T I want to set up a local LLM and train it with CircuitPython libraries and example code so that it will be better at writing code that is specific to CircuitPython. I have experience running local LLMs with ollama CLI tool, but no experience training them on specific data. Can you tell me what I would need to do in order to train a model on this code?

Training a local LLM on CircuitPython-specific code involves several approaches, from simple to more complex. Here's what you need to know:

Fine-tuning vs RAG (Recommended Starting Point)

RAG (Retrieval-Augmented Generation) is often the better starting approach because:

- Much easier to set up and maintain
- Requires no model training
- Can be updated easily with new code examples
- Works well with existing models

You can create a local RAG system that searches through CircuitPython documentation and code examples, then provides that context to your existing ollama models.

If You Want to Fine-tune

For actual model training, you have a few options:

1. LoRA Fine-tuning (Most Practical)

- **Tools**: Use `unsloth`, `axolotl`, or `llama-factory`
- **Hardware**: Needs a GPU with 12GB+ VRAM for decent-sized models
- **Data prep**: Convert CircuitPython code into instruction-following format
- **Time**: Hours to days depending on model size and data amount

2. Full Fine-tuning

- **Hardware**: Requires significant GPU resources (multiple high-end GPUs)
- **Cost**: Very expensive
- **Complexity**: Much more involved

Data Preparation Steps

1. **Collect CircuitPython code**:
   - Official Adafruit CircuitPython libraries

- Example projects from GitHub
- Documentation code snippets
- Community projects

2. **Format for training:**

```
json
{
  "instruction": "Write CircuitPython code to control an LED",
  "input": "Use pin D13 and make it blink every second",
  "output": "import board\nimport digitalio\nimport time\n\nled = digitalio.Digi
}
```

3. **Quality control:**
   - Ensure code actually works
   - Remove duplicates
   - Add variety in coding styles and approaches

Recommended Approach

Start with **RAG using ollama:**

1. Set up a vector database (like Chroma or Qdrant)
2. Index CircuitPython documentation and code examples
3. Create a simple interface that queries the database and provides context to your ollama model
4. This gives you CircuitPython-aware responses without model training

Would you like me to elaborate on any of these approaches, or do you have a preference for which direction to pursue first?

T Yes, can you expand on the RAG approach some?

RAG Approach for CircuitPython Code Assistant

RAG works by retrieving relevant code examples and documentation when you ask a question, then feeding that context to your existing LLM to generate better, more specific

answers.

Architecture Overview

```
Your Question → Vector Search → Relevant Code Examples → LLM + Context →
CircuitPython-specific Answer
```

Step-by-Step Implementation

1. Data Collection and Preparation

Sources to gather:

- CircuitPython core libraries documentation
- Adafruit CircuitPython library examples
- Community projects from GitHub
- Hardware-specific guides (different boards, sensors, etc.)

Data formats to collect:

- `.py` files with good comments
- Markdown documentation with code blocks
- API reference documentation
- Troubleshooting guides

2. Vector Database Setup

Popular options:

- **Chroma** (easiest to start with)
- **Qdrant** (more features)
- **Weaviate** (good for complex queries)

Chunking strategy:

- Split code files into logical functions/classes
- Keep related code together (imports + usage)
- Include docstrings and comments
- Typical chunk size: 500-1000 tokens

3. Embedding Creation

Use a good embedding model like:

- `sentence-transformers/all-MiniLM-L6-v2` (fast, good quality)
- `text-embedding-ada-002` (if using OpenAI)
- `nomic-embed-text` (works well with code)

4. Simple Implementation Example

Here's a basic structure using Python:

python

```python
# Basic RAG setup
import chromadb
from sentence_transformers import SentenceTransformer
import ollama

class CircuitPythonRAG:
    def __init__(self):
        self.client = chromadb.Client()
        self.collection = self.client.create_collection("circuitpython_code")
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')

    def add_code_example(self, code, description, metadata=None):
        # Add code to vector database
        embedding = self.embedder.encode(f"{description}\n{code}")
        self.collection.add(
            embeddings=[embedding.tolist()],
            documents=[code],
            metadatas=[metadata or {}],
            ids=[f"code_{len(self.collection.get()['ids'])}"]
        )

    def query(self, question, model="llama3.1"):
        # Search for relevant code
        query_embedding = self.embedder.encode(question)
        results = self.collection.query(
            query_embeddings=[query_embedding.tolist()],
            n_results=3
        )

        # Build context from retrieved examples
        context = "\n\n".join(results['documents'][0])
```

```
        # Generate response with ollama
        prompt = f"""Based on these CircuitPython examples:

{context}

Question: {question}

Provide a specific CircuitPython solution:"""

        response = ollama.chat(model=model, messages=[{
            'role': 'user',
            'content': prompt
        }])

        return response['message']['content']
```

5. Advanced Features to Consider

Metadata filtering:

- Filter by hardware (ESP32, RP2040, etc.)
- Filter by component type (sensors, displays, motors)
- Filter by complexity level

Query enhancement:

- Parse questions to identify hardware mentioned
- Expand abbreviations (LED → Light Emitting Diode)
- Include related concepts (PWM when asking about motors)

Context ranking:

- Score results by code quality
- Prefer recent/maintained examples
- Weight by community usage

Tools and Frameworks

Ready-made solutions:

- **LangChain** - Full RAG pipeline
- **LlamaIndex** - Document-focused RAG

- **Haystack** - Enterprise-ready pipeline

For ollama integration:

- Most support ollama as a backend LLM
- Can use ollama's embedding models too

Benefits Over Fine-tuning

1. **Immediate updates** - Add new examples without retraining
2. **Transparency** - See exactly what examples influenced the answer
3. **Lower resource requirements** - No GPU needed for training
4. **Modular** - Can swap LLM backends easily
5. **Quality control** - Can curate and verify examples manually

Getting Started Steps

1. **Week 1**: Set up basic Chroma + ollama integration
2. **Week 2**: Collect and chunk CircuitPython examples
3. **Week 3**: Test and refine retrieval quality
4. **Week 4**: Add metadata filtering and query enhancement

Would you like me to dive deeper into any specific part, like the data collection process or show you a more complete code example?

T Yes, can you go deeper on how to collect and chunk the examples?

Data Collection and Chunking for CircuitPython RAG

Data Collection Strategy

Primary Sources (High Quality)

1. Official Adafruit Libraries

bash

```
# Clone the main repository bundle
git clone https://github.com/adafruit/Adafruit_CircuitPython_Bundle.git
```

```
# Individual libraries have better examples
git clone https://github.com/adafruit/Adafruit_CircuitPython_NeoPixel.git
git clone https://github.com/adafruit/Adafruit_CircuitPython_Motor.git
# ... etc for specific components you care about
```

2. CircuitPython Core Documentation

bash

```
# Official docs with code examples
git clone https://github.com/adafruit/circuitpython.git
# Look in: docs/shared-bindings/ and examples/
```

3. Learn Guides Code

- Scrape or manually collect from learn.adafruit.com
- These have complete, tested projects with explanations

Secondary Sources

GitHub Projects:

python

```
import requests
import os

def collect_github_circuitpython():
    # Search for CircuitPython projects
    query = "circuitpython language:python"
    headers = {"Authorization": f"token {os.getenv('GITHUB_TOKEN')}"}

    url = f"https://api.github.com/search/repositories?q={query}&sort=stars"
    response = requests.get(url, headers=headers)

    for repo in response.json()['items'][:50]:  # Top 50 starred repos
        # Clone or download specific files
        print(f"Found: {repo['full_name']}")
```

Community Forums:

- Reddit r/circuitpython code snippets
- Adafruit forums troubleshooting solutions

