production-grade spatial database (e.g., PostgreSQL with PostGIS) would allow for a dynamic pipeline where new scans can be processed and added incrementally.

   This would be paired with a true RAG system (Lewis et al., 2020). Instead of loading all rooms, the agent would first perform a "filtering query" on the database (e.g., `SELECT room_id, floor_area FROM rooms WHERE room_type = 'bedroom'`) and load *only* the results of that query into the LLM's context. This two-step retrieval process would allow the agent to scale to thousands of rooms without exceeding token limits.

2. **Advanced 2D-to-3D Multimodal Mapping:** The current script to give semantics to a room based on the panoramas only performs high-level image classification (e.g., this room is a "kitchen"). The next generation of this agent should implement a true 2D-to-3D mapping. This pipeline would:

   - Run advanced segmentation models like the Segment Anything models, SAM/SAM2 (Kirillov et al., 2023; Ravi et al., 2024) on 2D panorama images to detect and mask individual objects (e.g., "chair," "table").
   - Use the scan's camera pose data to project these 2D masks into the 3D point cloud.
   - This programmatic 2D-to-3D labeling would be a significant step towards a true 3D-LLM (Hong et al., 2023a). This process would automatically and accurately assign semantic labels to the geometric clusters in the `objects` table (e.g., `0-7-12` would be programmatically identified as a "chair"). This enhancement would automate the most time-consuming part of the data-labeling process and vastly improve the agent's semantic understanding.

3. **Conversational Memory:** To overcome the limited contextual memory, a persistent chat history should be implemented. In this model, the conversation history (e.g., the last 3 query-response pairs) would be appended to subsequent prompts. This would enable the agent to handle anaphoric references and conversational follow-ups (e.g., Query 1: "What is the volume of object 0-7-12?", Query 2: "What is *its* dominant color?").
4. **Parallel Tool Execution:** To resolve the sequential execution bottleneck, the agent's tool-calling logic should be re-architected. Instead of iterating and calling the API for each object one-by-one, the agent should first identify all necessary tool calls. It could then execute these calls in parallel (e.g., using an asynchronous `asyncio` event loop or a thread pool). This would change the total latency from a sum of all calls to being dependent only on the *longest* individual call, dramatically improving performance for complex queries.

6.5 2D SAM2 Masking to 3D Filtered Points

Future improvements could focus on enhancing the robustness and accuracy of SAM2-based 3D filtering in indoor environments. Enforcing multi-view consistency by using masks from overlapping images could mitigate occlusions and incomplete regions, while incorporating material- and texture-aware cues such as surface normals or reflectance priors could improve segmentation of low-texture or reflective surfaces. Geometry-constrained projection methods, such as plane fitting or clustering in the 3D point cloud, could help maintain mask completeness and alignment with underlying structures. Confidence-weighted back-projection and point cloud densification techniques could further reduce noise and compensate for sparse LiDAR sampling. Additionally, fine-tuning SAM2 on indoor datasets and integrating 2D-3D learning frameworks may enable more reliable prediction of object-level 3D masks, ultimately improving spatial reasoning and interaction with complex indoor scenes.

6.6 User Interface

The user interface can be significantly enhanced by integrating the 3D viewer with a GUI that leverages an AIagent and 2D-3D projection capabilities. This combination will allow users to interact more intuitively with complex 3D scenes, query object using natural language, and receive correct answers. By linking 2D visual information with 3D geometry, the interface can provide more precise object selection, improved visualization, and streamlined spatial reasoning. This all, will contribute to a more accessible, intelligent user experience.

