MASTER IN ADVANCED COMPUTATION FOR ARCHITECTURE AND DESIGN

MaCAD - Thesis

2024

Vocal Commands for AutoDesk Revit

BARCELONA

Institute for advanced architecture of Catalonia

MASTER IN ADVANCED COMPUTATION
FOR ARCHITECTURE AND DESIGN

Project Title: Revit Voice

MaCAD 2024 - Thesis

Faculty: David Andres Leon

Faculty Assistant: Gabriella Rossi

Dominic Large

James Foo

INDEX

01 - Problem ····· pg. 11

02 - Proposed Solution ····· pg. 15

- 02 | 1 State of the Art

03 - Methodology ····· pg. 21

- 03 | 1 Technology Stack
- 03 | 2 Pseudo-Code

04 - Transcription ····· pg. 27

05 - Tokenizer ····· pg. 31

- 05 | 1 Tokenizer Outline
- 05 | 2 Variable Identification
- 05 | 3 Dictionary Creation

06 - LLM / RAG ····· pg. 39

- 06 | 1 LLM / RAG Pseudo-Code
- 06 | 2 RAG Overview
- 06 | 3 Elements to JSON
- 06 | 4 Vector Embedding
- 06 | 5 Query LLM

07 - Revit ····· pg. 51

08 - Conclusions ····· pg. 55

- 06 | 2 Comparisons
- 06 | 3 Next Steps

Abstract

In the past several years, Building Information Modeling (BIM) has continued to expand and transform the architectural and structural design worlds. This has been achieved through the use of software like AutoDesk Revit, ArchiCAD, and Vectorworks, however, the introduction of technology like machine learning (ML) and artificial intelligence (AI) have begun to make these programs out-dated in terms of efficiency and productivity. One major development in this regard is the advancing proficiency of speech recognition technology through the use of coding languages such as Python, C#, JavaScript and HTML. These advancements offer the potential to eliminate manual tasks, optimize workflows, and make BIM processes more widely accessible and user friendly.

This research project aims to understand the potential benefits a coding language like Python, in combination with both speech recognition technology and existing AutoDesk Revit software, might offer the architectural and design community. By using speech recognition to automate manual computer aided design (CAD), designers can accelerate their efficiency and production of future projects.

Research Question & Objective

The main question driving this research project is: *Can machine learning through the use of Python be used to create a trained model that successfully and quickly recognizes and implements commands within Revit Family parameters using only the Human Voice and a prompt?*

In order to answer this proposed research question, the main objective will be to explore the most proficient and accessible ways to implement speech recognition technology into a Revit friendly plug-in. This plug-in will be designed to automate specific tasks; Specifically, creating and / or modifying Revit family elements.

Thesis Milestones

In order to realize this project, three major milestones must be met; The first being **Voice-to-Text Conversion.** This milestone will require training a machine learning model to accurately transcribe spoken prompts and use grammatical practices to identify both specific words that refer to Revit family parameters and their corresponding values. A quick path to achieving this will likely be to use a natural language processing tool (NLP) that can be run within Python on virtually any machine with a microphone.

The second major milestone will be a **Text-to-Revit Command** process. Once prompts are recorded, they must be broken down and translated into a form that Revit will be able to understand and use as commands. The simplest route to achieve this will most likely be converting identified parameters and their values into a JSON file that can then be read by Revit using PyRevit.

Finally, successfully deploying a plug-in with a **Human-Centered User Interface** (UI). Ensuring that this tool is easy to use and can run efficiently for other designers will complete the scope of this project and prove the reason to use the tool and continue its expansion.

Future Opportunity

Due to the minimal time frame of the project, the initial focus will remain on successfully translating spoken words into viable commands within Revit, there will continue to be opportunity to expand the thesis.

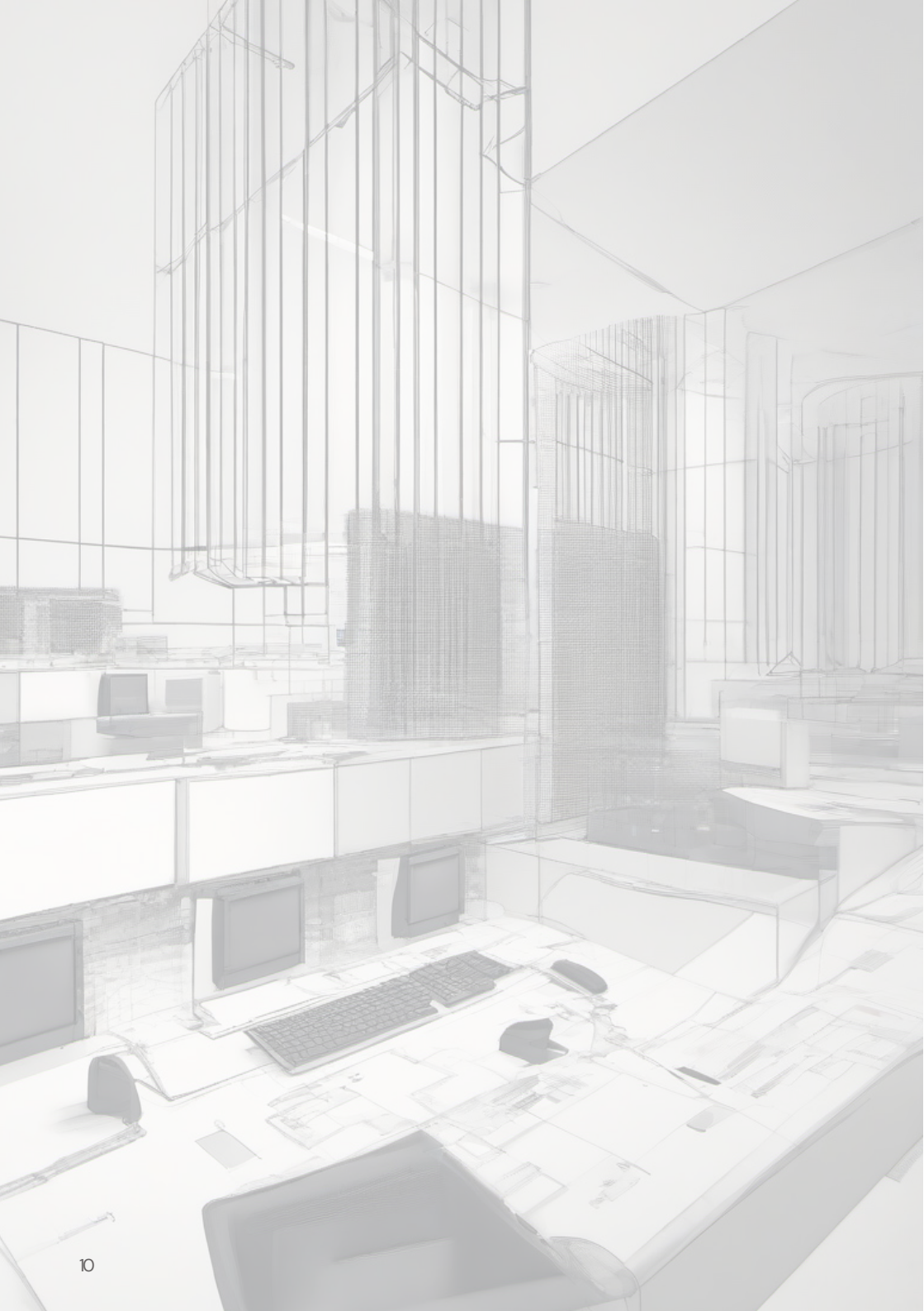
In addition to modifying single types of elements within a family and updating their parameters, the opportunity to expand **Workflow**

Automation would streamline user experience even farther. Expanding the intelligence of the model through further Python exploration, the model would be able to handle entire chains of command rather than being limited to single-sentence spoken prompts.

Finally, the task of **Scheduled Maintenance.** Keeping with regular updates to keep concurrent with adapting technologies and softwares will be essential in further developments of the tool.

Conclusion

The integration of Speech Recognition within Revit, powered by Python and ML, will dramatically increase the efficiency of designers. Not only will the opportunity to let Human-Centered UI drive advancements, designers will also be able to streamline their projects, saving time and money.



