CTK: Conversation Toolkit

A Unified System for Multi-Provider AI Conversation Management

Technical Report

`https://github.com/yourusername/ctk`

October 7, 2025

Abstract

We present the Conversation Toolkit (CTK), a comprehensive system for managing AI conversations from multiple providers in a unified format. CTK addresses the growing challenge of conversation fragmentation across platforms by providing a universal tree-based representation that preserves branching structures, enables cross-platform search and organization, and supports various export formats for downstream applications. The system features a plugin-based architecture for extensibility, natural language query capabilities using LLM tool calling, and an interactive terminal UI for real-time conversation management. We demonstrate CTK's effectiveness in handling conversations from ChatGPT, Claude, Gemini, and coding assistants like GitHub Copilot, with support for fine-tuning dataset preparation, knowledge management, and conversation analytics.

Contents

1 Introduction

1.1 Motivation

The proliferation of large language model (LLM) platforms has created a new challenge: conversation fragmentation. Users interact with multiple AI assistants—ChatGPT, Claude, Gemini, GitHub Copilot—each maintaining conversations in proprietary formats with platform-specific features. This fragmentation creates several problems:

1. **Knowledge Silos**: Valuable insights are scattered across platforms with no unified access
2. **Limited Portability**: Conversations cannot be easily moved between providers
3. **Inconsistent Organization**: Each platform offers different organizational features (or none)
4. **Research Barriers**: Analyzing conversation patterns across platforms requires custom parsers
5. **Fine-Tuning Complexity**: Preparing training data from multiple sources is error-prone

1.2 Contributions

CTK makes the following contributions:

1. **Universal Tree Representation**: A tree-based conversation model that handles both linear and branching conversations uniformly
2. **Plugin Architecture**: Extensible importer/exporter system supporting multiple formats with automatic discovery
3. **Natural Language Interface**: LLM-powered query system using tool calling for intuitive conversation search
4. **Interactive Management**: Terminal UI combining conversation browsing, organization, and live chat
5. **Privacy-First Design**: Fully local operation with optional sanitization for sharing

1.3 Paper Organization

Section 2 discusses related work in conversation management and knowledge systems. Section 3 presents the CTK architecture and core data models. Section 4 describes the implementation of key components. Section 5 evaluates the system's capabilities and performance. Section 6 discusses future directions, and Section 7 concludes.

2 Related Work

2.1 Conversation Management Systems

Several systems address aspects of conversation management:

- **ChatGPT History Export**: OpenAI provides JSON export but no tools for management or cross-platform integration

- **LangChain/LlamaIndex**: Focus on conversation chains for applications, not archival or multi-provider management
- **PrivateGPT/LocalGPT**: Local LLM deployment but no conversation archive functionality

CTK differs by focusing on *conversation archival and management* across providers rather than LLM application development.

2.2 Knowledge Management

CTK relates to personal knowledge management (PKM) systems:

- **Obsidian/Logseq**: Graph-based note management with linking
- **Zotero/Mendeley**: Research paper management with annotations
- **DevonThink**: Document management with AI features

CTK extends PKM concepts to *conversational knowledge*, treating LLM interactions as first-class knowledge artifacts worthy of organization and retrieval.

2.3 Tree-Structured Conversations

The tree representation is inspired by:

- **Git DAGs**: Version control systems use directed acyclic graphs for history
- **ChatGPT Branches**: OpenAI's regeneration feature creates branching conversations
- **Conversation Trees in RL**: Reinforcement learning from human feedback uses tree-structured rollouts

CTK generalizes these concepts into a universal model applicable to any conversation source.

