Organizational Chart – An Interactive RAG

Sri Sakthi Thirumagal Poovannan

srisakth@buffalo.edu

May 2026

Abstract

This paper describes an interactive retrieval augmented generation (RAG) framework that generates a queryable knowledge graph from an image of an organizational chart. The pipeline consists of (1) a fine tuned DETR detector which identifies employee boxes on the chart, (2) EasyOCR text extraction of OCR processed text from images and contrast limited images, (3) a node assembly which combines fragmented OCR tokens within each identified box, and (4) a hierarchy building step that combines pixel level line tracing and a vision language retrieval pass to establish parent child relationships among identified employees. The finished hierarchy is embedded using a sentence transformer model and stored in a FAISS inner product index, which is made available through a Flask web application, which provides a D3 rendered tree view, a chat panel, a chart vs. chart comparison view, and a time travel reconstruction slider. The primary engineering contribution is the ability to answer questions related to the structure of the organization from the reconstructed graph, and to answer descriptive questions by semantically retrieving node text chunks from FAISS using the reconstructed hierarchy. The paper also includes important limitations of implementation, including the ambiguity of using hierarchy based strategies.

Keywords: retrieval augmented generation, organizational charts, DETR, EasyOCR, FAISS, hierarchy reconstruction, vector search, Flask, large language models

Acknowledgments

I gratefully thank my project advisors, Dr. David Doermann, for his guidance, technical feedback, and support throughout this project. I also thank the Artificial Intelligence Innovation Laboratory (A2IL) for supporting the development environment and broader project context.

1 Introduction

Organizations, hospitals, and educational institutions utilize organizational charts as a way to communicate their hierarchy. Commonly, these charts are shown as still images attached to presentation slides, brochures or public relation materials. Even though these charts can be easily read by the human eye, they're not able to be queried in a data format. Therefore, if someone would like to find out who reports to a specific vice president, which positions are two levels below to a CEO, or how an organizational chart has changed between two fiscal years, that individual has to either scan the chart by eye or enter the organizational structure into a spreadsheet.

An interactive system has been implemented to create a query able graph structure from an organization chart (as a static image) that has been structured using the RAG approach[1] and subsequently described in other surveys [2].It includes the following two components specific to chart images: 1) a complete vision front end using computer vision[3][4] 2) the reconstruction of an explicit hierarchy by identifying visual connection lines between the boxes in the organization chart, falling back on the visual to language pass due to incomplete visual connection line evidence, which will require the LLM to read the organization chart top down.

This project aims to enhance the use of static image queries like databases by allowing a user to explore their static image as easily and confidently as they are able to do with a database, while still providing enough insight to provide confidence in those answers through the visualisation of the reconstructed tree, retrieved supporting chunks, and the hierarchy strategy that led to each parent/child link.

2 Project Goals

The project has four practical goals.

First, accept any plausible org chart image and recover its structure. The system should not assume a specific chart style. It must work on rectangular boxes, rounded boxes, mixed colours, and charts produced by different drawing tools, without per source heuristics.

Second, separate exact relationships from descriptive text. Questions such as "who reports to the CEO?" need structured graph traversal, while questions such as "what does the operations team do?" need descriptive retrieval over node text. The system uses both paths instead of forcing every question into a single retrieval model.

Third, expose the pipeline state to the user. The web interface shows the reconstructed tree, the hierarchy construction strategy used (line tracing, vision language, or coordinate fallback), and the retrieved chunks for each answer, so that users can judge confidence rather than treat the model output as a black box.

Fourth, support cross chart workflows. Real organizations evolve. The system retains every previously uploaded chart in memory and exposes a comparison view that diffs two hierarchies, a time travel view that replays the breadth first build order of a chart, and a what if simulator that lets the user add, remove, move, or merge roles before re querying.

3 High Level Architecture

The system architecture is shown in figure 1. The same image that you uploaded will go through a visual pipe to an embedded knowledge base. When a user asks a natural language question, the question will be routed by a small routing function to determine if the question is about simulation, role category, or intelligence style, in which case it will be sent to the routed function. If the question is not routed, then the question will be embedded through FAISS and the top 5 chunks will be passed back along with the user's question as input to an LLM for generation of the final response based on the user's question and the relevant portions of the chunks.