A Appendix

A.1 Segmented classes

```
self.label_to_names = {
    0: "unclassified",
    1: "ceiling",
    2: "floor",
    3: "wall",
    4: "wall_ext",  # external walls
    5: "beam",
    6: "column",
    7: "window",
    8: "door",  # door frame
    9: "door_leaf",  # the door panel both open and closed
    10: "plant",
    11: "curtain",
    12: "stairs",
    13: "clutter",  # other stuff in a house that are not really furniture, e.g.
        books, rug
    14: "noise",
    15: "person",  # people etc
    16: "kitchen_cabinet",  # both lower and upper cabinets only where kitchen
       faucet is
    17: "lamp",
    18: "bed",
    19: "table",
    20: "chair",
    21: "couch",
    22: "monitor",  # also TV's
    23: "cupboard",  # closed closets, wardrobe, cabinets etc.
    24: "shelves",  # open cabinets, shelves, kitchen shelves
    25: "builtin_cabinet",  # Not in geoslam data, classify as wall
    26: "tree",  # only trees, not low vegetations like shrubs
    27: "ground",
    28: "car",  # car, truck, etc.
    29: "grass",  # low vegetation, bush, shrubs, lawn
    30: "other",
}
```

Bibliography

Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M., & Savarese, S. (2016). 3d semantic parsing of large-scale indoor spaces. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1534–1543. https://doi.org/10.1109/CVPR.2016.170

Cao, Y., Wang, Y., Xue, Y., Zhang, H., & Lao, Y. (2022). Fec: Fast euclidean clustering for point cloud segmentation. *Drones*, *6*(11), 325. https://doi.org/10.3390/drones6110325

Cohen-Steiner, D., & Da, T. K. F. (2004). A greedy delaunay-based surface reconstruction algorithm. https://doi.org/10.1007/s00371-003-0224-7

Hong, Y., et al. (2023a). 3D-LLM: Injecting the 3D World into Large Language Models.

Hong, Y., Zhen, H., Chen, P., Zheng, S., Du, Y., Chen, Z., & Gan, C. (2023b). 3d-llm: Injecting the 3d world into large language models. https://arxiv.org/abs/2307.12981

Kazhdan, M., & Hoppe, H. (2013). Screened poisson surface reconstruction. *ACM Transactions on Graphics*, *32*(3), 1–13. https://doi.org/10.1145/2487228.2487237

Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., Dollár, P., & Girshick, R. (2023). Segment anything. *arXiv preprint arXiv:2304.02643*.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Nogueira, R., Kiela, D., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, *33*, 9459–9474.

Mao, Y., Zhong, J., Fang, C., Zheng, J., Tang, R., Zhu, H., Tan, P., & Zhou, Z. (2025). Spatiallm: Training large language models for structured indoor modeling. https://doi.org/10.48550/arXiv.2506.07491

Nan, L., & Wonka, P. (2017). Polyfit: Polygonal surface reconstruction from point clouds. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2372–2380. https://doi.org/10.1109/ICCV.2017.258

Point Cloud Library contributors. (2025). *Point cloud library (pcl)* (Version 1.15.1). https://github.com/PointCloudLibrary/pcl

Ravi, N., Mao, H., Dolhansky, B., Zhao, J., Kirillov, A., & Girshick, R. (2024). Segment anything 2: A foundation model for image and video segmentation. *arXiv preprint arXiv:2408.00714*.

Rusu, R. B., & Cousins, S. (2011). 3d is here: Point cloud library (pcl). *IEEE International Conference on Robotics and Automation (ICRA)*. https://doi.org/10.1109/ICRA.2011.5980567

The CGAL Project. (2025). *CGAL user and reference manual* (6.1). CGAL Editorial Board. https://doc.cgal.org/6.1/Manual/

vis.gl contributors. (2025). *Deck.gl: Gpu-powered webgpu/webgl2 data visualization framework* (Version 9.2.2) [MIT License]. https://github.com/visgl/deck.gl

Wang, Y. (2019). Deck.gl: Large-scale web-based visual analytics made easy. *arXiv preprint arXiv:1910.08865*. https://doi.org/10.48550/arXiv.1910.08865

Yamada, Y., Bao, Y., Lampinen, A. K., Kasai, J., & Yildirim, I. (2024). Evaluating spatial understanding of large language models. https://arxiv.org/abs/2310.14540

Yang, J., Jimenez, C. E., et al. (2024). Swe-agent: Agent-computer interfaces enable automated software. *arXiv preprint arXiv:2405.06283*.

Yao, S., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models.

Colophon

This document was typeset using LaTeX, using a modified the KOMA-Script class `scrbook` available at https://github.com/tudelft3d/msc_geomatics_thesis_template. The main font is Palatino.