Exploring Retrieval-Augmented Code Generation for Procedural Modelling in Houdini

Xuwen Dong

MSc Computer Animation and Visual Effects

NCCA - Bournemouth University

Supervisors: Jian Chang and Jon Macey

August 2026

Word count: 4564

Abstract

Procedural modelling in Houdini can produce editable and reusable three-dimensional assets, but constructing valid node networks requires specialised knowledge of node types, parameters, data flow and the Houdini Python API. This project investigated whether retrieval-augmented generation (RAG) could support a small local code model when generating procedural Houdini networks from natural-language requests. The prototype combined Houdini with an external FastAPI retrieval and generation service, a manually verified Markdown knowledge base stored in ChromaDB, and a quantised Qwen2.5-Coder-7B model served through llama.cpp. Generated code remained subject to manual review before execution. The exploratory evaluation used a small set of development cases documented through logs, token and timing records where available, node-network inspection and visual geometry inspection. Straight-staircase reuse was successful in a recorded RAG run, while spiral-staircase cases remained mixed as spatial composition became more demanding. Some outputs failed to execute because of API or VEX details; others executed but produced geometrically incorrect results. These observations suggest that RAG can improve access to domain knowledge and support reuse of verified procedural patterns. The comparison suggests that smaller local models and larger cloud models may benefit from different RAG configurations, as their ability to interpret and compose retrieved knowledge differs.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Background and Context

Procedural modelling offers a powerful way to construct editable three-dimensional assets through rules, parameters and repeatable node networks. In Houdini, this approach can produce many variations from a compact procedural definition, but it also requires knowledge of node types, input conventions, parameter names and the Houdini Python API. Large language models provide a possible natural-language interface to this process: a user describes an intended asset, the model writes Python, and Houdini executes that code to construct a procedural network. The difficulty is that general code-generation ability does not guarantee reliable knowledge of a specialized and version-sensitive digital content creation environment.

1.2 Proposed Approach

This project explores whether retrieval-augmented generation (RAG) can support code generation for procedural modelling in Houdini. The prototype uses a small, quantised Qwen2.5-Coder-7B model running locally through llama.cpp. Rather than training or fine-tuning the model on Houdini data, the system retrieves manually prepared and verified procedural examples, inserts them into the prompt, and asks the model to generate executable Houdini Python. The resulting code can then be reviewed and executed from a Houdini interface.

1.3 Research Aim and Question

The central research question is: how far can retrieval of domain-specific Houdini knowledge improve a small code model's ability to create procedural node networks, and where does that support stop being sufficient? The investigation is deliberately exploratory. Its evidence consists of a small set of progressively more complex tasks, development logs, manually recorded benchmark notes and screenshots of node networks and geometry. It therefore does not attempt to estimate a general success rate.

The main argument is that RAG can improve domain knowledge access and encourage reuse of verified procedural patterns, particularly when failure is caused by an unknown API convention or node-connection pattern. It does not, however, make the model learn a new modelling skill or replace its ability to compose several patterns into a coherent procedure. The Copy to Points experiment provides the main case study: an underspecified baseline failed repeatedly, increasingly explicit prompt revisions eventually succeeded, and a later RAG-enabled run also succeeded by retrieving a relevant connection pattern. Straight and

spiral staircase tasks then show that results become mixed as procedural composition grows more demanding. Throughout the thesis, executable code and visually correct geometry are treated as separate outcomes.

1.4 Thesis Overview

Chapter 2 reviews retrieval-augmented code generation and language-driven procedural 3D modelling. Chapter 3 introduces Houdini, local model inference and the RAG pipeline. Chapter 4 describes the prototype design and implementation. Chapter 5 presents the exploratory evaluation, followed by discussion, limitations and future work in Chapter 6. Chapter 7 concludes the thesis.

2. Previous Work

This project lies between retrieval-augmented code generation and language-driven procedural 3D modelling. Prior work in these areas motivates the architecture, but most published systems operate at a much larger scale, use stronger models, or target Blender and general-purpose programming benchmarks rather than Houdini. The present prototype is therefore best understood as a small domain-specific investigation rather than a direct reproduction of those systems.

2.1 Retrieval-Augmented Code Generation

Retrieval-augmented generation separates information storage from model parameters. In the original RAG formulation, relevant passages are retrieved from an external collection and supplied to a generator at inference time (Lewis et al., 2020). For code generation, this is attractive because libraries and APIs change, while a model's parametric knowledge is fixed after training. DocPrompting retrieves documentation from a pool and conditions natural-language-to-code generation on the selected passages. Zhou et al. (2023) show that documentation can support previously unseen libraries and usages without retraining the generator. This idea maps closely to Houdini, where a small model may understand the requested operation while still hallucinating a method, node type or parameter.

Other code-oriented systems retrieve source-code examples rather than documentation alone. ReACC combines retrieval and generation for code completion, using similar external code as context for an unfinished program (Lu et al., 2022). RepoCoder iterates between repository retrieval and generation (Zhang et al., 2023), suggesting a possible future direction for iterative retrieval or repair. These systems also support the interpretation adopted here: retrieved material is external evidence for generation, not evidence that the underlying model has learned the retrieved procedure.

CodeRAG-Bench provides an especially relevant caution. Wang et al. (2025) separate retrieval quality from end-to-end generation quality and report that high-quality context can improve code generation, while retrievers may still fail to fetch useful material and generators may fail to use what is retrieved. The benchmark also examines different document sources and chunking strategies. Its distinction between finding appropriate context and using that context effectively informs the analysis in this thesis. A low embedding distance does not prove that a Houdini example will be followed correctly, and a lower-ranked example may still dominate the generated result.

2.2 Language Models for Procedural 3D Modelling

Research on language-driven 3D creation demonstrates the value of code as an editable representation. 3D-GPT uses multiple language-model agents to decompose instructions and control procedural functions in Blender (Sun et al., 2024). SceneCraft generates Blender scripts from scene graphs and uses rendered-image feedback to refine spatial constraints; it also accumulates reusable functions in a skill library (Hu et al., 2024). LL3M similarly coordinates planning, retrieval, coding, debugging and visual refinement agents, with a Blender documentation knowledge base used to improve code correctness and modelling capability (Lu et al., 2025). These systems show that script generation can preserve editability and that feedback, planning and retrieval are complementary rather than interchangeable components.

2.3 Positioning of This Work

The Houdini prototype follows this procedural perspective at a smaller scale by generating editable SOP node networks rather than lists of vertices. Unlike the cited multi-agent systems, however, it does not implement visual critique, automated repair, constraint solving or model training. The project primarily investigates retrieval-supported generation with a lightweight local model, while selected cloud-model experiments are included to examine more demanding cases involving longer retrieved contexts and greater procedural complexity.

3. Technical Background

3.1 Procedural Modelling in Houdini

Houdini represents many modelling operations as directed node networks. At the Object level, a Geometry container holds Surface Operator (SOP) nodes that create, transform and combine geometry. A procedural asset is defined not only by the nodes that exist but also by their connections, parameters and the final display or render flag. Houdini's Python API

exposes these operations through the hou module. Generated code must therefore satisfy several layers of constraints: valid Python syntax, valid Houdini API calls, valid node types and parameter names, valid input ordering, and a network whose output corresponds to the requested shape.

3.2 **Language Model Inference**

The local generator is Qwen2.5-Coder-7B-Instruct in Q4_K_M GGUF form. It is served by llama.cpp through an OpenAI-compatible HTTP endpoint and is sampled at low temperature in the recorded implementation. Quantisation allows the model to run on the available RTX 4080 with 16 GB of graphics memory, but its small size makes missing or inaccurate domain knowledge visible. The system also supports selected cloud-model comparisons through a configurable API connection.

3.3 Retrieval-Augmented Generation Pipeline

The RAG pipeline converts each user query into an embedding using the multilingual paraphrase-MiniLM-L12-v2 sentence-transformer. ChromaDB stores embeddings for Markdown knowledge documents and returns the nearest documents with source metadata and distance values. A prompt builder combines fixed Houdini code-generation rules, the retrieved context and the current user request. In this project, top-k denotes how many retrieved documents are inserted. Increasing top-k may add useful components, but it also increases prompt length and can introduce competing patterns.

