AI Gateway Frontend - Complete Development Roadmap

Executive Summary

This roadmap details the development of a local desktop AI Gateway application for Ubuntu 24.04. The application provides a unified interface for interacting with OpenRouter.ai models, featuring conversation management, model switching, file uploads, and PDF export capabilities.

Estimated Timeline: 12-15 working days **Primary Developer**: Single hobbyist using LLM-assisted coding **Tech Stack**: Python 3.11+, CustomTkinter, LangChain, OpenRouter API

Technology Stack & Justification

Core Framework Selection

GUI Framework: CustomTkinter

- **Why**: Built on Python's standard Tkinter, making it familiar to LLMs trained on extensive Python documentation
- **LLM-Friendly**: Simpler API than PyQt6, more predictable code generation
- **Aesthetics**: Modern, clean widgets out-of-the-box with dark/light theme support
- **Ubuntu Compatibility**: Native support, minimal dependencies
- **Learning Curve**: Gentle for hobbyist developers

Backend: LangChain

- **Why**: Required by constraints; provides model abstraction and message formatting
- **OpenRouter Integration**: Clean API wrapper for multiple model providers
- **Tool Support**: Built-in function calling and file handling capabilities
- **Conversation Memory**: Native conversation history management

Supporting Libraries

Library	Purpose	Justification
`langchain-openai`	OpenRouter API integration	Direct ChatOpenAI interface compatible with OpenRouter
`python-dotenv`	Environment/config management	Secure API key storage
`Pillow (PIL)`	Image handling for GUI	Icons, logo display
`reportlab`	PDF generation	High-quality PDF export with typography control
`ttkbootstrap`	Additional UI components	Enhanced widgets for settings panels
`json` (stdlib)	Conversation persistence	Standard, human-readable format
`pathlib` (stdlib)	File system operations	Modern path handling

Library	Purpose	Justification
`datetime` (stdlib)	Timestamps	Conversation metadata

Not Used

- **SQLite/Database**: JSON files provide simpler debugging and manual editing for a single-user local app
- **PyQt6/PySide6**: Overkill for this scope; licensing complexity; harder for LLMs to generate correct code
- **Electron/Web**: Not a web app per constraints
- **FastAPI/Flask**: No server component needed for local desktop app

---

Project Architecture Overview

```
ai-gateway-frontend/
├── main.py                         # Entry point, app initialization
├── config/
│   ├── settings.py                 # Settings management, API key handling
│   └── models.json                 # Available OpenRouter models list
├── gui/
│   ├── main_window.py              # Primary application window
│   ├── chat_panel.py               # Chat interface and message display
│   ├── sidebar.py                  # Conversation list and folder tree
│   ├── settings_window.py          # Settings/preferences dialog
│   └── prompt_manager.py           # System prompt management UI
├── core/
│   ├── conversation.py             # Conversation data model
│   ├── langchain_handler.py        # LangChain integration and API calls
│   ├── file_handler.py             # File upload and attachment handling
│   └── model_manager.py            # Model selection and parameter management
├── storage/
│   ├── conversation_storage.py     # JSON persistence layer
│   └── folder_manager.py           # Folder/project organization
├── export/
│   └── pdf_exporter.py             # PDF generation with reportlab
├── data/
│   ├── conversations/              # JSON conversation files
│   │   ├── project_a/
│   │   └── project_b/
│   └── prompts/                    # Saved system prompts
└── assets/
    └── icons/                      # UI icons and images
```

Implementation Roadmap

Phase 1: Foundation & Core Infrastructure (Days 1-3)

Day 1: Project Setup & Configuration System

Work Package 1.1: Project Initialization

- **Duration**: 2 hours
- **Deliverables**:

    - Project directory structure
    - Virtual environment setup
    - `requirements.txt` with all dependencies
    - Basic `.gitignore`

LLM Prompt:

```
Create a Python project structure for an AI Gateway desktop application on
Ubuntu 24.04. Set up:
1. A virtual environment setup script (setup.sh)
2. A requirements.txt file including: customtkinter, langchain, langchain-
openai, python-dotenv, Pillow, reportlab, requests
3. A complete directory structure matching this layout: [paste architecture tree
above]
4. Create empty __init__.py files in all package directories
5. Add a .gitignore file that excludes: venv/, data/, __pycache__/, *.pyc, .env,
*.log

Include installation instructions for Ubuntu 24.04 in a README.md.
```

Work Package 1.2: Settings & Configuration Manager

- **Duration**: 4 hours
- **Deliverables**:
    - `config/settings.py` with Settings class
    - Environment variable loading (.env file handling)
    - Encrypted API key storage mechanism
    - Default configuration values

