Embedding Implementation Plan for Spring Test App

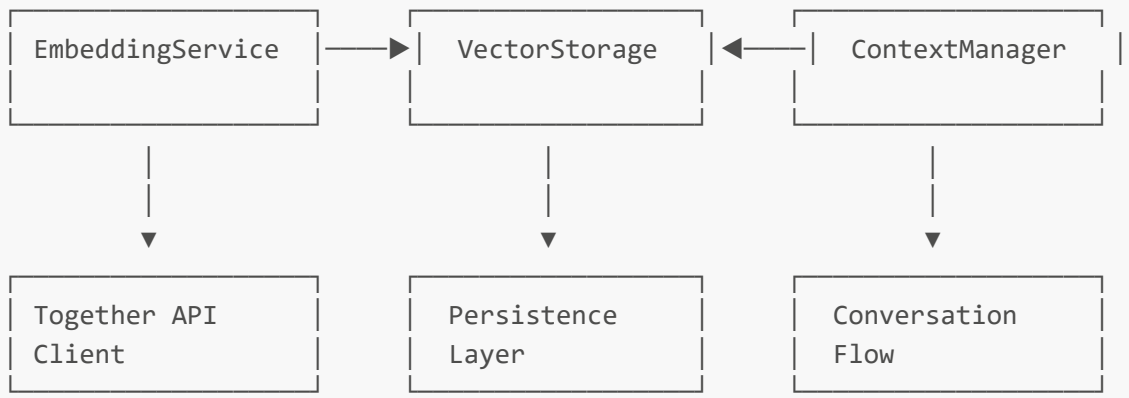
Overview

This document outlines the implementation plan for adding semantic search capabilities to the Spring Test App using Together.ai's embedding models. The embeddings will enhance the application's ability to maintain context, understand user intent, and provide more intelligent responses by:

1. Storing and retrieving relevant context from previous conversations
2. Keeping track of spring specifications mentioned in conversations
3. Relating new requests to similar past interactions
4. Understanding various ways users might express the same concepts

Architecture

Core Components



Integration Points

The embedding system will:

1. Extend the `TogetherAPIClient` to generate embeddings
2. Create a new `EmbeddingService` to manage embeddings operations
3. Add a `VectorStorage` class for storing and querying vector embeddings
4. Integrate with `ChatService` to enhance conversation context
5. Update `SequenceGenerator` to leverage relevant context

File Structure and New Components

```
spring_test_app/
├── services/
│   ├── embedding_service.py       # NEW - Main service for embedding operations
│   └── context_manager.py         # NEW - Service for managing conversation
context
```

```
│
├── utils/
│   ├── together_api_client.py     # MODIFIED - Add embedding API support
│   └── vector_storage.py          # NEW - In-memory vector store with persistence
│
├── models/
│   └── data_models.py             # MODIFIED - Add embedding-related data models
│
└── logs/
    └── app.log                    # EXISTING - Leverage for conversation tracking
```

Implementation Details

1. `embedding_service.py`

```python
class EmbeddingService:
    """Service for generating and managing embeddings."""

    DEFAULT_MODEL = "togethercomputer/bge-base-en-v1.5"

    def __init__(self, api_client, vector_storage=None):
        """Initialize embedding service.

        Args:
            api_client: TogetherAPIClient instance.
            vector_storage: Optional VectorStorage instance.
        """
        self.api_client = api_client
        self.vector_storage = vector_storage or VectorStorage()
        self.embedding_cache = {}  # Cache to avoid duplicate API calls

    def generate_embedding(self, text, model=None):
        """Generate an embedding for text using Together.ai API.

        Args:
            text: Text to embed.
            model: Optional model to use.

        Returns:
            Embedding vector.
        """
        # Use cache if available
        cache_key = f"{text}:{model or self.DEFAULT_MODEL}"
        if cache_key in self.embedding_cache:
            return self.embedding_cache[cache_key]

        # Get embedding from API
        embedding = self.api_client.generate_embedding(
            text,
            model or self.DEFAULT_MODEL
        )
```

```
        # Cache the result
        self.embedding_cache[cache_key] = embedding

        return embedding

    def index_conversation(self, message, metadata=None):
        """Index a conversation message in the vector store.

        Args:
            message: ChatMessage instance or text.
            metadata: Optional metadata.

        Returns:
            ID of the indexed document.
        """
        # Extract text from ChatMessage if needed
        text = message.content if hasattr(message, 'content') else message

        # Generate embedding
        embedding = self.generate_embedding(text)

        # Store in vector storage
        doc_id = self.vector_storage.add(
            embedding=embedding,
            text=text,
            metadata=metadata or {}
        )

        return doc_id

    def search_similar(self, query, limit=3):
        """Search for similar content in vector storage.

        Args:
            query: Query text or embedding.
            limit: Maximum number of results.

        Returns:
            List of tuples (score, document).
        """
        # Generate embedding if query is text
        if isinstance(query, str):
            query = self.generate_embedding(query)

        # Search in vector storage
        return self.vector_storage.search(query, limit=limit)
```

