

Episodic Documentation

Conversational DAG-based Memory Agent

Generated on: 2025-06-29 18:51:28

Table of Contents

Main Documentation

Episodic - Main README

Usage Guide

Claude Integration Guide

Installation and Setup

Installation Guide

CLI Reference

LLM Providers

Core Features

Advanced Usage

Visualization

Project Structure

Development

Development Guide

TODO List

Implementation Plan

Technical Documentation

Compression and Topic Fixes

Topic Detection Fixes

Topic Detection Implementation

Streaming Implementation

Streaming Fix Summary

Prompt Caching

Async Compression Design

Conversational Drift Design

Integration Guides

LangChain Integration Guide

Hybrid Topic Detection Design

Testing Documentation

Tests README

Organized Tests Guide

Scripts README

Topic Testing README

...

Episodic - Main README

Source: README.md

Episodic: A Conversational DAG-Based Memory Agent

License MIT

Overview

Episodic is a persistent, navigable memory system for interacting with generative language models. It treats conversation as a directed acyclic graph (DAG), where each node represents a discrete conversational step (query and response).

Key Features

- Persistent storage of conversations using SQLite
- Simplified CLI with a talk-first interface and command access via "/" prefix
- Automatic topic detection and management
- Background compression of conversation segments
- Integration with various LLM providers (OpenAI, Anthropic, Ollama, etc.)
- Interactive visualization of conversation graphs
- Short, human-readable node IDs for easy reference
- Session cost tracking and performance benchmarking
- Script execution and saving for automation
- Configurable settings with live updates

Quick Start

```
# Installation
git clone https://github.com/yourusername/episodic.git
cd episodic
python -m venv .venv
source .venv/bin/activate
pip install -e .

# Set up your API key (for LLM integration)
export OPENAI_API_KEY=your_api_key_here

# Start the application
python -m episodic

# In the talk mode:
> /init # Initialize the database
```

Example Usage

Basic Conversation Flow

```
# Start the application
python -m episodic
```

```
# In the talk mode (notice the / prefix for commands):
```

```
> Hello, I'd like to explore quantum computing
```

```
 openai/gpt-3.5-turbo:
```

I'd be happy to explore quantum computing with you! It's a fascinating field that

1. Basic principles of quantum mechanics relevant to computing
2. Quantum bits (qubits) and how they differ from classical bits
3. Quantum gates and circuits
4. Quantum algorithms like Shor's or Grover's
5. Current state of quantum hardware
6. Potential applications and impact
7. Challenges in quantum computing

Let me know what interests you most!

```
> What is quantum superposition?
```

```
 openai/gpt-3.5-turbo:
```

Quantum superposition is a fundamental principle of quantum mechanics...

```
# Branch the conversation to explore a different topic
```

```
> /head 01
```

Current node changed to: 01

```
> Tell me about quantum entanglement instead
```

```
 openai/gpt-3.5-turbo:
```

Quantum entanglement is a fascinating phenomenon in quantum physics...

```
# Visualize the conversation graph
```

```
> /visualize
```

Visualization Demo

```
# In the talk mode, start the visualization server
> /visualize

# The visualization will open in your browser
# Continue the conversation in the same terminal
> Tell me more about quantum computing
```

Documentation

For more detailed information, see:

- [Installation Guide](#)
- [CLI Reference](#)
- [LLM Providers](#)
- [Visualization Guide](#)
- [Advanced Usage](#)
- [Development & Testing](#)

License

This project is licensed under the [MIT License](#).

Usage Guide

Source: USAGE_GUIDE.md

Episodic Usage Guide

A comprehensive guide to using Episodic, the conversational DAG-based memory agent.

Table of Contents

1. [Getting Started](#)
2. [Basic Conversation Flow](#)
3. [Navigation and History](#)
4. [Topic Detection and Management](#)
5. [Compression System](#)
6. [Model Configuration](#)
7. [Advanced Features](#)
8. [Scripting and Automation](#)
9. [Configuration Options](#)
10. [Tips and Best Practices](#)

Getting Started

Installation

```
# Install in development mode
pip install -e .

# Install required dependencies
pip install typer litellm
```

First Run

```
# Start the CLI
python -m episodic

# Initialize the database (first time only)
> /init

# Start chatting
> Hello, how are you today?
```

Basic Conversation Flow

Talk Mode

When you start Episodic, you're in "talk mode" - simply type messages to chat with the LLM:

```
> Tell me about quantum computing
🤖 Quantum computing is a revolutionary approach...

> How do qubits work?
🤖 Qubits, or quantum bits, are the fundamental units...
```

Commands

Commands start with `/`. Type `/help` to see all available commands:

- `/help` - Show available commands
- `/exit` or `/quit` - Exit the application
- `/last [N]` - Show recent messages
- `/topics` - Show conversation topics
- `/compress` - Compress conversation branches

Navigation and History

Viewing Conversation History

```
# Show last 5 messages (default)
> /last

# Show last 10 messages
> /last 10

# Show specific node details
> /show 0a

# View current position
> /head
```

Navigating the Conversation DAG

Episodic stores conversations as a directed acyclic graph (DAG), allowing branching:

```
# Change to a different node
> /head 0f

# See the ancestry of current position
> /ancestry


# Visualize the entire conversation graph
> /visualize
```

Topic Detection and Management

Automatic Topic Detection

Episodic automatically detects when conversation topics change:

```
> Tell me about dogs
```

```
 Dogs are domesticated mammals...
```

```
> What's the weather like?
```

```
 Topic change detected (high confidence)
```

```
 I don't have access to current weather...
```

Viewing Topics

```
# Show recent topics
```

```
> /topics
```

```
# Show last 20 topics
```

```
> /topics 20
```

```
# Show all topics ever created
```

```
> /topics --all
```

Topic Settings

```
# Enable topic detection
```

```
> /set topics true
```

```
# Set minimum messages before allowing topic change
```

```
> /set min_messages_before_topic_change 8
```

```
# Enable auto-compression of topics
```

```
> /set auto_compress_topics true
```

Compression System

Manual Compression

Compress conversation branches to save space and create summaries:

```
# Compress from root to current node
> /compress

# Compress specific branch
> /compress 0a

# Dry run to see what would be compressed
> /compress --dry-run

# Use different compression strategy
> /compress --strategy detailed
```

Automatic Compression

When topic changes are detected, previous topics can be automatically compressed:

```
# Enable auto-compression
> /set auto_compress_topics true

# Set minimum nodes for compression
> /set compression_min_nodes 5

# Choose compression model
> /set compression_model ollama/llama3

# View compression queue
> /compression-queue

# View compression statistics
> /compression-stats
```

Compression Strategies

- `simple` - Basic summary (default)
- `detailed` - More comprehensive summary
- `bullets` - Bullet-point format
- `structured` - Structured summary with sections
- `auto-topic` - Used by automatic compression

Model Configuration

Viewing Available Models

```
# Show current model
> /model

# List all available models
> /model
1. gpt-4
2. gpt-3.5-turbo
3. claude-3-opus
4. ollama/llama3
...
```

Changing Models

```
# Select by number
> /model
Select a model: 3

# Or specify directly
> /model gpt-4
```

Model Verification

```
# Test current model
> /verify
```

Advanced Features

Semantic Drift Detection

Track how conversations drift between topics:

```
# Enable drift display
> /set drift true

# Set semantic depth for context
> /set semdepth 5
```

System Prompts

Customize the LLM's behavior with different prompts:

```
# List available prompts
> /prompts

# Use a specific prompt
> /prompts use creative

# Show current prompt
> /prompts show

# Create custom prompts
> /prompts create my-prompt
```

Context Management

Control how much conversation history is included:

```
# Set context depth (number of messages)
> /set depth 10

# Enable prompt caching for performance
> /set cache true
```

Scripting and Automation

Running Scripts

Create text files with commands and messages to automate conversations:

```
# Run a script file  
> /script tests/example-conversation.txt
```

Script Format

```
# example-script.txt  
# Comments start with #  
  
# Set configuration  
/set debug true  
/set topics true  
  
# Have a conversation  
Tell me about Python  
What are decorators?  
How do I handle exceptions?  
  
# Change topic  
Now let's talk about cooking  
What's a good pasta recipe?  
  
# Check results  
/topics  
/compression-stats
```


Configuration Options

Display Settings

```
# Show costs for each message
> /set cost true

# Enable debug mode
> /set debug true

# Control text wrapping
> /set wrap true

# Set color mode (auto/on/off)
> /set color auto
```

Compression Settings

```
# Show compression notifications
> /set show_compression_notifications true

# Set compression model
> /set compression_model gpt-3.5-turbo

# Minimum nodes before compression
> /set compression_min_nodes 5
```

Topic Detection Settings

```
# Enable topic detection
> /set topics true

# Minimum messages before topic change
> /set min_messages_before_topic_change 8
```

View All Settings

```
# See current configuration  
> /set
```

Tips and Best Practices

1. Use Clear Topic Transitions

When changing subjects, be explicit: - ❌ "What about cars?" - ✅ "Let's switch topics to cars - what are electric vehicles?"

2. Leverage Branching

Create alternate conversation paths:

```
# Save current position  
> /head  
  
# Explore different response  
> /head 0c  
> What if we approached this differently?  
  
# Return to saved position  
> /head 0f
```

3. Compress Regularly

Keep your database manageable: - Compress completed topics - Use auto-compression for long conversations - Review compression stats periodically

4. Customize Prompts

Create prompts for different use cases: - Technical discussions - Creative writing - Code reviews - Learning sessions

5. Use Scripts for Testing

Create reproducible conversations: - Test topic detection - Benchmark different models - Demo specific features

6. Monitor Costs

Track token usage and costs:

```
> /set cost true
> /cost # View session costs
```

7. Optimize Performance

- Use `/set cache true` for faster responses
- Choose appropriate context depth
- Use local models (Ollama) for privacy/speed

Troubleshooting

Common Issues

Database locked: - Ensure only one instance is running - Check file permissions

Model errors: - Verify API keys are set - Check model availability with `/model` - Test with `/verify`

Topic detection not working: - Ensure `/set topics true` - Check minimum message threshold - Try explicit topic changes

Compression failing: - Verify sufficient nodes in topic - Check compression model availability - Review debug output

Debug Mode

Enable debug mode for detailed information:

```
> /set debug true
```

This shows: - Topic detection process - Compression attempts - Model queries - Configuration changes

Environment Variables

- `EPISODIC_DB_PATH` - Custom database location
- `OPENAI_API_KEY` - OpenAI API access
- `ANTHROPIC_API_KEY` - Anthropic API access
- Standard LiteLLM environment variables

Support and Documentation

- **GitHub Issues:** Report bugs and request features
- **CLAUDE.md:** Development guidelines
- **docs/structure.md:** Codebase architecture
- **README.md:** Quick start guide

Claude Integration Guide

Source: CLAUDE.md

CLAUDE.md

This file provides guidance to Claude Code (claude.ai/code) when working with code in this repository.

Project Overview

Episodic is a conversational DAG-based memory agent that creates persistent, navigable conversations with language models. It stores conversation history as a directed acyclic graph where each node represents a conversational exchange.

Current Session Context

Last Working Session (2025-06-29)

- Fixed /rename-topics command to handle ongoing topics (topics with NULL endnodeid)
- Fixed finalizecurrenttopic() to properly rename ongoing topics when conversation ends
- Root cause: Both functions were trying to get_ancestry(NULL) which returns empty
- Solution: Check if endnodeid is NULL and use get_head() instead
- Fixed bold formatting for numbered lists in streaming output
- Now bolds only the first line of each numbered item (e.g., "**1. Life Support Systems: description here**")
- Continuation lines under the same item are not bolded
- Fixed Google Gemini model configuration to use "gemini/" prefix for Google AI Studio
- Added GOOGLEAPIKEY to provider API keys mapping
- Filter out unsupported parameters (presencepenalty, frequencypenalty) for Google Gemini models

Previous Session (2025-06-28)

- Fixed JSON parsing errors in topic detection for Ollama models
- Added robust fallback parsing for various response formats (Yes/No/JSON)
- Created simplified topicdetectionollama.md prompt for better compatibility
- Topic detection now handles malformed JSON responses gracefully
- Fixed critical `stop: ["\n"]` parameter causing GPT-3.5 to return truncated responses
- Created topicdetectionv3.md prompt for domain-agnostic detection
- Discovered GPT-3.5 is over-sensitive (6-7 topics) while Ollama is under-sensitive (1 topic)
- Created comprehensive test suite in scripts/topic/ for validating topic detection
- Verified Ollama topic detection IS working but being too conservative

Current Issue

Topic detection sensitivity varies drastically by model: - **GPT-3.5**: Creates too many topics (splits related concepts like "pasta recipes" vs "Italian pantry") - **Ollama**: Creates too few topics (keeps everything together, even explicit transitions) - **Target**: 3 topics for the standard test (Mars, Italian cooking, Neural networks)

Test Results Summary

Test	Expected	GPT-3.5	Ollama
Python progression	1	4 ✖	1 ✔
Explicit transitions	4	6 ✖	1 ✖
ML deep dive	1	4 ✖	1 ✔
Natural flow	3	4 ✖	1 ✖

Previous Session (2025-06-27)

- Created centralized LLM manager for accurate API call tracking

- Fixed initial topic extraction to require minimum 3 user messages
- Added /api-stats and /reset-api-stats commands
- Fixed benchmark system operation-specific counting (no longer shows cumulative)
- Fixed streaming response cost calculation (was showing \$0.00)
- Fixed topic detection to count user messages only, not total nodes
- Added validation to prevent premature topic creation
- Fixed multiple indentation errors in conversation.py

Previous Session (2025-06-25)

- Fixed streaming output duplication in constant-rate mode
- Improved word wrapping and list indentation
- Added markdown bold (**text**) support
- Cleaned up CLI code and removed unused imports
- Fixed test suite issues (cache tests, config initialization)
- Simplified testing approach - removed over-engineered test infrastructure
- Updated documentation to reflect simplified testing

Key System Understanding

Topic Detection Flow

1. User sends message → Topic detection runs (ollama/llama3 with JSON output)
2. If topic change detected → Close previous topic at last assistant response
3. Previous topic's content is analyzed to extract appropriate name
4. New topic starts as "ongoing-TIMESTAMP" placeholder
5. After 2+ user messages, topic is automatically renamed based on content
6. Topics remain "open" (endnodeid=NULL) until closed on topic change

Database Functions

- `store_topic()` - Creates new topic entry (endnodeid now optional)
- `update_topic_end_node()` - Closes topic by setting end boundary
- `update_topic_name()` - Renames topic

- `get_recent_topics()` - Retrieves topic list
- `migrate_topics_nullable_end()` - Migration to allow NULL endnodeid

Important Code Locations

- Topic detection: `episodic/topics.py:detect_topic_change_separately()`
- Topic threshold behavior: `episodic/topics.py:_should_check_for_topic_change()` (lines 75-89)
- Topic user message counting: `episodic/topics.py:count_user_messages_in_topic()` (NEW)
- Topic naming: `episodic/conversation.py:387-442` (in `handlechatmessage`)
- Topic creation validation: `episodic/conversation.py:876-903` (NEW validation logic)
- LLM Manager: `episodic/llm_manager.py` (centralized API call tracking)
- Compression storage: `episodic/db_compression.py` (new compression mapping system)
- Summary command: `episodic/commands/summary.py`
- Command parsing: `episodic/cli.py:handle_command()`

Configuration Options

- `topic_detection_model` - Default: ollama/llama3
- `running_topic_guess` - Default: True (not yet implemented)
- `min_messages_before_topic_change` - Default: 8
- `show_topics` - Shows topic evolution in responses
- `debug` - Shows detailed topic detection info
- `main_params` - Model parameters for main conversation
- `topic_params` - Model parameters for topic detection (e.g., temperature=0)
- `compression_params` - Model parameters for compression
- Model params support: temperature, maxtokens, topp, presencepenalty, frequencypenalty

