RAG System Debugging Guide: Common Bugs & Solutions

> **For ML Engineering Interviews & Production Deployments**
> Comprehensive documentation of real bugs encountered during development and how to prevent them.

Table of Contents

Overview

This document chronicles the actual bugs encountered while building a production-grade RAG system with Endee vector database, providing insights valuable for:

- **Technical interviews** discussing debugging experience
- **Code reviews** and best practices
- **Onboarding** new ML engineers to vector DB projects
- **Production deployment** troubleshooting

System Architecture Context

```
Documents → Ingestion → Embeddings → Endee Storage → Retrieval → LLM → Answer
```

Tech Stack:

- Python 3.13
- Endee vector database (Docker)
- sentence-transformers (all-MiniLM-L6-v2)
- OpenAI GPT-3.5-turbo
- PyPDF for document processing

Bugs Encountered During Development

Bug #1: Variable Name Shadowing

Severity: 🔴 High
Error Type: `UnboundLocalError`
Discovery Phase: Runtime execution

The Problem

```python
# In src/ingest.py
def chunk_text(text: str, chunk_size: int, overlap: int) -> List[str]:
    """Function to split text into chunks"""
    # ... implementation
    return chunks

def process_documents(directory: Path = None) -> List[Dict[str, Any]]:
    for doc in documents:
        chunks = chunk_text(doc["content"], CHUNK_SIZE, CHUNK_OVERLAP)  # ✓ This works

        # BUG: Loop variable shadows the function name!
        for i, chunk_text in enumerate(chunks):  # ✗ chunk_text now refers to string, not function
            chunk_obj = {
                "id": create_chunk_id(doc["name"], i),
                "text": chunk_text,  # This is fine
                # ...
            }
```

Error Message:

```
UnboundLocalError: cannot access local variable 'chunk_text' where it is not associated with a value
```

Root Cause Analysis

1. **Scope pollution**: The loop variable `chunk_text` (a string) shadows the module-level function `chunk_text()`
2. **Python's name resolution**: When Python sees `chunk_text` used as a variable in the loop, it treats ALL occurrences in that scope as local variables
3. **Forward reference issue**: The call to `chunk_text()` function happens BEFORE the loop, but Python's compiler sees the variable assignment in the loop and assumes `chunk_text` is a local variable everywhere in the function

The Fix

```python
# BEFORE (Broken)
for i, chunk_text in enumerate(chunks):
    chunk_obj = {"text": chunk_text, ...}

# AFTER (Fixed)
for i, text_chunk in enumerate(chunks):  # Renamed to avoid shadowing
    chunk_obj = {"text": text_chunk, ...}
```

Prevention Strategies

✅ Use descriptive, unique variable names

- Avoid reusing function names as variables
- Follow naming conventions: `text_chunk`, `chunk_data`, `chunk_content`

✅ Linting tools

```bash
# Use pylint to catch shadowing
pylint src/ingest.py
# Warning: Redefining name 'chunk_text' from outer scope
```

✅ Type hints help IDE detection

```python
def chunk_text(text: str, ...) -> List[str]: ...
for i, text_chunk: str in enumerate(chunks): ...
```

Interview Talking Points

- Demonstrates understanding of **Python's scoping rules** (LEGB: Local, Enclosing, Global, Built-in)
- Shows **debugging methodology**: read error messages carefully, trace variable usage
- Highlights importance of **code review** and **static analysis tools**

Bug #2: API Response Format Mismatch

Severity: 🟠 Medium
Error Type: `AttributeError`
Discovery Phase: Runtime execution with Endee server running

The Problem

```python
# In src/store.py
def initialize_endee_index(index_name: str = None):
    existing_indexes = client.list_indexes()

    # BUG: Assumed list_indexes() returns list of dicts
    if index_name in [idx.get('name') for idx in existing_indexes]:  # ✗ Breaks!
        # ...
```