2. vector_storage.py

```python
class VectorStorage:
    """Simple in-memory vector store with persistence."""

    def __init__(self, storage_path=None):
        """Initialize vector storage.

        Args:
            storage_path: Path where vectors will be persisted.
        """
        self.storage_path = storage_path or os.path.join(
            os.path.dirname(os.path.dirname(os.path.abspath(__file__))),
            "appdata",
            "vector_store.dat"
        )
        self.documents = {}  # id -> {embedding, text, metadata}
        self.load()

    def add(self, embedding, text, metadata=None):
        """Add a document to the vector store.

        Args:
            embedding: Vector embedding.
            text: Original text.
            metadata: Optional metadata.

        Returns:
            Document ID.
        """
        # Generate ID if not in metadata
        doc_id = metadata.get("id", str(uuid.uuid4()))

        # Store document
        self.documents[doc_id] = {
            "embedding": embedding,
            "text": text,
            "metadata": metadata or {},
            "timestamp": datetime.now().isoformat()
        }

        # Save to disk
        self.save()

        return doc_id

    def search(self, query_embedding, limit=3):
        """Search for similar documents.

        Args:
            query_embedding: Query embedding vector.
            limit: Maximum number of results.

        Returns:
            List of tuples (score, document).
```

```
        """
        results = []

        for doc_id, doc in self.documents.items():
            # Calculate cosine similarity
            score = self._cosine_similarity(query_embedding, doc["embedding"])
            results.append((score, doc))

        # Sort by score (descending)
        results.sort(reverse=True, key=lambda x: x[0])

        return results[:limit]

    def _cosine_similarity(self, a, b):
        """Calculate cosine similarity between two vectors."""
        dot_product = sum(x * y for x, y in zip(a, b))
        magnitude_a = math.sqrt(sum(x * x for x in a))
        magnitude_b = math.sqrt(sum(x * x for x in b))

        if magnitude_a == 0 or magnitude_b == 0:
            return 0

        return dot_product / (magnitude_a * magnitude_b)

    def save(self):
        """Save vector store to disk."""
        try:
            # Ensure directory exists
            os.makedirs(os.path.dirname(self.storage_path), exist_ok=True)

            # Serialize data
            with open(self.storage_path, "wb") as f:
                pickle.dump(self.documents, f)

            # Also create JSON representation for inspection
            from medium.dat_json_utility import convert_dat_to_json
            convert_dat_to_json(self.storage_path)

            return True
        except Exception as e:
            logging.error(f"Error saving vector store: {str(e)}")
            return False

    def load(self):
        """Load vector store from disk."""
        if not os.path.exists(self.storage_path):
            return

        try:
            with open(self.storage_path, "rb") as f:
                self.documents = pickle.load(f)
        except Exception as e:
            logging.error(f"Error loading vector store: {str(e)}")
            self.documents = {}
```