3.4 **Interpreting Execution and Geometric Outcomes**

During development, experiment results were recorded mainly through execution behaviour, node-network inspection and visual inspection of the generated geometry. These records were not organised using a formal two-category evaluation rubric. However, the experiments revealed an important distinction between whether generated code could execute and whether the resulting geometry was actually correct. Some scripts failed because of Python, Houdini API or VEX errors, while others executed successfully but produced incorrect placement, orientation or overall shape. This distinction is used later in the thesis as an analytical lens for interpreting the recorded results, rather than as a formally predefined evaluation scheme.

4. System Design and Implementation

The implemented system is an exploratory research prototype. It evolved from a direct Python-to-model script into a separated architecture because Houdini's embedded Python environment was not suitable for loading the heavier machine-learning and vector-database

dependencies. The final prototype therefore uses an external FastAPI-based RAG and generation server between Houdini and the model/retrieval stack. Its main components are summarised below.

- Houdini client and execution environment, including the review and execution interface.
- FastAPI RAG and generation server, which handles retrieval, prompt construction, model requests and returned generation results.
- Retrieval stack, including the Markdown knowledge base, embedding model and ChromaDB vector store.
- Language-model backend, using the local Qwen model as the primary generator, with optional cloud-model support for selected exploratory comparisons.

4.1 **Initial Direct Code Generation**

The earliest end-to-end pipeline read a prompt, sent it to the local llama.cpp server, extracted the response text, removed optional Markdown code fences and executed the result in Houdini. A Shelf Tool provided a convenient trigger (Figure 3). This proved that natural language could cause the model to create a Geometry container and a Box SOP, shown in Figure 4, but it also exposed an API hallucination: the generated script called a non-existent setPos method after creating the nodes. The partial success established the central engineering problem. Output cleaning could remove formatting noise, but it could not verify whether an API call or procedural network was correct.

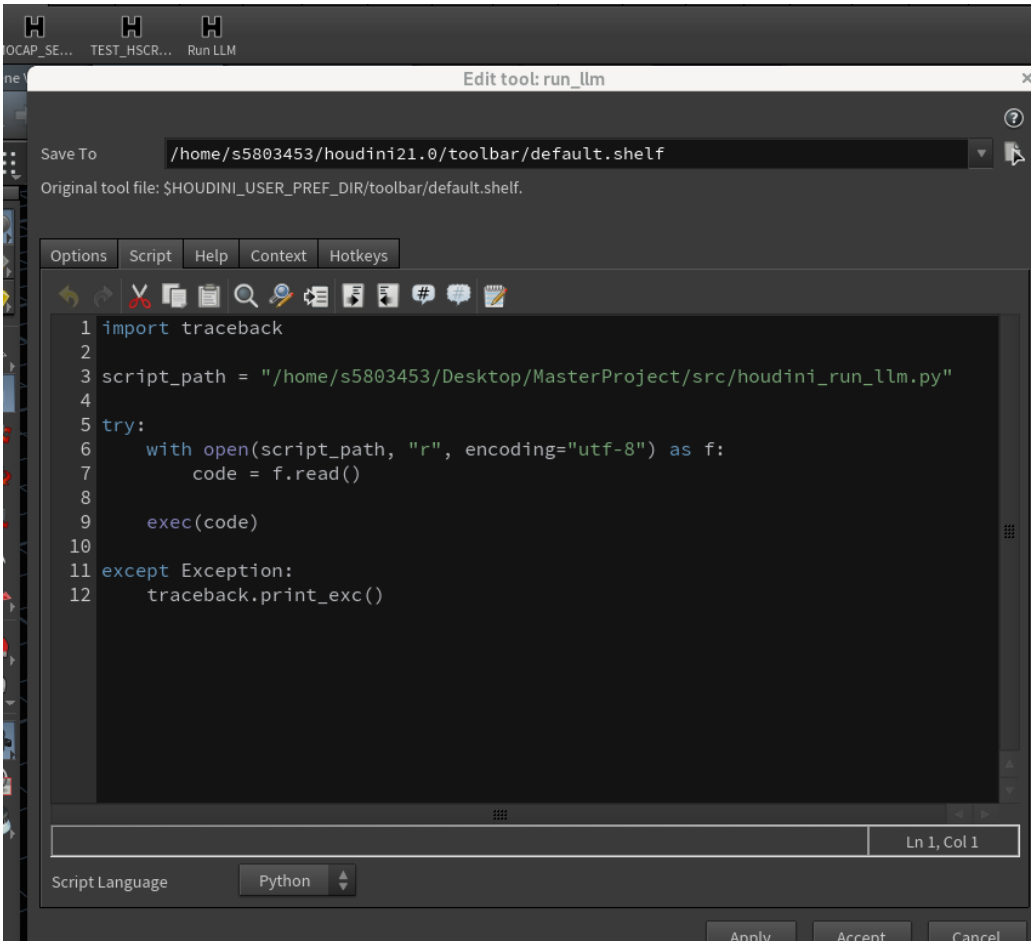


Figure 1. Initial Shelf Tool Script Editor screenshot.

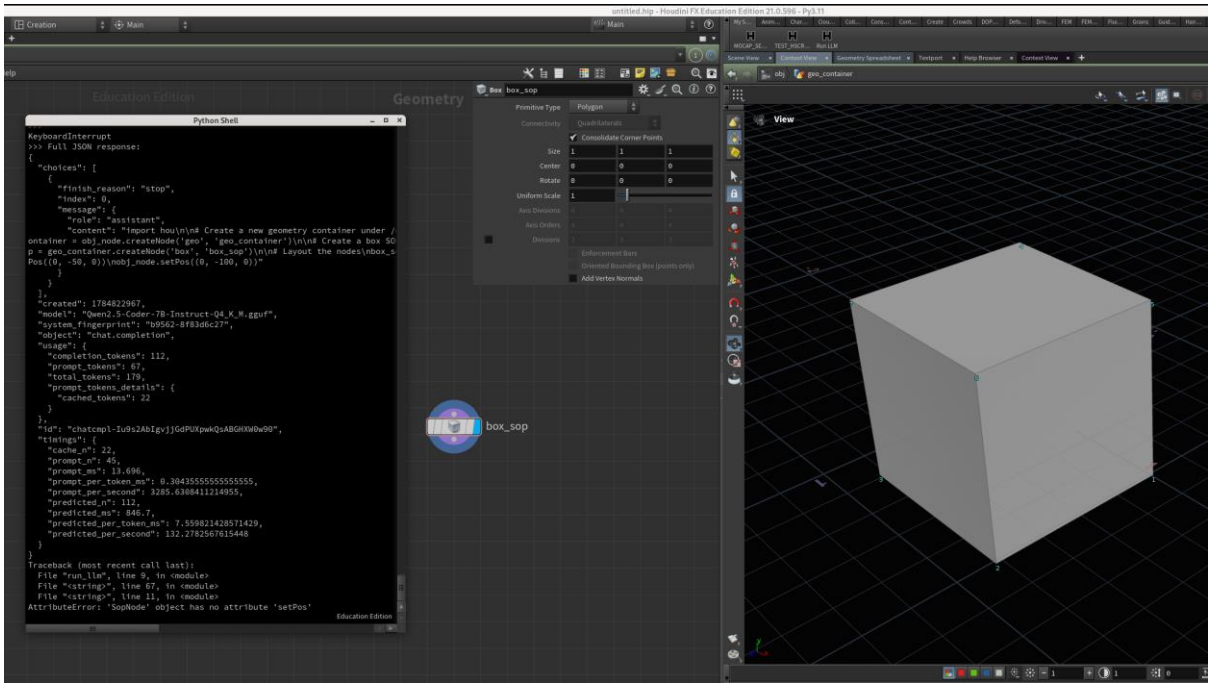


Figure 2. Initial Box SOP generation result.

4.2 **Motivation for Retrieval-Augmented Generation**

The motivation for retrieval emerged most clearly from the original Copy to Points benchmark. The initial natural-language prompt repeatedly produced plausible but procedurally incorrect networks, including missing nodes, reversed inputs and incorrect display settings. Increasingly explicit revisions eventually produced a successful result by naming variables, prescribing the connection order and identifying the final displayed node. By that stage, however, the prompt had become increasingly solution-specific and close to a procedural recipe. RAG was introduced as a way to supply verified Houdini conventions and reusable connection patterns from an external knowledge base, while allowing the user request to remain focused on the intended modelling task.