- Discord community shared code

Automated Collection Script

Here's a comprehensive collector:

python

```python
import os
import git
import glob
from pathlib import Path
import requests
import time

class CircuitPythonCollector:
    def __init__(self, output_dir="circuitpython_data"):
        self.output_dir = Path(output_dir)
        self.output_dir.mkdir(exist_ok=True)

    def collect_official_libraries(self):
        """Collect from official Adafruit libraries"""
        libraries = [
            "Adafruit_CircuitPython_NeoPixel",
            "Adafruit_CircuitPython_Motor",
            "Adafruit_CircuitPython_BME280",
            "Adafruit_CircuitPython_Display_Text",
            "Adafruit_CircuitPython_HID",
            # Add more as needed
        ]

        for lib in libraries:
            repo_url = f"https://github.com/adafruit/{lib}.git"
            local_path = self.output_dir / lib

            if not local_path.exists():
                print(f"Cloning {lib}...")
                git.Repo.clone_from(repo_url, local_path)

            # Extract examples
            self._extract_examples(local_path, lib)

    def _extract_examples(self, repo_path, library_name):
        """Extract code examples from a repository"""
```

```
        examples_dir = repo_path / "examples"
        if examples_dir.exists():
            for py_file in examples_dir.glob("**/*.py"):
                self._process_python_file(py_file, library_name)

        # Also check for examples in README or docs
        for readme in repo_path.glob("**/README*"):
            if readme.suffix.lower() in ['.md', '.rst']:
                self._extract_code_blocks(readme, library_name)

    def _process_python_file(self, file_path, library_name):
        """Process individual Python files"""
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                content = f.read()

            # Skip if too short or not actually CircuitPython
            if len(content) < 100 or not self._is_circuitpython_code(content):
                return

            metadata = {
                'source': str(file_path),
                'library': library_name,
                'type': 'example',
                'file_name': file_path.name
            }

            # Save to our processed examples
            output_file = self.output_dir / "processed" / f"{library_name}_{file_pa
            output_file.parent.mkdir(exist_ok=True)

            with open(output_file, 'w') as f:
                f.write(content)

            # Store metadata separately
            import json
            with open(f"{output_file}.meta", 'w') as f:
                json.dump(metadata, f)

        except Exception as e:
            print(f"Error processing {file_path}: {e}")

    def _is_circuitpython_code(self, content):
```

```
        """Check if code is actually CircuitPython"""
        circuitpython_indicators = [
            'import board',
            'import digitalio',
            'import analogio',
            'import busio',
            'import microcontroller',
            'import neopixel',
            'import adafruit_'
        ]

        content_lower = content.lower()
        return any(indicator in content_lower for indicator in circuitpython_indica

    def _extract_code_blocks(self, readme_path, library_name):
        """Extract code blocks from markdown/rst files"""
        try:
            with open(readme_path, 'r', encoding='utf-8') as f:
                content = f.read()

            # Extract Python code blocks
            import re

            # Markdown code blocks
            md_pattern = r'```python\n(.*?)\n```'
            rst_pattern = r'\.\. code-block:: python\n\n((?:    .*\n)*)'

            for pattern in [md_pattern, rst_pattern]:
                matches = re.findall(pattern, content, re.DOTALL)
                for i, code in enumerate(matches):
                    if self._is_circuitpython_code(code):
                        output_file = self.output_dir / "processed" / f"{library_na
                        with open(output_file, 'w') as f:
                            f.write(code.strip())

        except Exception as e:
            print(f"Error extracting from {readme_path}: {e}")
```

Smart Chunking Strategies

The key is chunking code in a way that preserves meaning and context:

1. Function-Level Chunking

python

```python
import ast
import re

class CircuitPythonChunker:
    def __init__(self):
        self.chunk_size_limit = 1000  # tokens

    def chunk_python_file(self, file_content, metadata):
        """Chunk Python file intelligently"""
        chunks = []

        try:
            tree = ast.parse(file_content)

            # Extract imports (always include these)
            imports = self._extract_imports(tree)

            # Process each top-level element
            for node in tree.body:
                if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
                    chunk = self._create_function_chunk(node, file_content, imports
                    if chunk:
                        chunks.append({
                            'content': chunk,
                            'metadata': {
                                **metadata,
                                'chunk_type': 'function',
                                'function_name': node.name
                            }
                        })

                elif isinstance(node, ast.Assign):
                    # Handle global configurations or setup code
                    chunk = self._create_assignment_chunk(node, file_content, impor
                    if chunk:
                        chunks.append({
                            'content': chunk,
                            'metadata': {
                                **metadata,
```

```
                            'chunk_type': 'setup'
                        }
                    })

            # If file is short enough, also include as whole file
            if len(file_content) < self.chunk_size_limit:
                chunks.append({
                    'content': file_content,
                    'metadata': {
                        **metadata,
                        'chunk_type': 'complete_example'
                    }
                })

        except SyntaxError:
            # If can't parse, fall back to simple chunking
            chunks = self._simple_chunk(file_content, metadata)

        return chunks

    def _extract_imports(self, tree):
        """Extract all import statements"""
        imports = []
        for node in tree.body:
            if isinstance(node, (ast.Import, ast.ImportFrom)):
                imports.append(ast.unparse(node))
        return imports

    def _create_function_chunk(self, func_node, file_content, imports):
        """Create a chunk for a function with context"""
        func_code = ast.unparse(func_node)

        # Add docstring context if available
        description = ""
        if (ast.get_docstring(func_node)):
            description = f"# {ast.get_docstring(func_node)}\n"

        # Combine imports + description + function
        chunk = "\n".join(imports) + "\n\n" + description + func_code

        # Add usage example if found in comments
        usage_example = self._find_usage_example(func_node.name, file_content)
        if usage_example:
```

```
            chunk += f"\n\n# Usage example:\n{usage_example}"

        return chunk

    def _find_usage_example(self, function_name, file_content):
        """Look for usage examples of a function in the file"""
        lines = file_content.split('\n')
        for i, line in enumerate(lines):
            if function_name in line and 'def ' not in line:
                # Found usage, extract a few lines of context
                start = max(0, i-2)
                end = min(len(lines), i+3)
                return '\n'.join(lines[start:end])
        return None
```

2. Semantic Chunking by Hardware Component

python

```
def chunk_by_component(self, file_content, metadata):
    """Group code by hardware components used"""

    component_patterns = {
        'led': [r'led\s*=', r'digitalio\.DigitalInOut', r'board\.LED'],
        'neopixel': [r'neopixel', r'NeoPixel'],
        'sensor': [r'bme280', r'temperature', r'humidity', r'pressure'],
        'motor': [r'motor', r'servo', r'stepper'],
        'display': [r'display', r'oled', r'lcd', r'tft'],
        'audio': [r'audio', r'wav', r'mp3', r'tone'],
    }

    chunks = []
    lines = file_content.split('\n')
    current_chunk = []
    current_components = set()

    for line in lines:
        line_components = set()

        # Check what components this line relates to
        for component, patterns in component_patterns.items():
            if any(re.search(pattern, line, re.IGNORECASE) for pattern in patterns):
```

```
                line_components.add(component)

        if line_components:
            if current_components and not line_components.intersection(current_comp
                # Different component, create chunk
                if current_chunk:
                    chunks.append({
                        'content': '\n'.join(current_chunk),
                        'metadata': {
                            **metadata,
                            'components': list(current_components),
                            'chunk_type': 'component_focused'
                        }
                    })
                current_chunk = [line]
                current_components = line_components
            else:
                current_chunk.append(line)
                current_components.update(line_components)
        else:
            current_chunk.append(line)

    # Don't forget the last chunk
    if current_chunk:
        chunks.append({
            'content': '\n'.join(current_chunk),
            'metadata': {
                **metadata,
                'components': list(current_components) if current_components else [
                'chunk_type': 'component_focused'
            }
        })

    return chunks
```

3. Complete Processing Pipeline

python

```
def process_all_collected_data():
    """Complete pipeline to process collected data"""
    collector = CircuitPythonCollector()
```

```
    chunker = CircuitPythonChunker()

    # Collect data
    collector.collect_official_libraries()

    # Process each collected file
    processed_chunks = []

    for py_file in Path("circuitpython_data/processed").glob("*.py"):
        with open(py_file, 'r') as f:
            content = f.read()

        # Load metadata
        with open(f"{py_file}.meta", 'r') as f:
            metadata = json.load(f)

        # Chunk the content
        chunks = chunker.chunk_python_file(content, metadata)
        processed_chunks.extend(chunks)

    # Save processed chunks for RAG system
    with open("rag_ready_chunks.json", 'w') as f:
        json.dump(processed_chunks, f, indent=2)

    print(f"Processed {len(processed_chunks)} chunks ready for RAG")
```

Quality Control

Add these filters to ensure good chunks:

python

```
def filter_quality_chunks(chunks):
    """Remove low-quality chunks"""
    filtered = []

    for chunk in chunks:
        content = chunk['content']

        # Skip if too short or too long
        if len(content) < 50 or len(content) > 2000:
            continue
```

```
        # Skip if mostly comments
        lines = content.split('\n')
        comment_ratio = sum(1 for line in lines if line.strip().startswith('#')) /
        if comment_ratio > 0.7:
            continue

        # Skip if no actual CircuitPython imports
        if not any(imp in content for imp in ['import board', 'import digitalio', '
            continue

        # Must have some actual code
        code_lines = [line for line in lines if line.strip() and not line.strip().s
        if len(code_lines) < 3:
            continue

        filtered.append(chunk)

    return filtered
```

This approach gives you:

- **High-quality, tested code examples**
- **Proper context preservation**
- **Component-specific retrieval**
- **Scalable collection process**

Would you like me to show you how to integrate these chunks into the actual RAG system next?