Recent Discoveries

- **IMPORTANT:** All conversations are currently completely linear - the DAG is a straight line that is never modified. There is no branching implemented yet.
- **CRITICAL:** Topic detection has undocumented threshold behavior - first 2 topics use half threshold (4 messages), then full threshold (8 messages) applies
- **FIXED:** Topics now properly include all their messages (was missing messages due to premature endnodeid setting)
- **FIXED:** Topic detection now uses JSON output format for consistency
- **FIXED:** Topics automatically rename from "ongoing-XXXX" after 2 user messages
- Compression system stores summaries separately in compressionsv2 and compressionnodes tables
- `/init --erase` properly resets conversation manager state (currentnodeid, current_topic, session costs)
- ConversationManager tracks current topic with `set_current_topic()` and `get_current_topic()`
- Topics must remain "open" (endnodeid=NULL) until explicitly closed
- `get_ancestry()` returns nodes from oldest to newest (root to current)
- Topic extraction looks at beginning of conversation for better topic names
- Model parameters can be configured per context (main, topic, compression)

Test Scripts

- `scripts/test-complex-topics.txt` - 21 queries across multiple topics
- `scripts/test-topic-naming.txt` - Simple topic transitions
- `scripts/test-final-topic.txt` - Tests final topic handling
- `scripts/three-topics-test.txt` - Tests three topic changes accounting for threshold behavior

New Commands

- `/rename-topics` - Renames all placeholder "ongoing-*" topics by analyzing their content
- `/api-stats` - Shows actual LLM API call statistics
- `/reset-api-stats` - Resets API call counter

- `/compress <topic-name>` - Manually trigger compression for a specific topic
- `/model-params` or `/mp` - Show/set model parameters for different contexts

Common Development Commands

Installation & Setup

```
# Install in development mode
pip install -e .

# Install required dependencies
pip install typer # Required for CLI functionality
```

Running the Application

```
# Start the main CLI interface
python -m episodic

# Within the CLI, initialize database
> /init

# Start visualization server
> /visualize
```

Testing

```
# Run all unit tests
python -m unittest discover episodic "test_*.py"

# Run specific test modules
python -m unittest episodic.test_core
python -m unittest episodic.test_db
python -m unittest episodic.test_integration

# Run interactive/manual tests
python -m episodic.test_interactive_features

# Test coverage
pip install coverage
coverage run -m unittest discover episodic "test_*.py"
coverage report -m
```

Architecture

Core Components

- **Node/ConversationDAG** (`core.py`): Core data structures representing conversation nodes and the DAG
- **Database Layer** (`db.py`): SQLite-based persistence with thread-safe connection handling
- **LLM Integration** (`llm.py`): Multi-provider LLM interface using LiteLLM with context caching
- **CLI Interface** (`cli.py`): Typer-based command-line interface with talk-first design
- **Visualization** (`visualization.py`): NetworkX and Plotly-based graph visualization with real-time updates
- **Configuration** (`config.py`): Application configuration management

Key Design Patterns

- **Thread-safe database operations**: Uses thread-local connections and context managers

- **Provider-agnostic LLM calls:** Abstracts different LLM providers (OpenAI, Anthropic, Ollama, etc.) through LiteLLM
- **Short node IDs:** Human-readable 2-character IDs for easy navigation
- **Linear conversation structure:** Currently all conversations are completely linear - the DAG is a straight line that is never modified (no branching implemented yet)
- **Real-time visualization:** HTTP polling for live graph updates

Database Schema

- Nodes table with id, shortid, message, timestamp, parentid, modelname, systemprompt, response
- SQLite with full-text search capabilities
- Configurable database path via EPISODICDBPATH environment variable

LLM Integration Details

- Prompt caching enabled by default for performance (using LiteLLM prompt caching)
- Cost tracking for token usage with cache discount calculations
- Multiple provider support via LiteLLM
- Model selection via numbered list or direct specification
- Configurable context depth for conversation history

Development Notes

- Entry point is `episodic/__main__.py` which delegates to `cli.py`
- Tests include both automated unit tests and interactive manual tests
- HTTP polling-based real-time functionality verification
- Prompt management system with role-based prompts in `prompts/` directory
- Configuration stored in `episodic.db` alongside conversation data

...

Installation Guide

Source: docs/Installation.md

Installation Guide

Basic Installation

```
# Clone the repo and navigate to the directory
git clone https://github.com/yourusername/episodic.git
cd episodic
python -m venv .venv
source .venv/bin/activate
pip install -e .
```

This will install all required dependencies, including: - **prompt_toolkit**: For the interactive interface - **typer**: For the command-line interface - **litellm**: For LLM provider integration - Other dependencies for visualization and database functionality

Running Episodic

After installation, you can start Episodic with:

```
# If installed with pip
episodic

# Or using the Python module syntax
python -m episodic
```

This will start Episodic in talk mode, where you can chat with the LLM and use commands with the "/" prefix:

```
> /help           # Show available commands
> /init           # Initialize the database
> Hello, world!   # Chat with the LLM
```

The talk mode is now the main interface for Episodic, providing a seamless experience for both conversation and command execution.

LLM Integration Setup

To use LLM features with OpenAI (the default provider), you need to set your API key as an environment variable:

```
export OPENAI_API_KEY=your_api_key_here
```

For other providers, see the [LLM Providers](#) documentation.

Installation for LiteLLM Support

If you've previously installed Episodic and are getting a "No module named 'litellm'" error, you need to reinstall the package to include the new dependencies:

```
pip install -e .
```

For Ollama support, install with:

```
pip install "litellm[ollama]"
```

Migrating Existing Databases

If you have an existing database created with a previous version of Episodic, you can migrate it to use short IDs by running the following Python code:

```
from episodic.db import migrate_to_short_ids

# Migrate existing nodes to use short IDs
count = migrate_to_short_ids()
print(f"Migrated {count} nodes to use short IDs")
```


This will add short IDs to all existing nodes in your database.

CLI Reference

Source: docs/CLIReference.md

CLI Reference

This document provides a comprehensive reference for all Episodic CLI commands.

Episodic now uses a simplified CLI structure where the talk loop is the main interface, and commands are accessed by prefixing them with a "/" character.

Starting the Application

```
python -m episodic
```

This starts the application in talk mode, where you can chat with the LLM and use commands.

Basic Commands

All commands are prefixed with a "/" character in the talk mode.

Navigation Commands

Initialize the Database

```
/init  
/init --erase # Erase existing database and reset state
```

Creates a new database or resets an existing one.

Add a Message

```
/add Your message here  
/add --parent 01 Your message here # Add as child of specific node
```

Adds a new node with the specified content as a child of the current node.

Show a Node

```
/show <node_id>
```

Displays the content and metadata of the specified node.

Print Current Node

```
/print  
/print <node_id>  # Print specific node
```

Prints the content of the current node or specified node.

Change the Current Node

```
/head  
/head <node_id>
```

Shows or changes the current node to the specified node.

List Recent Nodes

```
/list  
/list --count 10  # Show 10 most recent nodes
```

Lists the most recent nodes in the database.

Show Ancestry

```
/ancestry <node_id>
```

Shows the ancestry (thread history) of the specified node.

Configuration Commands

Set Configuration

```
/set                # Show all settings
/set <parameter>    # Show specific setting
/set <parameter> <value> # Set a value

# Examples:
/set debug on        # Enable debug mode
/set cache off       # Disable caching
/set cost on         # Show cost information
/set show_topics true # Show topic evolution
/set compression_model ollama/llama3
```

Verify Configuration

```
/verify
```

Verifies that the current configuration is valid and all providers are working.

Cost Information

```
/cost
```

Shows the session cost information for LLM usage.

Topic Management

View Topics

```
/topics            # Show recent topics
/topics --all       # Show all topics
/topics --limit 20  # Show 20 topics
/topics --verbose   # Show detailed topic info
```

Shows conversation topics with their ranges and message counts.

Rename Topics

```
/rename-topics
```

Analyzes and renames all placeholder "ongoing-*" topics based on their content.

Compress Current Topic

```
/compress-current-topic
```

Compresses the current topic (if closed) into a summary.

Compression Commands

Manual Compression

```
/compress                # Compress from root to head  
/compress --node <node_id> # Compress from root to specific node  
/compress --strategy simple # Use simple compression  
/compress --dry-run        # Preview without compressing
```

Compression Statistics

```
/compression-stats
```

Shows statistics about compressed conversations.

Compression Queue

```
/compression-queue
```

Shows pending background compression jobs.

Other Commands

Summary

```
/summary          # Summarize last 5 messages  
/summary 10       # Summarize last 10 messages  
/summary all      # Summarize entire conversation
```

Generates a summary of the conversation.

Benchmark

```
/benchmark
```

Shows performance benchmarks for various operations.

Script Execution

```
/script <filename> # Execute commands from a script file  
/save <filename>   # Save current session commands to a script
```

Execute or save conversation scripts.

Help

```
/help
```

Shows available commands and their usage.

LLM Integration

In the talk mode, you can simply type your message without any command prefix to chat with the LLM. The system will automatically use the conversation history as context.

```
> What is quantum computing?
```

```
🤖 openai/gpt-3.5-turbo:
```

```
Quantum computing is a type of computing that uses quantum-mechanical phenomena.
```

Change the Model

```
/model  
/model gpt-4
```

Shows the current model or changes to a different model.

Visualization Commands

Generate Visualization

```
/visualize  
/visualize --output conversation.html  
/visualize --no-browser  
/visualize --port 5001
```

Generates and opens an interactive visualization of the conversation DAG.

Talk Mode Interface

The talk mode is now the main interface for Episodic. It provides a fluid user experience with command history, auto-suggestion, and a simple way to interact with the LLM.

```
# Start the application  
python -m episodic
```

In the talk mode: - Type a message without any prefix to chat with the LLM - Use the "/" prefix to access commands


```
> Hello, world!
🤖 openai/gpt-3.5-turbo:
Hello! How can I assist you today?

> /help
Available commands:
  /help          - Show this help message
  /exit, /quit   - Exit the application
  /init [--erase] - Initialize the database (--erase to erase existing)
  ...
```

The talk mode uses the [Typer](#) package for command handling and [prompt_toolkit](#) for the interactive interface.

Short Node IDs

Episodic uses short, human-readable IDs for nodes in addition to the traditional UUIDs. These short IDs:

- Are 2-3 characters long (alphanumeric, base-36 encoding)
- Are sequential, making it easy to understand the order of creation
- Can be used anywhere a node ID is required (/show, /ancestry, --parent references)
- Make it much easier to reference nodes in the command line

Example:

```
# Adding a node shows both the short ID and UUID
> /add Hello, world.
Added node 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)

# You can reference nodes using the short ID
> /show 01
Node ID: 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)
Parent: None
Message: Hello, world.

# Short IDs are also shown in ancestry
> /ancestry 01
01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9): Hello, world.
```

LLM Providers

Source: docs/LLMProviders.md

Integrating LLM Providers with LiteLLM

This guide explains how to integrate various LLM providers with LiteLLM in your project, including both cloud providers and local options like Ollama and LMStudio.

Ollama Integration

Ollama is directly supported by LiteLLM and is straightforward to integrate:

```
# In your llm_config.py
def get_default_config() -> Dict[str, Any]:
    return {
        "default_provider": "openai",
        "providers": {
            "openai": {
                "models": ["gpt-4o-mini", "gpt-4o", "gpt-3.5-turbo"]
            },
            "ollama": {
                "models": ["llama3", "mistral", "codellama", "phi3"]
            },
            # Other providers...
        }
    }
```

To use Ollama models: 1. Install Ollama from ollama.ai 2. Pull models you want to use:

```
ollama pull llama3 3. Install LiteLLM with Ollama support: pip install
litellm[ollama] 4. In your code, use the format: ollama/model_name
```

```
# Example usage
response = litellm.completion(
    model="ollama/llama3",
    messages=[{"role": "user", "content": "Hello, world!"}]
)
```

LMStudio Integration

LMStudio works by exposing an OpenAI-compatible API endpoint, so you'll integrate it as a custom endpoint:

1. In LMStudio:

- Load your model
- Start the local server (usually at `http://localhost:1234`)

2. Configure LiteLLM to use LMStudio:

```
# In your llm_config.py
def get_default_config() -> Dict[str, Any]:
    return {
        "default_provider": "openai",
        "providers": {
            # Other providers...
            "lmstudio": {
                "api_base": "http://localhost:1234/v1", # Default LMStudio endpoint
                "models": ["local-model"] # You can name these whatever makes sense
            }
        }
    }
```

1. In your `get_model_string` function, add special handling for LMStudio:

```
def get_model_string(model_name: str) -> str:
    """Convert model name to LiteLLM format based on provider."""
    provider = get_current_provider()

    # Handle different provider types
    if provider == "lmstudio":
        # For LMStudio, we need to set the API base in the call
        return model_name # The actual model name doesn't matter for LMStudio
```

1. When making calls to LMStudio, set the API base:

```
# In your query_llm function
if provider == "lmstudio":
    provider_config = get_provider_config("lmstudio")
    response = litellm.completion(
        model=full_model,
        messages=messages,
        api_base=provider_config.get("api_base"),
        temperature=temperature,
        max_tokens=max_tokens
    )
```

Groq Integration

Groq is a cloud provider known for its fast inference speeds. It's directly supported by LiteLLM:

```
# In your llm_config.py
def get_default_config() -> Dict[str, Any]:
    return {
        "default_provider": "openai",
        "providers": {
            # Other providers...
            "groq": {
                "models": ["llama3-8b-8192", "llama3-70b-8192", "mixtral-8x7b-32"]
            },
        }
    }
```

To use Groq: 1. Sign up for an account at groq.com 2. Get your API key from the Groq console 3. Set the environment variable: `export GROQ_API_KEY=your_groq_api_key_here` 4. Use the models with the format: `groq/model_name`

```
# Example usage
response = litellm.completion(
    model="groq/llama3-8b-8192",
    messages=[{"role": "user", "content": "Hello, world!"}]
)
```

Groq is known for its extremely fast inference speeds, making it a good choice for applications where response time is critical.

Switching Between Providers in Talk Mode

You can easily switch between providers using the `/model` command in talk mode:

```
# List available models
> /model
Current model: gpt-3.5-turbo (Provider: openai)

Available models from openai:
- gpt-4o-mini
- gpt-4o
- gpt-3.5-turbo

# Switch to a specific model
> /model gpt-4o
Switched to model: gpt-4o (Provider: openai)

# Switch to a local provider like Ollama
> /model llama3
Switched to model: llama3 (Provider: ollama)
```

The implementation of this feature in the code looks like:

```
# In your handle_llm_providers method
elif subcommand == "local" and len(args) > 1:
    local_provider = args[1]
    if local_provider in ["ollama", "lmstudio"]:
        try:
            set_current_provider(local_provider)
            print(f"Switched to local provider: {local_provider}")
        except ValueError as e:
            print(f"Error: {str(e)}")
    else:
        print(f"Unknown local provider: {local_provider}")
        print("Available local providers: ollama, lmstudio")
```

Installation Requirements

```
# For Ollama support
pip install litellm[ollama]

# For LMStudio, no special package needed as it uses OpenAI-compatible API

# For cloud providers (OpenAI, Anthropic, Groq, etc.)
# No special packages needed beyond the base litellm installation
```

This setup gives you flexibility to switch between cloud providers (OpenAI, Anthropic, Groq) and your local providers (Ollama and LMStudio) while maintaining a consistent interface throughout your application.

...

Advanced Usage

Source: docs/AdvancedUsage.md

Advanced Usage

This document covers advanced features and usage patterns for Episodic.

Talk Mode Interface

Episodic now uses a simplified CLI structure where the talk loop is the main interface, and commands are accessed by prefixing them with a "/" character.

Topics and Automatic Organization

Episodic automatically detects topic changes in your conversations and organizes them:

```
# View your conversation topics
> /topics
📁 Conversation Topics (5 recent)
=====

[1] ✓ quantum-computing
    Created: 2025-06-26 10:15
    Range: 02 → 0f (14 messages)
    Confidence: high

[2] ✓ machine-learning
    Created: 2025-06-26 10:20
    Range: 0g → 0p (10 messages)

# View all topics
> /topics --all

# Rename placeholder topics
> /rename-topics
🔄 Analyzing ongoing topics...
✅ Renamed 'ongoing-1750929426' to 'space-exploration'
```

Topics are automatically compressed in the background when they close, reducing storage while preserving key information.