The implementation of the system uses the following technologies: Python, Flask, vanilla JS without HTMX, D3.js, PyTorch[9]. Facebook Research's DETR[3], EasyOCR[4], Sentence Transformers[5](MinimL L6 v2)[6], FAISS[7] (for vector search), OpenAI's GPT 4o[8](for generating answers to user questions, also used for hierarchical reasoning based on the visual content). Information on how Flask manages file uploads and JSON endpoint can be found in the Flask documentation[10]. The renderer for tree view in the browser is D3.js[11].

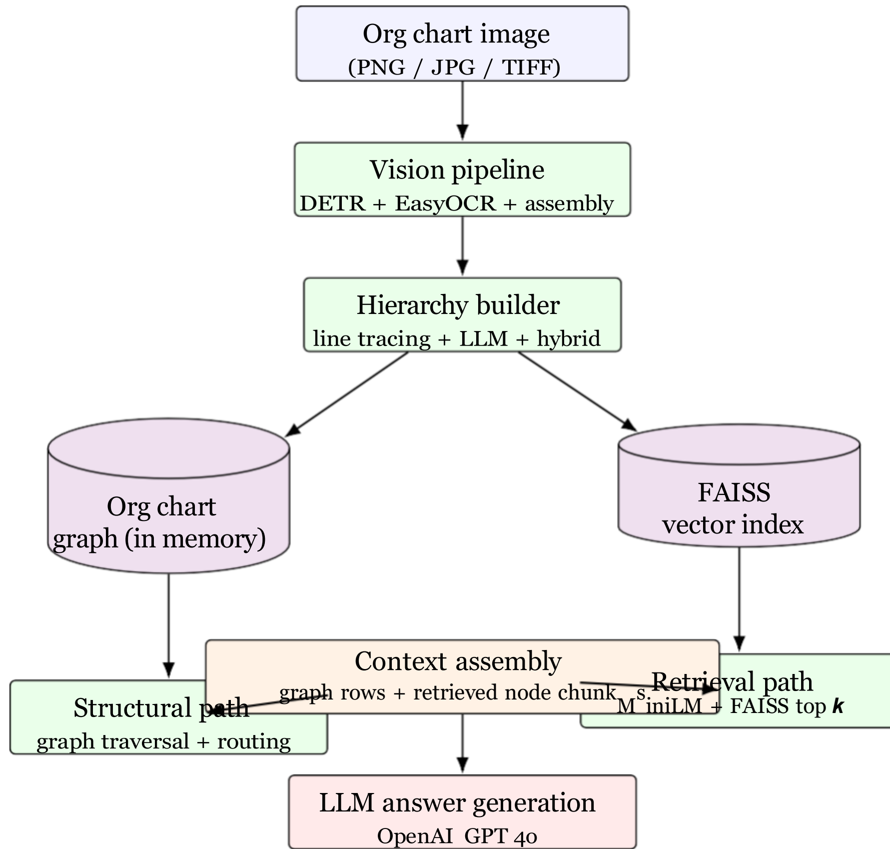

Figure 1: Organizational chart RAG architecture. The vision pipeline reconstructs a hierarchy from the image; the same hierarchy is exposed for graph traversal and embedded into a FAISS index. At query time both paths run, and a single LLM call merges their evidence.

4 Corpus Inventory

The greatest amount of intermediate data resides in the processed output/ folder, which contains 124 * txt result.csv files. Each of these files is made up of a box that has been detected and the composite OCR text. These files serve as the connection between the visual pipeline and the RAG (Regularity Assessment Group) part of the system; when a new CSV is created due to the upload of fresh material, the RAG indexer is able to rebuild the FAISS store using any combination of those CSVs in a batch mode. The FAISS Index that supports the RAG's knowledge base has already been built and is in the repository (rag knowledge base.faiss, 2.3 MB), along with a separate JSON file with corresponding metadata (545 KB), making it possible for the demo path to execute rag demo.py and provide answers to questions without having to re run the visual pipeline.

5 Data Acquisition Pipeline

The visual pipeline is implemented in five stages.