4.3 **RAG Architecture and Service Separation**

The codebase was subsequently separated by responsibility. Model communication and response cleaning became utilities; ordinary generation and RAG-assisted generation became service functions; command-line, HTTP and Houdini entry points called those functions. Knowledge ingestion and retrieval were kept in a separate RAG package. This structure made it possible to run the same task with and without retrieval and reduced the risk that Houdini-specific entry code would accumulate model and database logic.

A FastAPI service created a process boundary between Houdini and the retrieval stack. Houdini sends a JSON request containing the prompt, whether RAG is enabled, the chosen provider and a top-k value. When retrieval is enabled, the service embeds the query, searches ChromaDB, formats the selected documents and sends the augmented prompt to the model. It then returns cleaned Python and retrieval metadata. Houdini needs only its standard Python environment, the hou module and an HTTP client. Figure 3 summarises the implemented architecture and the manual review point before execution.

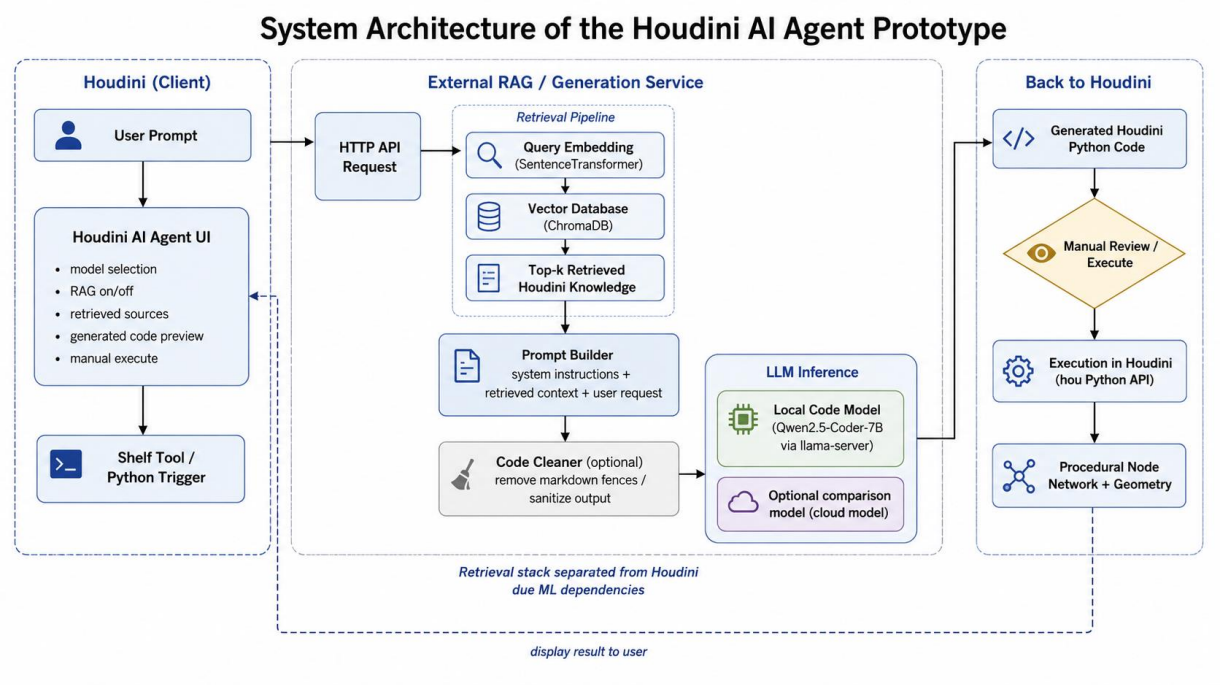


Figure 3. System architecture of the prototype. Houdini remains the client and execution environment, while retrieval and language-model dependencies run in an external service; generated code returns to a manual review gate before execution.

4.4 Knowledge Base and Retrieval Design

Knowledge Sources and Representation

The knowledge base was manually constructed from Houdini learning resources and project examples rather than from a single source. External source material included a procedural staircase tutorial by Chetal Gazdar (Gazdar, 2023) and SideFX's *Sci Fi Stair Generator* tutorial by Simon Verstraete (Verstraete, 2020). Houdini-Agent (Kazama-Suichiku, n.d.) was used as a supporting tool during knowledge preparation, assisting with tasks such as reading scene and node-network structure, renaming and organising nodes, and summarising procedural concepts and keywords. The resulting material was then manually reviewed and, where applicable, tested in Houdini before being reformatted into Markdown documents for retrieval.

The initial knowledge base used Markdown files as whole-document retrieval units. Each document contained elements such as purpose, relevant user requests, knowledge type, recommended procedural patterns, key concepts and selected Python examples. A multilingual embedding model was used so that Chinese task descriptions and mainly English technical material could be represented within the same retrieval pipeline.

Knowledge Quality and Execution Context

Knowledge quality proved as important as retrieval relevance. The first staircase document was relatively focused, although it initially contained a conventional main-entry pattern that did not work correctly in Houdini's exec-based environment. Later examples were derived from more complete tutorial workflows and included additional process-specific content, such as intermediate construction steps and node-layout code. These documents were therefore simplified and reformatted around clearer purpose, requests, procedural patterns and key concepts to make them more suitable for retrieval. The later evaluation examines how these changes affected document ranking.

Query Construction

Later experiments introduced more varied staircase documents and exposed retrieval-design issues. When a long code-generation prompt was embedded directly, generic instructions distorted its semantic representation and a straight-staircase document ranked above the expected spiral example. Using the concise task phrase as the retrieval query produced the intended ordering. Document formatting also affected ranking: adding consistent purpose, request, knowledge-type and concept sections moved the spiral example above less relevant alternatives. These observations motivated a practical separation between the query used for retrieval and the full prompt used for generation.

Chunking and Top-k

Chunking was introduced because some complete procedural examples became too long to use efficiently as retrieval context. In one long spiral-staircase case, even the cloud model required substantially more tokens and generation time, while the smaller local model struggled more noticeably with the same large context. To reduce this burden, the full example was split into separate core, railing and baluster documents. This allowed top-k to control how much procedural knowledge was inserted into the prompt. Retrieving only the core reduced the context and produced the central staircase structure in a recorded cloud-model run, while retrieving all three chunks produced a more complete result but at a much higher token cost. The experiment therefore treated chunking as a way to balance context size and procedural coverage rather than as a claim that smaller chunks are always better.

4.5 Prompt Construction and Code Execution

Prompt construction retained fixed constraints even when RAG was enabled. The system instructs the model to return executable Houdini Python only, create SOP nodes inside a Geometry container, use explicit node variables and explicit connections, set display and render flags on the intended final node, and treat the user's current request as higher priority

than example-specific parameter values. These instructions define the output contract. Retrieved examples supply domain-specific implementation patterns, but they do not override the task.

4.6 **User Interface and Safety**

The final interface is a minimal PySide6 GUI launched from a Houdini Shelf Tool on the main UI thread. It provides a prompt field, model selector, RAG toggle, Generate and Execute controls, a generated-code preview, execution status, and a Retrieved Knowledge panel showing document names and distances. Generated code is not executed automatically. The user can inspect the response and decide whether to run it, which provides a basic human confirmation gate. Figure 4 shows this interface with a retrieved parameterised-staircase document and the resulting Houdini geometry.