3. context_manager.py

```
class ContextManager:
    """Manages conversation context using embeddings."""

    def __init__(self, embedding_service, chat_service):
        """Initialize context manager.

        Args:
            embedding_service: EmbeddingService instance.
            chat_service: ChatService instance.
        """
        self.embedding_service = embedding_service
        self.chat_service = chat_service
        self.current_context = {
            "spring_specs": {},
            "recent_topics": [],
            "active_conversation": True
        }

    def process_message(self, message):
        """Process a new message, extracting context and updating state.

        Args:
            message: ChatMessage instance.

        Returns:
            Updated context.
        """
        # Index message for future reference
        self.embedding_service.index_conversation(
            message,
            metadata={
                "role": message.role,
                "timestamp": message.timestamp.isoformat()
            }
        )

        # Extract spring specifications if present
        if message.role == "user":
            specs = self._extract_specifications(message.content)
            if specs:
                self.current_context["spring_specs"].update(specs)

        # Update recent topics
        self.current_context["recent_topics"] =
self._extract_topics(message.content)

        return self.current_context
```

```
    def get_relevant_context(self, message):
        """Get context relevant to the current message.

        Args:
            message: Current message text or ChatMessage.

        Returns:
            Dictionary with relevant context.
        """
        # Extract text from ChatMessage if needed
        text = message.content if hasattr(message, 'content') else message

        # Search for similar past interactions
        similar = self.embedding_service.search_similar(text)

        # Format context for prompt
        return {
            "specifications": self.current_context["spring_specs"],
            "recent_topics": self.current_context["recent_topics"],
            "similar_exchanges": [
                {
                    "text": doc["text"],
                    "metadata": doc["metadata"],
                    "similarity": score
                }
                for score, doc in similar
            ]
        }

    def _extract_specifications(self, text):
        """Extract spring specifications from text."""
        # Use existing text parser patterns
        from utils.text_parser import extract_spring_specs
        return extract_spring_specs(text) or {}

    def _extract_topics(self, text):
        """Extract main topics from text."""
        # Simple keyword extraction for now
        # Could be enhanced with embeddings clustering
        keywords = []
        for pattern in PARAMETER_PATTERNS.values():
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                keywords.extend(matches)
        return keywords[:5]  # Limit to top 5
```

4. Update `TogetherAPIClient` in `together_api_client.py`

Add this method to the existing `TogetherAPIClient` class:

```
def generate_embedding(self, text, model="togethercomputer/bge-base-en-v1.5"):
    """Generate an embedding using Together.ai API.

    Args:
        text: Text to generate embedding for.
        model: Model to use for embedding.

    Returns:
        Embedding vector or None if request failed.
    """
    try:
        # Prepare request
        url = "https://api.together.xyz/v1/embeddings"
        payload = {
            "model": model,
            "input": text
        }

        # Make request
        response = self.session.post(
            url,
            headers=self.get_headers(),
            json=payload,
            timeout=30
        )
        response.raise_for_status()

        # Parse response
        data = response.json()

        # Extract embedding vector
        embedding = data.get("data", [{}])[0].get("embedding", [])

        return embedding
    except Exception as e:
        logging.error(f"Error generating embedding: {str(e)}")
        return None
```

5. Update `ChatService` Integration

Add this to `ChatService.__init__`:

```
# Set up context and embedding services if available
self.context_manager = None
self.embedding_service = None
```

Then add these methods:

```
def set_embedding_service(self, embedding_service):
    """Set the embedding service for enhanced context.

    Args:
        embedding_service: EmbeddingService instance.
    """
    self.embedding_service = embedding_service

    # Create context manager if not already present
    if not self.context_manager and self.embedding_service:
        from services.context_manager import ContextManager
        self.context_manager = ContextManager(
            embedding_service=self.embedding_service,
            chat_service=self
        )

def add_message_with_context(self, role, content):
    """Add a message with context processing.

    Args:
        role: Message role.
        content: Message content.

    Returns:
        Added message.
    """
    # Add the message
    message = self.add_message(role, content)

    # Process for context if context manager is available
    if self.context_manager:
        self.context_manager.process_message(message)

    return message

def get_relevant_context(self, message_text):
    """Get context relevant to a message.

    Args:
        message_text: Message text.

    Returns:
        Dictionary with relevant context or None.
    """
    if not self.context_manager:
        return None

    return self.context_manager.get_relevant_context(message_text)
```

Storage Strategy

The vector storage will be persisted in two ways:

1. **Primary Storage**: Native Python pickle format in `appdata/vector_store.dat`
2. **Inspection Format**: JSON version in `medium/vector_store.json` (via the existing conversion utilities)

The system will NOT require external vector databases like FAISS or ChromaDB for this implementation. Instead, it uses a simple in-memory vector store with serialization to maintain simplicity while providing the core functionality.