Stage 1 – DETR Box Detection:

The model is built using a DETR ResNet 50 backbone architecture that has been trained on org charts (originally saved/not under encryption). In this case, we are only considering 2 classes (employees and their title boxes), 100 queries to detect (detr) and 716×716 input size, and the box score at least 0.25 indicates confidence level. Finally, we have our bounding box being detected.

Stage 2 – Text Recognition (OCR Extractor)
EasyOCR [4] is used to retrieve the text inside each box that was detected. The image patch is up scaled and processed via CLAHE contrast normalisation, before the recogniser is run and is subsequently sharpened lightly. The result of the OCR pass is a list of {bbox, text, score} entries that all have a score of 0.3 or above.
Stage 3: Node Assembly
When OCR scans a page, it frequently finds multiple fragments in a DETR box—a person's title may appear as one of the two lines in a single fragment or be separated into two separate fragments. As with most fragments, when the assemblers combine a number of fragments inside a DETR box; for example, if a chart contains non standard visual shapes, but only has a couple of boxes returned from DETR then use an OCR only adaptive proximity fallback to group the fragments based on their vertical proximity, horizontal proximity, and the distance away from each other.
Stage 4 – Hierarchy reconstruction. This is the most algorithmically interesting stage and is described in detail in Section 6.
Stage 5 – Persistence. The reconstructed hierarchy is written to a node level CSV in processed output/ and registered with the in memory OrgChartGraph object. The same hierarchy is also flattened into text chunks ready for embedding.
The pipeline is orchestrated by pipeline runner.PipelineRunner. It exposes a single run() method that calls each stage in order and records which hierarchy strategy succeeded. Spell checking is supported but disabled by default the code comment explains the trade off: "spellchecker can corrupt proper nouns and organisation titles".

6 Hierarchy Graph Construction

The hierarchy stage is implemented as three independent builders plus a hybrid arbitrator.

6.1 Node Attributes

Each node carries a title string, an integer depth level, a parent identifier, a list of child identifiers, and a path to root cached for fast lookup. A separate title normalizer.py cleans common OCR artefacts in titles ("oj"→"of", "0f"→"of", "dept"→"Department", and bracket/semicolon cleanup) before any downstream comparison.

6.2 Edge Semantics

Edges encode reporting lines: a directed edge from a parent node to a child node means “the child reports to the parent.” Sibling order is preserved by sorting children left to right on the original image, which keeps the rendered tree visually faithful to the source chart.

6.3 Hierarchy Builders

Line tracing : The pure visual builder is the default. It detects vertical stems exiting the bottom of each detected box, traces them to horizontal “buses,” and then connects each bus to the tops of the boxes below it. The rule is strict: a node is a parent only if a vertical line exits from its bottom. Proximity is explicitly disallowed, because nodes that simply happen to sit above another are not necessarily parents.

Vision language pass : When line tracingfails, for example on charts where connection lines are dashed, low contrast, or partially missing – a vision language builder takes over. It annotates every detected box with a numeric identifier, base64 encodes the resized image, and asks an LLM[8] to list, for each node, the identifiers of its children. The top down children list framing is deliberate: an earlier “what is the parent of *X*?” framing was discarded because the model tended to pick the nearest node above, not the truly connected one.

Coordinate fallback : If both the line and LLM builders are unavailable :for example, when no model API key is configured and the chart has no clean lines, the system falls back to a coordinate clustering scheme: a per row cluster threshold of 150 pixels groups nodes into levels, and each node is assigned the nearest box on the row above as its parent. The user interface labels this case “coordinate fallback (inaccurate),” so the user does not silently trust a low confidence reconstruction.

Hybrid arbitration : When more than one builder produces a result, the hybrid layer arbitrates using either a majority vote or a highest confidence policy. Disagreement and low confidence thresholds are applied before a final parent map is emitted. PipelineRunner is last hierarchy result records the strategy used for every chart, the number of LLM derived parent links, and the number of line derived parent links, and this record is surfaced in the web interface.