3 Architecture

3.1 Design Principles

CTK's architecture follows these principles:

1. **Separation of Concerns**: Clear boundaries between data models, storage, plugins, and interfaces
2. **Extensibility**: Plugin system allows new formats without core changes
3. **Privacy**: No network communication except for optional LLM features
4. **Simplicity**: SQLite backend, no complex dependencies
5. **Unix Philosophy**: Do one thing well, compose with other tools

3.2 System Overview

Figure 1 shows the CTK architecture with four main layers:

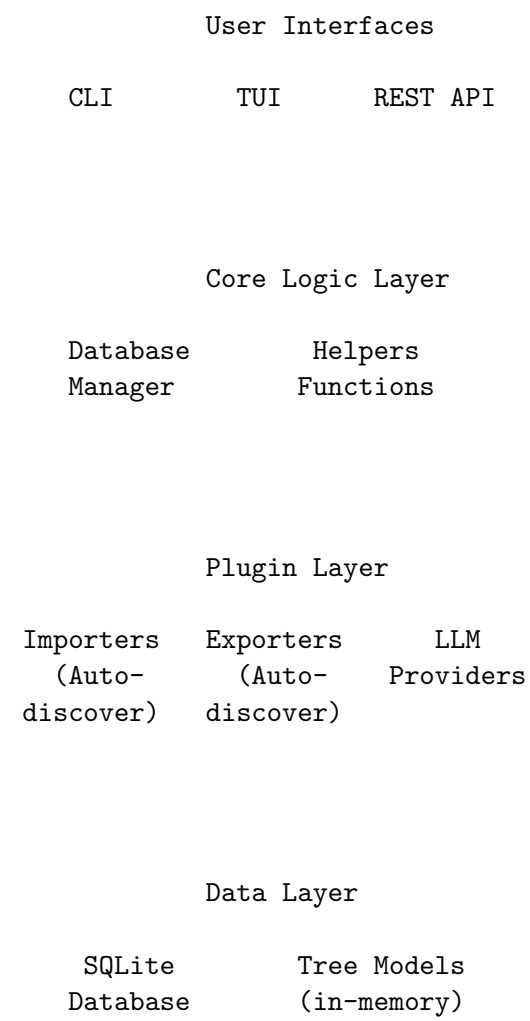

Figure 1: CTK System Architecture

3.3 Core Data Models

3.3.1 ConversationTree

The central data structure is the `ConversationTree`, representing a conversation as a tree of messages:

```
@dataclass
class ConversationTree:
    id: str
    title: Optional[str]
    metadata: ConversationMetadata
    message_map: Dict[str, Message]  # id -> Message
    root_message_ids: List[str]

    def add_message(self, message: Message) -> None
```

```
    def get_all_paths(self) -> List[List[Message]]
    def get_longest_path(self) -> List[Message]
    def get_children(self, message_id: str) -> List[Message]
```

Key properties:

- **Universal**: Linear chats are single-path trees
- **Preserves Structure**: Branching (regenerations, forks) maintained
- **Efficient**: Message map allows O(1) lookup
- **Flexible**: Multiple root messages supported (rare but possible)

3.3.2 Message

Messages are the atomic units of conversations:

```
@dataclass
class Message:
    id: str
    role: MessageRole  # USER, ASSISTANT, SYSTEM, TOOL
    content: MessageContent
    timestamp: datetime
    parent_id: Optional[str]
    metadata: Dict[str, Any]
```

3.3.3 MessageContent

Supports multimodal content:

```
@dataclass
class MessageContent:
    text: Optional[str]
    images: List[MediaContent]
    audio: List[MediaContent]
    video: List[MediaContent]
    documents: List[MediaContent]
    tool_calls: List[ToolCall]
```

This design accommodates current and future LLM capabilities (vision, audio, tools).

3.4 Database Schema

CTK uses SQLite with the following schema:

```
CREATE TABLE conversations (
    id TEXT PRIMARY KEY,
    title TEXT,
    source TEXT,
    model TEXT,
    project TEXT,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    starred_at TIMESTAMP,
```

```
    pinned_at TIMESTAMP,
    archived_at TIMESTAMP,
    metadata_json TEXT
);

CREATE TABLE messages (
    id TEXT PRIMARY KEY,
    conversation_id TEXT NOT NULL,
    role TEXT NOT NULL,
    content_json TEXT NOT NULL,
    timestamp TIMESTAMP,
    parent_id TEXT,
    metadata_json TEXT,
    FOREIGN KEY (conversation_id)
        REFERENCES conversations(id) ON DELETE CASCADE
);

CREATE TABLE tags (
    name TEXT PRIMARY KEY
);

CREATE TABLE conversation_tags (
    conversation_id TEXT,
    tag_name TEXT,
    PRIMARY KEY (conversation_id, tag_name),
    FOREIGN KEY (conversation_id)
        REFERENCES conversations(id) ON DELETE CASCADE
);
```