Error Message:

```
AttributeError: 'str' object has no attribute 'get'
```

Root Cause Analysis

1. **Incorrect assumption about API response format**
   - Expected: `[{"name": "index1", ...}, {"name": "index2", ...}]`
   - Actual: `["index1", "index2"]` (simple list of strings)
2. **Lack of API documentation verification**
   - Didn't check Endee SDK source code or test responses
   - Made assumptions based on common patterns from other vector DBs
3. **Integration testing gap**
   - Bug only appeared when Endee server was running
   - Mock tests would have missed this

The Fix

```python
# BEFORE (Broken)
existing_indexes = client.list_indexes()
if index_name in [idx.get('name') for idx in existing_indexes]:  # Assumes dict

# AFTER (Fixed)
existing_indexes = client.list_indexes()
# list_indexes() returns a list of index names (strings)
if index_name in existing_indexes:  # Direct string comparison
```

Debugging Process

Step 1: Print the actual response

```python
existing_indexes = client.list_indexes()
logger.info(f"Existing indexes: {existing_indexes}")  # Added logging
# Output: {'indexes': [{'name': 'rag_documents', ...}]}
```

Step 2: Discovered nested structure

```
# Actually, the response was even more complex!
# It's a dict with 'indexes' key containing the list
if index_name in existing_indexes:  # This worked because string was there
```

Prevention Strategies

✅ Always inspect API responses first

```
# Add defensive logging during development
response = client.list_indexes()
print(f"DEBUG: list_indexes response type: {type(response)}")
print(f"DEBUG: list_indexes response: {response}")
```

✅ Read SDK source code

```
# Check the actual Endee SDK implementation
from endee import Endee
import inspect
print(inspect.getsource(Endee.list_indexes))
```

✅ Use type hints and validate responses

```
from typing import List, Dict, Any

def initialize_endee_index(index_name: str = None) -> Any:
    existing_indexes: List[str] = client.list_indexes()
    # Type checker would warn if we try .get() on strings
```

✅ Integration tests with real services

```
# Don't just mock - test against actual Endee server
def test_list_indexes_format():
    client = get_endee_client()
    indexes = client.list_indexes()
    assert isinstance(indexes, (list, dict))  # Validate structure
```

Interview Talking Points

- Shows ability to **debug third-party APIs** and handle documentation gaps
- Demonstrates **iterative debugging**: add logging → inspect data → fix
- Highlights difference between **unit tests vs integration tests**
- Real-world example of **defensive programming**

Bug #3: Enum Value Mismatch

Severity: 🟠 Medium
Error Type: `AttributeError`
Discovery Phase: Index creation

The Problem

```
# In src/store.py
from endee import Precision

def get_precision_enum(precision_str: str) -> Precision:
    precision_map = {
        "BINARY": Precision.BINARY,  # ✗ This doesn't exist!
        "INT8D": Precision.INT8D,
        # ...
    }
    return precision_map.get(precision_str, Precision.INT8D)

# Usage
client.create_index(
    name=index_name,
    precision=get_precision_enum(config.PRECISION),  # Breaks here
)
```

Error Message:

```
AttributeError: type object 'Precision' has no attribute 'BINARY'. Did you
mean: 'BINARY2'?
```

Root Cause Analysis

1. **SDK version differences or documentation outdated**
   - Documentation might have referenced `BINARY`
   - Actual SDK uses `BINARY2`
2. **Insufficient enum introspection**
   - Didn't verify available enum values before using them
3. **No validation of configuration values**
   - `config.PRECISION = "INT8D"` worked fine
   - Would have failed if someone set `config.PRECISION = "BINARY"`

The Fix

```
# BEFORE (Broken)
precision_map = {
    "BINARY": Precision.BINARY,  # Doesn't exist
    # ...
}

# AFTER (Fixed)
```

```
precision_map = {
    "BINARY": Precision.BINARY2,   # Corrected to actual enum value
    "BINARY2": Precision.BINARY2,  # Support both names
    "INT8D": Precision.INT8D,
    "INT16D": Precision.INT16D,
    "FLOAT16": Precision.FLOAT16,
    "FLOAT32": Precision.FLOAT32
}
```

Debugging Process

Step 1: Introspect the enum

```
# Command-line debugging
python -c "from endee import Precision; print(dir(Precision))"
# Output: ['BINARY2', 'FLOAT16', 'FLOAT32', 'INT16D', 'INT8D', ...]
```