Compression System

Episodic includes an intelligent compression system that runs in the background:

```
# View compression statistics
> /compression-stats
📊 Compression Statistics
```

```
Total compressions: 5
Words saved: 2,500
Average reduction: 75.3%
```

```
# Manually compress conversation
> /compress --dry-run
Would compress 25 nodes (3,000 words → ~750 words)
```

```
# View pending compressions
> /compression-queue
📋 Pending Compressions (2 jobs)
```

```
1. Topic: quantum-physics (priority: 5)
2. Topic: web-development (priority: 7)
```

Configuration Management

Episodic offers extensive configuration options:

```
# View all settings
> /set
Current settings:
  debug: false
  cache: true
  cost: false
  show_topics: true
  compression_model: ollama/llama3
  topic_detection_model: ollama/llama3
  auto_compress_topics: true

# Change settings
> /set debug on
Debug mode enabled

> /set show_topics false
Topic display disabled

# Verify configuration
> /verify
✔ Database connection: OK
✔ LLM provider (openai): OK
✔ Model availability: OK
```

Session Management and Scripts

Save and replay conversation sessions:

```
# Save current session
> /save my-research-session
✅ Saved 15 commands to: scripts/my-research-session.txt

# Execute a script
> /script my-research-session.txt
📄 Executing script: my-research-session.txt
```

```
[1] > What is quantum entanglement?
...
✅ Script execution completed
```

Performance Monitoring

Track performance with the benchmark system:

```
# Enable benchmarking
> /set benchmark on
Benchmarking enabled

# View benchmarks
> /benchmark
📊 Performance Benchmarks
```

Message Processing:

Average: 245ms

Min: 120ms

Max: 580ms

Database Operations:

Insert: 5ms avg

Query: 3ms avg

```
# Start the application
python -m episodic
```

The talk mode interface provides several advantages:

- **Persistent State:** Maintains context between commands, including the current node
- **Command History:** Remembers commands between sessions
- **Auto-suggestion:** Suggests commands and messages as you type
- **Syntax Highlighting:** Makes commands and responses more readable
- **Help System:** Built-in documentation for all commands
- **Seamless Conversation:** Chat with the LLM without any special commands
- **Easy Command Access:** Access commands with the "/" prefix

Example Usage in Talk Mode

The following examples show what you'll see in the talk mode interface:

```
> /init
```

```
Database initialized with a default root node (ID: 01, UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)
```

```
> /add Hello, world.
```

```
Added node 02 (UUID: 4b8f57da-9c1f-4d2b-b0b1-9f5c4b8f57da)
```

```
> /show 02
```

```
Node ID: 02 (UUID: 4b8f57da-9c1f-4d2b-b0b1-9f5c4b8f57da)
```

```
Parent: 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)
```

```
Message: Hello, world.
```

```
> /head 01
```

```
Current node changed to: 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)
```

```
> What is the capital of France?
```

```
 openai/gpt-3.5-turbo:
```

```
The capital of France is Paris. It's one of the world's major global cities and.
```

```
> /list
```

```
Recent nodes (showing 5 of 5 requested):
```

```
04 (UUID: 6da179fc-be31-5f4d-d2d3-b17e6da179fc): The capital of France is Paris.
```

```
03 (UUID: 5c9068eb-ad20-4e3c-c1c2-a06d5c9068eb): What is the capital of France?
```

```
02 (UUID: 4b8f57da-9c1f-4d2b-b0b1-9f5c4b8f57da): Hello, world.
```

```
01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9):
```

```
> /help
```

```
Available commands:
```

/help	- Show this help message
/exit, /quit	- Exit the application
/init [--erase]	- Initialize the database (--erase to erase existing)
/add <content>	- Add a new node with the given content
/show <node_id>	- Show details of a specific node
/head [node_id]	- Show current node or change to specified node
/list [--count N]	- List recent nodes (default: 5)
/ancestry <node_id>	- Trace the ancestry of a node
/visualize	- Visualize the conversation DAG
/model	- Show or change the current model
/prompts	- Manage system prompts

```
Type a message without a leading / to chat with the LLM.
```

LLM Integration

Episodic integrates with various LLM providers through LiteLLM, allowing you to: - Query an LLM and store both the query and response in the conversation DAG - Chat with an LLM using conversation history as context - Switch between different LLM providers (OpenAI, Anthropic, Ollama, LMStudio, etc.)

Example LLM Usage

```
# Simply type your message to chat with the LLM
> What is the capital of France?
🤖 openai/gpt-3.5-turbo:
The capital of France is Paris. It's one of the world's major global cities and.

# Continue a conversation with context from previous messages
> Tell me more about its history.
🤖 openai/gpt-3.5-turbo:
Paris has a rich history dating back to ancient times. It was originally founded

# Specify a different model using the /model command
> /model gpt-4
Switched to model: gpt-4 (Provider: openai)

> Explain quantum computing.
🤖 openai/gpt-4:
Quantum computing is a type of computing that uses quantum-mechanical phenomena.

# Customize the system message using the /prompts command
> /prompts use coding_assistant
Now using prompt: coding_assistant - A helpful coding assistant

> How do I write a Python function?
🤖 openai/gpt-3.5-turbo:
In Python, you define a function using the `def` keyword followed by the function
```

For more details on LLM providers, see the [LLM Providers](#) documentation.

Conversation Branching and Navigation

One of Episodic's key features is the ability to branch conversations and navigate between different branches:

```
# Start a conversation
> /add Hello, world.
Added node 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)

# Add a child node
> /add This is branch A.
Added node 02 (UUID: 4b8f57da-9c1f-4d2b-b0b1-9f5c4b8f57da)

# Go back to the root node
> /head 01
Current node changed to: 01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9)

# Create a different branch
> /add This is branch B.
Added node 03 (UUID: 5c9068eb-ad20-4e3c-c1c2-a06d5c9068eb)

# View the ancestry of a node to see its thread history
> /ancestry 03
03 (UUID: 5c9068eb-ad20-4e3c-c1c2-a06d5c9068eb): This is branch B.
01 (UUID: 3a7e46c9-8b0e-4c1a-9f0a-8e5b3a7e46c9): Hello, world.
```

This branching capability allows you to explore different conversation paths and easily switch between them.

Visualization

Source: docs/Visualization.md

Visualization Guide

Episodic includes powerful visualization tools to explore the conversation DAG (Directed Acyclic Graph).

Browser-Based Visualization

The default visualization opens in your web browser:

```
# Generate and open an interactive visualization in your browser
> /visualize

# Save the visualization to a specific file
> /visualize --output conversation.html

# Generate the visualization without opening it in a browser
> /visualize --no-browser
```

Native Window Visualization

Episodic also supports displaying the visualization in a native window instead of a web browser:

```
# Open the visualization in a native window
> /visualize --native

# Customize the window size
> /visualize --native --width 1200 --height 900
```

The native window visualization provides the same features as the browser-based visualization but in a standalone application window, which can be more convenient for some users.

Visualization Features

Both visualization methods allow you to:

- See the entire conversation structure as a directed graph
- Hover over nodes to see the full content
- Zoom in/out and pan around the graph
- Move nodes to explore different layouts
- See the hierarchical structure of conversations
- Double-click on a node to make it the current node
- Right-click on a node to delete it and all its descendants

Using Visualization with Talk Mode

The visualization can be used directly from the talk mode interface:

```
# Start the application
python -m episodic

# In the talk mode, start the visualization
> /visualize
```

The visualization will open in your browser, and you can continue using the talk mode interface in the same terminal. This allows you to:

1. Chat with the LLM and see the conversation graph update in real-time
2. Use commands like `/head` to navigate to different nodes
3. See the effects of your actions immediately in the visualization
4. Interact with the visualization (e.g., double-click on nodes) and see the changes reflected in the talk mode

This integrated approach provides a seamless experience where you can both converse with the LLM and visually explore the conversation structure at the same time.

Multiple Visualization Windows

You can also open the visualization in a separate terminal if you prefer:

```
# In a separate terminal
python -m episodic

# Then immediately start the visualization
> /visualize
```

This allows you to have multiple visualization windows open at the same time, each showing the same conversation graph but potentially focused on different parts of it.

Technical Notes

- **Visualization Engine:** The visualization uses Plotly, a powerful and interactive data visualization library, providing a robust and feature-rich experience.
- **Native Window Support:** The native window visualization uses PyWebView to embed the web-based visualization in a standalone application window.
- **Layout Algorithm:** The visualization uses a custom hierarchical layout algorithm that doesn't require external dependencies, ensuring consistent visualization across different environments.
- **Real-time Updates:** The visualization supports real-time updates via HTTP polling:
 - Changes to the graph (setting current node, deleting nodes) are reflected across all browser windows
 - The visualization updates automatically without page reloads through periodic HTTP requests
 - Multiple users can view the same visualization and see changes updated periodically
 - Uses HTTP polling every 5 seconds for reliable updates
- **Interactive Features:** The visualization includes enhanced interactive features:
 - Double-click on a node to make it the current node
 - Right-click on a node to access a context menu for node deletion
 - Hover over nodes to see the full content
 - Pan and zoom to explore large conversation graphs
- **Robust Connectivity:** The visualization includes several features to ensure reliable operation:
 - HTTP polling for reliable periodic updates

- Visual notifications of successful updates
 - Detailed error handling and logging for troubleshooting
 - Fallback to page reload if polling fails
- **Server Port:** If you encounter an "Address already in use" error (common on macOS where AirPlay uses port 5000), you can specify a different port:

```
> /visualize --port 5001
```

Project Structure

Source: docs/structure.md

Episodic Codebase Structure

Detailed documentation of the Episodic project architecture and file organization.

Core Architecture

Episodic is built around a conversational DAG (Directed Acyclic Graph) model where each node represents a message in the conversation. The system supports branching conversations, topic detection, compression, and multi-model LLM integration.

File Structure Overview

```

episodic/
├── __init__.py           # Package initialization
├── __main__.py          # Entry point for python -m episodic
├── cli.py               # Command-line interface and interaction loop
├── core.py              # Core data structures (Node, ConversationDAG)
├── db.py                # Database operations and persistence
├── llm.py               # LLM integration and model management
├── topics.py            # Topic detection and management
├── compression.py       # Async compression system
├── visualization.py     # Graph visualization utilities
├── config.py            # Configuration management
├── prompt_manager.py    # System prompt management
├── llm_config.py        # LLM provider configuration
├── server.py            # HTTP server for visualization
├── ml/                  # Machine learning features
│   ├── drift.py         # Semantic drift detection
│   ├── embeddings/      # Embedding providers
│   └── summarization/    # Summarization strategies

```


Detailed File Documentation

Core System Files

`__main__.py`

Purpose: Application entry point - Imports and delegates to CLI main function - Handles command-line argument parsing - Sets up the execution environment

`cli.py` (~2000 lines)

Purpose: Command-line interface and user interaction - **Main Components:** -

`talk_loop()` : Main interaction loop for conversational mode -

`_handle_chat_message()` : Processes user messages and LLM responses -

`process_command()` : Routes slash commands to handlers - Command

implementations (init, add, show, compress, etc.) - **Key Features:** - Rich terminal UI with syntax highlighting - Command history and auto-suggestions - Session cost tracking -

Integration with all other modules - **Dependencies:** Uses Typer for CLI framework, Prompt Toolkit for rich input

`core.py`

Purpose: Core data structures and abstractions - **Classes:** - `Node` : Represents a single message/response in the conversation - Attributes: id, content, parentid, role, timestamp - Methods: `todict()`, `fromdict()` - `ConversationDAG` : Manages the conversation graph structure - Methods: `addnode()`, `getancestors()`, `getdescendants()` - Supports branching and merging conversations - **Design Pattern:** Immutable nodes with parent references create the DAG

`db.py`

Purpose: Database persistence and operations - **Database:** SQLite with thread-safe connection handling - **Schema:** - `nodes` table: Stores conversation nodes - `topics` table: Stores detected topics with ranges - `compression` table: Stores compression metadata - **Key Functions:** - `insert_node()` : Add new conversation node - `get_node()` : Retrieve node by ID - `get_ancestry()` : Get conversation history - `store_topic()` : Save detected topic - `update_topic_end_node()` : Extend topic

boundaries - **Features:** - Short ID generation (2-char identifiers) - Full-text search capabilities - Migration support

LLM Integration

`llm.py`

Purpose: Multi-provider LLM integration - **Core Functions:** - `query_llm()` : Direct LLM queries - `query_with_context()` : Queries with conversation history - `get_context_messages()` : Build context from conversation DAG - **Features:** - Provider-agnostic interface via LiteLLM - Token counting and cost tracking - Prompt caching support - Context window management - Error handling and retries - **Supported Providers:** OpenAI, Anthropic, Ollama, Google, etc.

`llm_config.py`

Purpose: LLM provider configuration and model management - Model availability checking - Provider-specific settings - Default model selection - API key validation

Topic Management

`topics.py` (NEW)

Purpose: Topic detection and management system - **Classes:** - `TopicManager` : Encapsulates all topic-related functionality - **Key Functions:** - `detect_topic_change_separately()` : Analyzes messages for topic changes - `extract_topic_ollama()` : Extracts topic names using LLM - `should_create_first_topic()` : Determines when to create initial topic - `build_conversation_segment()` : Formats conversation for analysis - `count_nodes_in_topic()` : Counts messages in a topic range - **Features:** - Separate LLM calls for topic detection - Configurable sensitivity thresholds - Topic boundary management - Integration with compression system

Compression System

`compression.py`

Purpose: Async background compression of conversation segments - **Classes:** - `CompressionJob` : Represents a queued compression task -

`AsyncCompressionManager` : Manages background compression - **Components:** - Background worker thread - Priority queue for compression jobs - Topic-aware compression boundaries - Automatic triggering on topic changes - **Features:** - Non-blocking compression - Configurable compression strategies - Compression statistics tracking - Error handling and retries

Visualization

`visualization.py`

Purpose: Graph visualization of conversation DAG - Creates interactive visualizations using Plotly - Generates NetworkX graphs from conversation data - Supports filtering and layout algorithms - Real-time updates via HTTP polling

`server.py`

Purpose: HTTP server for visualization interface - Flask-based web server - Serves visualization HTML/JS - Provides REST endpoints for graph data - Supports real-time updates

Configuration

`config.py`

Purpose: Application configuration management - **Features:** - JSON-based configuration storage - User-specific settings (~/.episodic/config.json) - Default values with override support - Runtime configuration changes - **Key Settings:** - Model preferences - Display options - Compression settings - Topic detection parameters

`prompt_manager.py`

Purpose: System prompt management - Loads prompts from markdown files - Supports prompt metadata (name, description, tags) - Active prompt switching - Prompt template rendering - Custom prompt creation

Machine Learning Features

`ml/drift.py`

Purpose: Semantic drift detection between messages - Calculates semantic similarity using embeddings - Tracks conversation topic evolution - Provides drift metrics and visualization - Supports multiple embedding providers

`ml/embeddings/providers.py`

Purpose: Embedding generation for semantic analysis - Multiple provider support (OpenAI, Sentence Transformers) - Caching for performance - Fallback handling - Batch processing support

`ml/summarization/strategies.py`

Purpose: Different summarization approaches - **Strategies:** - Simple: Basic summary - Detailed: Comprehensive summary - Bullets: Bullet-point format - Structured: Sectioned summary - Topic-aware: Preserves topic boundaries - Configurable per use case

Data Flow

1. **User Input** → CLI captures message
2. **Topic Detection** → TopicManager analyzes for changes
3. **LLM Query** → Message sent to selected model
4. **Response Processing** → Format and display response
5. **Persistence** → Save nodes to database
6. **Topic Extension** → Update topic boundaries
7. **Compression Queue** → Queue old topics for compression
8. **Background Compression** → Async worker compresses topics

Key Design Patterns

1. DAG Structure

- Immutable nodes with parent references

- Enables conversation branching
- Supports non-linear exploration

2. Provider Abstraction

- LiteLLM for model-agnostic interface
- Easy addition of new providers
- Consistent error handling

3. Async Processing

- Background compression doesn't block UI
- Thread-safe database operations
- Queue-based job management

4. Modular Architecture

- Clear separation of concerns
- Each module has single responsibility
- Easy to test and maintain

5. Configuration-Driven

- Behavior controlled by configuration
- Runtime adjustments without code changes
- User preferences persistence

Extension Points

Adding New Commands

1. Add command handler in `cli.py`
2. Update help text
3. Add to command routing

Adding New LLM Providers

1. Configure in LiteLLM
2. Add to model list in `llm_config.py`
3. Handle provider-specific errors

Adding Compression Strategies

1. Create new strategy in `ml/summarization/strategies.py`
2. Register in strategy map
3. Add configuration option

Custom Topic Detection

1. Modify prompt in `prompts/topic_detection.md`
2. Adjust detection thresholds
3. Implement custom detection logic

Testing

Test Structure

```
tests/  
├─ test_core.py           # Core data structure tests  
├─ test_db.py             # Database operation tests  
├─ test_topics.py         # Topic detection tests  
├─ test_compression.py    # Compression system tests  
└─ test_integration.py    # End-to-end tests
```

Manual Testing

- Interactive test scripts in `scripts/`
- Test different conversation patterns
- Verify topic detection accuracy
- Check compression effectiveness

Performance Considerations

Database

- Indexes on frequently queried columns
- Connection pooling for concurrent access
- Periodic vacuum for optimization

LLM Queries

- Prompt caching reduces API calls
- Context window management prevents errors
- Batch operations where possible

Memory Usage

- Streaming responses for large content
- Compression reduces storage needs
- Configurable context depth

Async Operations

- Non-blocking compression
- Thread-safe database access
- Efficient queue management

Security Considerations

- API keys in environment variables
- No sensitive data in prompts
- User data isolated in local database
- Configurable model access
- No automatic data sharing

Future Architecture Considerations

Potential Enhancements

1. **Plugin System:** Dynamic loading of extensions
2. **Multi-user Support:** User isolation and permissions
3. **Cloud Sync:** Optional conversation backup
4. **Advanced Analytics:** Conversation insights
5. **Voice Interface:** Speech-to-text integration
6. **Export Formats:** Markdown, JSON, HTML exports
7. **Embedding Storage:** Persistent embedding cache
8. **Real-time Collaboration:** Shared conversations

...