T Right now I am focused on function based chunking not semantic hardware based. I found that the function based chunking code you provided raises this error:

```
AttributeError: 'CircuitPythonChunker' object has no attribute
'_create_assignment_chunk'
```

Can you provide a corrected version?

You're right, I included a call to `_create_assignment_chunk` but didn't implement that method. Here's the corrected version with all missing methods implemented:

python

```python
import ast
import re

class CircuitPythonChunker:
    def __init__(self):
        self.chunk_size_limit = 1000  # tokens

    def chunk_python_file(self, file_content, metadata):
        """Chunk Python file intelligently"""
        chunks = []

        try:
            tree = ast.parse(file_content)

            # Extract imports (always include these)
            imports = self._extract_imports(tree)

            # Process each top-level element
            for node in tree.body:
                if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
                    chunk = self._create_function_chunk(node, file_content, imports
                    if chunk:
                        chunks.append({
                            'content': chunk,
                            'metadata': {
                                **metadata,
                                'chunk_type': 'function',
                                'function_name': node.name
                            }
                        })

                elif isinstance(node, ast.Assign):
                    # Handle global configurations or setup code
                    chunk = self._create_assignment_chunk(node, file_content, impor
                    if chunk:
                        chunks.append({
                            'content': chunk,
                            'metadata': {
                                **metadata,
                                'chunk_type': 'setup'
                            }
```

```
                    })

            # If file is short enough, also include as whole file
            if len(file_content) < self.chunk_size_limit:
                chunks.append({
                    'content': file_content,
                    'metadata': {
                        **metadata,
                        'chunk_type': 'complete_example'
                    }
                })

        except SyntaxError:
            # If can't parse, fall back to simple chunking
            chunks = self._simple_chunk(file_content, metadata)

        return chunks

    def _extract_imports(self, tree):
        """Extract all import statements"""
        imports = []
        for node in tree.body:
            if isinstance(node, (ast.Import, ast.ImportFrom)):
                imports.append(ast.unparse(node))
        return imports

    def _create_function_chunk(self, func_node, file_content, imports):
        """Create a chunk for a function with context"""
        func_code = ast.unparse(func_node)

        # Add docstring context if available
        description = ""
        if (ast.get_docstring(func_node)):
            description = f"# {ast.get_docstring(func_node)}\n"

        # Combine imports + description + function
        chunk = "\n".join(imports) + "\n\n" + description + func_code

        # Add usage example if found in comments
        usage_example = self._find_usage_example(func_node.name, file_content)
        if usage_example:
            chunk += f"\n\n# Usage example:\n{usage_example}"
```

```
            return chunk

        def _create_assignment_chunk(self, assign_node, file_content, imports):
            """Create a chunk for important assignments (pin configs, constants, etc.)"
            assignment_code = ast.unparse(assign_node)

            # Only create chunks for assignments that look like CircuitPython setup
            if self._is_circuitpython_assignment(assignment_code):
                # Find related assignments in the same area
                context_lines = self._get_assignment_context(assign_node, file_content)

                if context_lines:
                    chunk = "\n".join(imports) + "\n\n" + "\n".join(context_lines)
                    return chunk

            return None

        def _is_circuitpython_assignment(self, assignment_code):
            """Check if assignment is CircuitPython-related"""
            circuitpython_keywords = [
                'board.', 'digitalio.', 'analogio.', 'busio.',
                'neopixel', 'adafruit_', 'microcontroller.',
                '.DigitalInOut', '.AnalogIn', '.PWMOut'
            ]

            return any(keyword in assignment_code for keyword in circuitpython_keywords

        def _get_assignment_context(self, assign_node, file_content):
            """Get related assignments around this one"""
            lines = file_content.split('\n')

            # Find the line number of this assignment
            target_line = None
            assignment_str = ast.unparse(assign_node)

            for i, line in enumerate(lines):
                if assignment_str.strip() in line.strip():
                    target_line = i
                    break

            if target_line is None:
                return [assignment_str]
```

```
        # Collect related lines (assignments, imports, comments)
        context_lines = []

        # Look backwards for related setup
        for i in range(max(0, target_line - 5), target_line):
            line = lines[i].strip()
            if line and (line.startswith('#') or '=' in line or line.startswith('im
                context_lines.append(lines[i])

        # Add the target line
        context_lines.append(lines[target_line])

        # Look forwards for related setup
        for i in range(target_line + 1, min(len(lines), target_line + 5)):
            line = lines[i].strip()
            if line and (line.startswith('#') or '=' in line):
                context_lines.append(lines[i])
            elif line == "":
                continue
            else:
                break  # Stop at first non-setup line

        return context_lines

    def _find_usage_example(self, function_name, file_content):
        """Look for usage examples of a function in the file"""
        lines = file_content.split('\n')
        for i, line in enumerate(lines):
            if function_name in line and 'def ' not in line:
                # Found usage, extract a few lines of context
                start = max(0, i-2)
                end = min(len(lines), i+3)
                return '\n'.join(lines[start:end])
        return None

    def _simple_chunk(self, file_content, metadata):
        """Fallback chunking when AST parsing fails"""
        # Split by blank lines or logical breaks
        sections = re.split(r'\n\s*\n', file_content)
        chunks = []

        current_chunk = ""
        for section in sections:
```

```
            if len(current_chunk) + len(section) < self.chunk_size_limit:
                current_chunk += section + "\n\n"
            else:
                if current_chunk:
                    chunks.append({
                        'content': current_chunk.strip(),
                        'metadata': {
                            **metadata,
                            'chunk_type': 'simple'
                        }
                    })
                current_chunk = section + "\n\n"

        # Don't forget the last chunk
        if current_chunk:
            chunks.append({
                'content': current_chunk.strip(),
                'metadata': {
                    **metadata,
                    'chunk_type': 'simple'
                }
            })

        return chunks
```

Now you can test it like this:

python

```
# Test the chunker
chunker = CircuitPythonChunker()

sample_code = """
import board
import digitalio
import time

# LED setup
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

def blink_led(times=5):
```

```
    '''Blink the LED a specified number of times'''
    for _ in range(times):
        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)

def setup_pin(pin_name):
    '''Setup a digital output pin'''
    pin = digitalio.DigitalInOut(getattr(board, pin_name))
    pin.direction = digitalio.Direction.OUTPUT
    return pin

# Main execution
if __name__ == "__main__":
    blink_led(3)
"""

metadata = {'source': 'test.py', 'library': 'core'}
chunks = chunker.chunk_python_file(sample_code, metadata)

for i, chunk in enumerate(chunks):
    print(f"\n--- Chunk {i+1} ({chunk['metadata']['chunk_type']}) ---")
    print(chunk['content'])
```

This corrected version includes:

- `_create_assignment_chunk()` - handles pin setups and configuration assignments
- `_is_circuitpython_assignment()` - filters for relevant assignments
- `_get_assignment_context()` - gathers related setup lines
- `_simple_chunk()` - fallback chunking method

The chunker will now create separate chunks for functions, important assignments/setup code, and fall back to simple chunking if the Python parsing fails.

T I am moving on to the RAG step now. I saw in the basic RAG setup code that you provided it has the function `add_code_example` but the function isn't called by the rest of the code. Can you tell me what that function is for and how to use it?

You're absolutely right! The `add_code_example` function is how you populate the vector database with your chunked CircuitPython code, but I didn't show how to connect it to the chunking process. Let me explain and show you how to use it.