Log Integration

The implementation will leverage the existing `app.log` file for:

1. Tracking embedding operations for debugging
2. Monitoring context updates and extraction
3. Performance metrics of vector search operations

Example log format for embedding operations:

```
2025-03-29 10:15:32,123 [INFO] Generated embedding for text (len=82) using model
togethercomputer/bge-base-en-v1.5
2025-03-29 10:15:33,456 [INFO] Indexed conversation message with ID 123e4567-e89b-
12d3-a456-426614174000
2025-03-29 10:15:45,789 [INFO] Vector search found 3 relevant documents with
scores [0.92, 0.87, 0.65]
```

Implementation Phases

Phase 1: Core Infrastructure (1-2 days)

1. Implement `TogetherAPIClient.generate_embedding()`
2. Create `VectorStorage` class
3. Implement `EmbeddingService`
4. Add unit tests for core components

Phase 2: Context Management (2-3 days)

1. Implement `ContextManager`
2. Integrate with `ChatService`
3. Update prompt templates to use context
4. Add relevant context to UI debug panel

Phase 3: Application Integration (1-2 days)

1. Update `SequenceGenerator` to use context
2. Integrate with main application flow
3. Add context visualization to UI (optional)
4. Enhance specification extraction with embedding similarity

Phase 4: Testing & Refinement (1-2 days)

1. Performance testing and optimization
2. Edge case handling
3. User experience refinements
4. Documentation updates

Integration Guide for Existing Applications

If you're adding embeddings to an existing application, follow these steps to minimize disruption:

1. Add the New Files

First, add the new files without modifying existing ones:

```
# Create the required files
touch utils/vector_storage.py
touch services/embedding_service.py
touch services/context_manager.py
touch test_embedding.py
```

2. Add the Required Imports

Update your import requirements to include any new packages:

```
# In requirements.txt
# Add any new dependencies, but there are none for our implementation
```

3. Modify TogetherAPIClient

Add the embedding functionality to the existing API client:

```
# In utils/together_api_client.py
def generate_embedding(self, text, model="togethercomputer/bge-base-en-v1.5"):
    """Generate an embedding using Together.ai API."""
    # Add implementation as described above
```

4. Update Main Initialization

Update your application entry point to initialize the embedding system:

```
# In main.py
# After initializing other services
if api_key:
    try:
```

```
        # Initialize embedding system
        vector_storage = VectorStorage()
        embedding_service = EmbeddingService(api_client, vector_storage)
        chat_service.set_embedding_service(embedding_service)
    except Exception as e:
        logging.error(f"Failed to initialize embedding system: {str(e)}")
```

5. Update ChatService

Update ChatService to work with the embedding system:

```
# In services/chat_service.py
def set_embedding_service(self, embedding_service):
    """Set the embedding service for context enhancement."""
    # Add implementation as described above
```

6. Update Prompt Generation

Modify prompt generation to include context:

```
# In TogetherAPIClientWorker.run method
# Add context to the prompt:
context_text = ""
if "context" in self.parameters:
    context_text = self.parameters["context"]

user_prompt = f"""
{parameter_text}

{test_type_text}

{context_text}

My message: {original_prompt}
...
"""
```

7. Update ChatPanel

Modify the ChatPanel to extract and include context:

```
# In on_send_clicked method
# Get relevant context
if hasattr(self.chat_service, 'get_relevant_context'):
    context = self.chat_service.get_relevant_context(message_text)
    if context:
        context_text = self.chat_service.format_context_for_prompt(context)
```

```
        if context_text:
            parameters["context"] = context_text
```

8. Testing

Test the embedding system with your test script:

```
python test_embedding.py --api-key YOUR_API_KEY
```

Benefits

1. **Enhanced Context Understanding**: The AI will maintain awareness of specifications even across multiple messages
2. **Consistent Conversation Flow**: Users can refer back to previous discussions naturally
3. **Improved Specification Extraction**: Catch more variations in how users express specifications
4. **Related Sequence Finding**: Similar past sequences can inform new generation
5. **Human Language Tolerance**: Better handling of synonyms, paraphrasing, and imprecise language

Technical Considerations

1. **Performance**: The in-memory vector store is suitable for hundreds to low thousands of documents without requiring external databases.
2. **Memory Usage**: Store only essential context, not complete conversation history.
3. **API Usage**: Cache embeddings to minimize API calls to Together.ai.
4. **Security**: No additional security concerns as data remains in the existing storage system.