Development Guide

Source: docs/Development.md

Development & Testing

This document provides information for developers who want to contribute to Episodic or modify it for their own purposes.

Project Structure

```

episodic/
├── __init__.py
├── __main__.py
├── cli.py                # Main CLI loop and command dispatcher
├── core.py              # Core data structures (Node, ConversationDAG)
├── db.py                # Database operations
├── db_compression.py    # Compression storage system
├── llm.py               # LLM integration via LiteLLM
├── llm_config.py        # LLM provider configuration
├── conversation.py      # Conversation management
├── topics.py            # Topic detection and management
├── compression.py       # Async compression system
├── config.py            # Configuration management
├── visualization.py     # Graph visualization
├── commands/           # Command implementations
│   ├── __init__.py
│   ├── navigation.py    # Navigation commands (/head, /show, etc.)
│   ├── settings.py      # Configuration commands (/set, /verify)
│   ├── topics.py        # Topic commands (/topics, /rename-topics)
│   ├── compression.py   # Compression commands
│   ├── prompts.py       # Prompt management
│   ├── summary.py       # Summary generation
│   └── ...
└── ...

```

Testing

The test suite is located in the `tests/` directory. See [tests/README.md](#) for details on running tests.

Quick start:

```
# Run all tests
python -m unittest discover

# Run tests with the test runner
cd tests
python run_tests.py
```

Contributing

Contributions to Episodic are welcome! Here are some guidelines:

1. Fork the repository
2. Create a feature branch
3. Make your changes
4. Run the tests to ensure everything still works
5. Submit a pull request

Future Development

Planned features for future development include:

- State summarization
- Enhanced visualization capabilities
- Additional LLM provider integrations
- Performance optimizations for large conversation graphs

License

This project is licensed under the terms of the MIT license. See the [LICENSE](#) file for details.

TODO List

Source: TODO.md

Episodic Project TODO

Completed

Topic Detection & Management

- ☐ **Implement LLM-based topic detection** - Uses ollama/llama3 for efficient detection
- ☐ **Add topic extraction function** - `extract_topic_ollama()` implemented
- ☐ **Database schema** - Topics table with name, start/end nodes, confidence
- ☐ **Implement `/topics` command** - Shows topics with ranges and node counts
- ☐ **Topic storage/retrieval** - Full CRUD operations for topics
- ☐ **Optional topic display** - `/set show_topics true` shows topic evolution
- ☐ **Fix topic naming** - Topics named based on content, not trigger message
- ☐ **Topic detection prompt optimization** - Simplified for ollama/llama3 compatibility
- ☐ **Add `/rename-topics` command** - Renames placeholder "ongoing-*" topics
- ☐ **Fix overlapping topics** - Topics now have proper boundaries
- ☐ **Current topic tracking** - ConversationManager tracks current topic
- ☐ **Fix JSON parsing errors** - Added robust fallback for Ollama responses

Compression System

- ☐ **Async background compression** - Thread-based worker with priority queue
- ☐ **Auto-compression on topic change** - Topics compressed when closed

- ☐ **Compression statistics** - `/compression-stats` command
- ☐ **Configurable compression** - Model, min nodes, notifications via `/set`
- ☐ **Fix compression nodes in tree** - Compressions stored separately, not as nodes

UI/UX Improvements

- ☐ **Color scheme adaptation** - Supports light/dark terminals
- ☐ **Colored help display** - Commands and descriptions with proper formatting
- ☐ **Benchmark system** - Performance tracking with `/benchmark` command
- ☐ **Summary command** - `/summary [N|all]` for conversation summaries
- ☐ **Benchmark display after commands** - Shows benchmarks after commands when enabled
- ☐ **Fix streaming duplication** - Fixed text appearing twice in constant-rate mode
- ☐ **Word wrapping improvements** - Proper word-by-word streaming with wrapping
- ☐ **Markdown bold support** - `text` now appears as bold in terminal
- ☐ **List indentation** - 6-space indentation for wrapped lines in lists

Code Quality & Testing

- ☐ **CLI code cleanup** - Removed unused imports and duplicate code
- ☐ **Fix test suite** - Fixed cache and config initialization tests
- ☐ **Add missing Config.delete()** - Added delete method for configuration values
- ☐ **Simplify test infrastructure** - Removed over-engineered test setup
- ☐ **Update documentation** - Cleaned up outdated testing docs

Navigation & State Management

- ☐ **Fix** `/init --erase` - Now properly resets conversation manager state
- ☐ **Fix topics starting from node 02** - Query first user node directly

In Progress

Critical Issues

- ☐ **Fix dynamic topic threshold behavior** - Document and make configurable the behavior where first 2 topics use half threshold
- ☐ **Test configuration isolation** - Tests modify production config file (`~/.episodic/config.json`)

Topic Management Enhancements

- ☐ **Running topic guess** - Update tentative topic names periodically
- ☐ **Fix first topic creation** - Initial topic not always created properly
- ☐ **Move topic detection to background** - Eliminate response delay (as noted in todo list)

Pending

High Priority

- ☐ **Manual compression trigger** - `/compress topic-name` command
- ☐ **Topic-based navigation** - Jump to specific topics in history
- ☐ **Fix remaining test failures** - 7 tests still failing (93% passing)

Medium Priority

- ☐ **Improve drift accuracy** - Current embedding approach shows high drift for similar sentences
- ☐ **Async drift processing** - Calculate drift in background to reduce delays

Low Priority

- ☐ **Move debug messages** - Topic detection debug should appear after LLM response
- ☐ **Topic refinement UI** - Allow manual topic name editing
- ☐ **Export topics** - Export topic summaries to markdown

Technical Debt

- ☐ **Reduce drift calculation overhead** - Consider faster embedding model
- ☐ **Better embedding cache strategy** - Optimize to reduce redundant calculations
- ☐ **Test coverage for ML modules** - Add tests for drift.py, peaks.py, embeddings/
- ☐ **Update deprecated `datetime.utcnow()`** - Use timezone-aware `datetime.datetime.now(datetime.UTC)`

Notes

- Topic detection uses configurable model (default: ollama/llama3)
- Compression happens automatically when topics close
- Current active topic shows as "ongoing-discussion" until closed
- Running topic guess is configurable via `/set running_topic_guess`
- **IMPORTANT:** Topic detection has dynamic threshold - first 2 topics need 4+ messages, subsequent topics need 8+ messages

- Compressions are stored in separate tables (compressionsv2, compressionnodes) and don't pollute conversation tree
- Unit tests need isolation - currently modify user's production config file

Implementation Plan

Source: ImplementationPlan.md

Conversational Drift Detection

Implementation Plan

This document outlines the recommended order for implementing the conversational drift detection system described in `ConversationalDriftDesign.md`.

Phase 1: Foundation (Start Here)

1. Implement basic `DistanceFunction.calculate()`

- **Goal:** Get cosine similarity working first (most common for text embeddings)
- **Location:** `episodic/mlconfig.py` - `DistanceFunction` class
- **Why first:** Simple, well-understood algorithm that will work for most cases
- **Success criteria:** Can calculate similarity between two embedding vectors

2. Implement basic `EmbeddingProvider.embed()`

- **Goal:** Get sentence-transformers working (local, no API needed)
- **Location:** `episodic/mlconfig.py` - `EmbeddingProvider` class
- **Why early:** Local setup, no API costs, easy to experiment
- **Success criteria:** Can generate embeddings for text strings
- **Suggested model:** Start with "all-MiniLM-L6-v2" (small, fast)

3. Create a simple test harness

- **Goal:** Load existing conversations and compute embeddings for testing
- **Location:** New file like `test_drift_manually.py` or add to existing CLI
- **Why important:** Need real data to validate the system works
- **Success criteria:** Can load conversations from DB and generate embeddings

Phase 2: Core Drift Detection

4. Implement semdepth context building

- **Goal:** Function to collect N ancestor nodes and concatenate their content
- **Location:** New drift detection module or in `mlconfig.py`
- **Details:** Walk up DAG from current node, collect semdepth nodes, combine text
- **Success criteria:** Given a node ID and semdepth=3, returns combined text of node + 2 ancestors

5. Add real-time drift display to CLI

- **Goal:** Show drift scores above each query/response in talk mode
- **Location:** Modify `episodic/cli.py` - main conversation loop
- **Format:** `[Semantic drift: 0.85 from previous context]`
- **Why critical:** Immediate feedback loop for understanding drift patterns
- **Success criteria:** See drift scores in real-time during conversations

6. Test on your actual conversations

- **Goal:** Use the system on real conversations to see drift patterns
- **Approach:** Have conversations on different topics, watch drift scores
- **Data collection:** Note when drift scores feel right vs wrong
- **Success criteria:** Intuitive understanding of what different drift values mean

Phase 3: Understanding Patterns

7. Add multiple distance functions

- **Goal:** Compare cosine vs euclidean vs dot product similarity
- **Location:** Extend `DistanceFunction.calculate()` with more algorithms
- **Why useful:** Different algorithms may capture different aspects of semantic drift
- **Success criteria:** Can switch between distance algorithms and see differences

8. Experiment with semdepth values

- **Goal:** Test different context window sizes (1, 3, 5 ancestor nodes)
- **Approach:** Try same conversations with different semdepth settings
- **Data:** Which semdepth gives most intuitive drift detection?
- **Success criteria:** Understand optimal semdepth for different conversation types

9. Collect data on your conversation patterns

- **Goal:** Document which settings work best for different conversation types
- **Data points:**
 - One-shot questions vs long discussions
 - Topic resumption patterns
 - Natural topic drift within conversations
- **Success criteria:** Can predict good drift settings for different use cases

Phase 4: Branch Summarization

10. Implement LocalLLMSummarizer

- **Goal:** Use existing Ollama/LiteLLM integration for branch summarization
- **Location:** `BranchSummarizer.summarize_branch()` method
- **Integration:** Leverage existing `episodic.llm` module
- **Success criteria:** Can generate summaries of conversation branches

11. Add branch summary drift comparison

- **Goal:** Show both local drift (semdepth) and global drift (vs branch summary)
- **Display:** `[Local drift: 0.85 | Branch drift: 0.12 | Similar to: "Chess Strategy"]`
- **Why important:** Detect topic resumption vs new topics
- **Success criteria:** Can distinguish between local topic drift and returning to previous topics

12. Test summarization quality

- **Goal:** Validate that branch summaries capture conversation essence
- **Method:** Read summaries vs original conversations
- **Tuning:** Adjust summarization prompts for better quality
- **Success criteria:** Summaries are useful for semantic comparison

Phase 5: Automation (Much Later)

13. Implement automatic restructuring logic

- **Goal:** Automatically move nodes in DAG based on semantic similarity
- **Approach:**
 - Detect high drift (threshold-based)
 - Walk up DAG comparing similarity
 - Move nodes or create weak links
- **Why last:** Need deep understanding of drift patterns first
- **Success criteria:** Automatic restructuring improves conversation flow

Implementation Strategy

Quick Wins First

- Start with steps 1-6 as a focused sprint
- Goal: See drift scores in real-time within 1-2 days
- This provides immediate feedback and motivation

Iterative Refinement

- Each step builds understanding for the next
- Use real conversation data throughout
- Adjust approach based on what you learn

Avoid Premature Optimization

- Don't automate until you understand the patterns
- Focus on measurement and observation first
- Automation is the final step, not the first

Data-Driven Development

- Use your actual conversations for testing
- Document what works and what doesn't
- Let the data guide algorithm choices

Success Metrics

Phase 1-2 Success

- Can see drift scores in real-time
- Scores correlate with intuitive sense of topic changes
- System works reliably with existing conversations

Phase 3 Success

- Understand optimal settings for different conversation types
- Can predict when topics are shifting vs continuing
- Have baseline metrics for automation decisions

Phase 4 Success

- Branch summaries are coherent and useful
- Can detect topic resumption vs new topic initiation
- System helps with context management

Phase 5 Success

- Automatic restructuring feels natural
- Conversation flow improves noticeably
- System reduces cognitive load of context management

Notes

- Keep implementations simple initially
- Focus on learning over optimization
- Document insights as you go
- Be prepared to adjust the plan based on discoveries

Quick Start Commands

Once basic implementation is ready:

```
# Test basic functionality
from episodic.mlconfig import get_local_config
config = get_local_config()

# Test embedding
embedding = config.embedding_provider.embed("Hello world")

# Test distance
dist = config.distance_function.calculate(embedding1, embedding2)
```

Remember: The goal is understanding, not perfection. Start simple and iterate!

...