Step 2: Update mapping

```
# Now we know the actual values available
```

Prevention Strategies

✅ Introspect enums programmatically

```
from endee import Precision

# List all valid values
valid_precisions = [p for p in dir(Precision) if not p.startswith('_')]
print(f"Valid precision values: {valid_precisions}")
```

✅ Validate configuration on startup

```
def validate_config():
    """Validate all config values before running application"""
    try:
        precision = get_precision_enum(config.PRECISION)
    except AttributeError as e:
        raise ValueError(f"Invalid PRECISION config: {config.PRECISION}. "
                        f"Valid values: {[p for p in dir(Precision) if not
p.startswith('_')]}")

# Call during app initialization
validate_config()
```

✅ Use hasattr() for defensive coding

```python
def get_precision_enum(precision_str: str) -> Precision:
    if hasattr(Precision, precision_str):
        return getattr(Precision, precision_str)
    else:
        logger.warning(f"Unknown precision '{precision_str}', using INT8D")
        return Precision.INT8D
```

Interview Talking Points

- Shows **SDK integration challenges** and version management
- Demonstrates **runtime introspection** techniques in Python
- Highlights importance of **configuration validation**
- Example of using Python's **reflection capabilities** (`dir()`, `hasattr()`, `getattr()`)

Bug #4: Race Condition in Index Creation

Severity: 🟡 Low
Error Type: `Conflict`
Discovery Phase: Retrieval pipeline

The Problem

```python
# In src/store.py
def initialize_endee_index(index_name: str = None):
    existing_indexes = client.list_indexes()

    if index_name in existing_indexes:
        index = client.get_index(name=index_name)
    else:
        client.create_index(name=index_name, ...)  # ✗ May fail if index created meanwhile
        index = client.get_index(name=index_name)
```

Error Message:

```
Conflict: Index with this name already exists for this user
```

Root Cause Analysis

1. **Time-of-check to time-of-use (TOCTOU) race condition**

   ```
   Time 1: Check if index exists → NO
   Time 2: (Another process creates index)
   Time 3: Try to create index → CONFLICT!
   ```

2. **Stateless API design**
   - Between `list_indexes()` and `create_index()`, state can change

    - No atomic "create if not exists" operation

3. **Multiple processes accessing same Endee instance**
    - Ingestion script and query script both call `initialize_endee_index()`

The Fix #1: Exception Handling

```
# BEFORE (Broken)
else:
    client.create_index(name=index_name, ...)
    index = client.get_index(name=index_name)

# AFTER (Fixed with try-catch)
else:
    try:
        client.create_index(name=index_name, ...)
        logger.info(f"Index '{index_name}' created successfully")
    except Exception as create_error:
        # Index might have been created between list and create calls
        if "already exists" in str(create_error).lower():
            logger.warning(f"Index '{index_name}' was created by another process, retrieving it")
        else:
            raise  # Re-raise if it's a different error

    index = client.get_index(name=index_name)
```

The Fix #2: Idempotent Design

```
def get_or_create_index(index_name: str):
    """Idempotent index retrieval - always safe to call"""
    try:
        # Try to get index first (most common case in production)
        return client.get_index(name=index_name)
    except NotFoundError:
        # Index doesn't exist, create it
        try:
            client.create_index(name=index_name, ...)
            return client.get_index(name=index_name)
        except ConflictError:
            # Created by another process, just get it
            return client.get_index(name=index_name)
```

Prevention Strategies

✅ Design for idempotency

```
# Operations should be safely repeatable
# get_or_create pattern is common in production systems
```

✅ Graceful error handling for conflicts

```
# Don't fail on conflicts that can be resolved
# Log warnings instead of errors for expected race conditions
```

✅ **Use retries for transient failures**

```
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(stop=stop_after_attempt(3), wait=wait_exponential(min=1, max=10))
def initialize_endee_index(index_name: str):
    # Automatic retries for transient network issues
    # ...
```

Interview Talking Points

- Demonstrates understanding of **concurrency and race conditions**
- Shows **idempotent design patterns** for distributed systems
- Example of **exception handling** vs **exception prevention**
- Relevant to **microservices architecture** where multiple instances may compete