What `add_code_example` does

This function takes your processed code chunks and stores them in the vector database so they can be retrieved later when you ask questions. It:

1. Creates embeddings (vector representations) of the code + description
2. Stores the actual code content
3. Saves metadata for filtering/context
4. Assigns unique IDs for retrieval

How to connect it to your chunked data

Here's how to populate your RAG system with the chunked data:

python

```python
import json
from pathlib import Path

class CircuitPythonRAG:
    def __init__(self):
        self.client = chromadb.Client()
        self.collection = self.client.create_collection("circuitpython_code")
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')

    def add_code_example(self, code, description, metadata=None):
        """Add a single code example to the vector database"""
        # Create embedding from description + code
        embedding_text = f"{description}\n{code}"
        embedding = self.embedder.encode(embedding_text)

        self.collection.add(
            embeddings=[embedding.tolist()],
            documents=[code],
            metadatas=[metadata or {}],
            ids=[f"code_{len(self.collection.get()['ids'])}"]
        )

    def populate_from_chunks(self, chunks_file="rag_ready_chunks.json"):
```

```
        """Load all your processed chunks into the RAG system"""
        print("Loading chunks into vector database...")

        with open(chunks_file, 'r') as f:
            chunks = json.load(f)

        for i, chunk_data in enumerate(chunks):
            code = chunk_data['content']
            metadata = chunk_data['metadata']

            # Create a good description for embedding
            description = self._create_description(code, metadata)

            # Add to vector database
            self.add_code_example(
                code=code,
                description=description,
                metadata={
                    **metadata,
                    'chunk_id': i
                }
            )

            if (i + 1) % 100 == 0:
                print(f"Processed {i + 1} chunks...")

        print(f"Successfully loaded {len(chunks)} code examples!")

    def _create_description(self, code, metadata):
        """Create a good description for embedding"""
        descriptions = []

        # Add chunk type info
        chunk_type = metadata.get('chunk_type', 'code')
        descriptions.append(f"CircuitPython {chunk_type}")

        # Add library info
        if 'library' in metadata:
            descriptions.append(f"using {metadata['library']} library")

        # Add function name if available
        if 'function_name' in metadata:
            descriptions.append(f"function: {metadata['function_name']}")
```

```
        # Add components if available
        if 'components' in metadata:
            components = ', '.join(metadata['components'])
            descriptions.append(f"components: {components}")

        # Extract key concepts from the code itself
        code_concepts = self._extract_code_concepts(code)
        if code_concepts:
            descriptions.append(f"concepts: {', '.join(code_concepts)}")

        return ' '.join(descriptions)

    def _extract_code_concepts(self, code):
        """Extract key concepts from code for better searchability"""
        concepts = []

        # Look for common CircuitPython patterns
        patterns = {
            'LED control': ['led', 'digitalio.DigitalInOut', 'board.LED'],
            'PWM': ['pwmio.PWMOut', 'duty_cycle'],
            'I2C communication': ['busio.I2C', 'i2c'],
            'SPI communication': ['busio.SPI', 'spi'],
            'Analog input': ['analogio.AnalogIn'],
            'NeoPixel': ['neopixel', 'NeoPixel'],
            'Servo control': ['servo', 'adafruit_motor.servo'],
            'Display': ['displayio', 'adafruit_display'],
            'Sensor reading': ['temperature', 'humidity', 'pressure'],
            'Time delays': ['time.sleep'],
            'Loop control': ['while True', 'for '],
        }

        code_lower = code.lower()
        for concept, keywords in patterns.items():
            if any(keyword.lower() in code_lower for keyword in keywords):
                concepts.append(concept)

        return concepts[:5]  # Limit to avoid too long descriptions

    def query(self, question, model="llama3.1", n_results=3):
        """Query the RAG system"""
        # Search for relevant code
        query_embedding = self.embedder.encode(question)
```

```
        results = self.collection.query(
            query_embeddings=[query_embedding.tolist()],
            n_results=n_results
        )

        if not results['documents'][0]:
            return "No relevant CircuitPython examples found."

        # Build context from retrieved examples
        context_parts = []
        for i, (doc, metadata) in enumerate(zip(results['documents'][0], results['m
            context_parts.append(f"Example {i+1} ({metadata.get('chunk_type', 'code

        context = "\n\n" + "="*50 + "\n\n".join(context_parts)

        # Generate response with ollama
        prompt = f"""Based on these CircuitPython examples:

{context}

Question: {question}

Provide a specific CircuitPython solution with explanations:"""

        response = ollama.chat(model=model, messages=[{
            'role': 'user',
            'content': prompt
        }])

        return response['message']['content']
```

Complete workflow example

Here's how to use everything together:

python

```
def setup_circuitpython_rag():
    """Complete setup process"""

    # Step 1: Collect and chunk data (if not already done)
    if not Path("rag_ready_chunks.json").exists():
        print("Collecting and chunking CircuitPython code...")
```

```
        collector = CircuitPythonCollector()
        chunker = CircuitPythonChunker()

        # This creates rag_ready_chunks.json
        process_all_collected_data()

    # Step 2: Create RAG system and populate it
    rag = CircuitPythonRAG()
    rag.populate_from_chunks("rag_ready_chunks.json")

    return rag

# Usage
if __name__ == "__main__":
    # Setup (run once)
    rag = setup_circuitpython_rag()

    # Test queries
    questions = [
        "How do I blink an LED?",
        "Show me how to read temperature from a sensor",
        "How to control a servo motor?",
        "Setup I2C communication with a display"
    ]

    for question in questions:
        print(f"\n{'='*60}")
        print(f"Question: {question}")
        print("="*60)
        answer = rag.query(question)
        print(answer)
```

Alternative: Manual population

If you want to add examples manually instead of from chunks:

python

```
rag = CircuitPythonRAG()

# Add individual examples
led_code = """
import board
```

```
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
"""

rag.add_code_example(
    code=led_code,
    description="Basic LED blinking example using digitalio",
    metadata={
        'library': 'core',
        'components': ['LED'],
        'difficulty': 'beginner'
    }
)
```

The key insight is that `add_code_example` is your **data ingestion** function - it's how you get your code examples into the system so they can be searched and retrieved later when you ask questions.

T I ended up with about 4300 chunks. I've noticed that processing the chunks into a vector database takes several minutes to complete. Is it possible to export or serialize the vector database and load it from a file more efficiently than processing it each time?

Yes! Processing 4300 chunks every time is definitely inefficient. ChromaDB (and most vector databases) support persistence so you can save and load the database. Here are several approaches:

1. ChromaDB Persistent Storage (Recommended)

The easiest solution is to use ChromaDB's built-in persistence:

python

```
import chromadb
from sentence_transformers import SentenceTransformer
import ollama
from pathlib import Path

class CircuitPythonRAG:
    def __init__(self, persist_directory="./chroma_db"):
        # Use persistent client instead of in-memory
        self.persist_directory = persist_directory
        self.client = chromadb.PersistentClient(path=persist_directory)

        # Try to get existing collection, create if doesn't exist
        try:
            self.collection = self.client.get_collection("circuitpython_code")
            print(f"Loaded existing collection with {self.collection.count()} examp
        except:
            self.collection = self.client.create_collection("circuitpython_code")
            print("Created new collection")

        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')

    def is_populated(self):
        """Check if database already has data"""
        return self.collection.count() > 0

    def populate_from_chunks(self, chunks_file="rag_ready_chunks.json", force_rebui
        """Load chunks only if database is empty or force_rebuild is True"""

        if self.is_populated() and not force_rebuild:
            print(f"Database already contains {self.collection.count()} examples. S
            print("Use force_rebuild=True to rebuild the database.")
            return

        if force_rebuild:
            # Clear existing data
            self.client.delete_collection("circuitpython_code")
            self.collection = self.client.create_collection("circuitpython_code")
            print("Rebuilding database from scratch...")

        # Rest of your population code stays the same
        print("Loading chunks into vector database...")
```

```
        with open(chunks_file, 'r') as f:
            chunks = json.load(f)

        # Process in batches for better performance
        batch_size = 100
        for i in range(0, len(chunks), batch_size):
            batch = chunks[i:i + batch_size]
            self._add_batch(batch, i)
            print(f"Processed {min(i + batch_size, len(chunks))} / {len(chunks)} ch

        print(f"Successfully loaded {len(chunks)} code examples!")

    def _add_batch(self, chunk_batch, start_index):
        """Add a batch of chunks efficiently"""
        embeddings = []
        documents = []
        metadatas = []
        ids = []

        for j, chunk_data in enumerate(chunk_batch):
            code = chunk_data['content']
            metadata = chunk_data['metadata']

            description = self._create_description(code, metadata)
            embedding_text = f"{description}\n{code}"
            embedding = self.embedder.encode(embedding_text)

            embeddings.append(embedding.tolist())
            documents.append(code)
            metadatas.append({**metadata, 'chunk_id': start_index + j})
            ids.append(f"code_{start_index + j}")

        # Add entire batch at once
        self.collection.add(
            embeddings=embeddings,
            documents=documents,
            metadatas=metadatas,
            ids=ids
        )

    # Rest of your methods stay the same...
    def query(self, question, model="llama3.1", n_results=3):
```

```
        # Same as before
        pass
```

Usage with persistence:

python

```python
def setup_circuitpython_rag():
    """Setup with persistence - much faster on subsequent runs"""

    # This will load existing database if it exists
    rag = CircuitPythonRAG(persist_directory="./my_circuitpython_db")

    # Only populate if database is empty
    if not rag.is_populated():
        print("First run - populating database (this will take a few minutes)...")
        rag.populate_from_chunks("rag_ready_chunks.json")
    else:
        print("Using existing database - ready to query!")

    return rag

# Usage
rag = setup_circuitpython_rag()  # Fast on subsequent runs!
answer = rag.query("How do I blink an LED?")
```