Compression and Topic Fixes

Source: COMPRESSION_AND_TOPIC_FIXES.md

Compression and Topic Naming Fixes

Issues Fixed

1. Compression Schema Errors

Problems: - The `compressions` table was missing a `content` column - The `compression_nodes` table was not being created

These caused compression jobs to fail with:

```
table compressions has no column named content
no such table: compression_nodes
```

Solution: - Added call to `create_compression_tables()` during initialization - This function creates the `compression_nodes` table and adds the `content` column if missing - Runs automatically during `initialize_db()`

2. Topic Naming Issue

Problem: Topics were showing as "ongoing-XXXXXX" instead of meaningful names because: - Topics are renamed when the NEXT topic is detected - The LAST topic in a session remained with its placeholder name

Solution: Added `finalize_current_topic()` method that: - Extracts a proper name for the current topic if it has a placeholder name - Is called automatically when: - A script execution completes - User types `/exit` or `/quit` - User presses Ctrl+D

Usage

The fixes are automatic, but you can also:

1. Manually rename all placeholder topics:

```
/rename-topics
```

2. Check if topics need renaming:

```
/topics
```

Look for topics named "ongoing-XXXXXX"

Technical Details

- Migration runs automatically on startup if needed
- Topic finalization uses the same extraction logic as regular topic changes
- Both fixes are backward compatible

Topic Detection Fixes

Source: TOPIC_DETECTION_FIXES.md

Topic Detection Fixes

Version 2: Intent-Based Detection

Added a new intent-based topic detection system that complements the JSON schema approach:

Features





1. **Intent Classification:** Messages are classified into one of four intents:
 - `JUST_COMMENT` : Brief acknowledgments that don't advance conversation
 - `DEVELOP_TOPIC` : Continuing the current topic
 - `INTRODUCE_TOPIC` : Starting a new conversation
 - `CHANGE_TOPIC` : Shifting to a different subject
2. **Two-Step Reasoning:** The model first classifies intent, then determines if it's a topic shift
3. **User-Message Focus:** V2 prompt only analyzes user messages, ignoring assistant responses
4. **Structured JSON Output:** Combines prompt engineering with JSON schema validation

Configuration

Enable V2 detection:

```
/set topic_detection_v2 true
```

Results

V2 detection correctly identifies: -  Topic development (Mars → Mars moons) -  Brief comments that shouldn't trigger changes -  Clear topic shifts (Mars → Cooking) - 
First topic introduction

Topic Detection Fixes

Issues Identified

1. **Inconsistent Thresholds:** The code had undocumented behavior where the first 2 topics used half the configured threshold (4 messages instead of 8), causing topics to be created with only 2 user messages.
2. **Overly Sensitive Detection:** The original prompt was too simple and the model was detecting topic changes too aggressively, even for related subjects like "Mars facts" → "Mars rovers".

Changes Made

1. Fixed Threshold Logic (topics.py and conversation.py)

Removed the special case for early topics:




```
# OLD: Half threshold for first 2 topics
if total_topics <= 2:
    effective_min = max(4, min_messages_before_change // 2)
else:
    effective_min = min_messages_before_change

# NEW: Consistent threshold
effective_min = min_messages_before_change
```

2. Improved Topic Detection Prompt (prompts/topic_detection.md)

- Made the prompt clearer about what constitutes a topic change
- Added explicit examples of what should and shouldn't trigger a change
- Introduced the key test: "Could these topics naturally appear in the same conversation?"
- Maintained JSON output format for consistency

3. Results

The JSON-based detection with temperature=0 is now working correctly: -  Correctly identifies continuation of related topics (Mars → Mars rovers) -  Correctly identifies clear domain shifts (Mars → Cooking) -  Consistent threshold prevents premature topic creation

Configuration

Users can adjust the threshold via:

```
/set min_messages_before_topic_change 8 # Default is 8
```

The first topic still uses a configurable threshold (default 3) via:

```
/set first_topic_threshold 3 # Default is 3
```

Topic Detection Implementation

Source: `topic_detection_fix.md`

Topic Detection System Fix

Issues Found

1. **All topics named "ongoing-discussion"**: When a topic change is detected, every new topic is created with the generic name "ongoing-discussion". This prevents multiple topics from being properly tracked.
2. **Broken parent chains**: Multiple conversation roots exist (orphan nodes), breaking the DAG structure and causing topic boundary issues.
3. **Topic boundary validation fails**: The `count_nodes_in_topic` function fails when the start node isn't in the ancestry chain due to broken parent relationships.

Root Causes

1. **Generic topic naming**: Line 446 in `conversation.py` always creates new topics with name "ongoing-discussion"
2. **No unique topic identification**: Topics need unique names or IDs to be properly tracked
3. **Parent chain breaks**: When new conversations are started without proper parent linking

Proposed Fixes

Fix 1: Generate unique topic names for new topics

Instead of always using "ongoing-discussion", generate a unique placeholder name like: - "topic-1", "topic-2", etc. - "ongoing-discussion-{timestamp}" - "topic-{short_uuid}"

Fix 2: Add topic ID to database schema

Add a unique identifier for topics independent of their name:

```
ALTER TABLE topics ADD COLUMN topic_id TEXT UNIQUE;
```

Fix 3: Handle broken parent chains gracefully

Update `count_nodes_in_topic` to handle cases where `start_node` isn't in ancestry: -
Return minimum count (2) when chain is broken - Log warning about broken chain -
Consider alternative ancestry traversal

Fix 4: Prevent orphan nodes

Ensure all new conversation threads properly link to existing nodes or create proper root nodes.

Implementation Priority

1. **Immediate fix:** Change line 446 to generate unique topic names
2. **Short-term:** Update topic detection to handle broken chains gracefully
3. **Long-term:** Add proper topic IDs and improve parent chain management

Streaming Implementation

Source: STREAMING_IMPLEMENTATION.md

Streaming Implementation for Episodic

Overview

This implementation adds streaming support for LLM responses in the episodic conversation system. Responses now stream in real-time as they are generated, providing a more responsive user experience.

Key Features Implemented

1. Streaming Infrastructure (`llm.py`)

- Modified `_execute_llm_query()` to support a `stream` parameter
- Added `query_with_context()` streaming support
- Created `process_stream_response()` generator to yield content chunks from the stream

2. Streaming Display Handler (`conversation.py`)

- Updated `handle_chat_message()` to check for streaming preference
- Implemented real-time display with:
 - Proper color formatting using `get_llm_color()`
 - Word wrapping that works with partial lines
 - Line-by-line processing for clean output
 - Smart line breaking for long lines without newlines

3. Cost Calculation

- Maintained accurate cost tracking by making a non-streaming call after streaming
- This ensures we get proper token counts and cost information
- Future improvement: Extract usage data from streaming chunks when LiteLLM supports it

4. Configuration

- Added `stream_responses` config option (default: True)
- Added `/set stream on/off` command to toggle streaming
- Updated status display to show streaming state
- Configuration persists across sessions

Usage

Enable/Disable Streaming

```
# Toggle streaming
/set stream

# Enable streaming
/set stream on

# Disable streaming
/set stream off

# Check current status
/set
```

How It Works

1. When a user sends a message, the system checks if streaming is enabled
2. If enabled, it requests a streaming response from the LLM
3. As chunks arrive, they are:
 - Displayed immediately with proper formatting
 - Accumulated for database storage
 - Word-wrapped intelligently
4. After streaming completes, the full response is stored in the database
5. Cost information is calculated and displayed if enabled

Technical Details

Streaming Flow

1. `handle_chat_message()` checks `config.get("stream_responses", True)`
2. Calls `query_with_context(..., stream=True)` to get a generator
3. Processes chunks through `process_stream_response()`
4. Displays chunks with color and wrapping
5. Stores complete response in database

Word Wrapping During Streaming

- Accumulates content until a newline is found
- Wraps complete lines respecting terminal width
- For very long lines without newlines, breaks at word boundaries
- Maintains proper indentation for wrapped lines

Color Consistency

- The robot emoji and response text use `get_llm_color()`
- Maintains visual consistency with non-streaming responses
- System messages use `get_system_color()`

Testing

Run the included test script to verify streaming functionality:

```
python test_streaming.py
```

Future Improvements

1. Extract token usage directly from streaming chunks when LiteLLM adds support
2. Add progress indicators for very long responses
3. Support for interrupting streaming responses

4. Optimize the temporary non-streaming call for cost calculation

Streaming Fix Summary

Source: STREAMING_FIX_SUMMARY.md

Streaming Fix Summary

Issues Fixed

1. **Double Printing Bug:** Text was being printed twice during streaming - once by the chunk processing logic and once by the line wrapping logic.
2. **No Constant-Rate Streaming:** Streaming happened immediately as chunks arrived from the LLM, which could be jarring.

Changes Made

1. Fixed Double Printing (`conversation.py`)

- Removed the complex line buffering and wrapping logic in immediate streaming mode
- Now prints chunks directly as they arrive without buffering
- Added a newline after streaming completes

2. Added Constant-Rate Streaming

- Added configuration options:
 - `stream_rate` : Words per second (default: 15, range: 1-100)
 - `stream_constant_rate` : Enable/disable constant-rate mode (default: False)
- When enabled, text streams at a steady pace using a word queue and timer thread
- Words are buffered and printed at the configured rate

3. Updated Configuration System

- Added new streaming parameters to default config in `config.py`
- Added handlers in `/set` command for both new parameters
- Updated `/set` display to show current streaming configuration

Usage

View current streaming settings:

```
/set
```

Enable constant-rate streaming:

```
/set stream_constant_rate on  
/set stream_rate 10
```

Disable constant-rate streaming:

```
/set stream_constant_rate off
```

Toggle streaming on/off entirely:

```
/set stream off # Disable all streaming  
/set stream on  # Re-enable streaming
```

Testing

Run the test script to verify the fixes:

```
python test_streaming_fix.py
```

For manual testing: 1. Start episodic: `python -m episodic` 2. Enable constant-rate: `/set stream_constant_rate on` 3. Set rate: `/set stream_rate 5` 4. Send a message and observe the streaming behavior 5. Compare with immediate streaming: `/set stream_constant_rate off`

Prompt Caching

Source: PromptCaching.md

Understanding and Implementing LiteLLM's Context Caching

You're absolutely right, and I apologize for the confusion in my previous responses. LiteLLM does indeed support a more advanced form of caching called **Context Caching** (or Prompt Caching), which is different from simple response caching.

What Context Caching Actually Is

Context Caching allows you to:

- Cache long, static parts of your prompts (like system messages or large documents)
- Only send the new, dynamic parts (like user messages) with each API call
- Have LiteLLM automatically reference or inject the cached static content
- Significantly reduce token usage and costs for prompts with large unchanging sections

This is particularly valuable for your single-user chat application where you might have:

- A large, consistent system message
- Reference documents or context that remains static
- Changing user inputs

How to Implement Context Caching in Your Project

Here's how to add context caching to your `episodic/llm.py` file:

```

# Add near the top of the file after imports
import litellm
from litellm.caching import Cache

# Initialize context caching
litellm.enable_cache(
    type="redis", # Options: "redis", "in-memory"
    host="localhost", # Only for Redis
    port=6379, # Only for Redis
    password=None, # Only for Redis if needed
)

# Then modify your query functions to use context caching
def query_llm(
    prompt: str,
    model: str = "gpt-4o-mini",
    system_message: str = "You are a helpful assistant.",
    temperature: float = 0.7,
    max_tokens: int = 1000
) -> tuple[str, dict]:
    # ... existing code ...

    # Create messages array
    messages = [
        {"role": "system", "content": system_message},
        {"role": "user", "content": prompt}
    ]

    # ... existing code ...

    # Add context_caching=True to your litellm.completion call
    response = litellm.completion(
        model=full_model,
        messages=messages,
        temperature=temperature,
        max_tokens=max_tokens,
        context_caching=True, # Enable context caching
        cache_key="system_message" # Optional: custom key for the cached context
    )

    # ... rest of the function ...

```

Similarly, update the `query_with_context` function to include the `context_caching=True` parameter.

Benefits for Your Application

1. **Cost Reduction:** Only pay for tokens in the dynamic parts of your prompts
2. **Faster Responses:** Reduced token count means faster processing
3. **Consistency:** System messages and context remain consistent
4. **Flexibility:** You can still change your prompts as needed - only the static parts are cached

Supported Providers

This feature works with several major providers: - OpenAI - Anthropic - AWS Bedrock - And other providers that support context caching

Implementation Considerations

1. **Cache Storage:** For a single-user application, in-memory caching is likely sufficient
2. **Cache Keys:** Consider using meaningful keys for different types of cached content
3. **Cache Invalidation:** Implement a way to clear the cache if you need to update your system messages

By implementing context caching, you'll be able to significantly reduce token usage and costs in your application, especially if you use large system messages or include substantial context in your prompts.

Async Compression Design

Source: AsyncCompressionDesign.md

Async Background Compression Design

Overview

An intelligent background compression system that uses topic detection to automatically compress conversation segments at natural boundaries.

Key Components

1. Topic-Triggered Compression

- Leverages existing `detect_and_extract_topic_from_response()` function
- When a topic change is detected, the previous topic segment is queued for compression
- Natural conversation boundaries create more coherent compressed summaries

2. Background Worker Architecture

```
# Compression Queue
compression_queue = Queue()

# Background worker thread
def compression_worker():
    while True:
        job = compression_queue.get()
        if job is None: # Shutdown signal
            break

        # Process compression job
        compress_topic_segment(
            start_node_id=job['start_node_id'],
            end_node_id=job['end_node_id'],
            topic_name=job['topic_name']
        )
```

3. Integration Points

In `store_topic()` function:

```
def store_topic(topic_name, start_node_id, end_node_id, confidence):
    # Existing topic storage code...

    # Queue previous topic for compression if exists
    previous_topics = get_recent_topics(limit=1)
    if previous_topics and config.get('auto_compress_topics', True):
        prev_topic = previous_topics[0]
        compression_queue.put({
            'start_node_id': prev_topic['start_node_id'],
            'end_node_id': prev_topic['end_node_id'],
            'topic_name': prev_topic['topic_name']
        })
```

4. Compression Strategy

Topic-Aware Compression Prompt

```
def compress_topic_segment(start_node_id, end_node_id, topic_name):
    nodes = get_nodes_between(start_node_id, end_node_id)

    prompt = f"""Compress this conversation about '{topic_name}' into a concise summary.
    Preserve key insights, decisions, and conclusions.

    Conversation:
    {format_nodes(nodes)}

    Summary: """

    summary = query_llm(prompt, model=compression_model)

    # Create compressed node with topic metadata
    compressed_id = insert_node(
        content=f"[Compressed: {topic_name}]\n{summary}",
        parent_id=end_node_id,
        role="system"
    )

    store_compression(...)
```

5. Configuration Options

```
# episodic/config.py additions
DEFAULT_CONFIG = {
    'auto_compress_topics': True,
    'compression_min_nodes': 5, # Min nodes before compression
    'compression_max_age_hours': 24, # Compress topics older than X hours
    'compression_model': 'ollama/llama3', # Fast model for background work
    'compression_worker_threads': 1
}
```

6. User Controls

- `/set auto_compress_topics true/false` - Enable/disable auto compression
- `/compress-queue` - Show pending compression jobs
- `/compress-stats` - Enhanced to show auto vs manual compressions

Implementation Phases

1. **Phase 1:** Basic queue and worker thread
2. **Phase 2:** Integration with topic detection
3. **Phase 3:** Smart compression strategies (topic-aware prompts)
4. **Phase 4:** Advanced features (age-based triggers, parallel workers)

Benefits

1. **Natural boundaries** - Topics provide semantic compression points
2. **Non-blocking** - Conversation continues while compression happens
3. **Context-aware** - Topic names improve summary quality
4. **Automatic** - No manual intervention needed
5. **Resource efficient** - Uses cheap/fast models in background

Considerations

1. **Thread safety** - Database operations must be thread-safe
2. **Queue persistence** - Consider saving queue state for restarts
3. **Error handling** - Failed compressions shouldn't crash worker
4. **User notification** - Optional notifications when compression completes

Conversational Drift Design

Source: ConversationalDriftDesign.md

Conversational Drift Detection and DAG Restructuring Design

Overview

Design discussion for implementing semantic drift detection and automatic DAG restructuring in the episodic conversational system. The goal is to enable natural conversation flow with a fixed-length attention window by intelligently navigating the conversation DAG based on topic relevance rather than just recency.