---

Bug #5: Dependency Version Conflicts

Severity: 🔴 High
Error Type: `ModuleNotFoundError`
Discovery Phase: Installation / Import

The Problem

Initial attempt:

```
pip install -r requirements.txt
# Success... but then:

python app.py --ingest
# ModuleNotFoundError: No module named 'pypdf'
```

Root cause:

```
# User's global Python environment had conflicts:
ERROR: pip's dependency resolver does not currently take into account all the
packages that are installed.
langchain-community 0.0.27 requires numpy<2,>=1, but you have numpy 2.2.6
llama-index-core 0.10.68 requires numpy<2.0.0, but you have numpy 2.2.6
```

Root Cause Analysis

1. **Global Python environment pollution**
    - Multiple ML projects installed incompatible packages

    - `numpy 2.x` required by some, `numpy 1.x` required by others

2. **Installation succeeded but imports failed**
    - `pip` installed packages despite warnings
    - But some packages failed to install correctly due to conflicts
3. **No environment isolation**
    - Running everything in system Python
    - Different projects competing for same dependencies

The Fix: Virtual Environment

```
# Create isolated environment
python3 -m venv venv

# Activate it
source venv/bin/activate

# Install dependencies in isolation
pip install -r requirements.txt
# Success! All packages install cleanly

# Run application
python app.py --ingest
# Works perfectly!
```

Prevention Strategies

✅ Always use virtual environments

```
# For every project
python3 -m venv venv
source venv/bin/activate

# Or use conda
conda create -n rag-system python=3.13
conda activate rag-system
```

✅ Pin dependency versions

```
# requirements.txt
endee==0.1.8          # Not endee>=0.1.0
numpy>=2.0.0,<3.0.0   # Explicit range
pypdf==6.6.2          # Exact version for reproducibility
```

✅ Use dependency management tools

```
# Modern alternatives
pip-tools  # pip-compile generates locked requirements
poetry     # All-in-one dependency + env management
uv         # Ultra-fast pip replacement
```

✅ **Verify installations**

```
# Add import checks at top of main module
try:
    import pypdf
    import endee
    import sentence_transformers
except ImportError as e:
    print(f"Missing dependency: {e}")
    print("Run: pip install -r requirements.txt")
    sys.exit(1)
```

Interview Talking Points

- Demonstrates **Python environment best practices**
- Shows understanding of **dependency hell** and how to solve it
- Relevant to **CI/CD pipelines** and **reproducible builds**
- Example of **debugging environment issues** vs code issues

Bug #6: API Key Configuration Issues

Severity: 🟡 Low
Error Type: `401 Unauthorized`
Discovery Phase: LLM generation

The Problem

```
# In .env file
OPENAI_API_KEY=sk-old-key-here

# User provides new key
OPENAI_API_KEY=sk-new-key-hereskold-keyappendedbyaccident  # Both keys concatenated!
```

Error Message:

```
Error code: 401 - Incorrect API key provided: sk-new-key-hereskold-keyappended...
```

Root Cause Analysis

1. **Text replacement error**

    - File edit tool replaced part of the line but concatenated old data
    - Resulted in malformed API key

2. **No validation of API key format**
    - OpenAI keys have specific format: `sk-proj-...` or `sk-...`
    - No regex check to validate format
3. **Secrets in plaintext**
    - `.env` file is not encrypted
    - Easy to make typos in long keys

The Fix

```
# Manual verification
cat .env | grep OPENAI_API_KEY
# Check the key looks correct

# Proper format
OPENAI_API_KEY=sk-proj-
OnQ_C5i9VCKQXbc3TSc4TxBWgmBWU88bVsZPJmQF3y5Hxym90MNtAZD5iExsgDyhI0epEBXb9bT3Bl
bkFJrXChaKfJIYYvDbqGCAAgVKC4TTrbhvOtVXNz9Lkwz4jVGWTuiSENXHiknUN9E5eFM9AVBVaksA
```