2. Export/Import Embeddings (Alternative approach)

If you want more control, you can save/load just the embeddings:

python

```python
import pickle
import numpy as np

class CircuitPythonRAG:
    def __init__(self):
        self.client = chromadb.Client()
        self.collection = self.client.create_collection("circuitpython_code")
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')
        self.cache_file = "circuitpython_embeddings.pkl"
```

```
    def save_embeddings_cache(self, chunks):
        """Save processed embeddings to avoid recomputation"""
        print("Computing and caching embeddings...")

        cached_data = []
        for i, chunk_data in enumerate(chunks):
            code = chunk_data['content']
            metadata = chunk_data['metadata']
            description = self._create_description(code, metadata)

            embedding_text = f"{description}\n{code}"
            embedding = self.embedder.encode(embedding_text)

            cached_data.append({
                'code': code,
                'metadata': metadata,
                'embedding': embedding,
                'id': f"code_{i}"
            })

            if (i + 1) % 100 == 0:
                print(f"Cached {i + 1} embeddings...")

        with open(self.cache_file, 'wb') as f:
            pickle.dump(cached_data, f)

        print(f"Saved embeddings cache with {len(cached_data)} items")
        return cached_data

    def load_embeddings_cache(self):
        """Load pre-computed embeddings"""
        try:
            with open(self.cache_file, 'rb') as f:
                cached_data = pickle.load(f)
            print(f"Loaded {len(cached_data)} cached embeddings")
            return cached_data
        except FileNotFoundError:
            return None

    def populate_from_cache_or_chunks(self, chunks_file="rag_ready_chunks.json"):
        """Use cache if available, otherwise compute from chunks"""

        cached_data = self.load_embeddings_cache()
```

```
    if cached_data is None:
        # Load chunks and create cache
        with open(chunks_file, 'r') as f:
            chunks = json.load(f)
        cached_data = self.save_embeddings_cache(chunks)

    # Populate database from cached embeddings (very fast)
    print("Populating database from cache...")

    batch_size = 500  # Larger batches since we're not computing embeddings
    for i in range(0, len(cached_data), batch_size):
        batch = cached_data[i:i + batch_size]

        embeddings = [item['embedding'].tolist() for item in batch]
        documents = [item['code'] for item in batch]
        metadatas = [item['metadata'] for item in batch]
        ids = [item['id'] for item in batch]

        self.collection.add(
            embeddings=embeddings,
            documents=documents,
            metadatas=metadatas,
            ids=ids
        )

    print("Database populated from cache!")
```

3. Performance Tips

For even better performance with large datasets:

python

```
def setup_fast_rag():
    """Optimized setup with performance improvements"""

    # Use a faster, smaller embedding model for development
    class FastCircuitPythonRAG(CircuitPythonRAG):
        def __init__(self, persist_directory="./chroma_db"):
            super().__init__(persist_directory)
            # Faster but slightly less accurate model
            self.embedder = SentenceTransformer('all-MiniLM-L6-v2')
```

```
        # Or even faster: 'paraphrase-MiniLM-L3-v2'

rag = FastCircuitPythonRAG()

if not rag.is_populated():
    # Use smaller subset for testing
    print("Populating with subset for testing...")
    with open("rag_ready_chunks.json", 'r') as f:
        chunks = json.load(f)

    # Use only first 500 chunks for quick testing
    test_chunks = chunks[:500]

    with open("test_chunks.json", 'w') as f:
        json.dump(test_chunks, f)

    rag.populate_from_chunks("test_chunks.json")

return rag
```

Recommended workflow:

1. **First run**: Use persistent ChromaDB - takes a few minutes to populate
2. **Subsequent runs**: Loads instantly from disk
3. **Development**: Use subset of chunks for faster iteration
4. **Production**: Use full dataset with persistence

The persistent ChromaDB approach is the simplest and most reliable. Your database will be saved in the `./chroma_db` directory and load instantly on subsequent runs!

T In the provided collector code it has this:

```
# Skip if too short or not actually CircuitPython
if len(content) < 100 or not self._is_circuitpython_code(content):
    return
```

Why are short files skipped? Do shorter code files not work well with RAG?

Good question! The length filter isn't strictly necessary for RAG to work - it's more of a quality heuristic. Let me explain the reasoning and when you might want to adjust or remove it:

Why short files are often skipped:

1. They're often not useful examples:

```python
# Example: A 50-character file might just be:
import board
import time
```

This doesn't teach you anything about actually using CircuitPython.

2. Metadata or config files: Short files in repositories are often:

- `__init__.py` (empty or just imports)
- License headers
- Placeholder files
- Build configuration snippets

3. Incomplete code snippets: Very short files might be fragments that don't make sense on their own.

When short files ARE valuable:

1. Minimal working examples:

```python
# This is only ~80 characters but very useful!
import board
import neopixel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 1)
pixels[0] = (255, 0, 0)
```

2. Simple demonstrations: Short, focused examples can actually be better for learning specific concepts.

3. Quick reference code: Concise examples of API usage are often more helpful than long, complex files.

Recommended approach:

Lower the threshold significantly or remove it entirely:

python

```python
def _process_python_file(self, file_path, library_name):
    """Process individual Python files"""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

        # Use a much lower threshold - even short examples can be valuable
        # Focus on CircuitPython relevance rather than length
        if not self._is_circuitpython_code(content):
            return

        # Optional: filter out ONLY truly useless files
        if len(content.strip()) < 20:  # Just a couple lines
            return

        # Check for common non-example patterns
        if self._is_metadata_file(file_path, content):
            return

        # ... rest of your code
```

Add a metadata file filter instead:

python

```python
def _is_metadata_file(self, file_path, content):
    """Check if this is a non-example file"""
    filename = file_path.name.lower()

    # Skip obvious non-examples
    if filename in ['__init__.py', 'setup.py', 'conf.py', 'version.py']:
        return True

    # Skip if it's just imports
    lines = [l.strip() for l in content.split('\n') if l.strip()]
    import_lines = [l for l in lines if l.startswith('import') or l.startswith('fro
```

```
    if len(lines) > 0 and len(import_lines) == len(lines):
        return True  # File is only imports

    return False
```

Better quality filter:

Instead of length, focus on **usefulness**:

python

```
def _is_useful_example(self, content):
    """Check if code is a useful example regardless of length"""

    # Must be CircuitPython
    if not self._is_circuitpython_code(content):
        return False

    # Should have some actual code (not just imports and comments)
    lines = content.split('\n')
    code_lines = [
        line for line in lines
        if line.strip()
        and not line.strip().startswith('#')
        and not line.strip().startswith('import')
        and not line.strip().startswith('from')
    ]

    # At least 2 lines of actual code
    if len(code_lines) < 2:
        return False

    # Bonus: Prefer files with common patterns
    useful_patterns = [
        'while True:',  # Main loop
        'def ',  # Function definition
        '.value =',  # Pin manipulation
        '= DigitalInOut(',  # Pin setup
        '= NeoPixel(',  # NeoPixel setup
    ]

    has_useful_pattern = any(pattern in content for pattern in useful_patterns)
```

```
    # If very short, must have a useful pattern
    if len(content) < 100 and not has_useful_pattern:
        return False

    return True
```

My recommendation:

Remove the length filter and replace it with a usefulness filter:

python

```
# Old approach:
if len(content) < 100 or not self._is_circuitpython_code(content):
    return

# Better approach:
if not self._is_useful_example(content):
    return
```

This way you'll capture valuable short examples while still filtering out genuinely useless files. Short, focused examples are often the **best** kind for RAG because they're:

- Easier to understand
- More focused on specific concepts
- Less likely to have irrelevant context
- Perfect for learning specific APIs or patterns

For your 4300 chunks, you might actually want to include more short examples rather than fewer - they could improve the quality of your RAG responses!

T Thank you for the explanation. I think I will remove the filter entirely. I am actually curating the example code from first hand sources known to be complete working examples.

That's an excellent approach! If you're curating from known-good sources with complete working examples, you definitely don't need that filter. In fact, removing it will likely improve your RAG system since you're ensuring quality at the source rather than trying to filter algorithmically.

Here's the simplified version without the length filter:

python

```python
def _process_python_file(self, file_path, library_name):
    """Process individual Python files"""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

        # Since you're curating from known-good sources, just check if it's Circuit
        if not self._is_circuitpython_code(content):
            return

        metadata = {
            'source': str(file_path),
            'library': library_name,
            'type': 'example',
            'file_name': file_path.name
        }

        # Save to our processed examples
        output_file = self.output_dir / "processed" / f"{library_name}_{file_path.r
        output_file.parent.mkdir(exist_ok=True)

        with open(output_file, 'w') as f:
            f.write(content)

        # Store metadata separately
        import json
        with open(f"{output_file}.meta", 'w') as f:
            json.dump(metadata, f)

    except Exception as e:
        print(f"Error processing {file_path}: {e}")
```

Since you're working with curated, first-hand complete examples, your RAG system should work really well. Those short, focused examples will be particularly valuable for answering specific "how do I..." questions. The quality of your source material is much more important than any algorithmic filtering!