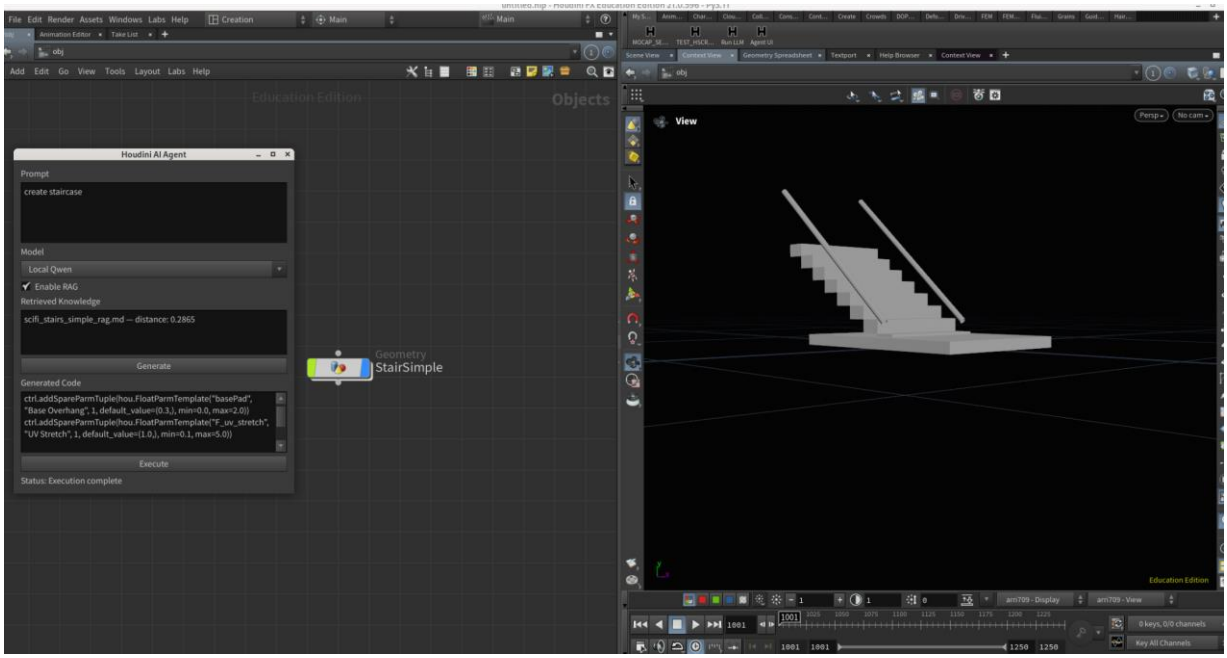


Figure 4. The PySide6 testing interface showing the selected local model, enabled RAG, the retrieved knowledge source and distance, generated-code preview, execution status and the resulting parameterised staircase in Houdini.

The current safety measures remain limited. A small blocked-word list rejects obvious operations such as subprocess calls or selected deletion functions, and manual review is encouraged before execution. This is not a sandbox and does not establish that generated code is safe. AST validation, an allow-list for Houdini operations and stronger isolation are future work. Likewise, the GUI is a testing interface rather than a finished product: configuration, error presentation, comparison views and experiment logging remain basic.

5. Exploratory Evaluation

5.1 Evaluation Setup

The evaluation was designed to probe failure modes rather than to report a statistically meaningful benchmark. Tasks progressed from creating a Box to connecting nodes, setting parameters, building a Copy to Points network, and constructing straight and spiral staircases. Records were accumulated during development and are uneven: some runs include token counts and timings, while others preserve only a result note and screenshot. The analysis therefore reports observed cases and avoids success-rate claims.

Because the records emerged during iterative development, the evaluation was not governed by a formal rubric defined in advance. Preserved runs were interpreted through the evidence available in each case: whether Python and VEX executed, whether inspection of the Houdini node network showed the requested nodes, connections, parameters and display flags, and whether visual inspection indicated that the resulting geometry matched the requested procedural structure. Execution behaviour and geometric outcome were treated separately, and screenshot-based judgements remain qualitative, particularly for borderline cases.

5.2 Copy to Points Case Study

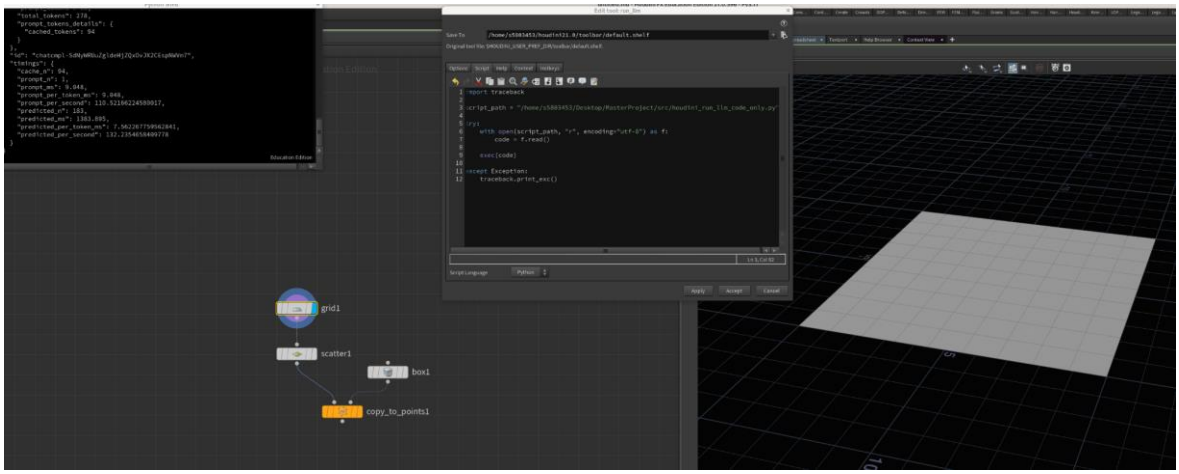
The Copy to Points task was the principal case study because it is simple in appearance but sensitive to procedural topology. The original baseline prompt asked the local model to create a Box, Grid, Scatter and Copy to Points network. Across the recorded attempts, outputs omitted the Box, displayed the Grid, reversed the two Copy to Points inputs, used an invalid multi-input connection, returned Markdown fences that caused a syntax error, or created nodes without connecting them. On the third attempt the expected nodes were present, but manual input correction was still required.

Prompt revisions progressively encoded the missing solution. Formatting errors motivated the code cleaner. Later revisions named the variables, specified the exact connection order in pseudocode-like form and required the final node to carry the display and render flags. The seventh recorded attempt succeeded. This result demonstrates that the model could follow an explicit procedural specification, but it weakens any claim that the model inferred the Houdini pattern from the original natural-language request. The prompt had become close to a recipe for the answer.

A later RAG-enabled Copy to Points test used a knowledge base whose relevant item was a verified staircase example. Although the final asset differed, that document contained the

same Box-to-Copy and points-to-Copy connection pattern. The recorded no-RAG comparison created an almost correct network but displayed only the Grid, whereas the later RAG-assisted run connected the network and produced copied geometry in the viewport (Figure 5). The no-RAG screenshot is evidence of that recorded comparison, not necessarily the very first baseline attempt. Together with the prompt-revision history, the case supports the narrower claim that retrieved examples can make a domain pattern available for reuse.

(a) Recorded **no-RAG outcome**



(b) RAG-assisted outcome

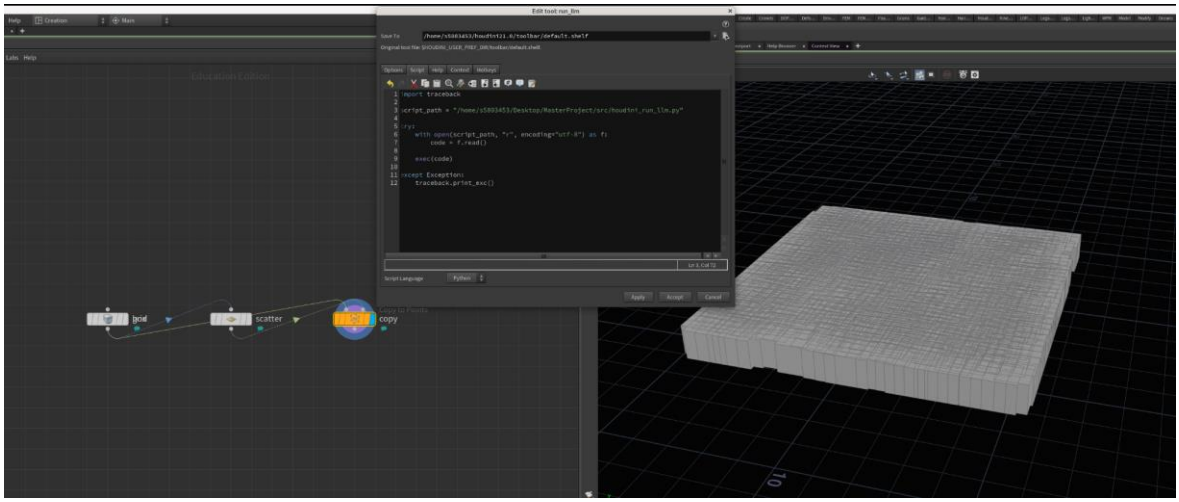


Figure 5. Recorded Copy to Points comparison. (a) **The no-RAG network contains the expected node types, but the viewport displays the Grid. (b) The RAG-assisted network displays the Copy to Points output and visible copied geometry.**

5.3 **Staircase and Spiral Staircase Experiments**

The staircase experiments tested whether that pattern could be reused at increasing levels of change. For a five-step linear staircase, the no-RAG model invented a polybox node and produced incorrect wrangle parameters, inputs and final output settings. The first RAG run also failed because of the incompatible entry guard in the knowledge document. Once the document was corrected, the model created the Box, Attribute Wrangle, Copy to Points, Grid

and Merge network and produced the requested geometry. Table 1 records the available token and timing values for these single runs. A subsequent 12-step request retained the same structure and changed the step count successfully (Figure 6). This is evidence of parameter substitution and structural reuse.