01 PROBLEM

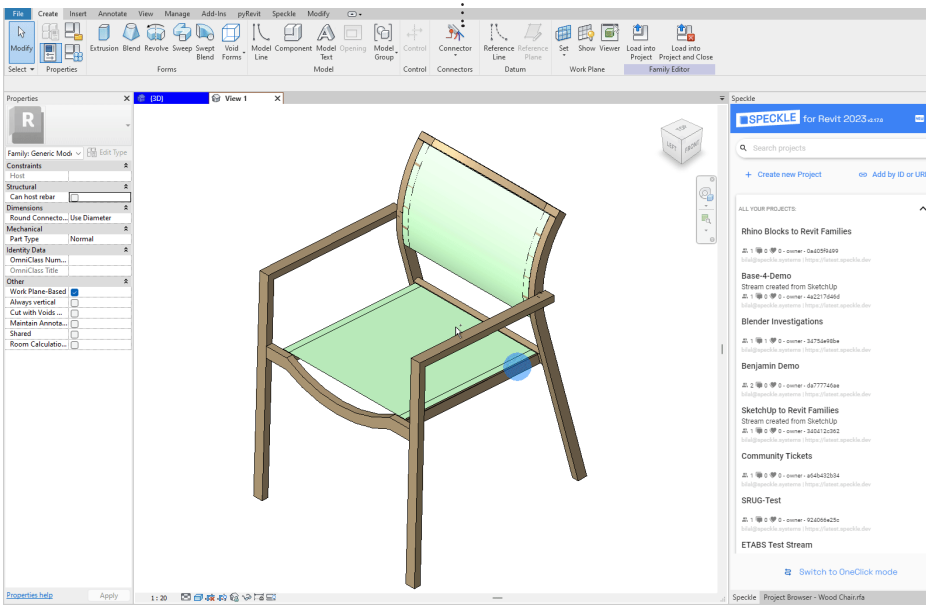
1979

2024

2060

?

RESEARCH FOCUS



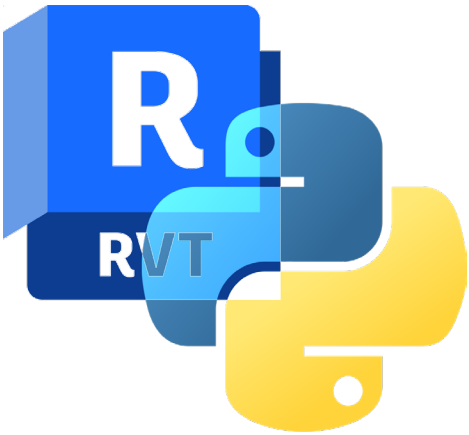
Unchanging User Experience

Despite decades of technological advancement, the core user experience between man and computer has remained nearly unchanged.

This tedious and repetitive process requires designers to spend extreme amounts of time that could be saved through updated methods.

This thesis explores a potential future for computer user experience, utilizing AI language models to interpret and execute commands.

02 SOLUTION



Voice 2 Command

Instead of the traditional method of manual CAD, the proposed combination of Speech Recognition, Python Coding, and AutoDesk Revit can **provide an alternative and more effective method of automated CAD.**

The automation of Revit commands through spoken prompts could lead to reduced design time required by Revit users. This will in turn produce an accelerated iteration process and faster turn around on future projects.

Speech Recognition

Python

+

Revit

STATE OF THE ART

Voice-to-Command

When considering current available technologies to consider as precedents for this research thesis, successful uses of Spoken Prompts translated into Computer Commands will be crucial. One impressive existing method is the Voice2CAD program. This is a stand alone software that allows users to program their own "spoken shortcuts" or "hot keys". While Voice2CAD does successfully interpret spoken prompts, the software specializes in dimension conversion; For example, instead of typing out "L2 [1/2 x 1 1/2 x 48] x (3)" to specify the dimensions of 3 members all labeled "L2", the user can simply speak that command: " Change the three L2 members to be one half by one and one half by 48." While this does save time in manual typing, this is a very specific case that has many limitations.

Another example of voice powered commands is the offering of Voice Recognition by AutoDesk within Fusion360. However, this product is still in its infancy with many limitations and inefficient results.

Natural Language Processing

In addition to Speech Recognition technologies offered by third party software, it is important to understand the Natural Language Processing (NLP) models that many of these programs utilize.

There are many NLP systems that can be utilized within Python code for Speech Recognition based on needs and uses. However, for the scope of this thesis, we will focus on three popular and effective NLP systems: nltk, spaCy, and Google Speech-Recognition.

While Google's Speech Recognition technology is extremely effective, the use of the product requires an active Internet connection as well as importing of specific Google dependencies. Because the goal of this product is to use the minimum required outlying dependencies and services, this avenue will be effective but not the main implementation to allow for maximum ease during user experience.

After researching the NLP spaCy, it became clear that this system was not only capable of easily running within a Python environment, but also extremely useful in terms of handling large amounts of information. However, due to its large amount of requires dependencies to be installed within the Python environment, the use of spaCy would likely lead to a drastic increase of CPU usage and time required to run.

Preliminary research and testing with the nltk NLP seemed to yield the best results. Because nltk requires minimal installed dependencies and is still able to handle lengthy volumes of information, this NLP will likely be the primary focus in initial Python iterations.

AI-Enhanced Modeling

Finally, when looking at precedents for artificial intelligence (AI) influenced programs that produce CAD models, AutoDesk's Fusion360 once again offers an example. While Fusion360 boasts Generative Design aided by AI within the native UI, it comes with its own limitations: The generative design tool is limited to basic functions and commands unless the user pays for additional subscriptions. In addition, the process remains reliant on the out dated manual CAD process.

Precedent Findings

When considering the current availabilities to designers, Speech Recognition and AI are being utilized but in very specific use cases with limited capabilities.

Taking these precedents and expanding upon their capacities will not only prove useful to this thesis, but future research as well.

Large Language Models

Large language models are advanced artificial intelligence systems designed to understand, generate, and manipulate human language. These models are built using massive neural networks trained on vast amounts of text data, allowing them to capture intricate patterns and nuances in language. They can perform a wide range of tasks, from answering questions and engaging in conversations to writing essays, translating languages, and even generating code.

The "large" in their name refers not only to the enormous datasets they're trained on but also to their immense number of parameters, often in the billions or trillions, which enable them to model complex linguistic relationships.

Vector Embedding

Building on the concept of large language models, their ability to understand and process language is deeply rooted in the use of vector embeddings. These embeddings are numerical representations of words or phrases in a high-dimensional space, where the relationships between words are encoded as mathematical relationships between vectors.

In this space, words with similar meanings or usage patterns are positioned closer together. This allows the model to capture the nuanced, contextual meaning of words, going far beyond simple dictionary definitions. For instance, the word "bank" might be represented differently depending on whether it's used in a financial context or referring to a river bank. The model can understand these distinctions by analyzing the surrounding words and the overall context of the sentence.

This contextual understanding enables LLMs to grasp subtle variations in meanings and even interpret figurative language to some extent. As a result, these models can engage in more human-like language processing, adapting their understanding based on the specific context in which words are used.

Vector Calculations

The vector representations in large language models enable powerful mathematical operations that contribute to the model's language understanding and generation capabilities. One of the most impressive examples is the ability to perform reasoning through vector calculations.

The classic example:

KING – MAN + WOMAN = QUEEN

This calcuation demonstrates how semantic relationships can be captured and manipulated in the vector space.

LLMs can also use vector similarity calculations, such as cosine similarity, to find words or concepts that are semantically related. This allows the model to suggest

synonyms, complete sentences, or find relevant information. Moreover, these vector operations extend beyond individual words to phrases and even entire sentences, enabling the model to understand and generate coherent text across longer contexts. A common use case of these vector similarity algorithms is in RAG, which will be discussed later in the thesis.

The model can also use these vector representations to perform tasks like clustering similar concepts, identifying outliers, or even translating between languages by mapping words from one language's vector space to another's. These mathematical operations on word vectors form the foundation of many advanced NLP tasks, from sentiment analysis to question answering, showcasing the power and versatility of vector-based representations in LLMs.

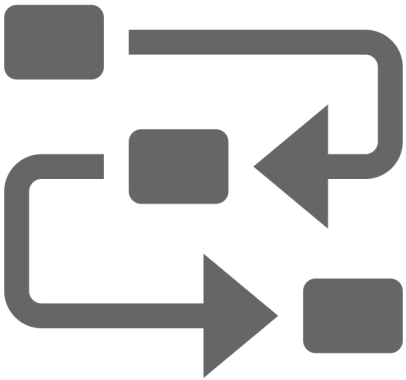
Popular Large Language Models

OPEN AI:
CHAT-GPT

ANTHROPIC
CLAUDE SONNET

META
LLAMA 3

03 METHODOLOGY



Divided Technology Stack

Dom - Python / Voice Tokenizer

Speech-to-Text

Audio / Prompt Recording

Text to Variables

Traditional coding method

Speech Tokenizers

Testing the performance of various speech tokenizers

Rhino.Inside Tests

Testing methodologies utilizing Rhino. Inside to execute the commands

Due to the scope of the thesis, encompassing both manually written Python code and Revit API knowledge, we determined that our time would most effectively be spent divided.