Future Enhancements

1. Integration with more sophisticated vector stores (FAISS, ChromaDB) for larger datasets
2. Clustering of specifications and sequences for advanced pattern recognition
3. Active learning from user feedback to improve context relevance
4. Multi-modal context incorporation (drawings, charts, etc.)

Leveraging app.log for Context Mining

The Spring Test App maintains a detailed log file at `logs/app.log` which can be a valuable source of context information. Here's how we can leverage it:

1. Log Mining for Conversation History

The log file contains records of user-assistant interactions that can provide additional context:

```
def mine_logs_for_conversation_context():
    """Extract conversation patterns from app.log file."""
    log_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), "logs",
"app.log")
```

```
    if not os.path.exists(log_file):
        logging.warning(f"Log file not found at {log_file}")
        return {}

    conversation_records = []
    current_conversation = []

    try:
        with open(log_file, 'r', encoding='utf-8') as f:
            for line in f:
                # Look for message indicators
                if "Adding user message:" in line:
                    # Extract timestamp and message content
                    timestamp = line.split("[INFO]")[0].strip()
                    message_content = line.split("Adding user message:")
[1].strip()

                    # Add to current conversation
                    current_conversation.append({
                        "timestamp": timestamp,
                        "role": "user",
                        "content": message_content
                    })

                elif "Adding assistant message:" in line:
                    # Extract timestamp and message content
                    timestamp = line.split("[INFO]")[0].strip()
                    message_content = line.split("Adding assistant message:")
[1].strip()

                    # Add to current conversation
                    current_conversation.append({
                        "timestamp": timestamp,
                        "role": "assistant",
                        "content": message_content
                    })

                # Look for conversation boundaries
                elif "Starting new conversation" in line or "Clearing chat
history" in line:
                    # Save previous conversation if not empty
                    if current_conversation:
                        conversation_records.append(current_conversation)
                        current_conversation = []

        # Add the last conversation if not empty
        if current_conversation:
            conversation_records.append(current_conversation)

        return conversation_records

    except Exception as e:
```

```
        logging.error(f"Error mining logs for context: {str(e)}")
        return []
```

2. Extracting Specification Patterns

The log file can reveal common specification patterns that we can use to enhance our extraction:

```
def extract_specification_patterns_from_logs():
    """Identify common specification patterns from log entries."""
    log_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), "logs",
"app.log")

    if not os.path.exists(log_file):
        return {}

    # Common specification patterns to look for
    pattern_indicators = {
        "free_length": ["free length", "free-length", "length when uncompressed"],
        "wire_diameter": ["wire diameter", "wire thickness", "wire gauge"],
        "outer_diameter": ["outer diameter", "OD", "outside diameter"],
        "inner_diameter": ["inner diameter", "ID", "inside diameter"],
        "spring_rate": ["spring rate", "k value", "stiffness"],
        "material": ["material", "made of", "composed of"]
    }

    # Track pattern frequency
    pattern_frequency = {key: 0 for key in pattern_indicators}
    matched_examples = {key: [] for key in pattern_indicators}

    try:
        with open(log_file, 'r', encoding='utf-8') as f:
            for line in f:
                if "user message:" in line:
                    message = line.split("user message:")[1].strip().lower()

                    # Check for each pattern
                    for pattern_key, indicators in pattern_indicators.items():
                        for indicator in indicators:
                            if indicator.lower() in message:
                                pattern_frequency[pattern_key] += 1

                                # Extract the context around the indicator (±50
chars)
                                start_idx = max(0, message.find(indicator) - 50)
                                end_idx = min(len(message),
message.find(indicator) + len(indicator) + 50)
                                context = message[start_idx:end_idx]

                                # Add to examples if not too many already
                                if len(matched_examples[pattern_key]) < 5:
                                    matched_examples[pattern_key].append(context)
```

```
            return {
                "frequency": pattern_frequency,
                "examples": matched_examples
            }
        except Exception as e:
            logging.error(f"Error extracting specification patterns from logs:
{str(e)}")
            return {}
```