7 Vector Indexing and Retrieval

Hierarchy reconstruction produces an OrgChartGraph object. The RAG indexer in rag pipeline.py flattens this graph into one chunk per node. The chunk format is fixed:

> *Title: {title} / Level: {level} / Reports To: {parent or "None (Root)"} / Subordinates: {comma joined children}*

Each chunk also carries metadata for the node’s full path to root and a coarse role category drawn from a small built in keyword table (*leadership*, *technical*, *administration*, *operations*). These role tags are not used to filter retrieval; they are used by the routing layer described in the next section.

Chunks are embedded with sentence transformers/all MiniLM L6 v2[6,5], which produces 384 dimensional vectors. Vectors are L_2 normalized and stored in a FAISS IndexFlatIP, giving cosine similarity through inner product [7]. The full index is persisted to rag knowledge base.faiss; its metadata (chunk text, node identifiers, role tags) is mirrored to a JSON sidecar so the system can re load without re embedding.

A single sentence encoder model is used for the entire corpus. This is appropriate for the org chart setting because chunks are short and structurally similar, and a heavier reranker would not earn its latency cost on tens to hundreds of nodes per chart.

8 Runtime Query Flow

Figure 3 shows the query time behaviour. The router runs first; if no specialized handler claims the question, the standard retrieve and generate path runs.

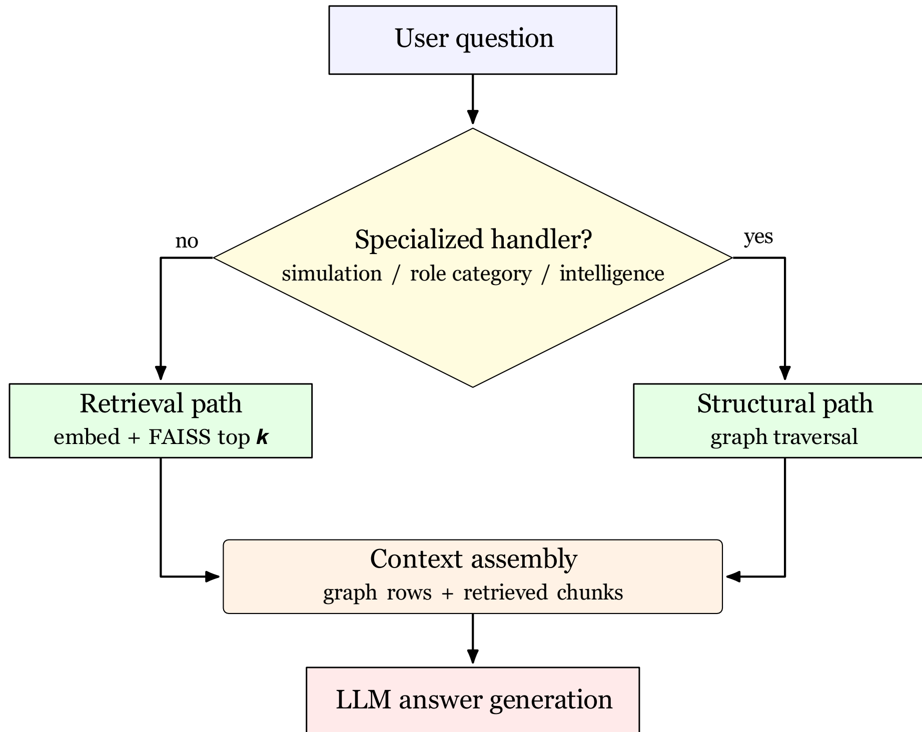

Figure 2: Runtime query flow. The router catches simulation, role category, and intelligence style questions before retrieval; otherwise the question is embedded and matched against the FAISS index, and the LLM receives both structural rows and retrieved chunks.

8.1 Specialized Handlers

Before any retrieval happens, four handlers run in order. Answer-simulation matches phrases like "what if we remove the CFO" and routes to the simulator.

8.2 Standard Retrieval

For everything else, the question is embedded with MiniLM, the top k chunks are retrieved from FAISS, and the chunks plus a short system prompt are sent to OpenAI gpt 4o with max tokens=1024. The system prompt explicitly instructs the model to use only the provided context and to refuse to

invent reporting lines that are not in the retrieved chunks. When the user phrase contains “list all” / “show all” / “enumerate”, the system automatically raises *k* up to a cap of 50 so that exhaustive list questions do not silently truncate.