With both team members work on opposite ends of the technology stack, task items could quickly be accomplished alongside one another while working toward a common goal. This also allowed for effective collaboration to determine best paths forward for both James and Dom during weekly catch up meetings.

James - Revit / LLM

Revit API

Controls Revit application through Custom Scripting

Preliminary Research

PyRevit

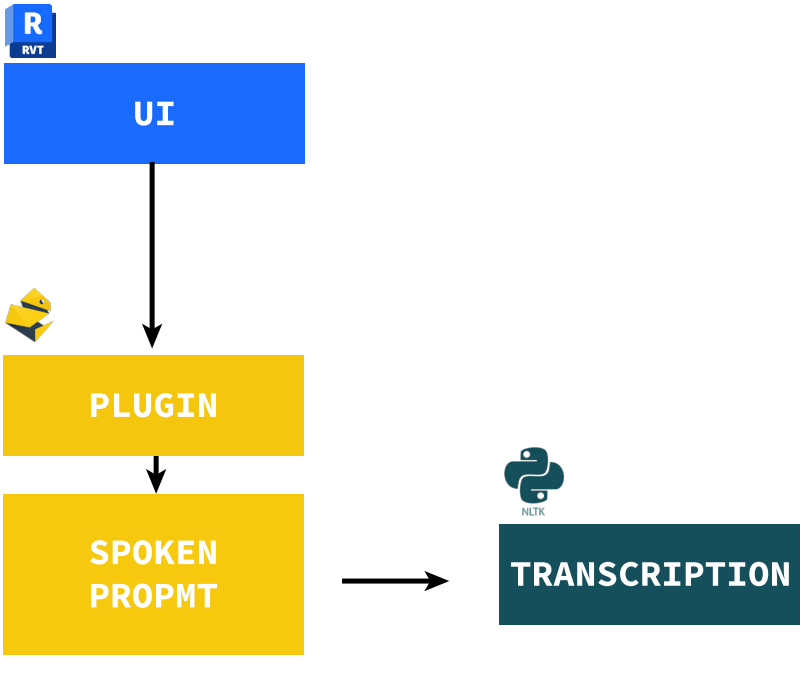
Running Python inside of the Revit environment

LLM / RAG

Encoding Revit file for LLM

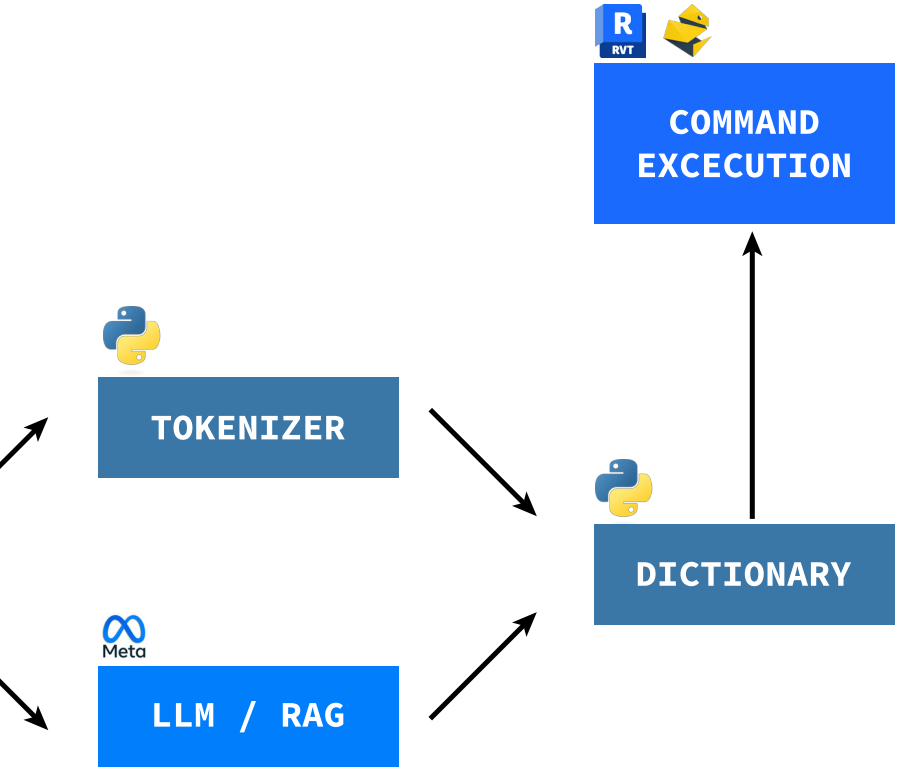
RAG retrieval of elements

Workflow Pseudo-Code

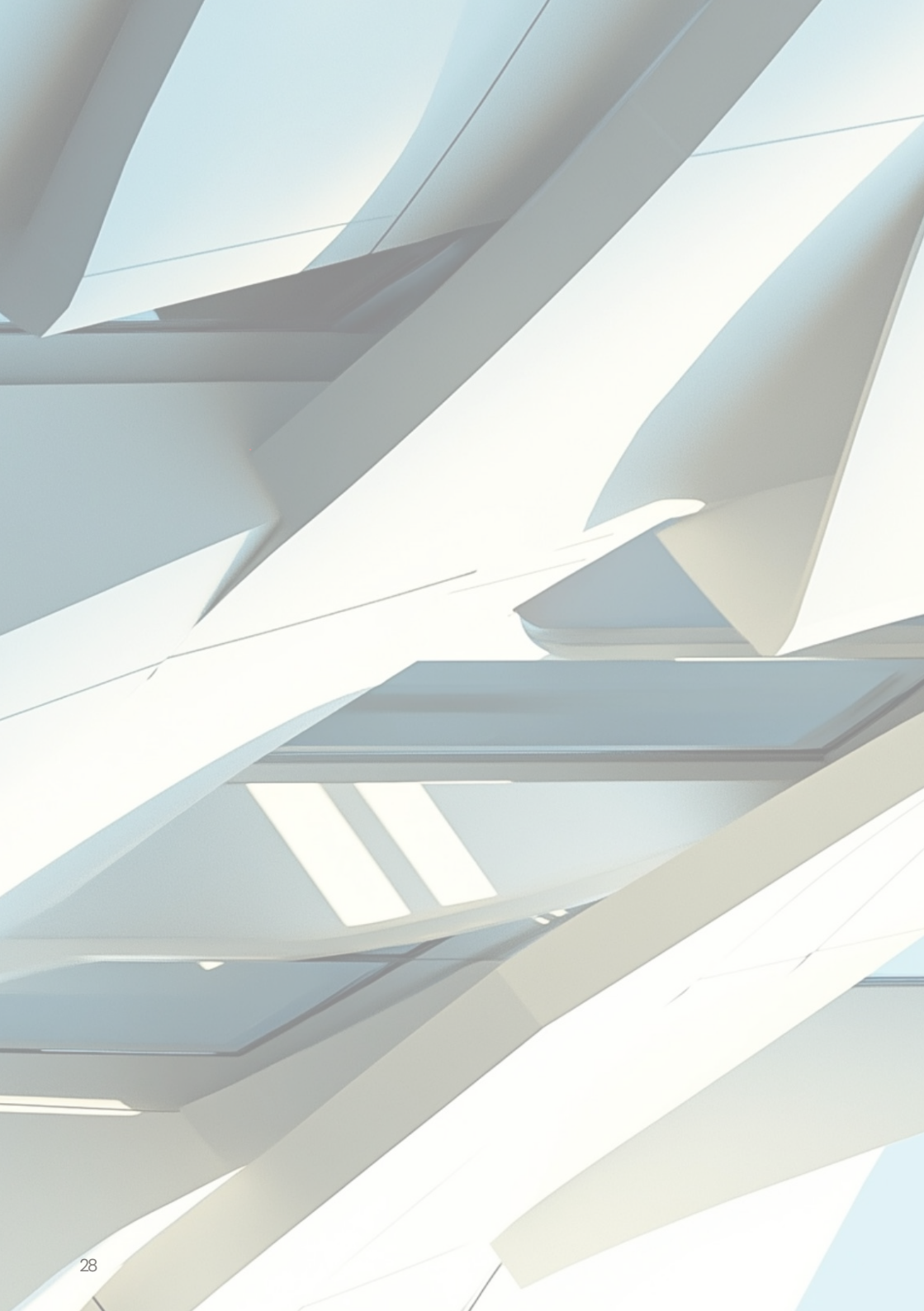


The diagram above shows the intended workflow process that will be executed during the final process. While being run inside the Revit environment, first the plug-in is installed via placing the dependent Python Notebooks into the Revit install directory. Once installed, the user will be able to easily record their prompt using their machine's microphone.