Table 1. Straight staircase comparison. Each row is one recorded run; values are not averages.

Condition	Outcome	Total tokens	Generation time
No RAG	Execution failed; invalid node type and incorrect procedural settings.	482	3.18 s
RAG	Executed and produced the requested staircase after the retrieved example was corrected for Houdini's execution context.	2,536	7.82 s

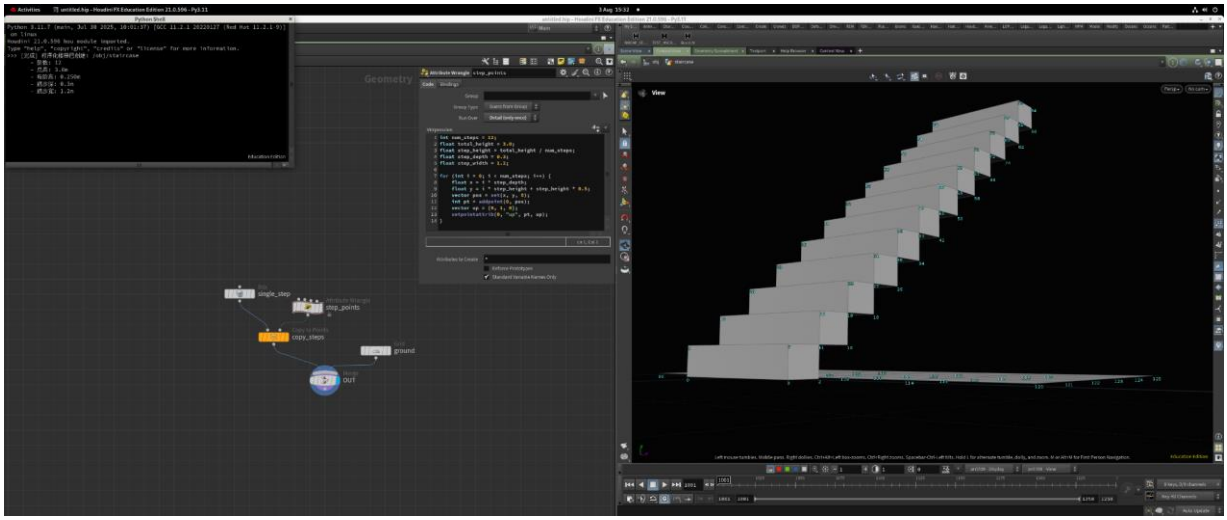
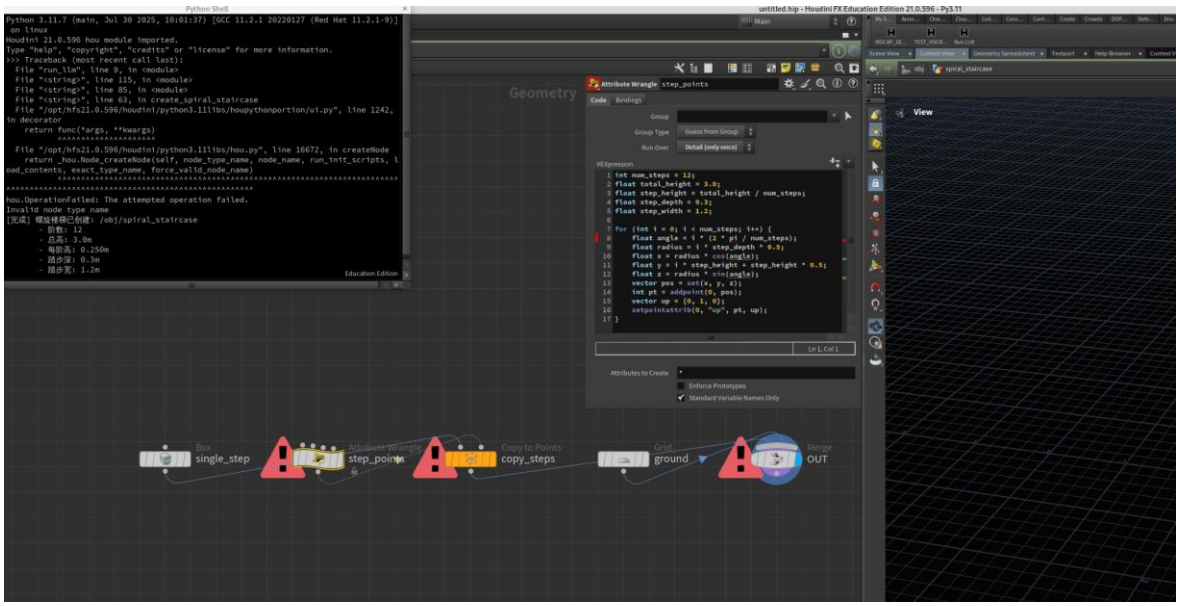


Figure 6. A recorded successful 12-step run shows the Box, Attribute Wrangle and Copy to Points network and the resulting linear staircase.

The spiral tasks required a different spatial composition. With only the linear example available, the model attempted to distribute points by angle and height but wrote the VEX constant as lowercase pi, causing compilation to fail. Manually correcting that detail produced a spiral form (Figure 7). A more explicit prompt later supplied radial distribution and quaternion rules. With RAG enabled, the generated network executed and placed steps along a curve, but discontinuities and local orientation errors remained. The experiments therefore moved between execution failure and partial geometric success without establishing a stable staircase procedure.

(a) Local Qwen output before correction



(b) Result after the recorded VEX constant correction

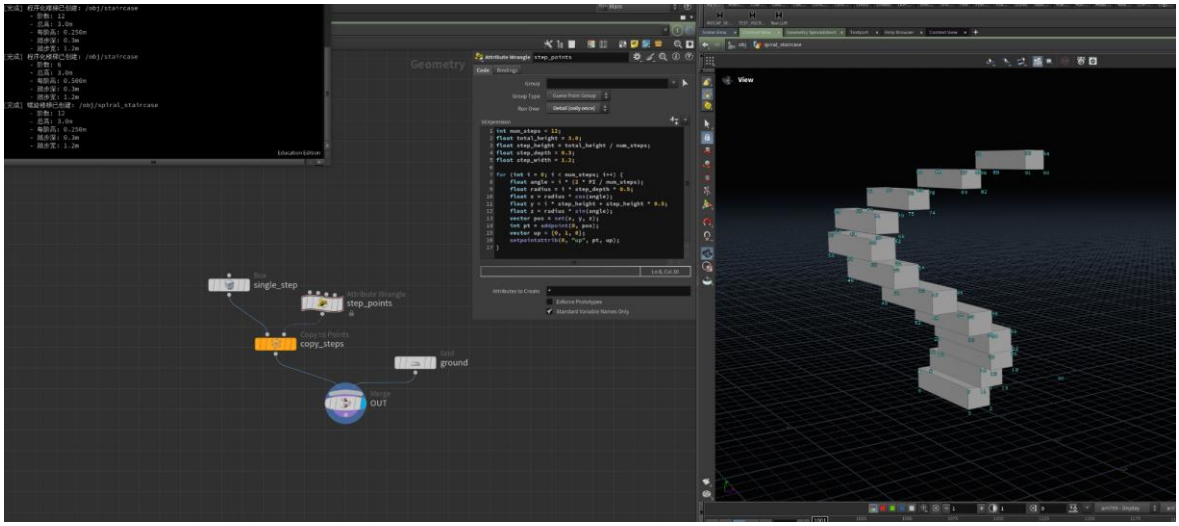


Figure 7. Recorded local spiral case. (a) **The run leaves error-marked nodes and no visible geometry while the VEX uses lowercase pi.** (b) Replacing **the constant allows the network to produce a spiral arrangement.**

5.4 **Model Comparison and Efficiency**

A single cloud-versus-local spiral comparison reinforces that distinction (Table 2). DeepSeek-V4-Flash produced executable Python and VEX, yet the staircase bent and overlapped incorrectly (Figure 8). The local Qwen model created the expected broad node structure but failed VEX compilation and produced no geometry because of the lowercase pi detail (Figure 7a). Since these are individual runs with different model behaviours and incomplete timing data, they cannot support a general model ranking. They show only that execution, geometric outcome and generation cost can diverge in the observed cases.