8.3 Fallback Without an API Key

If no API key is present, generate natural answer() produces a deterministic answer directly from the retrieved chunks’ structured metadata. The answer quality drops, but the system never crashes on a missing credential.

9 Web Interface

The web layer is implemented in Flask [10]. The maximum request size is set to 50 MB. Uploaded images are written to web uploads/, processed, and then deleted to avoid disk growth.

The HTTP surface is small. GET / returns the single page shell. POST /upload accepts a multipart image, runs the pipeline, and returns the reconstructed tree as JSON. POST /ask accepts a question and optional top k value, and returns the LLM answer plus the supporting sources, the current tree, a visual tree payload for D3, the simulation state, and a llm enabled flag. GET /tree, POST /reset, POST /clear history, and GET /status provide auxiliary operations. A second tier of endpoints supports multi chart workflows: GET /graphs lists every chart in memory, POST /set active switches the active chart, POST /compare diffs two charts, POST /ask compare asks an LLM to summarize the diff in natural language, and GET /build order returns the BFS sequence used by the time travel slider.

It uses D3.js [11] to render the tree, vanilla JavaScript for the chat panel, and tabbed views for the workspace, the comparison view, and the time travel view. Simulation buttons (*add role*, *remove role*, *move role*, *merge roles*) are wired directly to the GraphSimulator on the back end.

9.1 Session Model

The web module currently stores the active chart, the chat history, and the graph history as module level globals. This is acceptable for a demo or single user local run, but it is not a production session model – multiple users would share one chat history. A production deployment should move history into per session or per user storage and manage model clients through explicit application lifecycle hooks.

10 Simulation, Comparison, and Time Travel

Beyond simple Q&A, three operations on the reconstructed graph distinguish this system from a flat text RAG.

Simulation (graph simulator.py): GraphSimulator supports four operations: add a node under a chosen parent, remove a node (children promote one level), move a subtree, and merge two roles into one. Every operation returns a SimulationResult object that carries the updated graph, the list of affected nodes, a human readable change summary, and the operation name. This makes it easy to expose “preview” versus “commit” behaviour in the UI.

Comparison (graph diff.py): Two hierarchies are diffed with a normalized string match first, then a containment match, then a Jaccard similarity match at a 0.55 threshold. The output is a list of added, removed, moved, and matched nodes plus a single similarity score. The Jaccard fallback is

important because OCR variants of the same role – for example “Director of Nursing” and “Director oj Nursing” – should match instead of being reported as one removed and one added node.

Time travel: The /build order endpoint returns the BFS order in which the tree was constructed. The browser uses this to animate a slider that replays the chart from the root down to the leaves, which is helpful when a user wants to explain or verify the reconstruction visually.

11 Evaluation Harness

Unit tests live under tests/ replaces Sentence Transformers with a four dimensional encoder, builds a synthetic hospital hierarchy, and verifies the retrieval and answer flow end to end on OrgChartRAG. test graph simulator.py exercises every simulator operation on a small CEO VP Manager graph and confirms that child reassignment is correct after a removal. test hybrid hierarchy builder.py writes a temporary node text CSV with a CEO, two VPs, and a manager and asserts that the hybrid builder selects a valid parent map. test title normalizer.py covers the common OCR fixes ("oj"→"of", bracket/semicolon cleanup, and graph level title normalisation).

A separate batch eval.py script runs the RAG path over a directory of * txt result.csv files and records latency per question. The harness is deliberately deterministic where it can be: required mention checks are used for entity level facts such as role titles, and forbidden mention checks catch obvious hallucinations of roles that are not in the chart.

12 Implementation Scope

This paper describes the functionality that is present in the inspected repository. The implemented system includes the visual pipeline, the three hierarchy builders, the hybrid arbitrator, the FAISS backed RAG store, the Flask web application, the simulator, the diff view, the time travel view, and the four unit test files described above. It does not yet include every feature that would be expected in a hardened production deployment.