After successful installation and prompt recording, the plug-in then utilized the use of the nltk NLP in order to transcribe the user's spoken prompt. Once transcribed, one of two separate actions can be performed to create a usable Revit Dictionary: either Python is used to tokeninze specific phrases that are predefined, or a large language model (LLM) can be used in combination with a retrieval augmented generation (RAG) search to handle larger amounts of information.



Once the dictionary is created via one of the two processes, Revit is able to receive the dictionary and interpret its contents as commands and execute them.



04 Transcription

Transcription via NLTK

The diagram above shows the intended workflow process that will be executed during the final process. While being run inside the Revit environment, first the plug-in is installed via placing the dependent Python Notebooks into the Revit install directory. Once installed, the user will be able to easily record their prompt using their machine's microphone.

Recorder Class

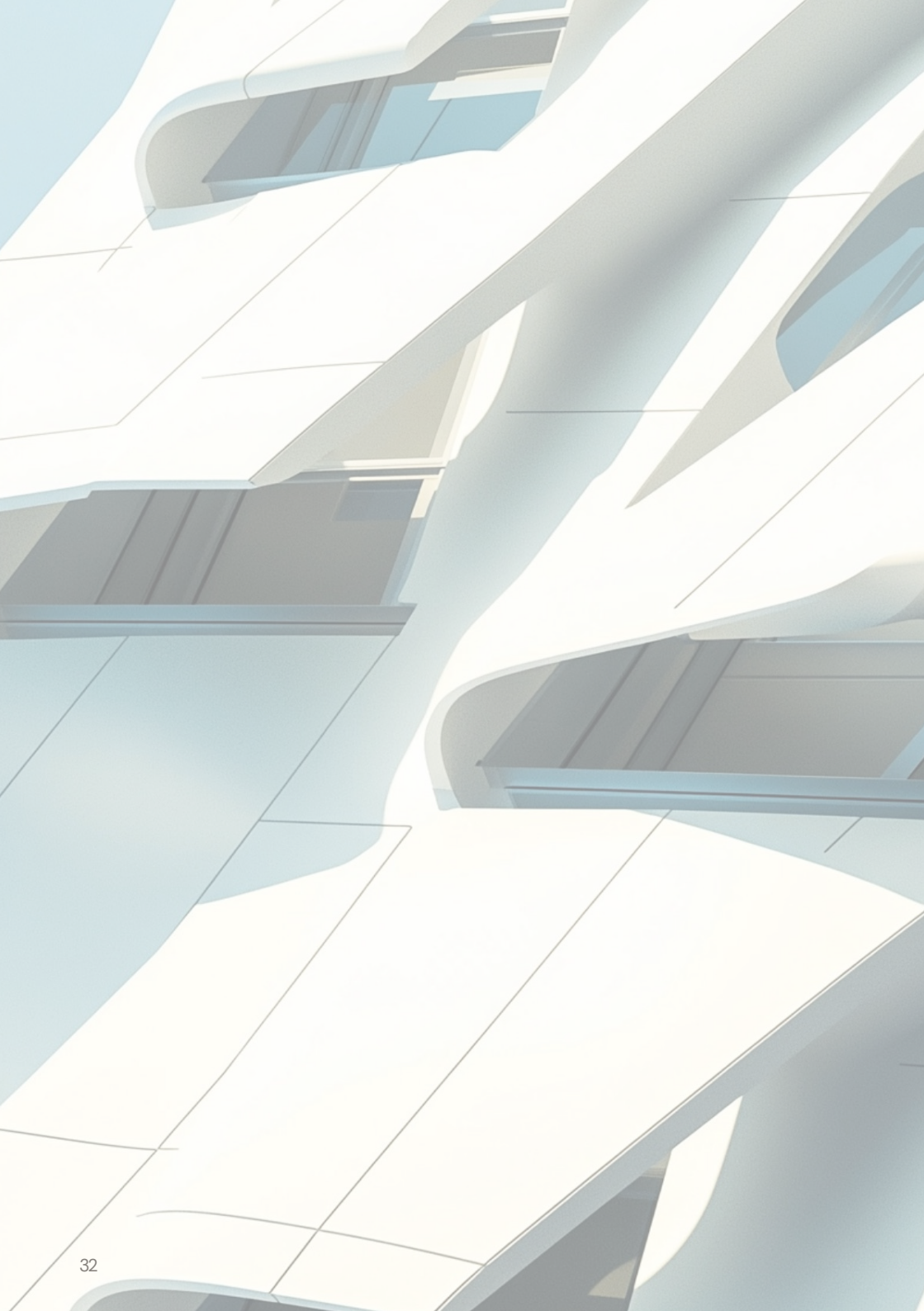
```
class Recorder:
    def __init__(self, silence_timeout=15):
        self.recognizer = sr.Recognizer()
        self.silence_timeout = silence_timeout  # Time to wait for silence before stopping

    def record_audio(self):
        with sr.Microphone() as source:
            print("Adjusting for ambient noise, please wait...")
            self.recognizer.adjust_for_ambient_noise(source)
            print("Recording... Speak now.")

            # Record audio until silence is detected
            audio_data = self.recognizer.listen(source, timeout=self.silence_timeout,
            phrase_time_limit=self.silence_timeout)
            print("Recording stopped.")

            return audio_data
```

Transcriber Class

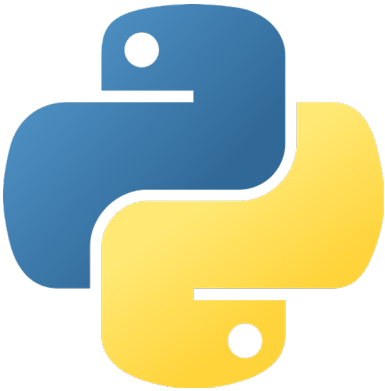
```
class Transcriber:
    def __init__(self):
        self.recognizer = sr.Recognizer()

    def transcribe_audio(self, audio_data):
        try:
            print("Transcribing...")
            prompt = self.recognizer.recognize_google(audio_data)
            print("You said: " + prompt)
            return prompt
        except sr.UnknownValueError:
            print("Google Speech Recognition could not understand the audio")
            return ""
        except sr.RequestError as e:
            print(f"Could not request results from Google Speech Recognition service; {e}")
            return ""
```

Transcription Output

```
Adjusting for ambient noise, please wait...
Recording... Speak now.
Recording stopped.
Transcribing...
You said: I need to change window number 13 to have a width of 2 ft
Transcribed Prompt: I need to change window number 13 to have a width of 2 ft
```



05 Tokenizer



Tokenizer Process Outline

DEPENDENCIES	RECORDER CLASS	TRANSCRIBER CLASS
nltk pyaudio / pydub SpeechRecognition word2number	Ambient Noise Detection “Recording... Speak Now” “Recording Stopped”	Speech Recognition “You Said _____”

The outline above illustrates the order of the Python code created to prioritize tokenizing specific words within a spoken prompt. This process ensures that viable parameters and their values can be pulled from the prompt and used within Revit. First, the necessary Dependencies are installed, enabling nltk and the corresponding requirements for Speech Recognition.

Next, the Recorder and Transcriber classes are defined; these classes allow for the Python environment to utilize the machine’s microphone in order to interpret what the user has spoken, and then translate that to a coded variable. The classes even offer options to update User Experience (UI) prompts such as: “Recording... Speak now” and “Recording Stopped”.