Good luck with your CircuitPython RAG system - sounds like you're building something really useful for the CircuitPython community!

"""Basic digital pin support The `digitalio` module contains classes to provide access to basic digital IO. All classes change hardware state and should be PASTED hen they are no longer needed if

T I also have `.pyi` stubs files like this one. Would using these make the RAG system more effective?

Great question! Yes, `.pyi` stub files can be valuable for your RAG system, but they serve a different purpose than regular example code. Let me explain the tradeoffs:

Benefits of including `.pyi` stubs:

1. Complete API reference:

- Shows all available methods, parameters, and return types
- Documents the full interface of a class/module
- Includes type hints that regular examples might not have

2. Authoritative documentation:

- Often has comprehensive docstrings
- Shows the "official" way to use APIs
- Includes parameter descriptions and warnings (like the RP2350 warning in your example)

3. Helps answer "what parameters does this take?" questions:

```
User: "What parameters does DigitalInOut.__init__ take?"
RAG: Can directly pull from the stub showing it takes a microcontroller.Pin
```

Limitations compared to working examples:

1. No actual usage patterns:

- Stubs show *what* exists, not *how* to use it

- No real-world context or complete programs

2. May be too technical:

- Type annotations might confuse beginners
- Lacks the "show me working code" aspect

3. Different retrieval patterns:

- Users asking "how do I blink an LED?" won't match well with stub signatures

Recommended approach: Include both, with different metadata

python

```python
def _process_python_file(self, file_path, library_name):
    """Process both .py and .pyi files with appropriate metadata"""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

        # Determine file type
        is_stub = file_path.suffix == '.pyi'

        if not self._is_circuitpython_code(content):
            return

        metadata = {
            'source': str(file_path),
            'library': library_name,
            'type': 'api_reference' if is_stub else 'example',
            'file_name': file_path.name,
            'is_stub': is_stub
        }

        # Save with appropriate naming
        suffix = 'stub' if is_stub else 'example'
        output_file = self.output_dir / "processed" / f"{library_name}_{suffix}_{fi
        output_file.parent.mkdir(exist_ok=True)

        with open(output_file, 'w') as f:
            f.write(content)

        import json
```

```
        with open(f"{output_file}.meta", 'w') as f:
            json.dump(metadata, f)

    except Exception as e:
        print(f"Error processing {file_path}: {e}")
```

Enhanced chunking for stubs:

Stubs should be chunked differently - by class/function rather than trying to keep everything together:

python

```python
def chunk_stub_file(self, file_content, metadata):
    """Special chunking for .pyi stub files"""
    chunks = []

    try:
        tree = ast.parse(file_content)

        # Get module-level docstring
        module_doc = ast.get_docstring(tree)

        # Extract imports
        imports = self._extract_imports(tree)

        # Process each class separately
        for node in tree.body:
            if isinstance(node, ast.ClassDef):
                chunk = self._create_class_chunk(node, imports, module_doc)
                if chunk:
                    chunks.append({
                        'content': chunk,
                        'metadata': {
                            **metadata,
                            'chunk_type': 'api_class',
                            'class_name': node.name
                        }
                    })

                # Also create chunks for each method
                for item in node.body:
```

```
                    if isinstance(item, ast.FunctionDef):
                        method_chunk = self._create_method_chunk(item, node.name, i
                        if method_chunk:
                            chunks.append({
                                'content': method_chunk,
                                'metadata': {
                                    **metadata,
                                    'chunk_type': 'api_method',
                                    'class_name': node.name,
                                    'method_name': item.name
                                }
                            })

        except SyntaxError:
            # Fallback for unparseable stubs
            chunks = self._simple_chunk(file_content, metadata)

        return chunks

    def _create_class_chunk(self, class_node, imports, module_doc):
        """Create a chunk for a class definition with full context"""
        class_doc = ast.get_docstring(class_node) or ""
        class_code = ast.unparse(class_node)

        # Build comprehensive chunk
        parts = []

        if module_doc:
            parts.append(f"# Module: {module_doc[:200]}...")  # Truncated module contex

        parts.extend(imports)
        parts.append("")
        parts.append(class_code)

        return "\n".join(parts)

    def _create_method_chunk(self, method_node, class_name, imports):
        """Create focused chunk for a single method"""
        method_doc = ast.get_docstring(method_node) or ""
        method_code = ast.unparse(method_node)

        parts = imports.copy()
        parts.append("")
```

```
    parts.append(f"# From class {class_name}:")
    parts.append(method_code)

    return "\n".join(parts)
```

Update your RAG query to use both:

python

```python
def query(self, question, model="llama3.1", n_results=5):
    """Query with both examples and API reference"""
    query_embedding = self.embedder.encode(question)
    results = self.collection.query(
        query_embeddings=[query_embedding.tolist()],
        n_results=n_results
    )

    if not results['documents'][0]:
        return "No relevant CircuitPython information found."

    # Separate examples from API references
    examples = []
    api_refs = []

    for doc, metadata in zip(results['documents'][0], results['metadatas'][0]):
        if metadata.get('type') == 'api_reference':
            api_refs.append((doc, metadata))
        else:
            examples.append((doc, metadata))

    # Build context - prioritize examples but include API reference
    context_parts = []

    if examples:
        context_parts.append("=== Working Examples ===")
        for i, (doc, meta) in enumerate(examples[:3]):  # Top 3 examples
            context_parts.append(f"\nExample {i+1}:\n{doc}")

    if api_refs:
        context_parts.append("\n=== API Reference ===")
        for doc, meta in api_refs[:2]:  # Top 2 API references
            context_parts.append(f"\n{doc}")
```

```
    context = "\n".join(context_parts)

    prompt = f"""You have access to CircuitPython examples and API documentation.

{context}

Question: {question}

Provide a specific CircuitPython solution with working code and explanations:"""

    response = ollama.chat(model=model, messages=[{
        'role': 'user',
        'content': prompt
    }])

    return response['message']['content']
```

My recommendation:

Include the stubs, but:

1. Mark them clearly with metadata (`type: 'api_reference'`)
2. Chunk them by class/method for focused retrieval
3. Use them as supporting information alongside examples
4. Don't let them dominate results - prioritize working examples

The combination will be powerful:

- **Working examples** answer "how do I...?" questions
- **API stubs** answer "what are the parameters?" and "what methods exist?" questions

Since you're curating from first-hand sources, having both will make your RAG system much more comprehensive!

T Can you suggest a method for extracting example code from the strings contained in the .pyi files? For instance this:

```
Here is blinky::

  import time
  import digitalio
  import board

  led = digitalio.DigitalInOut(board.LED)
  led.direction = digitalio.Direction.OUTPUT
  while True:
      led.value = True
      time.sleep(0.1)
      led.value = False
      time.sleep(0.1)
```

Is a fully working example code that is included within a string in the stubs file because those same files are used to build documentation in other formats. It would be great to be able to pull the examples like this out of the stubs to be included as examples.

Excellent idea! Extracting those embedded examples from the docstrings is very valuable. Here's a comprehensive solution:

python

```python
import re
from typing import List, Dict

class StubExampleExtractor:
    def __init__(self):
        # Patterns for different documentation styles
        self.rst_code_block = re.compile(
            r'::\s*\n\n((?:[ ]{2,}.*\n)+)',
            re.MULTILINE
        )

        self.explicit_code_block = re.compile(
            r'.. code-block:: python\s*\n\n((?:[ ]{2,}.*\n)+)',
            re.MULTILINE
        )
```

```
        self.example_section = re.compile(
            r'Example(?:\s+usage)?::\s*\n\n((?:[ ]{2,}.*\n)+)',
            re.MULTILINE | re.IGNORECASE
        )

    def extract_examples_from_stub(self, stub_content: str, metadata: Dict) -> List
        """Extract all code examples from a stub file"""
        examples = []

        # Get all docstrings from the file
        docstrings = self._extract_all_docstrings(stub_content)

        for i, (docstring, context) in enumerate(docstrings):
            # Try all patterns
            for pattern_name, pattern in [
                ('rst_double_colon', self.rst_code_block),
                ('explicit_code_block', self.explicit_code_block),
                ('example_section', self.example_section)
            ]:
                matches = pattern.findall(docstring)

                for j, match in enumerate(matches):
                    # Dedent the code (remove leading spaces)
                    code = self._dedent_code(match)

                    # Validate it's actually CircuitPython code
                    if self._is_valid_circuitpython_example(code):
                        examples.append({
                            'content': code,
                            'metadata': {
                                **metadata,
                                'type': 'extracted_example',
                                'source_type': 'stub_docstring',
                                'extraction_method': pattern_name,
                                'context': context,
                                'example_index': f"{i}_{j}"
                            }
                        })

        return examples

    def _extract_all_docstrings(self, stub_content: str) -> List[tuple]:
```

```
        """Extract all docstrings with their context"""
        docstrings = []

        try:
            import ast
            tree = ast.parse(stub_content)

            # Module-level docstring
            module_doc = ast.get_docstring(tree)
            if module_doc:
                docstrings.append((module_doc, 'module'))

            # Walk through all nodes
            for node in ast.walk(tree):
                docstring = ast.get_docstring(node)
                if docstring:
                    # Determine context
                    if isinstance(node, ast.ClassDef):
                        context = f"class:{node.name}"
                    elif isinstance(node, ast.FunctionDef):
                        # Get parent class if exists
                        parent = self._find_parent_class(tree, node)
                        if parent:
                            context = f"method:{parent.name}.{node.name}"
                        else:
                            context = f"function:{node.name}"
                    else:
                        context = "unknown"

                    docstrings.append((docstring, context))

        except SyntaxError:
            # Fallback: use regex to find docstrings
            docstrings = self._regex_extract_docstrings(stub_content)

        return docstrings

    def _find_parent_class(self, tree, target_node):
        """Find the parent class of a node"""
        for node in ast.walk(tree):
            if isinstance(node, ast.ClassDef):
                for item in node.body:
                    if item is target_node:
```

```
                    return node
        return None

    def _regex_extract_docstrings(self, content: str) -> List[tuple]:
        """Fallback regex-based docstring extraction"""
        # Match triple-quoted strings
        pattern = re.compile(r'"""(.*?)"""', re.DOTALL)
        matches = pattern.findall(content)
        return [(match, 'regex_extracted') for match in matches]

    def _dedent_code(self, indented_code: str) -> str:
        """Remove consistent leading indentation"""
        lines = indented_code.split('\n')

        # Find minimum indentation (excluding empty lines)
        non_empty_lines = [line for line in lines if line.strip()]
        if not non_empty_lines:
            return indented_code

        min_indent = min(len(line) - len(line.lstrip())
                         for line in non_empty_lines)

        # Remove that indentation from all lines
        dedented_lines = []
        for line in lines:
            if line.strip():  # Non-empty line
                dedented_lines.append(line[min_indent:])
            else:  # Empty line
                dedented_lines.append('')

        return '\n'.join(dedented_lines).strip()

    def _is_valid_circuitpython_example(self, code: str) -> bool:
        """Validate that extracted code is a real CircuitPython example"""
        # Must have actual code (not just comments or empty)
        code_lines = [
            line for line in code.split('\n')
            if line.strip() and not line.strip().startswith('#')
        ]

        if len(code_lines) < 2:
            return False
```

```
        # Should have CircuitPython imports or usage
        circuitpython_indicators = [
            'import board',
            'import digitalio',
            'import analogio',
            'import busio',
            'import microcontroller',
            'import time',
            'from digitalio',
            'from analogio',
            'board.',
            'digitalio.',
            'analogio.',
        ]

        code_lower = code.lower()
        if not any(indicator.lower() in code_lower for indicator in circuitpython_i
            return False

        # Try to parse as valid Python
        try:
            import ast
            ast.parse(code)
            return True
        except SyntaxError:
            return False

    def process_stub_file(self, file_path, library_name):
        """Process a stub file and extract examples"""
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                content = f.read()

            base_metadata = {
                'source': str(file_path),
                'library': library_name,
                'file_name': file_path.name
            }

            # Extract examples
            examples = self.extract_examples_from_stub(content, base_metadata)

            return examples
```

```
        except Exception as e:
            print(f"Error processing stub {file_path}: {e}")
            return []
```

Integration with your collector:

python

```python
class CircuitPythonCollector:
    def __init__(self, output_dir="circuitpython_data"):
        self.output_dir = Path(output_dir)
        self.output_dir.mkdir(exist_ok=True)
        self.example_extractor = StubExampleExtractor()

    def _process_python_file(self, file_path, library_name):
        """Process both .py and .pyi files"""
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                content = f.read()

            is_stub = file_path.suffix == '.pyi'

            # Process the stub file itself
            if not self._is_circuitpython_code(content):
                return

            metadata = {
                'source': str(file_path),
                'library': library_name,
                'type': 'api_reference' if is_stub else 'example',
                'file_name': file_path.name,
                'is_stub': is_stub
            }

            # Save the stub file
            suffix = 'stub' if is_stub else 'example'
            output_file = self.output_dir / "processed" / f"{library_name}_{suffix}
            output_file.parent.mkdir(exist_ok=True)

            with open(output_file, 'w') as f:
                f.write(content)
```

```
            import json
            with open(f"{output_file}.meta", 'w') as f:
                json.dump(metadata, f)

            # If it's a stub, also extract embedded examples
            if is_stub:
                examples = self.example_extractor.process_stub_file(file_path, libr

                # Save each extracted example
                for example in examples:
                    example_code = example['content']
                    example_meta = example['metadata']

                    # Create unique filename
                    context = example_meta.get('context', 'unknown')
                    index = example_meta.get('example_index', '0')
                    safe_context = context.replace(':', '_').replace('.', '_')

                    example_file = self.output_dir / "processed" / f"{library_name}

                    with open(example_file, 'w') as f:
                        f.write(example_code)

                    with open(f"{example_file}.meta", 'w') as f:
                        json.dump(example_meta, f)

                print(f"  Extracted {len(examples)} examples from {file_path.name}"

        except Exception as e:
            print(f"Error processing {file_path}: {e}")
```

Test the extractor:

python

```
# Test with your digitalio stub
extractor = StubExampleExtractor()

stub_content = """Your digitalio.pyi content here"""

examples = extractor.extract_examples_from_stub(
```

```
    stub_content,
    {'library': 'digitalio', 'source': 'test'}
)

for i, example in enumerate(examples):
    print(f"\n=== Extracted Example {i+1} ===")
    print(f"Context: {example['metadata']['context']}")
    print(f"Method: {example['metadata']['extraction_method']}")
    print("\nCode:")
    print(example['content'])
```

Enhanced pattern matching for edge cases:

python

```
def extract_examples_from_stub(self, stub_content: str, metadata: Dict) → List[Dic
    """Enhanced version with more pattern matching"""
    examples = []

    # Get all docstrings
    docstrings = self._extract_all_docstrings(stub_content)

    for i, (docstring, context) in enumerate(docstrings):
        # Pattern 1: Standard RST double-colon (your "Here is blinky::" example)
        matches = self.rst_code_block.findall(docstring)
        for j, match in enumerate(matches):
            code = self._dedent_code(match)
            if self._is_valid_circuitpython_example(code):
                # Try to extract a description from text before the ::
                description = self._extract_description_before_code(docstring, matc

                examples.append({
                    'content': code,
                    'metadata': {
                        **metadata,
                        'type': 'extracted_example',
                        'source_type': 'stub_docstring',
                        'context': context,
                        'description': description,
                        'example_index': f"{i}_{j}"
                    }
                })
```

```
        # Pattern 2: Inline examples with "Example usage::"
        # Pattern 3: Code blocks after "For example::"
        # ... etc

    return examples

def _extract_description_before_code(self, docstring: str, code_block: str) -> str:
    """Extract descriptive text before a code block"""
    # Find the position of the code block
    code_pos = docstring.find(code_block)
    if code_pos == -1:
        return ""

    # Get text before code
    before_text = docstring[:code_pos]

    # Look for description in last few lines before ::
    lines = before_text.split('\n')
    for line in reversed(lines[-3:]):  # Last 3 lines
        line = line.strip()
        if line and not line.endswith('::'):
            # Remove common prefixes
            line = re.sub(r'^(For example|Here is|Example):\s*', '', line, flags=re
            return line

    return ""
```

This solution will:

1. **Extract all code examples** from docstrings in stub files
2. **Preserve context** (which class/method the example relates to)
3. **Validate** that extracted code is actually valid CircuitPython
4. **Dedent properly** so code is ready to use
5. **Save as separate examples** that can be chunked and added to your RAG system

Since stub files often have the best, most concise examples (like your "blinky" example), extracting these will significantly improve your RAG system's quality!