Prevention Strategies

✅ Validate API keys on startup

```
import re

def validate_openai_key(key: str) -> bool:
    """Validate OpenAI API key format"""
    # OpenAI keys start with sk- or sk-proj-
    pattern = r'^sk-[a-zA-Z0-9]{32,}$|^sk-proj-[a-zA-Z0-9]{32,}$'
    return bool(re.match(pattern, key))

# In config.py
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")
if OPENAI_API_KEY and not validate_openai_key(OPENAI_API_KEY):
    logger.warning("OPENAI_API_KEY format looks invalid!")
```

✅ Use secrets management tools

```
# Production best practices
# 1. AWS Secrets Manager
# 2. HashiCorp Vault
# 3. Kubernetes Secrets
# 4. GitHub Secrets (for CI/CD)
```

✅ Add .env to .gitignore

```
# Never commit secrets!
.env
*.env
.env.local
.env.*.local
```

✅ **Provide .env.example template**

```
# .env.example (safe to commit)
OPENAI_API_KEY=your-api-key-here
ENDEE_BASE_URL=http://localhost:8080/api/v1
ENDEE_AUTH_TOKEN=

# .env (never commit, listed in .gitignore)
OPENAI_API_KEY=sk-proj-actual-secret-key-here
```

Interview Talking Points

- Shows **security-conscious development** practices
- Demonstrates **secrets management** awareness
- Example of **configuration validation** and fail-fast principles
- Relevant to **DevOps** and **production deployments**

Common Pitfalls for ML Engineers

Category 1: Vector Database Integration

Pitfall: Not Understanding Index Parameters

```
# ❌ Bad: Using defaults without understanding
client.create_index(name="my_index", dimension=384)

# ✅ Good: Configured based on use case
client.create_index(
    name="my_index",
    dimension=384,           # Match embedding model
    space_type="cosine",     # Normalized vectors
    precision=Precision.INT8D,  # 8x compression, minimal quality loss
    M=16,                    # HNSW parameter: higher = better recall
    ef_con=128              # Construction time tradeoff
)
```

Common mistakes:

- Wrong `dimension` (doesn't match embedding model)
- Wrong `space_type` (using L2 with normalized vectors)
- No understanding of HNSW parameters

How to prevent:

- Read vector DB documentation thoroughly
- Understand your embedding model's output dimension
- Benchmark different parameter settings

Pitfall: Batch Size Limits

```python
# ❌ Bad: Trying to upsert 10,000 vectors at once
index.upsert(all_10000_vectors)  # May timeout or fail

# ✅ Good: Respect API batch limits
def batch_upsert(index, vectors, batch_size=1000):
    for i in range(0, len(vectors), batch_size):
        batch = vectors[i:i + batch_size]
        index.upsert(batch)
        logger.info(f"Upserted batch {i//batch_size + 1}")
```

Common mistakes:

- Not reading API limits (Endee limit: 1000 vectors/request)
- No progress logging for long operations
- No error handling for partial failures

How to prevent:

- Check documentation for limits
- Always batch large operations
- Add progress bars (tqdm) for visibility

Category 2: Embedding Model Issues

Pitfall: Dimension Mismatch

```python
# ❌ Bad: Index and model don't match
index = client.create_index(dimension=384)  # all-MiniLM-L6-v2
model = SentenceTransformer("all-mpnet-base-v2")  # 768 dimensions!

# ✅ Good: Verify dimensions match
model = SentenceTransformer(config.EMBEDDING_MODEL_NAME)
model_dim = model.get_sentence_embedding_dimension()
assert model_dim == config.VECTOR_DIMENSION, \
    f"Model dimension ({model_dim}) != config ({config.VECTOR_DIMENSION})"
```

How to prevent:

- Store model name and dimension together in config
- Add assertion checks

- Log model metadata on startup

Pitfall: Embedding Efficiency

```
# ❌ Bad: Embedding one at a time
for doc in documents:
    embedding = model.encode(doc)  # Inefficient!

# ✅ Good: Batch encoding
all_texts = [doc["text"] for doc in documents]
embeddings = model.encode(all_texts, batch_size=32, show_progress_bar=True)
```

Common mistakes:

- Not using GPU when available
- No batch processing
- Not showing progress for long jobs