SPOKEN PROMPT	EXTRACT VARIABLES	DICTIONARY

Adjust for Ambient Noise
 	
You Said: “I need to change window number 13 to have a width of two feet instead of three”
	“I need to **change** **window** number 13 to have a **width of two feet** instead of three.”	{
		“Modify”
		“Window: 13”
		“Width: 2 ft”
		}

Once the two classes are defined, the user is able to proceed with speaking there prompt for their Revit command. After being activated, the code prompts the user to be silent to account for ambient noise before allowing the recording to begin. The model is also trained to know when the user has stopped speaking according to comparison with the measured ambient noise. Finally, the variable “prompt” is displayed and cached in the code, ready to be referenced.

After the variable is created, the tokenized values are compared to the variable to identify the parameters that need to be adjusted, as well as their corresponding values. Finally, all of this information is organized and cached within a JSON dictionary that is readable by Revit.

Variable Identification

Create or Modify

```
# Define keywords for actions
action_keywords = {
    "Create": ["create", "make", "build", "design", "generate"],
    "Modify": ["modify", "change", "adjust", "alter", "update"]
}
```

Input:
Processing text: “I need to update tables 5, 11, and 17”

Output:
Detected Action: “Modify”
Modified IDs: [5, 11, 17]

Because the approach uses ‘token’ words to identify variables and their values, those variables need to be identified.

The Python code above shows the defined token words to identify whether an item is to be Created or Modified. However, in order to expand the capabilities of the code, additional token words are included for each of the two options. This ensures that users do not have to adhere to the same prompt each time a command is spoken. Of course, these values are able to be updated for further expansion.

Variable / Value Identification

```
# Define possible synonyms for each parameter
parameter_synonyms = {
    "Height": ["height", "tall", "elevation", "high"],
    "Corner Fillet": ["corner fillet", "fillet", "rounded corner", "corner fillets", "fillets"],
    "Leg Insert": ["leg insert", "leg distance", "leg spacing", "inserts", "leg insert distance of "],
    "Sides": ["sides", "edges"],
    "Thickness": ["thickness", "thick", "depth"]
}
```

Input:
Processing text: “I need to update tables 5, 11, and 17”

Output:
{
“Height”: “3 feet”,
“Corner Fillet”: “5.25 inches”,
“Leg Insert”: “1.25 inches”,
“Sides”: “3.25 feet”,
“Thickness”: “0.75 inches”
}

Once the code identifies whether or not elements are to be created or modified, the variables themselves are identified. Similarly to the action keywords, parameter keywords are defined as well as their possible synonyms.

Once identified with their values, Parameters are organized in an easy to understand format, both for humans and Revit.

Dictionary Creation

Create

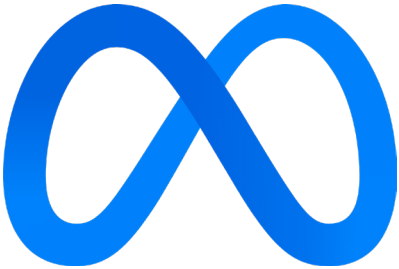
```
{
        “Action”: “Create”, .....
        “Table IDs”: [],.....
        “Parameters”: {.....
                “Height”: {
                “ParameterName”: “Height”,
                “Value”: “3”,
                “Unit”: “feet”
        },
        “Thickness”: {
                “ParameterName”: “Thickness”,
                “Value”: “0.75”,
                “Unit”: “inches”
        },
        “Sides”: {
                “ParameterName”: “Number of
Sides”,
                “Value”: “3.25”,
                “Unit”: “feet”
        },
        “Leg Insert”: {
                “ParameterName”: “Leg Insert”,
                “Value”: “1.25”,
                “Unit”: “inches”
        },
        “Corner Fillet”: {
                “ParameterName”: “Corner
Fillet”,
                “Value”: “5.25”,
                “Unit”: “inches”
        }
}
```

Once all the information is collected and ready, the Python code is then trained to organize and export the information in a Revit friendly dictionary. This ensures that the commands within Revit will successfully receive the information necessary to make updates based on the Spoken Prompt.

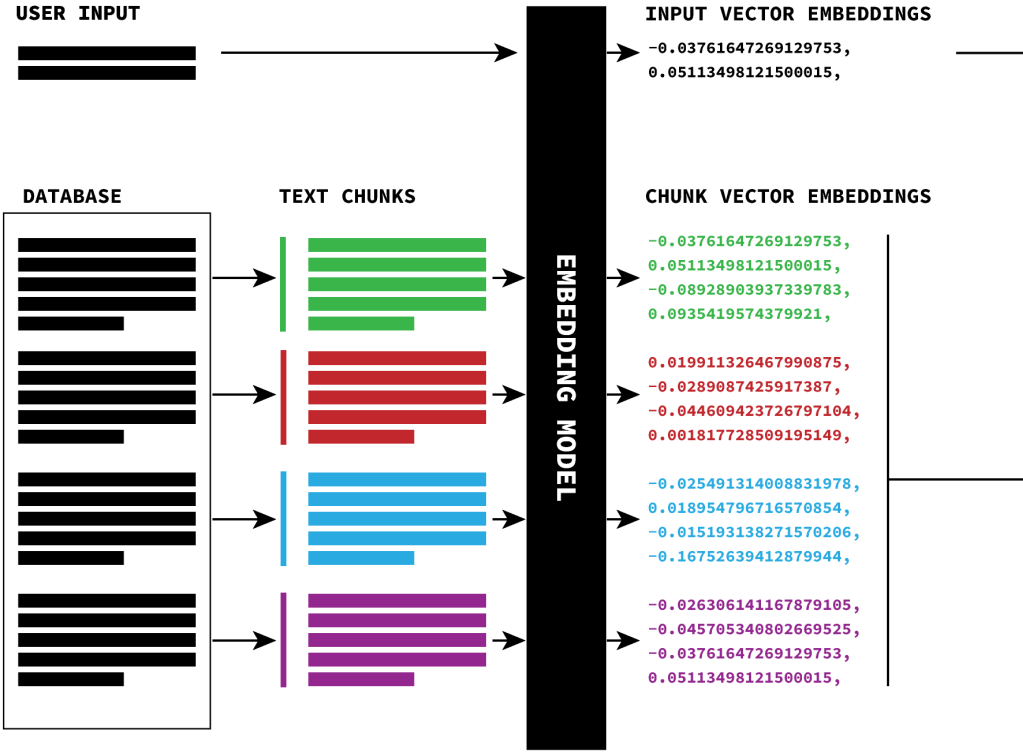
Modify

```
    {
..... “Action”: “Modify”,..... ACTION
..... “Table IDs”: [5, 11, 17],..... ELEMENT IDS
..... “Parameters”: {..... PARAMETERS
                    “Height”: {
                    “ParameterName”: “Height”,
                    “Value”: “3”,
                    “Unit”: “feet”
            },
            “Thickness”: {
                    “ParameterName”: “Thickness”,
                    “Value”: “0.75”,
                    “Unit”: “inches”
            },
            “Sides”: {
                    “ParameterName”: “Number of
    Sides”,
                    “Value”: “3.25”,
                    “Unit”: “feet”
            },
            “Leg Insert”: {
                    “ParameterName”: “Leg Insert”,
                    “Value”: “1.25”,
                    “Unit”: “inches”
            },
            “Corner Fillet”: {
                    “ParameterName”: “Corner
    Fillet”,
                    “Value”: “5.25”,
                    “Unit”: “inches”
            }
    }
```