LLM Prompt:

```
Create a Python settings manager (config/settings.py) for a desktop AI
application that:
1. Uses python-dotenv to load configuration from a .env file
2. Implements a Settings singleton class with these attributes:
   - openrouter_api_key (stored encrypted using cryptography.fernet)
   - default_model (string)
   - default_temperature (float, 0.0-2.0)
   - theme (string: "dark" or "light")
   - conversations_directory (pathlib.Path)
   - prompts_directory (pathlib.Path)
3. Provides methods: load_settings(), save_settings(), update_api_key(key)
4. Creates necessary directories if they don't exist
5. Includes error handling for missing or corrupted config files
6. Add the cryptography library to requirements if needed

Use type hints and include docstrings.
```

Work Package 1.3: Models Configuration

- **Duration**: 2 hours
- **Deliverables**:
    - `config/models.json` with OpenRouter model definitions
    - Model metadata (name, context length, pricing, capabilities)

LLM Prompt:

```
Create a JSON configuration file (config/models.json) that lists popular
OpenRouter.ai models with the following structure for each model:
{
  "model_id": "openai/gpt-4-turbo",
  "display_name": "GPT-4 Turbo",
  "provider": "OpenAI",
  "context_window": 128000,
  "supports_function_calling": true,
  "supports_vision": false,
  "input_price_per_1k": 0.01,
  "output_price_per_1k": 0.03,
  "description": "Most capable GPT-4 model"
}

Include at least 15 popular models from different providers (OpenAI, Anthropic,
Google, Meta, Mistral). Research current OpenRouter model IDs and pricing.
Organize them in a list under a "models" key.
```

Day 2: Data Models & Storage Layer

Work Package 2.1: Conversation Data Model

- **Duration**: 3 hours
- **Deliverables**:
  - `core/conversation.py` with Conversation and Message classes
  - Data validation and serialization methods

LLM Prompt:

```
Create Python data models (core/conversation.py) for an AI chat application with
these requirements:

1. Message class with attributes:
   - role (string: "user", "assistant", "system")
   - content (string)
   - timestamp (datetime)
   - model_used (string, optional)
   - attachments (list of file paths, optional)
   - token_count (int, optional)

2. Conversation class with attributes:
   - conversation_id (UUID)
   - title (string)
   - created_at (datetime)
   - updated_at (datetime)
   - messages (list of Message objects)
   - folder_path (string, e.g., "project_a/subfolder")
   - current_model (string)
   - current_temperature (float)
   - system_prompt (string, optional)
   - metadata (dict for extensibility)

3. Methods for both classes:
   - to_dict() for JSON serialization
   - from_dict() class method for deserialization
   - __repr__ for debugging

Use dataclasses or pydantic for clean implementation. Include type hints and
docstrings.
```

Work Package 2.2: JSON Persistence Layer

- **Duration**: 3 hours
- **Deliverables**:
    - `storage/conversation_storage.py` with save/load operations
    - Atomic file writing (to prevent corruption)
    - Auto-backup mechanism

LLM Prompt:

```
Create a conversation storage system (storage/conversation_storage.py) that:

1. Implements a ConversationStorage class with methods:
   - save_conversation(conversation: Conversation) -> None
     * Saves to data/conversations/{folder_path}/{conversation_id}.json
     * Uses atomic writes (write to temp file, then rename)
     * Creates directories as needed

   - load_conversation(conversation_id: str) -> Conversation
     * Loads and deserializes JSON
     * Handles file not found gracefully

   - list_conversations(folder_path: str = "") -> List[Dict]
     * Returns conversation metadata (id, title, updated_at) without full
content
     * Sorted by updated_at descending

   - delete_conversation(conversation_id: str) -> bool

   - search_conversations(query: str) -> List[Conversation]
     * Simple text search in titles and message content

2. Include error handling for corrupted JSON files
3. Use pathlib for all file operations
4. Add logging for file operations

Use type hints and comprehensive docstrings.
```

Work Package 2.3: Folder Management System

- **Duration**: 2 hours
- **Deliverables**:
    - `storage/folder_manager.py` with folder CRUD operations
    - Folder tree traversal utilities

LLM Prompt:

```
Create a folder management system (storage/folder_manager.py) that:

1. Implements a FolderManager class with methods:
   - create_folder(path: str) -> bool
     * Creates nested folder structure (e.g., "project_a/experiments/test_1")

   - rename_folder(old_path: str, new_path: str) -> bool
     * Moves all conversations to new location

   - delete_folder(path: str, delete_conversations: bool = False) -> bool
     * Optionally deletes or orphans conversations

   - list_folders() -> Dict[str, Any]
```

```
      * Returns nested dict representing folder tree structure

   - get_folder_stats(path: str) -> Dict
      * Returns conversation count, total messages, disk usage

2. All operations should update conversation metadata accordingly
3. Include validation to prevent invalid folder names (no special characters)
4. Use pathlib and os for file system operations

Include comprehensive error handling and docstrings.
```

Day 3: LangChain Integration & Model Manager

Work Package 3.1: LangChain Handler

- **Duration**: 4 hours
- **Deliverables**:
    - `core/langchain_handler.py` with OpenRouter integration
    - Streaming response support
    - Error handling for API failures