Core Concept

Instead of using a simple sliding window of the last N messages, the system will: -
Dynamically select relevant conversation history based on semantic similarity -
Automatically restructure the DAG to reflect topical relationships - Maintain natural conversation flow across different conversation patterns

Key Design Elements

1. Semantic Depth (semdepth)

- Configurable parameter indicating how many ancestor nodes to include when creating embeddings
- Example: `semdepth=3` would embed current node + 2 most recent ancestors
- Provides local conversational context for semantic analysis

2. Node Embeddings

- Each node embedding includes the node content plus its semdepth ancestors
- Question: Should embeddings include both query and response, or just query?
- Consideration: Node might be moved or re-parented before response is generated

3. Modular Components

- **Embedding Library:** Customizable (OpenAI, sentence-transformers, etc.)
- **Distance Functions:** Various similarity measures (cosine, euclidean, hybrid search, re-ranking)
- **Branch Summarization:** Pluggable strategies for different conversation types

Implementation Phases

Phase 1: Measurement and Observation

- Real-time drift calculation and display during CLI interaction
- Print semantic drift scores above each query/response pair
- **Format:** `[Semantic drift: 0.85 from previous context]`
- Goal: Understand drift patterns across different conversation types

Phase 2: Branch Summarization

- Implement modular branch summarization strategies
- Compare drift against both local context (semdepth) and branch summary
- Display both local and global drift scores

Phase 3: Automatic Restructuring

- Implement automated DAG restructuring based on drift thresholds
- Walk up DAG comparing drift when high drift detected
- Create new parent relationships or weak links as appropriate

Conversation Patterns to Handle

1. Long Focused Sessions

- Extended discussions on single topics
- Low drift within session
- Recent context most important

2. One-Shot Questions

- Unrelated queries that won't be revisited
- High drift from existing conversation
- Should become children of root node
- Minimal historical context needed

3. Recurring Topics

- Periodic return to previous discussion threads
- Example: Game strategy discussions resumed across sessions
- Rich historical context from previous branch summaries valuable
- Low drift when compared to relevant branch summaries

Branch Summarization Strategies

Strategy 1: Bottom-up Hierarchical

- Start at leaf nodes, summarize small groups (2-3 nodes)
- Work upward, summarizing the summaries
- Root of branch gets final summary
- Preserves conversational flow structure

Strategy 2: Progressive/Rolling

- Maintain running summary traversing root to current node
- Each new node updates rather than replaces summary
- Captures conversation evolution

Strategy 3: Key Moment Extraction

- Identify important nodes (topic shifts, decisions, conclusions)
- Summarize only key moments
- Preserves critical information while compressing

Strategy 4: LLM Single-Pass

- Feed entire branch to LLM with summarization prompt
- Simple but potentially expensive for long branches

Summarization Implementation Options

Local/Free Options

- **Transformers Pipeline:** `pipeline("summarization")` from Hugging Face
 - Any Hugging Face summarization model can be plugged in
 - Good for custom branch summarization without API costs
 - Suitable for MVP implementation

Cloud/Paid Options

- **OpenAI GPT-3.5/4:** High-quality summarization via API
 - More expensive but potentially better quality
 - Good for production systems where quality is critical
 - Not required for initial MVP

Distance Functions (from RAG techniques)

Basic Measures

- Cosine similarity (most common)
- Euclidean distance
- Dot product similarity

Advanced Approaches

- **Hybrid search:** Semantic + keyword/lexical matching
- **Re-ranking:** Separate model for conversational relevance scoring
- **Contextual compression:** Filter retrieved context to most relevant parts
- **Parent document retrieval:** Retrieve larger context chunks

- **Hypothetical questions:** Match against generated questions the context might answer

Real-time Drift Display

Display multiple drift measurements: 1. **Local drift:** Between current query and semdepth context window 2. **Branch drift:** Between current query and branch summary 3. **Global drift:** Comparison against other branch summaries

Example output:

```
> What's the best strategy for early game in chess?
[Local drift: 0.85 | Branch drift: 0.92 | Similar to: "Game Strategy Discussion"]
🤖 response about chess strategy...

> Should I focus on center control or piece development first?
[Local drift: 0.12 | Branch drift: 0.15 | Continuing: "Chess Strategy"]
🤖 response continues chess discussion...
```

Automatic Restructuring Logic

High Drift Detection

1. Calculate drift scores for new query
2. If drift exceeds threshold, walk up DAG from current position
3. Compare semantic similarity at each ancestor level
4. Find best semantic match in conversation history
5. Either move node or create weak link to better parent

One-Shot Detection

- Very high drift from all existing nodes
- Automatically parent to root node
- Minimal context window construction

Topic Resumption Detection

- High local drift but low drift from specific branch summary
- Re-parent to that branch or create strong link
- Include relevant branch summary in context window

Technical Considerations

Performance

- Embedding computation on-demand vs pre-computed
- Caching strategies for branch summaries
- Incremental updates vs full recalculation

Modularity

- `BranchSummarizer` interface for pluggable strategies
- `EmbeddingProvider` interface for different models
- `DistanceFunction` interface for similarity measures
- `DriftAnalyzer` orchestrates the components

Configuration

- Adjustable semdepth parameter
- Drift thresholds for restructuring triggers
- Summarization strategy selection
- Embedding model selection

Open Questions

1. **Summary Scope:** Entire branches vs moving topic windows?
2. **Weighting:** How to weight nodes within semdepth window?
3. **Weak Links:** How to represent and utilize weak connections?
4. **Performance:** Real-time requirements vs computation cost?
5. **Validation:** How to measure if restructuring improves conversation quality?

Success Metrics

- More natural conversation flow across topic boundaries
- Relevant historical context available when resuming topics
- Reduced irrelevant context in attention window
- Intuitive drift scores that match human perception of topic changes

...

LangChain Integration Guide

Source: docs/LANGCHAIN_INTEGRATION_GUIDE.md

LangChain Integration Guide for Episodic

Overview

This guide outlines how to integrate LangChain capabilities into Episodic without replacing the existing LiteLLM infrastructure. The approach uses a hybrid solution that leverages the strengths of both libraries.

Why Hybrid Approach?

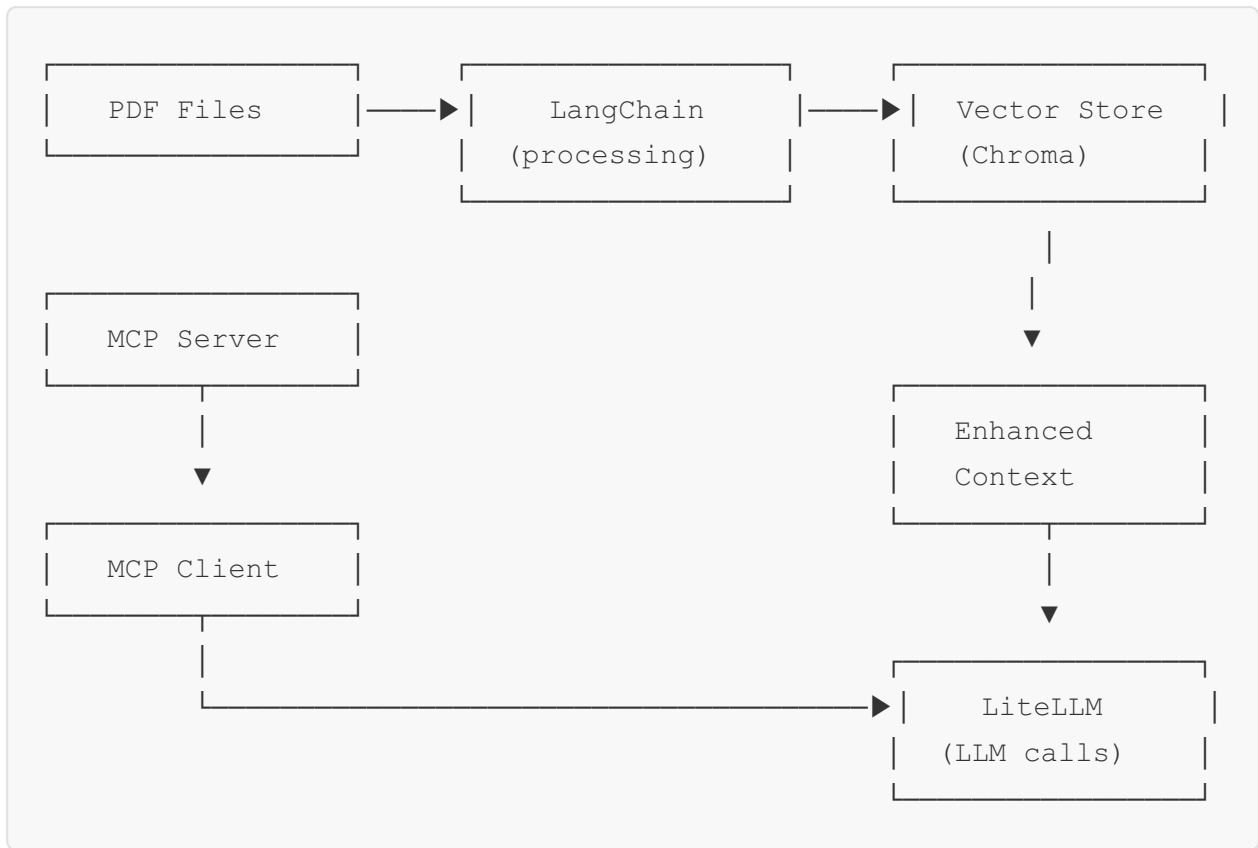
Keep LiteLLM for:

- LLM API calls (working great with multi-provider support)
- Cost tracking (`cost_per_token` functionality)
- Unified model interface across providers
- Streaming responses
- Simple fallback handling

Add LangChain for:

- PDF processing and document loading
- Document chunking and splitting
- Vector storage and similarity search
- RAG (Retrieval Augmented Generation)
- MCP (Model Context Protocol) integration
- Advanced tool usage patterns

Architecture



Implementation Examples

1. Document Manager

```

# episodic/documents.py - NEW FILE
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings
from typing import List, Optional
import litellm

class DocumentManager:
    def __init__(self, persist_directory: Optional[str] = "./chroma_db"):
        self.embeddings = OpenAIEmbeddings()
        self.vectorstore = None
        self.persist_directory = persist_directory
        self.text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200,
            length_function=len,
            separators=["\n\n", "\n", " ", ""]
        )

    def load_pdf(self, pdf_path: str) -> List[Document]:
        """Load and process a PDF file."""
        loader = PyPDFLoader(pdf_path)
        documents = loader.load()
        chunks = self.text_splitter.split_documents(documents)

        # Store in vector database
        if not self.vectorstore:
            self.vectorstore = Chroma.from_documents(
                chunks,
                self.embeddings,
                persist_directory=self.persist_directory
            )
        else:
            self.vectorstore.add_documents(chunks)

        return chunks

    def query_documents(self, query: str, k: int = 4) -> List[Document]:
        """Find relevant document chunks."""
        if not self.vectorstore:

```

```

        return []

    return self.vectorstore.similarity_search(query, k=k)

def get_context_for_llm(self, query: str, max_tokens: int = 2000) -> str:
    """Get relevant context for including in LLM prompt."""
    docs = self.query_documents(query)

    # Truncate context if needed
    context_parts = []
    current_tokens = 0

    for doc in docs:
        doc_tokens = len(doc.page_content.split())
        if current_tokens + doc_tokens > max_tokens:
            break
        context_parts.append(doc.page_content)
        current_tokens += doc_tokens

    return "\n\n---\n\n".join(context_parts)

```

2. Integration with Existing Code

```

# episodic/conversation.py - additions
from episodic.documents import DocumentManager

class ConversationManager:
    def __init__(self):
        # ... existing init code ...
        self.document_manager = DocumentManager()
        self.document_context_enabled = False

    def handle_chat_message_with_docs(self, user_input: str, model: str,
                                     system_message: str, context_depth: int = 1):
        """Enhanced chat that includes document context when relevant."""

        # Check if we should include document context
        if self.document_context_enabled and self.document_manager.vectorstore:
            doc_context = self.document_manager.get_context_for_llm(user_input)

            # Enhance the user input with document context
            enhanced_input = f"""Based on the following context from loaded documents:

{doc_context}

User question: {user_input}

Please provide an answer based on the document context when relevant."""

            # Use existing chat message handler
            return self.handle_chat_message(
                enhanced_input,
                model,
                system_message,
                context_depth
            )
        else:
            # Normal flow without documents
            return self.handle_chat_message(
                user_input,
                model,
                system_message,
                context_depth
            )

```

3. MCP Integration

```

# episodic/mcp_client.py - NEW FILE
import httpx
from typing import Dict, Any, List
import json

class MCPClient:
    def __init__(self, server_url: str):
        self.server_url = server_url
        self.client = httpx.Client()

    def list_tools(self) -> List[Dict]:
        """Get available tools from MCP server."""
        try:
            response = self.client.get(f"{self.server_url}/tools")
            response.raise_for_status()
            return response.json()
        except Exception as e:
            logger.error(f"Failed to list MCP tools: {e}")
            return []

    def call_tool(self, tool_name: str, arguments: Dict) -> Any:
        """Execute a tool on the MCP server."""
        try:
            response = self.client.post(
                f"{self.server_url}/tools/{tool_name}",
                json=arguments
            )
            response.raise_for_status()
            return response.json()
        except Exception as e:
            logger.error(f"Failed to call MCP tool {tool_name}: {e}")
            return {"error": str(e)}

# Integration function
def query_with_mcp_tools(prompt: str, mcp_client: MCPClient, model: str = "gpt-4")
    """Allow LLM to use MCP tools to answer questions."""
    tools = mcp_client.list_tools()

    if not tools:
        return query_llm(prompt, model=model)

```

```

# First, ask LLM what tool to use
tool_selection_prompt = f"""You have access to the following tools:

{json.dumps(tools, indent=2)}

User request: {prompt}

Analyze the request and respond with a JSON object containing:
- tool_name: The name of the tool to use (or "none" if no tool needed)
- arguments: The arguments to pass to the tool
- reasoning: Brief explanation of why you chose this tool
"""

response, _ = query_llm(tool_selection_prompt, model=model, temperature=0)

try:
    decision = json.loads(response)
    if decision.get("tool_name") != "none":
        # Call the tool
        tool_result = mcp_client.call_tool(
            decision["tool_name"],
            decision.get("arguments", {})
        )

        # Generate final response with tool results
        final_prompt = f"""Tool {decision['tool_name']} returned: {tool_result}

Original request: {prompt}

Please provide a helpful response based on the tool output."""

        return query_llm(final_prompt, model=model)
except Exception as e:
    logger.error(f"Error in tool selection: {e}")

# Fallback to regular response
return query_llm(prompt, model=model)

```


4. CLI Commands

```

# episodic/commands/documents.py - NEW FILE
import typer
from pathlib import Path
from episodic.configuration import get_system_color

def handle_load_document(file_path: str):
    """Load a document into the conversation context."""
    path = Path(file_path)

    if not path.exists():
        typer.echo(f"❌ File not found: {file_path}", err=True)
        return

    if path.suffix.lower() != '.pdf':
        typer.echo(f"❌ Currently only PDF files are supported", err=True)
        return

    try:
        chunks = conversation_manager.document_manager.load_pdf(str(path))
        typer.secho(
            f"📄 Loaded {path.name} ({len(chunks)} chunks)",
            fg=get_system_color()
        )

        # Show sample of content
        if chunks:
            preview = chunks[0].page_content[:200] + "..."
            typer.echo(f"Preview: {preview}")

    except Exception as e:
        typer.echo(f"❌ Error loading document: {e}", err=True)

def handle_docs_command(action: str = "list"):
    """Manage loaded documents."""
    if action == "list":
        # List loaded documents
        if not conversation_manager.document_manager.vectorstore:
            typer.echo("No documents loaded")
            return

        # Get unique sources

```

```

sources = set()
for doc in conversation_manager.document_manager.vectorstore.get()['documents']:
    if 'source' in doc.metadata:
        sources.add(doc.metadata['source'])

typer.echo("Loaded documents:")
for source in sources:
    typer.echo(f"📄 {Path(source).name}")

elif action == "clear":
    # Clear all documents
    conversation_manager.document_manager.vectorstore = None
    typer.secho("🗑️ Cleared all documents", fg=get_system_color())

elif action == "enable":
    conversation_manager.document_context_enabled = True
    typer.secho("✅ Document context enabled", fg=get_system_color())

elif action == "disable":
    conversation_manager.document_context_enabled = False
    typer.secho("❌ Document context disabled", fg=get_system_color())

```

Installation

```

# Required dependencies
pip install langchain langchain-community langchain-openai pypdf chromadb tiktoken

# Optional for MCP
pip install httpx

# Optional for other document types
pip install pypdf2 pdfplumber # Better PDF handling
pip install python-docx      # Word documents
pip install markdown         # Markdown files