3. Creating an Initial Vector Store from Logs

We can bootstrap our vector store using historical log data:

```
def initialize_vector_store_from_logs(embedding_service):
    """Populate initial vector store with historical log data."""
    # Mine logs for conversations
    conversations = mine_logs_for_conversation_context()

    indexed_count = 0

    for conv_idx, conversation in enumerate(conversations):
        # Skip very short conversations (likely noise)
        if len(conversation) < 2:
            continue

        # Index each message in the conversation
        for msg in conversation:
            # Index with conversation metadata
            metadata = {
                "role": msg["role"],
                "timestamp": msg["timestamp"],
                "conversation_id": f"log-conv-{conv_idx}",
                "type": "historical_log"
            }

            doc_id = embedding_service.index_conversation(
                msg["content"],
                metadata=metadata
            )

            if doc_id:
                indexed_count += 1

    logging.info(f"Initialized vector store with {indexed_count} historical
messages from logs")
    return indexed_count
```

4. Log Analysis Dashboard

For advanced users, we could add a log analysis dashboard to the application:

```
class LogAnalysisDashboard(QWidget):
    """Dashboard for analyzing log data and embeddings."""

    def __init__(self, embedding_service):
        super().__init__()
        self.embedding_service = embedding_service
        self.init_ui()

    def init_ui(self):
        """Initialize the UI."""
        # Set up layout
        layout = QVBoxLayout(self)

        # Add statistics section
        stats_group = QGroupBox("Embedding Statistics")
        stats_layout = QFormLayout()

        # Get statistics
        stats = self.embedding_service.get_statistics()

        # Add labels
        stats_layout.addRow("Total Documents:",
QLabel(str(stats["document_count"])))
        stats_layout.addRow("Cache Size:", QLabel(f"
{stats['cache_size']}/{stats['cache_limit']}"))
        stats_layout.addRow("Cache Hit Rate:", QLabel(f"
{stats['cache_hit_rate']:.2%}"))

        stats_group.setLayout(stats_layout)
        layout.addWidget(stats_group)

        # Add log mining section
        mining_group = QGroupBox("Log Mining")
        mining_layout = QVBoxLayout()

        # Button to mine logs
        mine_logs_btn = QPushButton("Mine Logs for Context")
        mine_logs_btn.clicked.connect(self.on_mine_logs_clicked)
        mining_layout.addWidget(mine_logs_btn)

        mining_group.setLayout(mining_layout)
        layout.addWidget(mining_group)

        # Add search test section
        search_group = QGroupBox("Search Test")
        search_layout = QVBoxLayout()

        # Input for search query
        self.search_input = QLineEdit()
        self.search_input.setPlaceholderText("Enter search query...")
        search_layout.addWidget(self.search_input)
```

```
        # Button to search
        search_btn = QPushButton("Search Similar Content")
        search_btn.clicked.connect(self.on_search_clicked)
        search_layout.addWidget(search_btn)

        # Results display
        self.search_results = QTextEdit()
        self.search_results.setReadOnly(True)
        search_layout.addWidget(self.search_results)

        search_group.setLayout(search_layout)
        layout.addWidget(search_group)

    def on_mine_logs_clicked(self):
        """Handle mine logs button click."""
        try:
            # Initialize vector store from logs
            count = initialize_vector_store_from_logs(self.embedding_service)
            QMessageBox.information(self, "Log Mining Complete", f"Indexed {count}
messages from logs.")
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to mine logs: {str(e)}")

    def on_search_clicked(self):
        """Handle search button click."""
        query = self.search_input.text().strip()
        if not query:
            return

        try:
            # Search for similar content
            results = self.embedding_service.search_similar(query, limit=5)

            # Display results
            result_text = ""
            for i, (score, doc) in enumerate(results):
                result_text += f"Result {i+1} (Score: {score:.4f}):\n"
                result_text += f"Text: {doc['text'][:200]}...\n"
                result_text += f"Metadata: {doc['metadata']}\n\n"

            self.search_results.setText(result_text or "No results found.")
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Search failed: {str(e)}")
```

Benefits of Log Integration

1. **Historical Context**: Leverage past conversations without starting from scratch
2. **Pattern Discovery**: Identify common specification patterns that might not be in our predefined list
3. **User Behavior Insights**: Understand how users naturally express specifications
4. **Continuous Improvement**: Use real-world data to refine extraction patterns
5. **Debugging Aid**: Trace context-related issues through detailed logging

6. **Privacy Control**: Process logs locally without sending additional data to external services