Overall, the evaluation supports three bounded observations. First, the small local model can complete simple direct-generation tasks. Second, explicit prompts and retrieved examples can both supply missing Houdini conventions. Third, increasing procedural complexity exposes failures that retrieval alone does not solve. The evidence does not support a general claim about reliability, production readiness or average performance across Houdini tasks.

Table 2. Exploratory single-run comparison for the spiral task.

Recorded field	DeepSeek-V4-Flash	Qwen2.5-Coder-7B
Prompt tokens	1,632	1,547
Completion tokens	6,065	1,027
Total tokens	7,697	2,574
Execution result	Python and VEX executed; geometry produced.	Network created, but VEX compilation failed; no final geometry.
Geometric result	Geometry present, but staircase shape incorrect.	Not assessable because no final geometry was produced.
Main observed failure	Step placement and orientation produced bending and overlap.	Lowercase pi constant caused VEX failure.

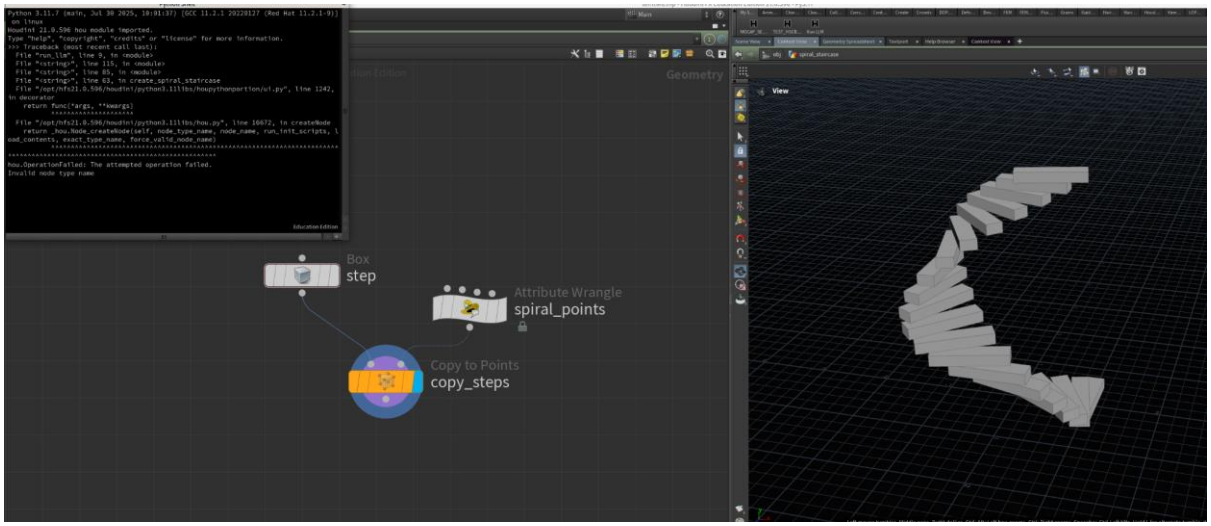


Figure 8. The cloud-model code executes and produces visible geometry, but the steps bend and overlap rather than forming a correct spiral staircase.

5.5 Retrieval Behaviour, Query Design and Chunking

Query Formulation and Document Ranking

Embedding the full code-generation prompt distorted retrieval in one recorded ranking test. The generic instructions caused staircase.md to rank first at distance 0.5020, while the intended spiral_core.md ranked lower at 0.5663. When the retrieval query was simplified to “Create a spiral staircase.”, spiral_core.md ranked first at 0.2631. This result motivated separating the concise semantic query from the fuller generation prompt. The distances are local to these recorded searches and should not be treated as absolute relevance scores.

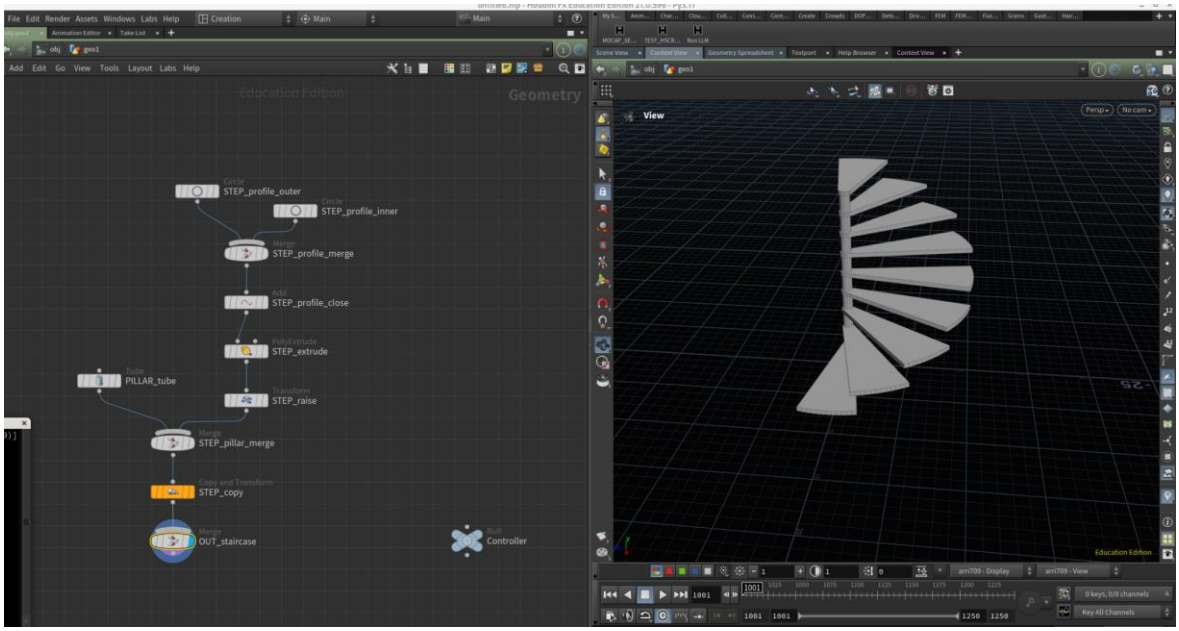
Chunking and Context Depth

Chunking exposed a second retrieval variable: how much context to return. A top-k = 1 run retrieved spiral_core.md and produced the core steps and central pillar. A top-k = 3 run also retrieved the railing and baluster documents and produced those additional components. Table 3 records the retrieved material, distances and token totals, and Figure 9 shows the two outputs. The comparison does not show that a higher top-k is generally better; the larger context was far more expensive and may introduce competing material. Retrieval quality and the generator's ability to use retrieved context remain separate questions. Figure 8 provides a related caution: a cloud-model response can execute with visible geometry while still producing an incorrect spiral shape.

Table 3. Recorded retrieval and context-depth cases. Distances and token totals are case-specific; higher top-k is not treated as generally better.

Stage	Condition	Top-ranked or retrieved knowledge	Distance	Total tokens	Observed result
Query ranking	Full code-generation prompt	staircase.md	0.5020	Not recorded	Intended spiral_core.md ranked lower at 0.5663.
Query ranking	Simplified query: 'Create a spiral staircase.'	spiral_core.md	0.2631	Not recorded	Ranking test only; no generation measured.
Context depth	top-k = 1	spiral_core.md	0.2719	4,989	Core steps and central pillar.
Context depth	top-k = 3	spiral_core.md; spiral_railing.md; spiral_baluster.md	0.2719; 0.2932; 0.3115	19,598	Steps, railings and balusters.