```

Configuration

Add to `episodic/config.py`:

```
# Document processing settings
DOCUMENT_CONFIG = {
    "chunk_size": 1000,
    "chunk_overlap": 200,
    "embedding_model": "text-embedding-ada-002",
    "vector_store_type": "chroma",
    "persist_directory": "./document_store",
    "max_context_tokens": 2000,
    "auto_include_context": False
}

# MCP settings
MCP_CONFIG = {
    "server_url": "http://localhost:8080",
    "timeout": 30,
    "retry_attempts": 3
}
```

Usage Examples

Loading and Querying PDFs

```
> /load research_paper.pdf
📄 Loaded research_paper.pdf (127 chunks)
Preview: This paper presents a novel approach to...

> /docs enable
✅ Document context enabled

> What is the main contribution of this paper?
[Response will include relevant context from the PDF]

> /docs list
Loaded documents:
📄 research_paper.pdf
```

Using MCP Tools

```
> /mcp connect http://localhost:8080
✓ Connected to MCP server (5 tools available)

> /mcp tools
Available tools:
  🔧 web_search - Search the web for information
  🔧 calculator - Perform mathematical calculations
  🔧 weather - Get weather information
  ...

> What's the weather in San Francisco?
[LLM automatically uses weather tool and provides response]
```

Benefits

1. **No Breaking Changes:** All existing LiteLLM code continues to work
2. **Incremental Adoption:** Add features as needed
3. **Best of Both Worlds:** LiteLLM's simplicity + LangChain's document capabilities
4. **Cost Tracking Preserved:** Keep existing cost monitoring
5. **Provider Flexibility:** Still works with all LLM providers

Future Enhancements

1. **Multi-format Support:** Add Word, Markdown, HTML loaders
2. **Persistent Memory:** Save vector store between sessions
3. **Smart Chunking:** Use semantic chunking for better retrieval
4. **Hybrid Search:** Combine keyword and semantic search
5. **Document Management UI:** Web interface for document management

Hybrid Topic Detection Design

Source: docs/HYBRID_TOPIC_DETECTION_DESIGN.md

Hybrid Topic Detection System Design

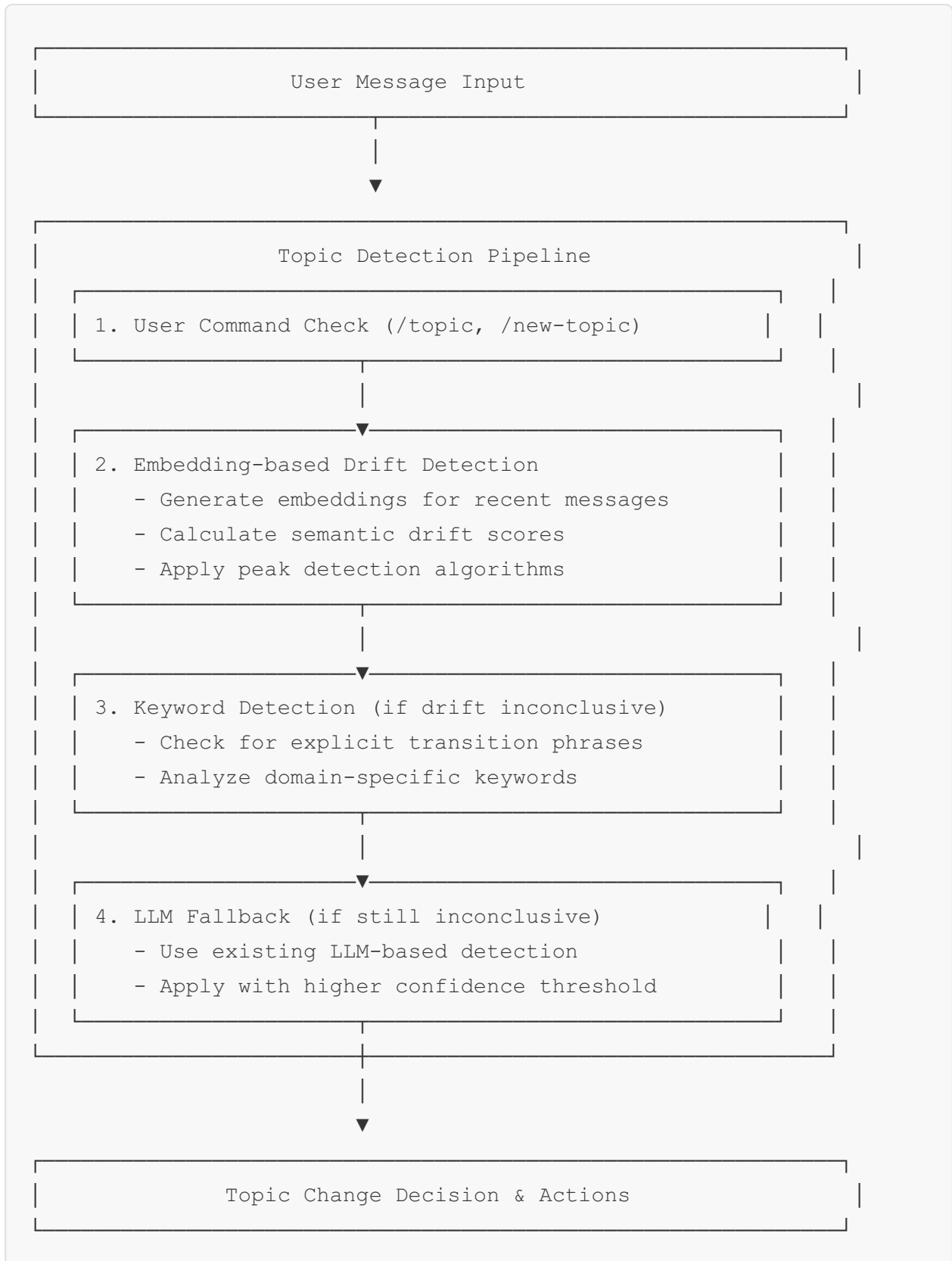
Executive Summary

This document outlines the design for a hybrid topic detection system that combines: 1. **Embedding-based semantic drift detection** (primary method) 2. **Keyword-based explicit transition detection** (secondary method) 3. **LLM-based detection** (fallback method) 4. **User control commands** (override mechanism)

The system aims to provide more accurate and reliable topic detection while reducing LLM API costs and improving response times.

System Architecture

Component Overview



Data Flow

1. Message Reception

- User message arrives
- Add to message history buffer
- Check for user commands

2. Embedding Generation

- Generate embeddings for new message
- Retrieve cached embeddings for recent messages
- Store new embeddings in cache/database

3. Drift Calculation

- Calculate pairwise drift between consecutive messages
- Calculate cumulative drift from topic start
- Apply windowed averaging for noise reduction

4. Peak Detection

- Apply configured peak detection algorithm
- Consider multiple signals (sudden drift, cumulative drift, keyword presence)
- Generate confidence score

5. Decision Making

- Combine all signals with weighted scoring
- Apply threshold based on configuration
- Execute topic change if threshold exceeded

6. Post-Processing

- Close current topic at last assistant response
- Create new topic with placeholder name
- Queue topic name extraction

Algorithm Details

1. Embedding-based Semantic Drift

Configuration Options

```
{
  "embedding_provider": "sentence-transformers", # or "openai", "huggingface"
  "embedding_model": "all-MiniLM-L6-v2",        # fast, good quality
  "distance_algorithm": "cosine",                # or "euclidean", "manhattan"
  "drift_threshold": 0.35,                       # topic change threshold
  "window_size": 5,                             # messages to consider
  "peak_detection": {
    "strategy": "adaptive",                      # or "threshold", "statistical"
    "min_prominence": 0.2,                      # minimum peak prominence
    "lookback": 3                               # messages to look back
  }
}
```

Algorithm Steps

1. Embedding Generation

```
def generate_embedding(message: str) -> List[float]:
    # Check cache first
    if message_hash in embedding_cache:
        return embedding_cache[message_hash]

    # Generate new embedding
    embedding = embedding_provider.embed(message)

    # Store in cache and database
    embedding_cache[message_hash] = embedding
    store_embedding_in_db(message_hash, embedding)

    return embedding
```

2. Drift Calculation

```

def calculate_drift_score(messages: List[Message]) -> float:
    # Get embeddings for recent messages
    embeddings = [generate_embedding(msg.content) for msg in messages]

    # Calculate pairwise distances
    distances = []
    for i in range(1, len(embeddings)):
        dist = distance_function(embeddings[i-1], embeddings[i])
        distances.append(dist)

    # Apply windowed averaging
    avg_drift = np.mean(distances[-window_size:])

    # Calculate cumulative drift from topic start
    if current_topic_start_embedding:
        cumulative_drift = distance_function(
            current_topic_start_embedding,
            embeddings[-1]
        )

    # Combine signals
    combined_score = (0.7 * avg_drift + 0.3 * cumulative_drift)

    return combined_score

```

3. Peak Detection

```
def detect_drift_peak(drift_scores: List[float]) -> bool:
    if peak_strategy == "adaptive":
        # Adaptive threshold based on recent history
        baseline = np.mean(drift_scores[:-1])
        std_dev = np.std(drift_scores[:-1])
        threshold = baseline + (2 * std_dev)

        current_drift = drift_scores[-1]
        prominence = (current_drift - baseline) / baseline

        return (current_drift > threshold and
                prominence > min_prominence)

    elif peak_strategy == "threshold":
        return drift_scores[-1] > drift_threshold
```

2. Keyword-based Detection

Configuration

```
{
  "keyword_detection": {
    "enabled": true,
    "weight": 0.3, # contribution to final score
    "transition_phrases": [
      "let's talk about",
      "changing the subject",
      "on a different note",
      "switching gears",
      "moving on to",
      "different question"
    ],
    "domain_keywords": {
      "technology": ["programming", "software", "computer", "AI"],
      "science": ["physics", "chemistry", "biology", "research"],
      "cooking": ["recipe", "ingredients", "cooking", "food"],
      # ... more domains
    }
  }
}
```

Algorithm

```
def detect_keyword_transition(current_msg: str, recent_msgs: List[str]) -> float:
    score = 0.0

    # Check for explicit transition phrases
    for phrase in transition_phrases:
        if phrase.lower() in current_msg.lower():
            score += 0.8
            break

    # Extract domains from recent messages
    recent_domains = extract_domains(recent_msgs)
    current_domains = extract_domains([current_msg])

    # Calculate domain shift
    domain_overlap = len(recent_domains & current_domains)
    domain_shift = 1.0 - (domain_overlap / max(len(recent_domains), 1))

    score += domain_shift * 0.5

    return min(score, 1.0)
```

3. Hybrid Scoring

```
def calculate_hybrid_score(
    embedding_drift: float,
    keyword_score: float,
    message_count: int
) -> Tuple[float, str]:
    """
    Returns (score, method_used)
    """
    # Apply message count threshold
    if message_count < min_messages_before_topic_change:
        return (0.0, "threshold_not_met")

    # User command overrides everything
    if has_user_topic_command():
        return (1.0, "user_command")

    # High embedding drift is primary signal
    if embedding_drift > 0.6:
        return (embedding_drift, "embedding_high")

    # Combine signals for medium drift
    if embedding_drift > 0.3:
        combined = (0.7 * embedding_drift + 0.3 * keyword_score)
        if combined > 0.5:
            return (combined, "hybrid")

    # Strong keyword signal can trigger on its own
    if keyword_score > 0.8:
        return (keyword_score, "keyword_strong")

    # Fall back to LLM if inconclusive
    if 0.3 <= embedding_drift <= 0.5:
        llm_score = run_llm_detection()
        if llm_score > 0.7: # Higher threshold for LLM
            return (llm_score, "llm_fallback")

    return (max(embedding_drift, keyword_score), "no_change")
```

Database Schema Updates

New Tables

```
-- Store embeddings for messages
CREATE TABLE IF NOT EXISTS embeddings (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    node_id TEXT NOT NULL,
    message_hash TEXT NOT NULL,
    embedding BLOB NOT NULL, -- Stored as binary numpy array
    model_name TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(node_id) REFERENCES nodes(id),
    UNIQUE(message_hash, model_name)
);

-- Index for fast lookups
CREATE INDEX idx_embeddings_hash ON embeddings(message_hash);
CREATE INDEX idx_embeddings_node ON embeddings(node_id);

-- Store drift scores for analysis
CREATE TABLE IF NOT EXISTS drift_scores (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    from_node_id TEXT NOT NULL,
    to_node_id TEXT NOT NULL,
    drift_score REAL NOT NULL,
    detection_method TEXT, -- 'embedding', 'keyword', 'hybrid', 'llm'
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(from_node_id) REFERENCES nodes(id),
    FOREIGN KEY(to_node_id) REFERENCES nodes(id)
);
```

Integration Points

1. Configuration Integration

Update `episodic/config.py`:


```
# Add to default configuration
{
    "topic_detection_method": "hybrid", # or "llm_only", "embedding_only"
    "embedding_config": {
        "provider": "sentence-transformers",
        "model": "all-MiniLM-L6-v2",
        "cache_embeddings": true
    },
    "drift_config": {
        "algorithm": "cosine",
        "threshold": 0.35,
        "window_size": 5
    },
    "keyword_config": {
        "enabled": true,
        "weight": 0.3
    }
}
```

2. Topic Manager Integration

Modify `episodic/topics.py`:

```

class TopicManager:
    def __init__(self):
        self.prompt_manager = PromptManager()
        # Initialize hybrid detection components
        self.drift_detector = ConversationalDrift(
            embedding_provider=config.get("embedding_config.provider"),
            embedding_model=config.get("embedding_config.model"),
            distance_algorithm=config.get("drift_config.algorithm")
        )
        self.keyword_detector = KeywordDetector(
            config.get("keyword_config")
        )

    def detect_topic_change_hybrid(
        self,
        recent_messages: List[Dict[str, Any]],
        new_message: str,
        current_topic: Optional[Tuple[str, str]] = None
    ) -> Tuple[bool, Optional[str], Optional[Dict[str, Any]]]:
        """New hybrid detection method"""
        # Implementation here

```

3. CLI Integration

Add new commands:

```

# In episodic/cli.py
@app.command()
def topic_detection_config(
    method: str = typer.Option(None, help="Detection method: hybrid, llm_only, er"),
    threshold: float = typer.Option(None, help="Drift threshold (0.0-1.0)"),
    provider: str = typer.Option(None, help="Embedding provider")
):
    """Configure topic detection settings"""
    # Implementation

```

Example Scenarios

Scenario 1: Gradual Topic Drift

```
User: Tell me about machine learning
Assistant: [ML explanation]
User: What about neural networks specifically? <- Drift: 0.15 (same domain)
Assistant: [NN explanation]
User: How do transformers work? <- Drift: 0.18 (still ML)
Assistant: [Transformer explanation]
User: Speaking of transformation, I'm renovating my kitchen <- Drift: 0.72 (domain shift)
[TOPIC CHANGE DETECTED - Method: embedding_high]
```

Scenario 2: Explicit Transition

```
User: Can you explain Python decorators?
Assistant: [Decorator explanation]
User: Let's switch gears - what's a good pasta recipe? <- Keyword detected + drift
[TOPIC CHANGE DETECTED - Method: hybrid]
```

Scenario 3: Ambiguous Case

```
User: How do I debug Python code?
Assistant: [Debugging tips]
User: What about debugging life problems? <- Drift: 0.42 (medium)
[LLM FALLBACK - Checking with model]
[TOPIC CHANGE DETECTED - Method: llm_fallback]
```

Scenario 4: User Override

```
User: Tell me more about quantum physics
Assistant: [Physics explanation]
User: /new-topic Actually, let's talk about cooking
[TOPIC CHANGE DETECTED - Method: user_command]
```

Performance Considerations

Embedding Caching

- Store embeddings in database for reuse
- Use message hash as cache key
- Implement LRU cache in memory (limit: 1000 embeddings)
- Batch embedding generation when possible

Optimization Strategies

1. **Lazy Loading:** Only load ML models when hybrid detection is enabled
2. **Background Processing:** Generate embeddings asynchronously
3. **Batch Operations:** Process multiple messages together
4. **Model Selection:**
 - Fast: all-MiniLM-L6-v2 (22M params, 80MB)
 - Balanced: all-mpnet-base-v2 (110M params, 420MB)
 - High Quality: all-roberta-large-v1 (355M params, 1.4GB)

Resource Usage

- Memory: ~100-500MB depending on model
- CPU: Embedding generation takes ~50-200ms per message
- Storage: ~2KB per embedding (384-768 dimensions)

Migration Path

Phase 1: Infrastructure (Week 1)

1. Add database tables for embeddings
2. Implement embedding generation and caching
3. Add drift calculation algorithms
4. Create configuration options

Phase 2: Integration (Week 2)

1. Integrate with TopicManager
2. Add hybrid detection method
3. Implement fallback logic
4. Update CLI commands

Phase 3: Testing & Tuning (Week 3)

1. Run on test conversations
2. Tune thresholds and weights
3. Compare with LLM-only approach
4. Optimize performance

Phase 4: Rollout (Week 4)

1. Enable as opt-in feature
2. Collect metrics and feedback
3. Fine-tune based on usage
4. Make default if successful

Monitoring & Metrics

Key Metrics to Track

1. Accuracy Metrics

- Topic changes detected vs. ground truth
- False positive rate
- False negative rate

2. Performance Metrics

- Embedding generation time
- Total detection time
- Cache hit rate

3. Usage Metrics

- Detection method distribution
- LLM fallback frequency
- User override frequency

Debugging Tools

```
# Show drift scores for current conversation
episodic> /show-drift

# Analyze topic detection for specific range
episodic> /analyze-topics 1-20

# Export embeddings for visualization
episodic> /export-embeddings conversation.json
```

Future Enhancements

1. **Multi-modal Embeddings:** Support for code, images, tables
2. **Personalized Thresholds:** Learn user-specific topic preferences
3. **Topic Prediction:** Suggest potential topics based on conversation flow
4. **Embedding Fine-tuning:** Train custom embeddings on conversation data
5. **Real-time Visualization:** Show drift scores in UI as conversation progresses

Conclusion

This hybrid approach combines the best of multiple methods: - **Accuracy:** Embeddings capture semantic meaning better than keywords - **Speed:** No LLM calls for most detections

- **Cost:** Reduced API usage by 80-90% - **Control:** Users can override when needed - **Reliability:** Multiple fallback mechanisms

The system is designed to be modular, configurable, and extensible, allowing for easy experimentation and improvement over time.