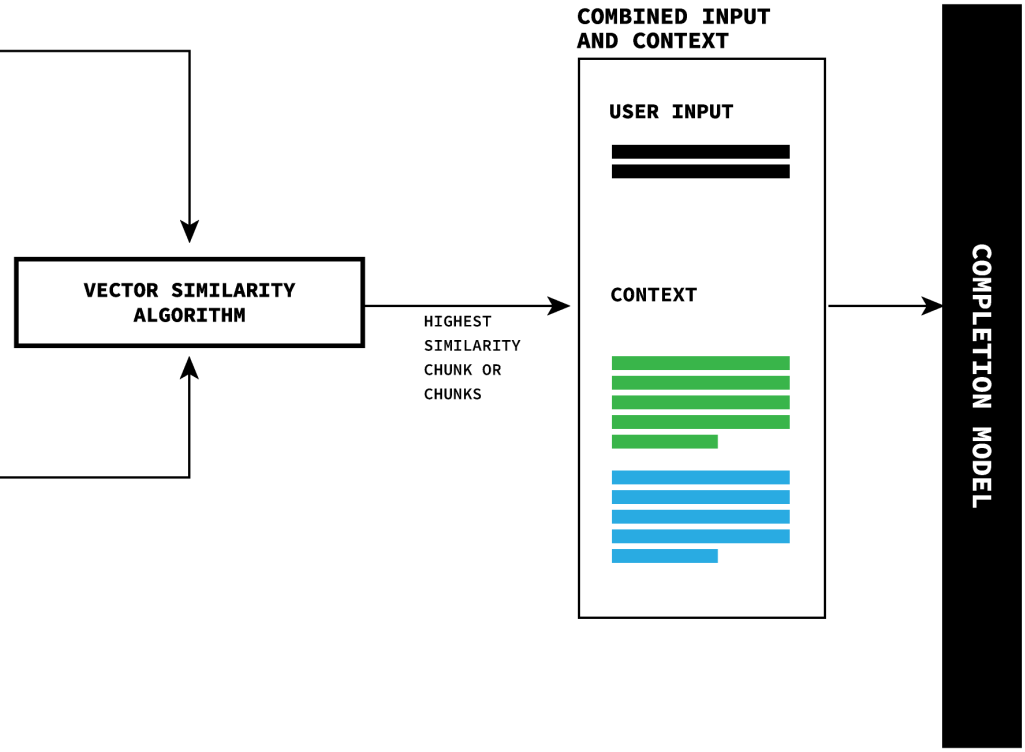
06 LLM / RAG



Retrieval Augmented Generation (RAG) Overview

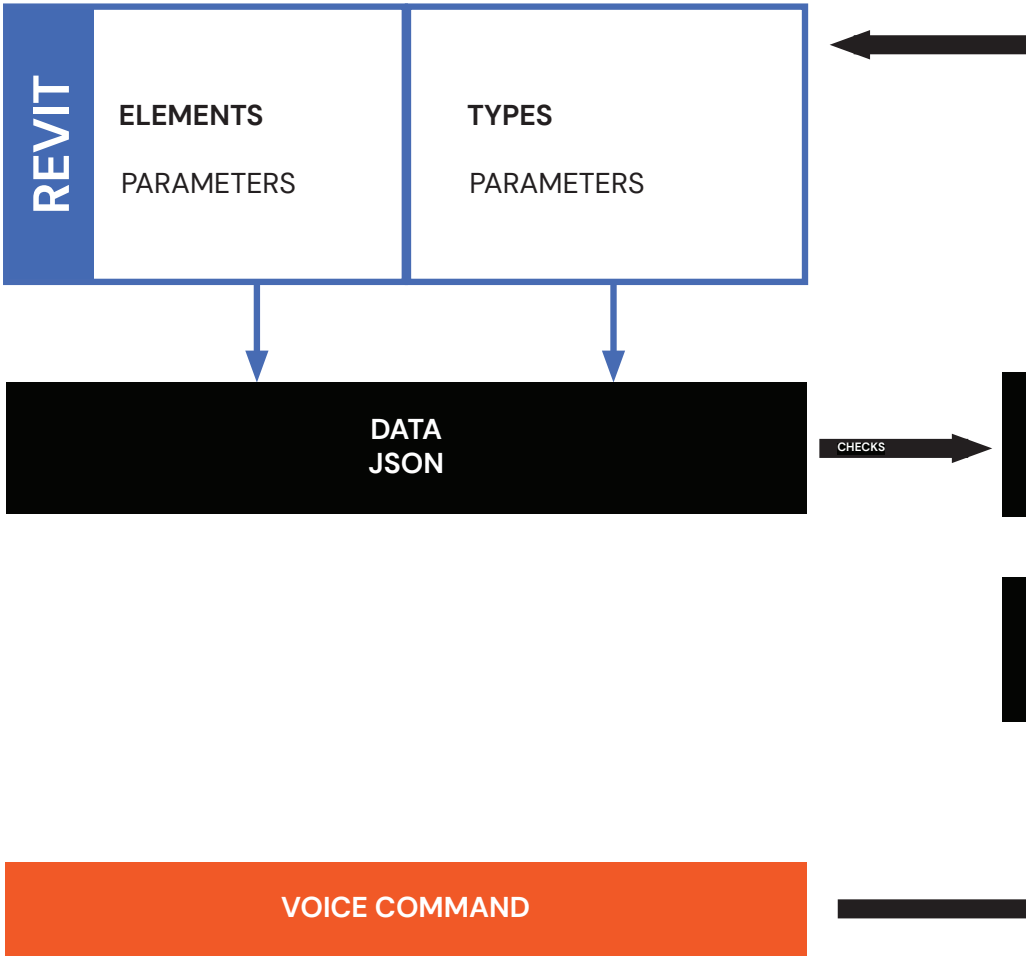


RAG (Retrieval Augmented Generation) is a enhancing an LLM's response and accuracy by connecting it to a external database. The first step in the process is to break the data base into different chunks of text, so specific information is discernible to the LLM model. Next, an embedding model is used to get the vector representations of the text chunks and are stored as a separate file. Simultaneously, the embedding model retrieves the vector embedding of the user's input.

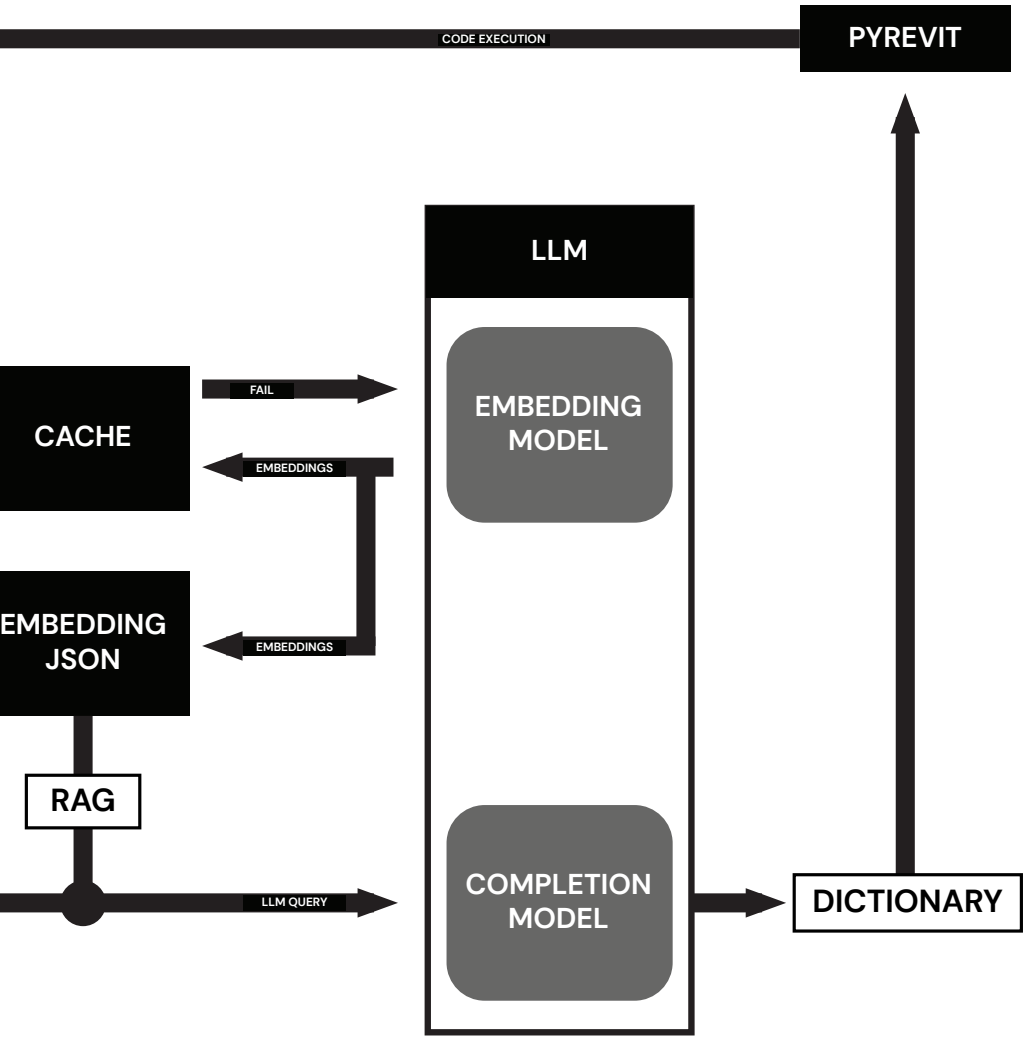


Once both the database and user input is converted to a vector similarity algorithm is used to determine which chunks from the data base are most relevant to the user input. These chunks are combined then combined with the user input and sent to the completion LLM, which returns a response.