(a) top-k = 1



(b) top-k = 3

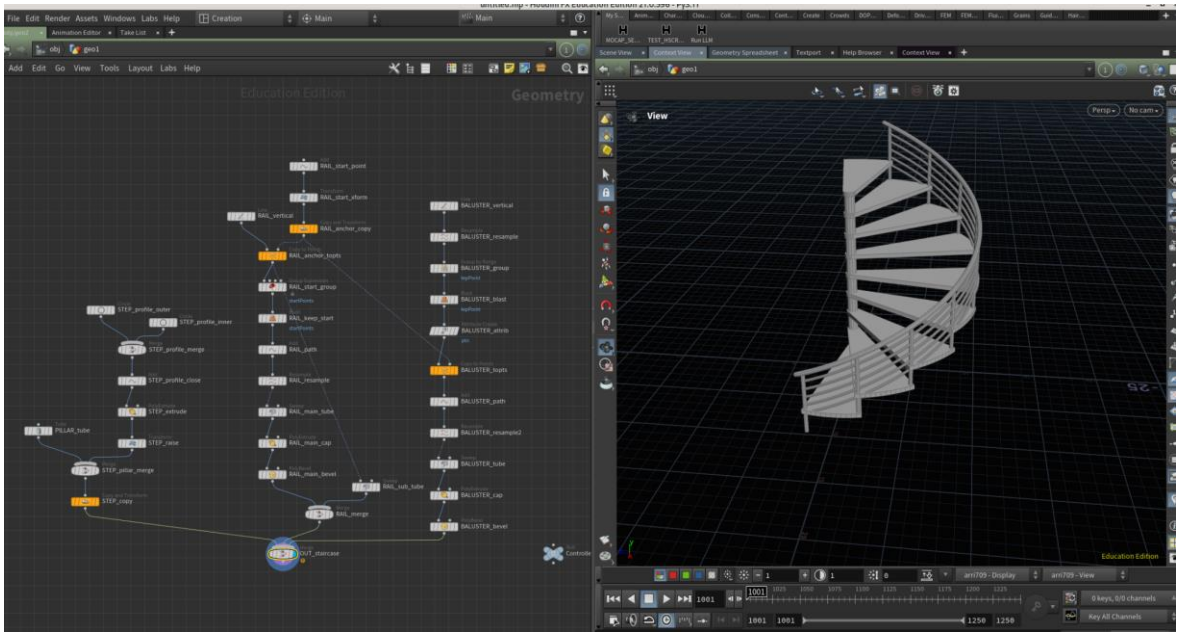


Figure 9. Recorded retrieval-depth comparison. (a) **The top-k = 1 run produces the staircase core and centre pillar. (b) The top-k = 3 run produces steps, railings and balusters.**

6. Discussion, Limitations and Future Work

6.1 Discussion

The experiments suggest that neither model size nor code executability is sufficient on its own to indicate task success. In the recorded spiral-staircase cases, a stronger cloud model could produce valid Python and VEX while still generating geometrically incorrect results,

whereas the smaller local model could capture parts of the intended procedural structure but fail on language or API details. This reinforces the need to evaluate generated Houdini procedures at both the code and modelling levels.

More broadly, the project raises a question about the role of AI within procedural content creation workflows. The potential value of a language model may lie less in replacing procedural modelling itself and more in interpreting user intent and translating it into appropriate procedural operations. For example, a request for a particular architectural style could involve selecting suitable procedural modules, choosing relevant parameters, or deciding when to use structures such as spiral staircases. In this role, the model acts as an interface between high-level intent and an existing procedural system.

The experiments also suggest that retrieval design cannot be considered independently of model capability. Prompt instructions, knowledge documents, chunking, top-k and possible future skill or repair mechanisms all influence how much procedural information is made available to the model and how effectively it can use that information. Larger models may tolerate longer or more complex context, while smaller local models may require more focused retrieval and stronger structural guidance. These questions remain open in the current prototype, but they define a practical direction for evaluating AI-assisted procedural workflows in terms of stability, efficiency and cost.

6.2 **Limitations**

The evaluation is limited by the small number of recorded cases and by the fact that experiments were conducted while the system was still evolving. Prompts, knowledge documents and model settings were not fixed across all comparisons, and several conditions were tested only once. Experimental records were also collected partly by hand, so token counts, timings and screenshots are available only for selected runs. These factors limit conclusions about stability, repeatability and comparative model performance.

The prototype also has practical limitations. The current GUI supports the basic prompt, retrieval, code preview and execution workflow, but experiment logging and performance measurement remain manual. The implementation was developed and tested in the current Linux laboratory environment and has not been validated across other operating systems or Houdini installations. Safety checks are limited to user review and a simple blocked-word mechanism, while the knowledge base remains relatively small and manually curated. Its retrieval behaviour is therefore sensitive to document structure, query formulation and the amount of context supplied to the model.

6.3 **Future Work**

Evaluation and Logging

A more systematic experiment pipeline is needed to make future comparisons reproducible. It should record prompts, retrieved documents and distances, model settings, token usage, generation time, generated code, execution errors and final outcomes in a structured format. Repeated runs under fixed conditions would make it possible to compare stability, cost and success across local and cloud models. Future evaluation should also distinguish between code execution, network validity and the quality of the resulting procedural geometry.

Retrieval, Model Capability and Repair

Future work should investigate how RAG configuration should change with model capability. Smaller local models may benefit from shorter, more focused context and stronger procedural guidance, while larger models may be able to use longer or more varied retrieved material. Further experiments could therefore compare document structure, chunk size, top-k and reranking across models under fixed conditions. The roles of prompts, retrieved knowledge and reusable procedural skills should also be separated more clearly. A repair loop could use Python, VEX or Houdini errors as feedback for another generation attempt, while preserving failed attempts for later analysis.

Interface, Safety and Workflow Integration

The prototype could also be developed toward a more practical procedural-modelling assistant. Generated code should be checked more carefully before execution while retaining explicit user confirmation. Cross-platform configuration would reduce the current dependence on the laboratory Linux environment, and the GUI could provide experiment history, model and retrieval settings, and direct comparison between outputs. More importantly, future work should examine where such an assistant is actually useful within a PCG workflow—for example, translating high-level user intent into procedural modules, parameter choices or asset variations—and whether the resulting reduction in manual work justifies the added generation time, token cost and iteration overhead.

7. Conclusion

This thesis explored whether retrieval-augmented generation can support procedural Houdini code generation, with a particular focus on a small locally hosted code model. The implemented prototype connected Houdini to an external RAG service, a manually curated

knowledge base and a local Qwen2.5-Coder-7B model, with selected cloud-model comparisons used for more demanding cases.

The experiments show that retrieval can be useful when the model lacks specific Houdini conventions or reusable procedural patterns. In the Copy to Points and straight-staircase cases, retrieved examples helped provide implementation knowledge that would otherwise need to be written directly into the prompt. However, the spiral-staircase experiments showed that making relevant knowledge available is not the same as being able to compose that knowledge into a correct procedure. Model capability, context size and the structure of retrieved knowledge all influenced the final result.

The project therefore suggests that RAG is best understood as a support mechanism for procedural code generation rather than a substitute for model capability. Its value lies in making domain knowledge available at inference time, while the language model remains responsible for interpreting, adapting and combining that knowledge into an executable Houdini workflow. For procedural content creation, this also means that successful code execution alone is not enough: the generated network must still produce the intended geometric and procedural result.

References

Gazdar, C. (2023) 'Procedural Staircase Modeling in Houdini: A Comprehensive Guide'. YouTube, 22 February. Available at: https://www.youtube.com/watch?v=VcAOqu7LYW8 (Accessed: 16 August 2026).

Hu, Z., Iscen, A., Jain, A., Kipf, T., Yue, Y., Ross, D. A., Schmid, C. and Fathi, A. (2024) 'SceneCraft: An LLM Agent for Synthesizing 3D Scenes as Blender Code'. Proceedings of the 41st International Conference on Machine Learning, PMLR 235, pp. 19252-19282. Available at: https://proceedings.mlr.press/v235/hu24g.html.

Kazama-Suichiku (n.d.) Houdini-Agent. GitHub repository. Available at: https://github.com/Kazama-Suichiku/Houdini-Agent (Accessed: 16 August 2026).

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W.-t., Rocktaschel, T., Riedel, S. and Kiela, D. (2020) 'Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks'. Advances in Neural Information Processing Systems, 33, pp. 9459-9474. Available at: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.-w. and Svyatkovskiy, A. (2022) 'ReACC: A Retrieval-Augmented Code Completion Framework'. Proceedings of the 60th Annual

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 6227-6240. doi: 10.18653/v1/2022.acl-long.431.

Lu, S., Chen, G., Dinh, N. A., Lang, I., Holtzman, A. and Hanocka, R. (2025) 'LL3M: Large Language 3D Modelers'. arXiv:2508.08228. Available at: https://arxiv.org/abs/2508.08228.