Category 3: Error Handling

Pitfall: Not Validating External Dependencies

```
# ❌ Bad: Assume everything works
def main():
    client = get_endee_client()
    store_embeddings(chunks)

# ✅ Good: Check dependencies first
def main():
    # Check Endee server is reachable
    try:
        client = get_endee_client()
        client.list_indexes()  # Test connection
    except Exception as e:
        logger.error("Cannot connect to Endee server")
        logger.error("Make sure Endee is running: docker compose up -d")
        sys.exit(1)

    # Check OpenAI key is configured
    if not os.getenv("OPENAI_API_KEY"):
        logger.error("OPENAI_API_KEY not set in .env file")
        sys.exit(1)
```

Category 4: Production Engineering

Pitfall: No Logging

```
# ❌ Bad: Silent failures
def process_documents(directory):
    docs = load_documents(directory)
    chunks = chunk_text(docs)
    return chunks

# ✅ Good: Comprehensive logging
def process_documents(directory):
    logger.info(f"Starting document processing from {directory}")
    docs = load_documents(directory)
    logger.info(f"Loaded {len(docs)} documents")

    chunks = chunk_text(docs)
    logger.info(f"Created {len(chunks)} chunks")

    return chunks
```

Debugging Methodology

Step-by-Step Process Used

1. **Read the error message carefully**

   ```
   AttributeError: 'str' object has no attribute 'get'
   ```

   - What type is it? (`str`)
   - What did we expect? (dict with `.get()`)
   - Where in the code? (line number in traceback)

2. **Add logging to inspect runtime values**

   ```
   logger.info(f"Type: {type(existing_indexes)}")
   logger.info(f"Value: {existing_indexes}")
   ```

3. **Test in isolation**

   ```
   # Quick REPL test
   from endee import Endee
   client = Endee()
   result = client.list_indexes()
   print(type(result), result)
   ```

4. **Check documentation/source code**

   ```
   import inspect
   print(inspect.getsource(Endee.list_indexes))
   ```

5. **Fix and verify**

  - Make minimal change
  - Test immediately
  - Don't fix multiple bugs at once

Prevention Strategies

Development Best Practices

✅ Use Type Hints

```
from typing import List, Dict, Any

def store_embeddings(chunks: List[Dict[str, Any]]) -> None:
    # Type checker catches many bugs before runtime
    pass
```

✅ Linting & Static Analysis

```
# Install tools
pip install pylint mypy black

# Run before commits
pylint src/
mypy src/
black src/ --check
```

✅ Unit Tests

```
def test_chunk_text():
    text = "A" * 1000
    chunks = chunk_text(text, chunk_size=100, overlap=10)
    assert len(chunks) > 0
    assert all(len(c) <= 110 for c in chunks)  # Max size + overlap
```

✅ Integration Tests

```
@pytest.mark.integration
def test_endee_roundtrip():
    """Test actual Endee connection"""
    client = get_endee_client()
    indexes = client.list_indexes()
    assert isinstance(indexes, (list, dict))
```

✅ Defensive Programming

```
def safe_operation():
    try:
        result = risky_operation()
    except SpecificError as e:
        logger.error(f"Expected error: {e}")
        # Handle gracefully
    except Exception as e:
        logger.exception("Unexpected error")
        raise  # Re-raise unexpected errors
```

Interview Talking Points

How to Discuss These Bugs in Interviews

Structure Your Response (STAR Method)

Situation: "While building a RAG system with Endee vector database..."

Task: "I needed to integrate the document storage and retrieval pipeline..."

Action: "I encountered an AttributeError when trying to list existing indexes. I debugged by adding logging to inspect the API response format, discovered it returned strings instead of dicts, and updated my code accordingly..."

Result: "This taught me to always verify API response formats rather than making assumptions, and I now use integration tests to catch these issues early."

Key Themes to Highlight

1. **Systematic Debugging**
    - "I use a methodical approach: read errors carefully, add logging, test in isolation, verify assumptions"
2. **Root Cause Analysis**
    - "I don't just fix symptoms - I investigate why the bug happened and how to prevent it class-wide"
3. **Production Awareness**
    - "I think about concurrent access, error handling, and monitoring from the start"
4. **Learning & Adaptation**
    - "Each bug taught me something: scope rules, API contracts, race conditions, dependency management"
5. **Tools & Best Practices**
    - "I use linting, type checking, virtual environments, and comprehensive logging as standard practice"

Sample Interview Questions & Answers

Q: Tell me about a challenging bug you've debugged.