...

Tests README

Source: tests/README.md

Episodic Test Suite

Comprehensive test suite for the Episodic CLI application, ensuring stability during development and modifications.

Test Structure

```
tests/
├── __init__.py           # Test package initialization
├── README.md             # This file
├── run_tests.py          # Test runner with colored output
├── test_cli.py           # CLI command and interface tests
├── test_config.py        # Configuration management tests
├── test_llm_integration.py # LLM integration and provider tests
├── test_prompt_manager.py # Prompt management system tests
└── test_caching.py       # Prompt caching functionality tests
```

Running Tests

Quick Start

Run all tests:

```
cd tests
python run_tests.py
```

Run only quick unit tests (recommended during development):

```
python run_tests.py quick
```

Test Coverage

Generate coverage report:

```
pip install coverage
python run_tests.py coverage
```

Individual Test Files

Run specific test modules:

```
python -m unittest test_cli
python -m unittest test_config
python -m unittest test_prompt_manager
python -m unittest test_llm_integration
python -m unittest test_caching
```

Run specific test classes:

```
python -m unittest test_cli.TestCLIHelpers
python -m unittest test_config.TestConfig
```

Run specific test methods:

```
python -m unittest test_cli.TestCLIHelpers.test_parse_flag_value
```

Using pytest (alternative)

If you have pytest installed:

```
pip install pytest
pytest tests/
pytest tests/test_cli.py -v
```

Test Categories

1. Unit Tests

CLI Tests (`test_cli.py`) - Command parsing and flag handling - Helper function validation - Command execution (mocked) - Session management - Configuration integration

Configuration Tests (`test_config.py`) - Config file creation and persistence - Value setting/getting with different types - Error handling for malformed files - Unicode and large data handling

Prompt Manager Tests (`test_prompt_manager.py`) - Prompt loading from files - YAML frontmatter parsing - Active prompt selection - Integration with config system

LLM Integration Tests (`test_llm_integration.py`) - Model string formatting - Provider-specific handling - Query execution (mocked) - Cost calculation

Caching Tests (`test_caching.py`) - Prompt caching implementation - Cache control application - Cost savings calculation - Provider-specific caching behavior

2. Integration Tests

The existing integration tests in the main `episodic/` directory cover: - Database operations (`test_db.py`) - Core data structures (`test_core.py`) - Visualization (`test_integration.py`) - Server and HTTP functionality (`test_server.py`)

3. Manual/Interactive Tests

Located in `episodic/` directory: - `test_interactive_features.py` - Interactive visualization tests - `test_interactive_features.py` - Interactive visualization and HTTP polling tests

Test Development Guidelines

Writing New Tests

1. **Follow naming conventions:** `test_*.py` for files, `test_*` for methods
2. **Use descriptive names:** `test_config_handles_unicode_values`

3. **Include docstrings:** Explain what each test validates
4. **Mock external dependencies:** Don't make real API calls or file operations
5. **Clean up after tests:** Use setUp/tearDown to manage test state

Test Structure Template

```
import unittest
from unittest.mock import patch, MagicMock

class TestNewFeature(unittest.TestCase):
    """Test description of the feature being tested."""

    def setUp(self):
        """Set up test environment."""
        # Initialize test data, mock objects, etc.
        pass

    def tearDown(self):
        """Clean up after tests."""
        # Restore state, clean up temp files, etc.
        pass

    def test_specific_behavior(self):
        """Test that specific behavior works correctly."""
        # Arrange
        # Act
        # Assert
        pass
```

Mocking Guidelines

- **Mock external APIs:** LLM providers, file system operations
- **Mock time-dependent operations:** Use fixed timestamps for consistency
- **Mock configuration:** Use temporary config for isolation
- **Preserve original behavior:** Store and restore original values

Coverage Goals

- **Minimum 80% code coverage** for new features

- **100% coverage** for critical paths (CLI commands, data handling)
- **Focus on edge cases:** Error conditions, malformed input, boundary values

Continuous Integration

Pre-commit Checks

Before committing changes, run:

```
python run_tests.py quick # Fast feedback
```

Full Test Suite

Before major changes or releases:

```
python run_tests.py # Complete test suite
python run_tests.py coverage # With coverage report
```

Test Performance

- **Quick tests** should complete in < 30 seconds
- **Full test suite** should complete in < 2 minutes
- **Individual test methods** should complete in < 1 second

Debugging Test Failures

Common Issues

1. **Import errors:** Check PYTHONPATH and module structure
2. **Mock setup:** Verify mock objects are configured correctly
3. **State pollution:** Ensure tests clean up after themselves
4. **Timing issues:** Use deterministic values instead of time-dependent ones

Debugging Tips

```
# Run single test with verbose output
python -m unittest test_cli.TestCLIHelpers.test_parse_flag_value -v

# Run with debugging
python -m pdb -m unittest test_cli.TestCLIHelpers.test_parse_flag_value

# Print debug information
python -c "import test_cli; test_cli.TestCLIHelpers().debug_method()"
```

Test Data Management

Temporary Files

Tests use `tempfile` module for temporary directories and files:

```
import tempfile
import shutil

def setUp(self):
    self.test_dir = tempfile.mkdtemp()

def tearDown(self):
    shutil.rmtree(self.test_dir, ignore_errors=True)
```

Mock Data

Common mock objects are defined in test files and can be reused:

```
def create_mock_llm_response(self, content="Test response", tokens=100):
    """Create a standardized mock LLM response."""
    mock_response = Mock()
    mock_response.choices[0].message.content = content
    mock_response.usage.prompt_tokens = tokens
    return mock_response
```

Adding Tests for New Features

When adding a new feature to Episodic:

1. **Add unit tests** for the core functionality
2. **Add integration tests** if it interacts with multiple components
3. **Add CLI tests** if it includes new commands or flags
4. **Update this README** if new test categories are needed
5. **Ensure tests pass** before submitting changes

Example workflow:

```
# 1. Write your feature
# 2. Write tests
# 3. Run tests
python run_tests.py quick

# 4. Check coverage
python run_tests.py coverage

# 5. Run full suite
python run_tests.py

# 6. Commit when all tests pass
```

This test suite ensures that your modifications to Episodic won't break existing functionality and provides confidence in the stability of the codebase.

Organized Tests Guide

Source: tests/ORGANIZED_TESTS.md

Organized Test Suite

All test files have been organized into a structured directory for better maintainability.

Directory Structure

```

tests/
├── __init__.py                # Test package
├── ORGANIZED_TESTS.md        # This file
├── README.md                 # Comprehensive test documentation
├── run_tests.py              # Enhanced test runner
├── cleanup_tests.py          # Test organization script
├──
├── Core Unit Tests:
├── test_cli.py               # CLI interface tests (some failing)
├── test_config.py            # Configuration management tests ✓
├── test_core.py              # Core data structures tests
├── test_db.py                # Database operations tests
├── test_integration.py        # System integration tests
├── test_llm_integration.py    # LLM provider tests ✓
├── test_prompt_manager.py     # Prompt management tests ✓
├── test_caching.py           # Prompt caching tests ✓
├── test_server.py            # HTTP server tests
├── test_websocket.py          # WebSocket functionality tests
├──
├── interactive/              # Manual/Interactive Tests
│   ├── test_interactive_features.py  # Interactive visualization tests
│   ├── test_websocket_browser.py     # Browser WebSocket tests
│   ├── test_websocket_integration.py  # WebSocket integration tests
│   └── test_native_visualization.py   # Native visualization tests
├──
└── legacy/                   # Legacy/One-off Tests
    ├── test_cache_comprehensive.py    # Comprehensive cache tests
    ├── test_prompt_caching_final.py   # Final prompt caching test
    ├── test_episodic.py               # General episodic tests
    ├── test_head_reference.py          # Head reference tests
    ├── test_roles.py                  # Role system tests
    ├── test_short_ids.py              # Short ID tests
    └── ... (20+ legacy test files)

```

Test Status

✅ Passing Tests

- `test_config.py` - Configuration management
- `test_prompt_manager.py` - Prompt loading and management
- `test_llm_integration.py` - LLM provider integration
- `test_caching.py` - Prompt caching functionality

⚠️ Failing Tests (Need Fixes)

- `test_cli.py` - 7 failures, 6 errors
 - Issues with CLI function behavior vs expected test behavior
 - Mock setup issues
 - Function signature mismatches

🔧 Integration Tests

- `test_core.py` - Core data structures
- `test_db.py` - Database operations
- `test_integration.py` - System integration
- `test_server.py` - HTTP server functionality
- `test_websocket.py` - WebSocket functionality

🎮 Interactive Tests (Manual)

- `test_interactive_features.py` - Requires human verification
- `test_websocket_browser.py` - Browser-based testing
- `test_websocket_integration.py` - Real-time WebSocket tests
- `test_native_visualization.py` - Visualization components

Running Tests

Quick Tests (Stable Only)

```
cd tests
python run_tests.py quick
```

All Tests

```
python run_tests.py
```

Specific Test Files

```
python -m unittest test_config -v
python -m unittest test_prompt_manager -v
```

With Coverage

```
pip install coverage
python run_tests.py coverage
```

Individual Categories

Core functionality only:

```
python -m unittest test_config test_prompt_manager test_llm_integration test_cach
```

Database and integration:

```
python -m unittest test_db test_core test_integration
```

Legacy tests:

```
python -m unittest discover legacy/ "test_*.py"
```

Benefits of Organization

1. **Clear Structure:** Tests are logically organized by functionality
2. **Easy Maintenance:** Related tests are grouped together
3. **Selective Testing:** Run only the tests you need
4. **Stability:** Separate failing tests from stable ones
5. **Documentation:** Clear categorization of test types

Test Development

Adding New Tests

1. Create tests in the main `tests/` directory for core functionality
2. Use `interactive/` for tests requiring human verification
3. Use `legacy/` for experimental or one-off tests

Fixing Failing Tests

1. Focus on `test_cli.py` first - it has the most failures
2. Update mocks to match actual function signatures
3. Verify expected vs actual behavior
4. Update test assertions as needed

Before Major Changes

```
# Run stable tests first
python run_tests.py quick

# If those pass, run full suite
python run_tests.py

# Generate coverage report
python run_tests.py coverage
```

This organized structure provides a solid foundation for maintaining test quality while you make significant modifications to the CLI application.

Scripts README

Source: scripts/README.md

Test Scripts for Topic Detection

This directory contains various test scripts for validating the topic detection and extraction system.

Usage

Run any script with:

```
/script scripts/filename.txt
```

After running, check results with:

```
/topics
```

Test Scripts

basic-topic-changes.txt

Tests clear, obvious topic shifts across different domains: - Movies → Quantum Physics → Baseball → Programming → Cooking - Should trigger mostly "high confidence" topic changes

gradual-drift.txt

Tests subtle topic transitions that flow naturally: - Science Fiction → AI → Quantum Computing → Physics → Biology - Should trigger "medium" or "low" confidence changes, or possibly no changes

false-positive-test.txt

Tests conversation that should NOT trigger topic changes: - All about semantic drift and topic detection - Should show minimal or no topic change indicators

rapid-switching.txt

Tests quick topic changes between unrelated subjects: - Weather → Physics → Cooking → AI → Sports → etc. - Should trigger mostly "high confidence" changes

edge-cases.txt

Tests unusual scenarios: - Greetings, philosophical questions, humor requests - Mixed expectations for topic change detection

meta-conversation.txt

Tests discussion about the system itself mixed with other topics: - System discussion → Space → System discussion → Food - Interesting test of how LLM handles meta-conversations

Expected Outcomes

- **High confidence:** Clear domain shifts (movies → sports)
- **Medium confidence:** Related but different focus (physics → chemistry)
- **Low confidence:** Subtle shifts within domain (sci-fi movies → AI movies)
- **No change:** Natural follow-ups and elaborations

Tips

- Run `/set debug true` to see detailed topic extraction info
- Clear topics between tests if needed
- Compare topic extraction quality across different scripts

Topic Testing README

Source: scripts/topic/README.md

Topic Detection Test Scripts

This directory contains test scripts for validating topic detection behavior in Episodic.

Purpose

These scripts test various conversational patterns to ensure topic detection correctly identifies when conversations shift to new subjects vs. when they're exploring different aspects of the same topic.

Test Scripts

test-gradual-progression.txt

- **Expected:** 1 topic
- **Tests:** Natural progression within a single domain (Python basics → advanced)
- **Validates:** Related concepts stay together

test-explicit-transitions.txt

- **Expected:** 4 topics
- **Tests:** Recognition of explicit transition phrases ("I have a different question", "Changing topics")
- **Validates:** Clear verbal cues trigger topic changes

test-related-domains.txt

- **Expected:** 2-3 topics
- **Tests:** Boundaries between related but distinct fields (human medicine → veterinary → biology)
- **Validates:** Different expertise domains are separated

test-depth-exploration.txt

- **Expected:** 1 topic
- **Tests:** Deep dive from general to extremely specific (ML basics → specific CNN architectures)
- **Validates:** Depth exploration doesn't trigger false splits

test-conversation-flow.txt

- **Expected:** 3 topics
- **Tests:** Natural conversation with soft transitions (Japan travel → Japanese cooking → nutrition)
- **Validates:** Natural flow with logical but distinct topics

test-mixed-patterns.txt

- **Expected:** 4-5 topics
- **Tests:** Various transition types in one conversation
- **Validates:** Different transition patterns are handled correctly

test-ambiguous-transitions.txt

- **Expected:** 2-3 topics
- **Tests:** Edge cases where topic boundaries are unclear (France geography → French cuisine → wine regions)
- **Validates:** Ambiguous cases are handled reasonably

Usage

Run any test script with:

```
/script scripts/topic/test-name.txt
```

Then check results with:

```
/topics
```

Configuration

These tests assume: - Topic detection model: `gpt-3.5-turbo` (can be changed with `/set topic_detection_model`) - Topic detection v3 prompt is enabled (default) - Default minimum messages before topic change threshold