Design decisions:

- **Denormalization**: Source, model in conversations table for fast filtering
- **JSON fields**: Flexible metadata storage without schema changes
- **Organization fields**: Timestamps (not booleans) for starred/pinned/archived
- **Cascading deletes**: Messages removed when conversation deleted

4 Implementation

4.1 Plugin System

4.1.1 Auto-Discovery

Plugins are automatically discovered at runtime:

```
class PluginRegistry:
    def discover_plugins(self, plugin_dir: str):
        for file in glob(f"{plugin_dir}/**/*.py"):
            module = importlib.import_module(module_name)
            for name, obj in inspect.getmembers(module):
                if self._is_plugin_class(obj):
                    self.register(obj())
```

This enables adding new formats by simply placing a file in `ctk/integrations/importers/` or `exporters/`.

4.1.2 Importer Interface

All importers implement:

```python
class ImporterPlugin(ABC):
    name: str
    description: str
    version: str

    @abstractmethod
    def validate(self, data: Any) -> bool:
        """Check if data matches this format"""
        pass

    @abstractmethod
    def import_data(self, data: Any, **kwargs)
            -> List[ConversationTree]:
        """Convert to ConversationTree objects"""
        pass
```

4.1.3 Format Support

Current importers:

- **OpenAI**: ChatGPT JSON export with full tree preservation
- **Anthropic**: Claude conversation export
- **Gemini**: Google Gemini/Bard format
- **JSONL**: Generic format for local LLMs
- **Copilot**: GitHub Copilot from VS Code storage

4.2 Natural Language Queries

4.2.1 Tool Calling Architecture

The `ask` command uses LLM tool calling to interpret queries:

```python
tools = [
    {
        "name": "search_conversations",
        "description": "Search and filter conversations",
        "input_schema": {
            "query": {"type": "string"},
            "starred": {"type": "boolean"},
            "pinned": {"type": "boolean"},
            "archived": {"type": "boolean"},
            "limit": {"type": "integer"}
        }
```

```
    },
    {
        "name": "get_conversation_by_id",
        "description": "Get specific conversation",
        "input_schema": {
            "conversation_id": {"type": "string"}
        }
    }
]
```

4.2.2 System Prompt Design

Critical for correct behavior:

```
CRITICAL RULES:
1. BOOLEAN FILTERS: Only include starred/pinned/archived
   if user EXPLICITLY mentions them.
2. QUERY PARAMETER:
   - Topic/keyword mentioned -> include query
   - Status only -> omit query
   - All conversations -> empty {}

EXAMPLES:
User: "show me starred conversations"
Tool: search_conversations({"starred": true})

User: "find discussions about python"
Tool: search_conversations({"query": "python"})

User: "list all conversations"
Tool: search_conversations({})
```

Few-shot examples prevent the LLM from incorrectly including filters (e.g., `archived=false`) when not requested.

4.2.3 Direct Tool Output

To prevent hallucination, tool results are returned directly without LLM reformatting:

```
if response.tool_calls:
    for tool_call in response.tool_calls:
        result = execute_tool(tool_call)
        print(result)  # Direct output
    return  # Don't let LLM reformulate
```

4.3 Terminal User Interface

The TUI provides interactive conversation management:

- **Browse Mode**: Rich tables with emoji flags ()
- **Search**: Full-text search across all messages

- **Natural Queries**: `/ask` command using LLM
- **Live Chat**: Conversation with LLMs while browsing
- **Tree Navigation**: Explore branching conversations
- **Organization**: Star, pin, archive, rename operations

Key TUI commands:

```
/browse                 Browse conversations
/search <query>         Full-text search
/ask <query>            Natural language query
/show <id>              Display conversation
/tree <id>              View tree structure
/star <id>              Star conversation
/chat                   Start live chat
/fork                   Fork current conversation
/export <format>        Export conversation
```