LLM / RAG Pseudo-Code

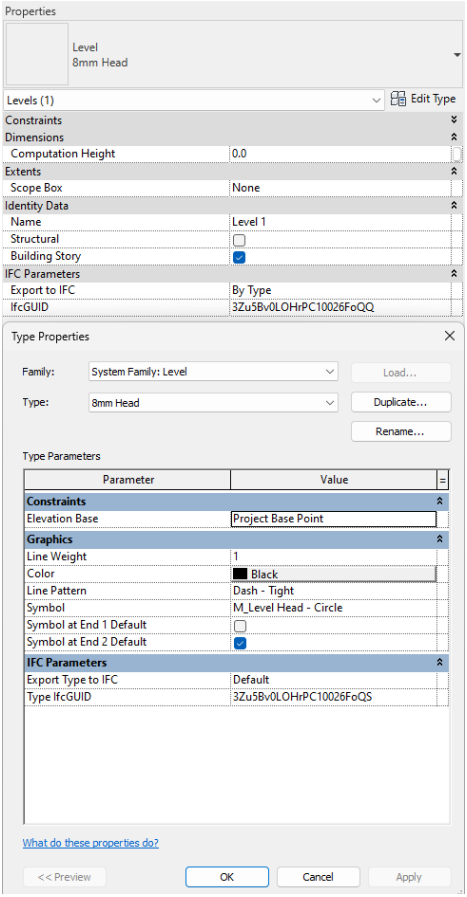


An alternative to using a tokenizer to identify variables, we can also use a Large Language Model (LLM). Rather than manually specifying keywords and actions, we can utilize an LLM such as Ollama or Mistral with Retrieval Augmented Generation (RAG) to identify and or key words within the Spoken Prompt.



For the Revit file to be used as a RAG database, relevant element data is extracted and stored in a JSON file. Each element is treated as a chunk of text in for the RAG. These chunks are then processed by the embedding model and stored in memory cache. This caching system prevents repeated texts chunks from being sent to the embedding model. Finally, the LLM uses RAG to create the dictionary used to edit the Revit elements.

Elements to JSON



When exporting elements into JSON format, the script extracts all of the element's instance and type parameters as well as important identifiers such as it's ID and Name. This allows for easy retrieval of data later.

```
{
  "MODEL_ELEMENTS": [
    {
      "ID": 311,
      "Name": "Level 1",
      "Category": "Levels",
      "Instance_Parameters": {
        "Category": "-2000240",
        "Export to IFC": "0",
        "IfcGUID": "3Zu5Bv0LOHrPC10026FoQQ",
        "Design Option": "-1",
        "Scope Box": "-1",
        "Name": "Level 1",
        "Structural": "0",
        "Building Story": "1",
        "Story Above": "-1",
        "Elevation": "0.0",
        "Computation Height": "0.0",
        "Family and Type": "305",
        "Family": "305",
        "Type": "305",
        "Family Name": "N/A",
        "Type Name": "N/A",
        "Type Id": "305"
      },
      "Type_Parameters": {
        "Category": "-2000240",
        "Export Type to IFC": "0",
        "Type IfcGUID": "3Zu5Bv0LOHrPC10026FoQS",
        "Design Option": "-1",
        "Symbol at End 1 Default": "0",
        "Symbol at End 2 Default": "1",
        "Elevation Base": "0",
        "Symbol": "24867",
        "Line Pattern": "498286",
        "Color": "0",
        "Line Weight": "1",
        "Family Name": "Level",
        "Type Name": "8mm Head"
      }
    },
```

Vector Embedding

Here is an example of the data before and after the vector embedding process.


```
{
 "MODEL_ELEMENTS": [
  {
   "ID": 311,
   "Name": "Level 1",
   "Category": "Levels",
   "Instance_Parameters": {
    "Category": "-2000240",
    "Export to IFC": "0",
    "IfcGUID": "3Zu5Bv0LOHrPC10026FoQQ",
    "Design Option": "-1",
    "Scope Box": "-1",
    "Name": "Level 1",
    "Structural": "0",
    "Building Story": "1",
    "Story Above": "-1",
    "Elevation": "0.0",
    "Computation Height": "0.0",
    "Family and Type": "305",
    "Family": "305",
    "Type": "305",
    "Family Name": "N/A",
    "Type Name": "N/A",
    "Type Id": "305"
   },
   "Type_Parameters": {
    "Category": "-2000240",
    "Export Type to IFC": "0",
    "Type IfcGUID": "3Zu5Bv0LOHrPC10026FoQS",
    "Design Option": "-1",
    "Symbol at End 1 Default": "0",
    "Symbol at End 2 Default": "1",
    "Elevation Base": "0",
    "Symbol": "24867",
    "Line Pattern": "498286",
    "Color": "0",
    "Line Weight": "1",
    "Family Name": "Level",
    "Type Name": "8mm Head"
   }
  },
```

```
{
  "content": "Element ID: 311\nName: Level 1\nCategory: Levels\nInstance Parameters:\n  Category: -2000240\n  Export to IFC: 0\n  IfcGUID: 3Zu5Bv0LOHrPC10026FoQQ\n  Design Option: -1\n  Scope Box: -1\n  Name: Level 1\n  Structural: 0\n  Building Story: 1\n  Story Above: -1\n  Elevation: 0.0\n  Computation Height: 0.0\n  Family and Type: 305\n  Family: 305\n  Type: 305\n  Family Name: N/A\n  Type Name: N/A\n  Type Id: 305\nType Parameters:\n  Category: -2000240\n  Export Type to IFC: 0\n  Type IfcGUID: 3Zu5Bv0LOHrPC-10026FoQS\n  Design Option: -1\n  Symbol at End 1 Default: 0\n  Symbol at End 2 Default: 1\n  Elevation Base: 0\n  Symbol: 24867\n  Line Pattern: 498286\n  Color: 0\n  Line Weight: 1\n  Family Name: Level\n  Type Name: 8mm Head",
  "vector": [
    -0.03761647269129753,
    0.05113498121500015,
    -0.08928903937339783,
    0.0935419574379921,
    0.019911326467990875,
    -0.0289087425917387,
    -0.044609423726797104,
    0.001817728509195149,
    -0.025491314008831978,
    0.018954796716570854,
    -0.015193138271570206,
    -0.16752639412879944,
    -0.026306141167879105,
    -0.037159327417612076,
    -0.03689078986644745,
    -0.030796362087130547,
    0.08261997252702713,
    0.041862212121486664,
    0.009968443773686886,
    -0.05334179848432541,
    -0.014689642004668713,
          ...
```

LLM Query

INPUT

```
prompt = f"""Based on the following context and verbal command,
output ONLY a Python dictionary with this exact format:
    {{
        "Command": "command_here",
        "ID": "id_here",
        "ElementType": "elementtype_here",
        "Family": "element_family_here",
        "Type": "type_here",
        "TypeID": "type_id_here",
        "Parameters": {{
            "ParameterName1": "Value1_units",
            "ParameterName2": "Value2_units"
        }},
        "Identifiers": ["identifier1", "identifier2"]
    }}

    IMPORTANT:

    1. Use the exact ID, Type ID, and Parameter names from the
context if available.
    2. Use any identifiers (such as Mark, Family and Type,
Type Mark, etc.) provided in the context to determine which
element is being referred to.
    4. If a parameter value is changed by the command, reflect
this in the output.
    5. Do not include any explanatory text. The output should
be a valid Python dictionary and nothing else.
    6. Always include units after a parameter value if they
are provided in the context.
    7. In the "Identifiers" list, include any identifying
information used to select this element.
    8. Units should be either feet (ft), Inches (in), Meters
(m) or Millimeters (mm)

    Context:
    {context} ———————————————— RAG OUTPUT

    Verbal Command: {query} ———————————— USER INPUT [VOICE]

    Python Dictionary Output:"""
```

OUTPUT

```
{
 “Command”: “Change all ottomans width to
be 4 feet”,
 “ID”: “983518, 982743”,
 “ElementType”: “Furniture”,
 “Family”: “Sofa - Ottoman”,
 “Type”: “W600XD600 2, W600XD600”,
 “TypeID”: “983518, 982743”,
 “Parameters”: {
 “Width”: “4 ft”
 },
 “Identifiers”: [
 “Similarity: 0.54”,
 “Similarity: 0.53”
 ]
}
```

The prompt on the left is the current version that produces the required dicitonary for editing Revit elements. It includes instructions on how the format the dictionary and what information to look for within the context.