A: "When building a RAG system with the Endee vector database, I encountered a subtle variable shadowing bug. I had a function called `chunk_text()` that split documents, and later in my processing loop, I used `chunk_text` as a loop variable. Python's scoping rules meant that once I assigned to `chunk_text` in the loop, all references to it in that function scope were treated as the local variable - even the earlier function call. This manifested as an UnboundLocalError.

I debugged this by carefully reading the traceback, which pointed to the function call. I added print statements to trace variable scopes and realized the shadowing issue. The fix was simple - rename the loop variable to `text_chunk` - but the lesson was valuable: always use unique, descriptive variable names and leverage linting tools like pylint which would have caught this during development."

Q: How do you handle ambiguous or poorly documented third-party APIs?

A: "I encountered this with Endee's `list_indexes()` method. The documentation wasn't entirely clear about the response format - I assumed it returned a list of dictionaries like `[{"name": "index1"}]`, but it actually returned a simple list of strings `["index1"]`.

My approach was:

1. **Add logging** to inspect the actual response at runtime
2. **Read the SDK source code** using Python's inspect module to see the implementation
3. **Write integration tests** against the real API to validate my assumptions
4. **Document my findings** in code comments for future maintainers

This experience reinforced that when working with third-party libraries, I should verify rather than assume, and integration tests are crucial alongside unit tests."

Q: How do you prevent bugs in production?

A: "I use multiple layers of defense:

Development time:

- Type hints with mypy for static type checking
- Pylint for catching common mistakes like variable shadowing
- Virtual environments to isolate dependencies

Testing:

- Unit tests for business logic
- Integration tests for external services
- Validation functions for configuration and API responses

Runtime:

- Comprehensive logging at appropriate levels (DEBUG, INFO, WARNING, ERROR)
- Input validation and assertion checks
- Graceful error handling with specific exception types

Example: In the RAG system, I validate the OpenAI API key format on startup, check that the Endee server is reachable before attempting operations, and use retry logic for transient failures. I also made operations idempotent - for instance, index creation handles the case where the index already exists rather than failing."

Summary Checklist for Interviews

When discussing this project, mention:

- ✅ Fixed 6 distinct bug classes (scoping, API contracts, enums, race conditions, dependencies, configuration)
- ✅ Used systematic debugging: logging, inspection, isolation testing
- ✅ Implemented production best practices: virtual environments, type hints, comprehensive testing
- ✅ Understood root causes: Python scoping (LEGB), API integration challenges, concurrency
- ✅ Applied defensive programming: validation, error handling, idempotent operations
- ✅ Used modern tooling: pylint, mypy, docker, git, virtual envs

Key Insight: "Building this RAG system taught me that production ML engineering is 20% algorithms and 80% engineering fundamentals - dependency management, error handling, API integration, and robust testing."

Additional Resources

Tools Used

- **pylint**: Static code analysis
- **mypy**: Static type checking
- **black**: Code formatting
- **pytest**: Testing framework
- **docker**: Containerization
- **tqdm**: Progress bars for long operations

Debugging Commands

```
# Inspect Python objects
python -c "from endee import Precision; print(dir(Precision))"

# Check import issues
```

```
python -c "import pypdf; print(pypdf.__version__)"

# Verify environment
which python
pip list | grep endee

# Check logs
tail -f rag_system.log

# Test Endee connection
curl http://localhost:8080/api/v1/index/list
```

Further Reading

- Python Scoping Rules (LEGB): https://realpython.com/python-scope-legb-rule/
- Vector Database Fundamentals: https://www.pinecone.io/learn/vector-database/
- HNSW Algorithm: https://arxiv.org/abs/1603.09320
- Production ML Best Practices: https://ml-ops.org/

Last Updated: 2026-01-30
Version: 1.0
Author: ML Engineering Team