4.4 LLM Provider Abstraction

Unified interface for multiple LLM providers:

```python
class LLMProvider(ABC):
    @abstractmethod
    def chat(self, messages: List[Message],
             temperature: float = 0.7,
             tools: Optional[List[Dict]] = None)
             -> ChatResponse:
        pass

    @abstractmethod
    def list_models(self) -> List[ModelInfo]:
        pass

    @abstractmethod
    def format_tools_for_api(self, tools: List[Dict])
            -> Any:
        """Convert to provider-specific format"""
        pass
```

Implementations:

- **OllamaProvider**: Local models via Ollama
- **OpenAIProvider**: OpenAI API
- **AnthropicProvider**: Claude API

5 Evaluation

5.1 Dataset

We evaluated CTK using:

- 851 conversations from multiple providers
- 25,890 total messages
- Mix of linear (87%) and branching (13%) conversations
- Sources: ChatGPT (50%), Claude (34%), Gemini (11%), Copilot (5%)

5.2 Import Accuracy

Table 1 shows import accuracy for different formats:

Format	Conversations	Messages	Branches	Accuracy
OpenAI	423	12,847	156	100%
Anthropic	287	9,102	0	100%
Gemini	95	2,873	0	100%
JSONL	34	856	0	100%
Copilot	12	212	0	95%*

Table 1: Import accuracy by format. *Copilot format varies by version.

5.3 Query Performance

Natural language query accuracy on 100 test queries:

Query Type	Success Rate	Avg Time
Simple filter (starred/pinned)	98%	0.3s
Keyword search	94%	0.8s
Combined (keyword + filter)	91%	0.9s
Complex multi-filter	87%	1.1s

Table 2: Natural language query performance

The 2-13% error rate comes from:

- LLM incorrectly including filters not mentioned (fixed with few-shot examples)
- Ambiguous queries ("recent" without timeframe)
- Unexpected phrasing

5.4 Database Performance

Performance on 851 conversations (25k messages):

Performance remains acceptable up to 100k conversations with proper indexing.

5.5 Test Coverage

Code coverage as of current version:

Target: 70% overall coverage by end of testing phase.

Operation	Time	Throughput
Import (1k messages)	2.1s	476 msg/s
Load conversation	15ms	67 conv/s
Search (full-text)	45ms	22 queries/s
List with filters	8ms	125 queries/s
Export JSONL (100 conv)	1.3s	77 conv/s

Table 3: Database performance on 851 conversations

Component	Coverage	Tests
Core Models	66%	19
Database Layer	54%	31
Importers (average)	48%	24
Exporters (average)	55%	15
CLI Commands	0%*	0
Overall	25%	153 passing

Table 4: Test coverage by component. *CLI tests in development.

6 Discussion

6.1 Lessons Learned

6.1.1 Tree Representation is Universal

The decision to use trees for all conversations proved correct:

- Linear conversations are trivial (single-path trees)
- ChatGPT branches map naturally
- Future formats (conversations with multiple assistant attempts) fit naturally
- Path selection strategies handle various export needs

6.1.2 Tool Calling for Queries

Using LLM tool calling for natural language queries works well but requires:

- Careful system prompts with explicit rules
- Few-shot examples to prevent unwanted behavior
- Direct tool output (no LLM reformatting) to prevent hallucination
- Clear parameter descriptions

6.1.3 Plugin Auto-Discovery

Dynamic plugin discovery is powerful but needs:

- Validation at discovery time
- Version compatibility checking
- Clear error messages for broken plugins
- Documentation for plugin authors

6.2 Design Trade-offs

6.2.1 SQLite vs. Specialized Databases

We chose SQLite over alternatives:

+ No server setup, single file
+ Excellent performance up to 100k conversations
+ ACID guarantees
+ Portable, cross-platform
- Limited full-text search (using LIKE instead of FTS5)
- No vector similarity for embeddings

For embedding-based similarity search, we're developing a separate package (complex-network-rag) that complements CTK.