Above the an example of a dicitionary output by the LLM.

07 Revit

Update Elements

EXTRACTS ELEMENT IDS

COLLECT ELEMENTS

UPDATE ELEMENT PARAMETERS

The process of running the dictionary within Revit relatively simple. It finds the right elements using the ID's in the dictionary looks for and exchanges the parameters specified by the dictionary.

```
def update_revit_element(doc, uidoc, llm_output):
    element_ids = extract_id(llm_output.get('ID')) if llm_output.get('ID') else None
    type_ids = extract_id(llm_output.get('TypeID')) if llm_output.get('TypeID') else None
    parameters_to_update = llm_output.get('Parameters', {})

    if not element_ids and not type_ids:
        print("No ElementID or TypeID provided in LLM output.")
        return

    from Autodesk.Revit.DB import ElementId, Transaction

    element_ids = [ElementId(id) for id in element_ids]
    type_ids = [ElementId(id) for id in type_ids] if type_ids else []

    elements = [doc.GetElement(id) for id in element_ids if doc.GetElement(id) is not None]
    element_types = [doc.GetElement(id) for id in type_ids if doc.GetElement(id) is not None]

    if not elements and not element_types:
        print(f"No elements or element types found with provided IDs.")
        return

    t = Transaction(doc, "Update Element Parameters")
    t.Start()

    try:
        for element in elements:
            update_element_parameters(doc, element, parameters_to_update)

        for element_type in element_types:
            update_element_parameters(doc, element_type, parameters_to_update)

        t.Commit()
        print(f"Update process completed for {len(elements)} elements and {len(element_types)}
element types")
    except Exception as e:
        t.RollBack()
        print(f"Error updating elements/types: {str(e)}")
```

08 Conclusions

Comparisons

While both the Tokenizer and LLM /RAG methods of Voice-to-Command within Revit proved successful, they come with their own use cases, advantages, and disadvantages.

Tokenizer

Python code utilizing NLPs such as nltk and spaCy are mainly beneficial for adjusting universal parameters such as: Width, Height, Length, Offset etc. More complex parameters with specialized characters and complex tasks often lead to errors.

Starting with the first tested approach, the Python Tokenizer utilizing nltk; The primary benefit of this approach, was the fact that it performed significantly faster when executing a spoken command.

However, while the Tokenizer performed actions with incredibly high speeds, it was heavily reliant on a high level of detail and precision when defining words to act as tokens within spoken prompts. In addition, the token method often struggled with specialized parameters; Because this method used patterns and token words, variations to these spoken prompts might cause incorrect commands to be executed.

LLM / RAG

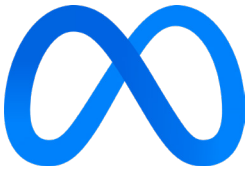
When considering the LLM approach combined with RAG searches, this method is preferred for complex tasks that the Tokenizer cannot handle. This is due to the use of the LLM and its extensive knowledge of predefined language.

The key two advantages of the use of LLM and RAG are flexibility and complexity; Because the LLM does not need to use token words defined in the code, it is far more capable in understanding drastically different spoken prompts. Not only is does the LLM provide

a deeper understanding of language, it also is able to interpret mispronounced words as well as unexpected synonyms. These two advantages make the Speech Recognition within the LLM model far superior to the Tokenizer method.

Although the LLM / RAG combination offers more complex language understanding, it still comes with its own drawbacks. The main disadvantage being, it is often slower to execute commands than the Tokenizer. In addition to lack of speed, this model often struggles find highly specific data and element IDs, which can leading to inaccuracies and inconsistency.

Additionally, due to RAG's chunk retrieval processing grabbing only a limited number of chunks, a limited number of elements can be edited at a time. While the number of chunks retrieved can be increased, this significantly slows the LLM's response time.



Next Steps

Python

There are several steps that could to taken to continue this research thesis in a meaningful and productive process. First, to improve the functionality of the Python Tokenizer model, utilization of an updated NLP system instead of nltk or spaCy could improve the token creation process to allow for a deeper understanding of Spoken Prompts.

Next, the combination of an NLP system with a more limited version of an LLM, specific to construction focused language. This combination could benefit from the adaptability of an LLM, while maintaining its speed when executing spoken commands.

LLM

While the Python improvements are somewhat simple and straightforward, the LLM model will require more in depth exploration.

First, the model simply needs to be fine tuned to improve the accuracy of the output. This could

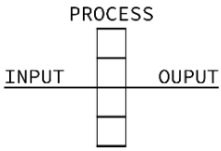
After model tuning, the LLM process could substantially benefit from improved prompt engineering. Tweaking the overall prompt that is fed into the LLM model may produce more consistently accurate results.

The LLM process may also be improved upon by testing with different LLM models. Currently, Ollama and Mistral have been used to test as database models for the LLM. However, there may be models that are substantially better suited for the use case of Voice-to-Command within Revit. During the scope of the thesis, experimentation was greatly limited by computer speed and resources. In addition to a new model, improved machinery would enhance the

capabilities and speed of the command execution.

Additionally, writing the code in C# rather than Python could speed up the process. Revit is built on the .NET framework, meaning the Python code needs to be run through an interpreter (IronPython). Translating the code from Python to C#, .NET's native programming language, would result in faster run times.

Finally, the implementation of a multi-threading caching process would drastically reduce the required time to execute commands via the LLM model. Specifically, using multiple CPU threads to checked the cached vectors would likely decrease the required time to run.



REFERENCES

"Benefits Icon - Free PNG & SVG 6176506 - Noun Project." The Noun Project, https://thenounproject.com/icon/benefits-6176506/. Accessed 13 Sept. 2024.

"Demo Icon - Free PNG & SVG 1626811 - Noun Project." The Noun Project, https://thenounproject.com/icon/demo-1626811/. Accessed 13 Sept. 2024.

Dominic, Large. 13 Sept. 2024, https://www.midjourney.com/jobs/dd26592b-a4b1-4e83-8dd2-c87a29756db3?index=0.

Generative Design AI Software. https://www.autodesk.com/solutions/generative-design-ai-software. Accessed 14 Sept. 2024.

GOKER, Mucahit Bilal. Rhino Block to Revit Family. 4 Jan. 2024, https://speckle.systems/tutorials/rhino-block-to-revit-family/#sending-from-rhino.

Google AI Vector Logo - Download Free SVG Icon | Worldvectorlogo. https://worldvectorlogo.com/logo/google-ai-1. Accessed 15 Sept. 2024.

"How It Works." Voice Activated Software | Voice2CAD, https://voice2cad.com/how-does-it-work/. Accessed 14 Sept. 2024.

How to Customize Command Aliases in AutoCAD. https://www.autodesk.com/support/technical/article/caas/sfdcarticles/sfdcarticles/How-to-customize-command-shortcuts-in-AutoCAD.html. Accessed 14 Sept. 2024.

"Language Icon - Free PNG & SVG 3786977 - Noun Project." The Noun Project, https://thenounproject.com/icon/language-3786977/. Accessed 14 Sept. 2024.

"Methodology Icon - Free PNG & SVG 300489 - Noun Project." The Noun Project, https://thenounproject.com/icon/methodology-300489/. Accessed 13 Sept. 2024.

"Opportunities Icon - Free PNG & SVG 7040307 - Noun Project." The Noun Project, https://thenounproject.com/icon/opportunities-7040307/. Accessed 13 Sept. 2024.

"Problem Icon - Free PNG & SVG 7197893 - Noun Project." The Noun Project, https://thenounproject.com/icon/problem-7197893/. Accessed 13 Sept. 2024.

"Recording Icon - Free PNG & SVG 4306258 - Noun Project." The Noun Project, https://thenounproject.com/icon/recording-4306258/. Accessed 13 Sept. 2024.

"Shapes Icon - Free PNG & SVG 6887669 - Noun Project." The Noun Project, https://thenounproject.com/icon/shapes-6887669/. Accessed 13 Sept. 2024.

"Simple Icon - Free PNG & SVG 3094259 - Noun Project." The Noun Project, https://thenounproject.com/icon/simple-3094259/. Accessed 13 Sept. 2024.

"Solution Icon - Free PNG & SVG 7118681 - Noun Project." The Noun Project, https://thenounproject.com/icon/solution-7118681/. Accessed 13 Sept. 2024.


REVIT
VOICE
Vocal Commands for
AutoDesk Revit