Sun, C., Han, J., Deng, W., Wang, X., Qin, Z. and Gould, S. (2023) '3D-GPT: Procedural 3D Modeling with Large Language Models'. arXiv:2310.12945. Available at: https://arxiv.org/abs/2310.12945.

Verstraete, S. (2020) 'Sci Fi Stair Generator'. SideFX Tutorials, 13 January. Available at: https://www.sidefx.com/tutorials/sci-fi-stair-generator/ (Accessed: 16 August 2026).

Wang, Z. Z., Asai, A., Yu, X. V., Xu, F. F., Xie, Y., Neubig, G. and Fried, D. (2025) 'CodeRAG-Bench: Can Retrieval Augment Code Generation?'. Findings of the Association for Computational Linguistics: NAACL 2025, pp. 3199-3214. doi: 10.18653/v1/2025.findings-naacl.176.

Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G. and Chen, W. (2023) 'RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation'. Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 2471-2484. doi: 10.18653/v1/2023.emnlp-main.151.

Zhou, S., Alon, U., Xu, F. F., Wang, Z., Jiang, Z. and Neubig, G. (2023) 'DocPrompting: Generating Code by Retrieving the Docs'. The Eleventh International Conference on Learning Representations (ICLR 2023). Available at: https://openreview.net/forum?id=ZTCxT2t2Ru.

Generative AI Use Statement

Generative AI tools were used during this project to support drafting, editing and code development. AI-generated or AI-assisted material was reviewed and modified by the author before inclusion. Generative AI was also used experimentally as part of the project itself, including local and cloud language models for Houdini code generation. Where applicable, AI-assisted code and content are acknowledged in accordance with NCCA coding standards.

Appendix A – Example RAG Knowledge Document

This appendix presents a representative knowledge document used by the retrieval-augmented generation (RAG) system. The example illustrates how procedural Houdini knowledge was prepared, structured and supplied to the generator as external retrieval context. The representative file is staircase.md.

A.1 Knowledge Sources and Preparation

The knowledge base was manually constructed from Houdini learning resources and project examples rather than from a single source. External source material included Chetal Gazdar's procedural staircase tutorial (Gazdar, 2023) and SideFX's Sci Fi Stair Generator tutorial by Simon Verstraete (Verstraete, 2020). Houdini-Agent (Kazama-Suichiku, n.d.) was used as a supporting tool during preparation of the knowledge material. Across the prepared cases, it assisted with tasks such as reading scene and node-network structure, renaming and organising nodes, and summarising procedural concepts and keywords. The resulting material was then manually reviewed and, where applicable, tested in Houdini before being reformatted as Markdown retrieval documents.

This workflow means that the knowledge base should not be interpreted as an automatically scraped corpus. It is a small, manually curated collection in which external learning material, scene information and project examples were transformed into retrieval-oriented procedural references.

A.2 Purpose

Create a procedural staircase in Houdini using repeated geometry.

A.3 Knowledge Type

- Type: Example
- Asset: Staircase
- Variant: Straight Basic

A.4 Key Concepts

- straight staircase
- linear repetition
- Box SOP
- Copy to Points
- repeated steps

A.5 Relevant User Requests

- Create a staircase.

- Generate procedural stairs.
- Create repeated steps.
- 创建程序化楼梯。
- 创建可调节台阶。

A.6 Recommended Node Pattern

1. Create one Geometry container under /obj.
2. Create one Box SOP as the source step geometry.
3. Create points representing step positions.
4. Use Copy to Points to copy the Box onto those points.
5. Set the final node as the display and render node.

A.7 Important Copy to Points Connection Rule

The knowledge document explicitly records the Houdini-specific input convention that became important in the Copy to Points experiments.

```
copy.setInput(0, box)       # input 0: source geometry
copy.setInput(1, wrangle)   # input 1: target points
copy.setDisplayFlag(True)
copy.setRenderFlag(True)
```

The source geometry must use input 0 and the target points must use input 1. These inputs should not be reversed.

A.8 Parameter Relationships

For step index i, the procedural relationship is represented by:

```
y = i * step_height
z = i * step_depth
```

The full example exposes step count, width, height and depth as controllable parameters.

A.9 Selected Houdini Python Excerpt

```
# Source step geometry
box = geo.createNode("box", "single_step")
# Target points
wrangle = geo.createNode("attribwrangle", "step_points")
# Copy to Points
copy = geo.createNode("copytopoints", "copy_steps")
copy.setInput(0, box)
copy.setInput(1, wrangle)
# Final output
merge = geo.createNode("merge", "OUT")
merge.setInput(0, copy)
merge.setDisplayFlag(True)
merge.setRenderFlag(True)
```

This excerpt is intentionally shortened for the appendix. The full knowledge document also contains the point-generation VEX, ground-plane construction, node layout, parameter defaults and immediate-execution code used in the original example.

A.10 Common Mistakes Recorded in the Knowledge Document

- Creating SOP nodes directly under /obj instead of inside a Geometry container.
- Reversing the Copy to Points inputs.
- Forgetting to set the final display flag.
- Creating geometry directly through node.geometry() instead of using SOP nodes.
- Using parameter names without verifying that they exist in the installed Houdini version.

A.11 Role in the Thesis

This document illustrates the type of manually prepared and reviewed procedural knowledge supplied to the generator at inference time. It supports the thesis interpretation of RAG as retrieval-supported pattern reuse rather than model learning.

A.12 Source References

Gazdar, C. (2023) Procedural Staircase Modeling in Houdini: A Comprehensive Guide. YouTube, 22 February 2023.

Kazama-Suichiku (n.d.) Houdini-Agent. GitHub repository. Accessed 16 August 2026.

Verstraete, S. (2020) Sci Fi Stair Generator. SideFX Tutorials, 13 January 2020.

Appendix B – Project Repository and Supplementary Materials

B.1 Project Repository

The complete project repository is available at: https://github.com/adalek/MasterProject

The repository contains the prototype source code, retrieval pipeline, knowledge files, experimental prompts, launch scripts and project documentation. It therefore acts as the main supplementary implementation artefact for the dissertation.

B.2 Repository Structure

- src/ Houdini GUI, FastAPI service, model communication, generation functions and response cleaning.
- rag/ RAG configuration, knowledge ingestion, embedding-based retrieval and prompt construction.
- knowledge/ Markdown knowledge documents used by the retrieval system.
- prompts/ Prompt files used during direct-generation and benchmark development.
- examples/full/ Long-form procedural examples used in selected experiments.
- scripts/ Shell scripts for starting the local model, starting the RAG server and rebuilding the vector database.
- docs/ Project and benchmark documentation.
- HoudiniCodes/ Additional Houdini-related code assets retained in the repository.

B.3 Main Implementation Files

src/ui.py – PySide6 Houdini interface.

src/rag_server.py – FastAPI endpoint used by the Houdini client.

src/ask_model.py – Model API communication and inference settings.

src/generate.py – Direct generation path.

src/generate_with_rag.py – Generation path with retrieval context.

src/clean_code.py – Response-cleaning utility.

rag/ingest.py – Knowledge ingestion and ChromaDB construction.

rag/retrieve.py – Embedding search and retrieval results.

rag/prompt_builder.py – Construction of the final generation prompt.

B.4 Reproducibility and Environment Notes

- Primary development environment: university Lab Linux.
- Houdini version recorded in the repository documentation: 21.0.596.

- Houdini Python version: 3.11.7.
- Local model: Qwen2.5-Coder-7B-Instruct-Q4_K_M.gguf.
- Local inference runtime: llama.cpp.
- Retrieval stack: Sentence Transformers and ChromaDB.
- Python dependencies are managed through uv, pyproject.toml and uv.lock.

The current repository includes environment-specific paths and launch assumptions from the university Lab Linux setup. As discussed in the thesis, the prototype has not yet been adapted or validated across multiple operating systems.

B.5 Supplementary Experimental Material

The repository provides supporting material that is too detailed to reproduce in the main thesis, including:

- original and revised benchmark prompts;
- RAG knowledge documents and chunked staircase examples;
- scripts used to start the local model and retrieval service;
- benchmark and development documentation;
- implementation code corresponding to the architecture described in Chapter 4.

B.6 Use of the Repository in Relation to the Thesis

The thesis presents the system design, selected experiments and bounded conclusions. The repository provides the corresponding implementation artefact and supplementary project material for readers who wish to inspect the code, prompt files and knowledge-base organisation in more detail. The repository should not be interpreted as replacing the evidence reported in the dissertation; rather, it complements the written evaluation.