In particular, the inspected code does not implement authentication, rate limiting, or a response safety classifier on the /ask route. The route runs the pipeline and returns the rendered answer, but it does not run a separate safety check before emitting it. A safety layer is useful future work, especially for deployment, but it should be implemented, logged, and tested before being described as part of the working system.
Using repository derived counts keeps the report reproducible and avoids relying on stale numbers from earlier development notes.

13 Limitations

Hierarchy ambiguity on noisy charts. On charts where the connecting lines are dashed, partially occluded, or rendered at low contrast, the line tracer may report no parent for some nodes, and the system depends entirely on the vision language pass. When the LLM is also unavailable, the coordinate fallback can over or under assign parents.

OCR fragility. Hand drawn, rotated, or stylized labels can confuse EasyOCR. The system mitigates this with CLAHE preprocessing, upscaling, and post hoc title normalization, but pathological inputs still degrade quality.

Single in memory web session. The Flask layer shares one in memory active chart and one chat history. This is fine for a demo but not for multi user deployment.

Incomplete safety layer. The /ask route does not run a response safety classifier before returning the answer. Guardrails should be added before any non demo deployment.

Operational dependencies. A complete run requires the fine tuned DETR checkpoint to be present on disk, valid model API credentials, and writable web uploads/ and processed output/ directories. The system can degrade gracefully when these are missing, but the user experience is meaningfully worse.

14 Future Work

The next engineering steps should focus on reliability and correctness.

First, the web layer should move to per session or per user state with explicit application lifecycle hooks. Second, the hierarchy builder should expose confidence scores per parent child edge in the UI rather than only at the strategy level, so that a user can audit individual reporting lines. Third, the evaluation harness should add component level metrics such as the precision and recall of parent child edges, the accuracy of OCR recovered role titles, and retrieval recall at *k*. Fourth, hierarchy reconstruction should be re audited after every new chart style is encountered, so that page structure changes do not silently degrade the graph. Fifth, a response safety guardrail should be integrated into the back end response path and covered by tests. Finally, the comparison view should expose graph level summary statistics – for example, span of control change and depth change – in addition to the current node level diff.

15 Conclusion

This paper has described an interactive RAG system for organizational charts. The visual pipeline – DETR detection, EasyOCR text recognition with CLAHE preprocessing, node assembly, and a three way hierarchy builder recovers a structured graph from a static chart image. The recovered graph is exposed both for direct traversal and through a FAISS indexed retrieval path, and the two paths are merged by a single grounded LLM call. The web interface exposes the reconstructed tree, the retrieved chunks, a what if simulator, a chart versus chart diff, and a time travel reconstruction slider.

The main lesson is that organizational chart RAG systems need more than a vector database. They need a robust visual front end, a hierarchy reconstruction stage that can fall back across multiple evidence types, source aware retrieval over structured node chunks, and a user interface that exposes the strategy used to produce each answer. The system described here provides that foundation while leaving clear next steps for production hardening.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval augmented generation for knowledge intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[2] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval augmented generation for large language models: A survey," arXiv:2312.10997, 2023.

[3] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End to end object detection with transformers," in *European Conference on Computer Vision (ECCV)*, pp. 213–229, 2020.

[4] JaidedAI, "EasyOCR: Ready to use OCR with 80+ supported languages,"

[5] N. Reimers and I. Gurevych, "Sentence BERT: Sentence embeddings using Siamese BERT networks," in *Proc. EMNLP IJCNLP*, pp. 3980–3990, 2019.

[6] Sentence Transformers, "all MiniLM L6 v2 model card," Hugging Face,https://huggingface.co/sentence transformers/all MiniLM L6 v2.

[7] J. Johnson, M. Douze, and H. Jégou, "Billion scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[8] OpenAI, "Hello GPT 4o,". https://openai.com/index/ hello gpt 4o/.

[9] A. Paszke et al., "PyTorch: An imperative style, high performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] Pallets, "Flask documentation,"https://flask. palletsprojects.com/.

[11] M. Bostock, V. Ogievetsky, and J. Heer, "D^3: Data driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.

[12] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proc. 4th Alvey Vision Conference*, pp. 147–151, 1988.

[13] Meta AI, "FAISS: A library for efficient similarity search,"