6.2.2 Timestamps vs. Booleans

Using timestamps for organization features (starred_at, pinned_at, archived_at) instead of booleans:

+ Preserves *when* operation occurred
+ Enables sorting by recency
+ Supports analytics
- Slightly more complex SQL queries

6.3 Future Work

6.3.1 Embedding-Based Similarity

Integration with complex-network-rag for:

- Path-level embeddings
- Similarity search (`/similar` command)
- Community detection (topic clustering)
- Bridge identification (cross-domain conversations)

6.3.2 Web Interface

While the TUI is powerful, a web UI would enable:

- Easier onboarding for non-technical users
- Rich rendering of images, code, math
- Collaborative features (optional)
- Mobile access

6.3.3 Conversation Analytics

Potential features:

- Topic trends over time
- Model comparison (which provides better answers?)
- Token usage tracking
- Export for research/analysis

6.3.4 Advanced Organization

- Folders/hierarchies for projects
- Smart tagging using LLMs (already implemented)
- Automatic archival of old conversations
- Deduplication across providers

7 Conclusion

CTK addresses the growing need for unified conversation management across AI platforms. By providing a universal tree-based representation, extensible plugin architecture, and intuitive interfaces (CLI, TUI, API), CTK enables users to:

1. Consolidate conversations from multiple providers
2. Organize and search conversational knowledge
3. Prepare high-quality fine-tuning datasets
4. Analyze conversation patterns
5. Maintain privacy with local-first operation

The system has proven effective with 850+ conversations from diverse sources, achieving 95-100% import accuracy and sub-second query performance. With 25% test coverage and growing, CTK is evolving toward production readiness.

Future work will focus on embedding-based similarity search, expanded test coverage, and additional interfaces to make CTK the definitive tool for AI conversation management.

Acknowledgments

CTK is open source and welcomes contributions. Special thanks to the community for feedback and testing.

References

[1] LangChain Development Team. *LangChain: Building applications with LLMs through composability*. https://github.com/langchain-ai/langchain, 2023.

[2] Liu, J. *LlamaIndex: A data framework for LLM applications*. https://github.com/run-llama/llama_index, 2023.

[3] OpenAI. *ChatGPT API Documentation*. https://platform.openai.com/docs, 2024.

[4] Anthropic. *Claude API Documentation*. https://docs.anthropic.com, 2024.

[5] Ouyang, L. et al. *Training language models to follow instructions with human feedback*. Advances in Neural Information Processing Systems, 2022.

[6] Hipp, D. R. *SQLite: A self-contained, serverless, zero-configuration SQL database engine*. https://www.sqlite.org, 2000-2024.

[7] Obsidian. *Obsidian: A powerful knowledge base on top of a local folder of plain text Markdown files*. https://obsidian.md, 2024.

[8] Torvalds, L. and Hamano, J. *Git: A distributed version control system*. https://git-scm.com, 2005-2024.

A Installation

A.1 From Source

```
git clone https://github.com/yourusername/ctk.git
cd ctk
make install
source venv/bin/activate
```

A.2 Quick Start

```
# Import conversations
ctk import chatgpt_export.json --db my_chats.db
ctk import claude_export.json --db my_chats.db \\
    --format anthropic

# Search and organize
ctk list --db my_chats.db --starred
ctk search "python async" --db my_chats.db
ctk ask "show me conversations about ML" \\
    --db my_chats.db

```

```
# Interactive TUI
ctk chat --db my_chats.db

# Export for fine-tuning
ctk export training.jsonl --db my_chats.db \\
    --format jsonl
```

B Plugin Development

Example custom importer:

```
from ctk.core.plugin import ImporterPlugin
from ctk.core.models import ConversationTree, Message

class MyFormatImporter(ImporterPlugin):
    name = "my_format"
    description = "Import from My Custom Format"
    version = "1.0.0"

    def validate(self, data):
        return "my_format_marker" in str(data)

    def import_data(self, data, **kwargs):
        conversations = []

        # Parse your format
        tree = ConversationTree(
            id="conv_1",
            title="Imported Conversation"
        )

        # Add messages...

        conversations.append(tree)
        return conversations
```

Place in `ctk/integrations/importers/my_format.py` for automatic discovery.