T I am trying to test the stubs example extractor code by itself first. It's raising this exception:

```
Traceback (most recent call last):
  File "stubs_example_extractor.py", line 199, in <module>
    examples =
example_extractor.process_stub_file(Path("circuitpython/circuitpython-
stubs/digitalio/__init__.pyi"), "digitalio")

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "stubs_example_extractor.py", line 188, in process_stub_file
    examples = self.extract_examples_from_stub(content, base_metadata)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "stubs_example_extractor.py", line 30, in
extract_examples_from_stub
    docstrings = self._extract_all_docstrings(stub_content)
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "stubs_example_extractor.py", line 76, in _extract_all_docstrings
    docstring = ast.get_docstring(node)
                ^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/ast.py", line 293, in get_docstring
    raise TypeError("%r can't have docstrings" %
node.__class__.__name__)
TypeError: 'Expr' can't have docstrings
```

The issue is that `ast.walk()` visits ALL nodes in the tree, including expression nodes that can't have docstrings. We need to filter to only check nodes that can actually have docstrings. Here's the corrected version:

python

```python
def _extract_all_docstrings(self, stub_content: str) -> List[tuple]:
    """Extract all docstrings with their context"""
    docstrings = []

    try:
        import ast
        tree = ast.parse(stub_content)

        # Module-level docstring
```

```
        module_doc = ast.get_docstring(tree)
        if module_doc:
            docstrings.append((module_doc, 'module'))

        # Walk through all nodes, but only check types that can have docstrings
        for node in ast.walk(tree):
            # Only these node types can have docstrings
            if isinstance(node, (ast.ClassDef, ast.FunctionDef, ast.AsyncFunctionDe
                docstring = ast.get_docstring(node)
                if docstring:
                    # Determine context
                    if isinstance(node, ast.ClassDef):
                        context = f"class:{node.name}"
                    elif isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
                        # Get parent class if exists
                        parent = self._find_parent_class(tree, node)
                        if parent:
                            context = f"method:{parent.name}.{node.name}"
                        else:
                            context = f"function:{node.name}"
                    else:
                        context = "unknown"

                    docstrings.append((docstring, context))

    except SyntaxError:
        # Fallback: use regex to find docstrings
        docstrings = self._regex_extract_docstrings(stub_content)

    return docstrings
```

The key change is replacing:

python

```
for node in ast.walk(tree):
    docstring = ast.get_docstring(node)
```

With:

python

```
for node in ast.walk(tree):
    if isinstance(node, (ast.ClassDef, ast.FunctionDef, ast.AsyncFunctionDef)):
        docstring = ast.get_docstring(node)
```

This ensures we only try to get docstrings from node types that can actually have them:

- `ast.Module` (handled separately at the top)
- `ast.ClassDef` (classes)
- `ast.FunctionDef` (functions/methods)
- `ast.AsyncFunctionDef` (async functions/methods)

The error occurred because `ast.walk()` returns ALL nodes including expressions, assignments, etc., and `ast.get_docstring()` raises a `TypeError` when called on nodes that can't have docstrings.

Try running your test again with this fix!

T It seems that the regex expressions don't work quite right. It appears they assume the example is complete when they find an empty line, but there are cases where the code contains an empty line with more code afterward that is part of the same example. For instance in this passage:

```
Basic digital pin support

The `digitalio` module contains classes to provide access to basic
digital IO.

All classes change hardware state and should be deinitialized when they
are no longer needed if the program continues after use. To do so,
either
call :py:meth:`!deinit` or use a context manager. See
:ref:`lifetime-and-contextmanagers` for more info.

For example::

  import digitalio
  import board

  pin = digitalio.DigitalInOut(board.LED)
  print(pin.value)

This example will initialize the the device, read
:py:data:`~digitalio.DigitalInOut.value` and then
:py:meth:`~digitalio.DigitalInOut.deinit` the hardware.
```

The regex is finding

```
  import digitalio
  import board
```

but not the rest of the example:

```
  pin = digitalio.DigitalInOut(board.LED)
  print(pin.value)
```

You're absolutely right! The regex pattern `((?:[ ]{2,}.*\n)+)` stops at the first line that doesn't match, including blank lines. We need to make it continue through blank indented lines. Here's the corrected version:

python

```python
class StubExampleExtractor:
    def __init__(self):
        # Updated patterns that handle blank lines within code blocks
        # Match indented lines OR blank lines, but stop at unindented text
        self.rst_code_block = re.compile(
            r'::\s*\n\n((?:(?:[ ]{2,}.*|[ \t]*)(?:\n|$))+)',
            re.MULTILINE
        )

        self.explicit_code_block = re.compile(
            r'.. code-block:: python\s*\n\n((?:(?:[ ]{2,}.*|[ \t]*)(?:\n|$))+)',
            re.MULTILINE
        )

        self.example_section = re.compile(
            r'Example(?:\s+usage)?::\s*\n\n((?:(?:[ ]{2,}.*|[ \t]*)(?:\n|$))+)',
            re.MULTILINE | re.IGNORECASE
        )
```

The key change is from:

- `((?:[ ]{2,}.*\n)+)` - stops at first non-matching line (including blank lines)

To:

- `((?:(?:[ ]{2,}.*|[ \t]*)(?:\n|$))+)` - continues through blank lines

This pattern matches:

- `[ ]{2,}.*` - lines with at least 2 spaces and any content
- `|` - OR
- `[ \t]*` - lines with only spaces/tabs (blank but still indented context)
- `(?:\n|$)` - followed by newline or end of string

Alternative: More robust pattern

If you still have issues, here's an even more robust version that explicitly handles the RST indentation rules:

python

```
def extract_examples_from_stub(self, stub_content: str, metadata: Dict) → List[Dic
    """Extract all code examples from a stub file"""
    examples = []

    # Get all docstrings from the file
    docstrings = self._extract_all_docstrings(stub_content)

    for i, (docstring, context) in enumerate(docstrings):
        # Find all code blocks in this docstring
        code_blocks = self._find_rst_code_blocks(docstring)

        for j, code in enumerate(code_blocks):
            # Validate it's actually CircuitPython code
            if self._is_valid_circuitpython_example(code):
                # Try to extract description
                description = self._extract_description_before_code(docstring, code

                examples.append({
                    'content': code,
                    'metadata': {
                        **metadata,
                        'type': 'extracted_example',
                        'source_type': 'stub_docstring',
                        'context': context,
                        'description': description,
                        'example_index': f"{i}_{j}"
                    }
                })

    return examples

def _find_rst_code_blocks(self, text: str) → List[str]:
    """Find all RST code blocks, handling blank lines correctly"""
    code_blocks = []
    lines = text.split('\n')

    i = 0
    while i < len(lines):
        line = lines[i]

        # Look for :: at end of line (RST code block indicator)
        if line.rstrip().endswith('::'):
```

```
            # Skip the blank line after ::
            i += 1
            if i < len(lines) and not lines[i].strip():
                i += 1

            # Now collect all indented lines (including blank lines within the bloc
            code_lines = []
            base_indent = None

            while i < len(lines):
                current_line = lines[i]

                # Determine indentation
                stripped = current_line.lstrip()
                if stripped:  # Non-empty line
                    indent = len(current_line) - len(stripped)

                    # Set base indentation from first code line
                    if base_indent is None:
                        base_indent = indent

                    # If indentation matches or is greater, it's part of the code l
                    if indent >= base_indent:
                        code_lines.append(current_line)
                        i += 1
                    else:
                        # Unindented line, end of code block
                        break
                else:
                    # Blank line - include it but don't break
                    # (might be blank line within code)
                    code_lines.append(current_line)
                    i += 1

                    # However, if we see TWO consecutive blank lines, stop
                    if i < len(lines) and not lines[i].strip():
                        break

            # Process collected code block
            if code_lines:
                # Remove trailing blank lines
                while code_lines and not code_lines[-1].strip():
                    code_lines.pop()
```

```
                if code_lines:
                    code = '\n'.join(code_lines)
                    dedented = self._dedent_code(code)
                    if dedented.strip():
                        code_blocks.append(dedented)
            else:
                i += 1

    return code_blocks
```

Test it with your example:

python

```
test_text = """Basic digital pin support

The `digitalio` module contains classes to provide access to basic digital IO.

For example::

  import digitalio
  import board

  pin = digitalio.DigitalInOut(board.LED)
  print(pin.value)

This example will initialize the device."""

extractor = StubExampleExtractor()
code_blocks = extractor._find_rst_code_blocks(test_text)

print(f"Found {len(code_blocks)} code blocks:")
for i, block in enumerate(code_blocks):
    print(f"\n=== Block {i+1} ===")
    print(block)
```

This should now correctly extract:

python

```
import digitalio
import board

pin = digitalio.DigitalInOut(board.LED)
print(pin.value)
```

The line-by-line parsing approach is more robust than regex for handling RST's indentation-based syntax, especially with blank lines within code blocks.

Retry

Claude can make mistakes. Please double-check responses.