LLM Prompt:

```
Create a LangChain integration module (core/langchain_handler.py) that:

1. Implements a LangChainHandler class with:
   - __init__(api_key: str)
   - send_message(
         messages: List[Message],
         model: str,
         temperature: float = 0.7,
         stream: bool = True,
         tools: List = None
     ) -> Generator[str, None, None] or str
     * Uses ChatOpenAI from langchain-openai
     * Configures base_url for OpenRouter: "https://openrouter.ai/api/v1"
     * Handles streaming vs non-streaming responses
     * Returns chunks for streaming, full response otherwise

   - estimate_tokens(text: str) -> int
      * Simple approximation (chars / 4)

   - supports_function_calling(model: str) -> bool
      * Checks model capabilities from models.json

2. Use langchain-openai's ChatOpenAI class with these parameters:
   - model_name=model
   - openai_api_key=api_key
   - openai_api_base="https://openrouter.ai/api/v1"
   - temperature=temperature
   - streaming=stream

3. Convert Message objects to LangChain message format
4. Handle API errors gracefully (rate limits, authentication, network)
5. Include retry logic with exponential backoff

Use type hints and add comprehensive error messages.
```

Work Package 3.2: Model Manager

- **Duration**: 3 hours
- **Deliverables**:
  - `core/model_manager.py` for model selection and metadata
  - Model filtering and search capabilities

LLM Prompt:

```
Create a model management system (core/model_manager.py) that:

1. Implements a ModelManager class with:
   - __init__() that loads config/models.json

   - get_all_models() -> List[Dict]
     * Returns all available models

   - get_model_by_id(model_id: str) -> Dict
     * Returns specific model metadata

   - filter_models(
       supports_function_calling: bool = None,
       supports_vision: bool = None,
       provider: str = None,
       max_price: float = None
     ) -> List[Dict]
     * Filters models by capabilities and constraints

   - search_models(query: str) -> List[Dict]
     * Searches model names and descriptions

   - get_model_display_name(model_id: str) -> str
     * Returns user-friendly name

   - estimate_cost(model_id: str, input_tokens: int, output_tokens: int) ->
float
     * Calculates approximate cost

2. Cache the models.json data in memory
3. Include validation for model_id existence
4. Add methods to group models by provider

Use type hints and docstrings.
```

Work Package 3.3: File Handler

- **Duration**: 1 hour
- **Deliverables**:
  - `core/file_handler.py` for file attachment processing
  - Base64 encoding for API submission

LLM Prompt:

```
Create a file handling module (core/file_handler.py) that:

1. Implements a FileHandler class with:
   - attach_file(file_path: str) -> Dict
     * Validates file exists and size < 10MB
     * Returns dict with: name, size, path, mime_type

   - encode_file_for_api(file_path: str) -> str
     * Reads file and returns base64 encoded string
   - get_file_info(file_path: str) -> Dict
     * Returns metadata: name, size, extension, mime_type

   - validate_file(file_path: str) -> Tuple[bool, str]
     * Checks if file is valid for upload
     * Returns (is_valid, error_message)

2. Support common file types: txt, pdf, docx, png, jpg, json, csv, py
3. Include size limits and type validation
4. Use mimetypes library for MIME type detection

Include error handling for missing files and permission issues.
```

Phase 2: GUI Foundation (Days 4-6)

Day 4: Main Window & Application Shell

Work Package 4.1: Main Application Window

- **Duration**: 4 hours
- **Deliverables**:
  - `gui/main_window.py` with primary window layout
  - Menu bar with File/Edit/View/Help menus
  - Window state persistence (size, position)

LLM Prompt:

```
Create the main application window (gui/main_window.py) using CustomTkinter:

1. Implement a MainWindow class inheriting from customtkinter.CTk:
   - Window title: "AI Gateway"
   - Default size: 1200x800
   - Minimum size: 800x600
   - Dark theme by default

2. Layout with three main sections (using grid):
   - Left sidebar (20% width): conversation list and folders
   - Center panel (60% width): chat interface
   - Right panel (20% width): model settings and options

3. Menu bar with:
   - File: New Conversation, Open Folder, Settings, Exit
   - Edit: Clear Conversation, Delete Conversation
   - View: Toggle Sidebar, Toggle Right Panel, Theme
   - Help: Documentation, About

4. Methods:
   - __init__()
   - setup_layout()
   - create_menu_bar()
   - load_window_state() / save_window_state()
   - on_closing()

5. Save window geometry to config on close
6. Include placeholder frames for each section with background colors for
visibility

Use CustomTkinter widgets and include comprehensive comments.
```

Work Package 4.2: Sidebar - Conversation List

- **Duration**: 4 hours
- **Deliverables**:
    - `gui/sidebar.py` with conversation tree view
    - Folder navigation and conversation selection
    - Right-click context menus

LLM Prompt:

```
Create a sidebar component (gui/sidebar.py) using CustomTkinter:

1. Implement a Sidebar class inheriting from customtkinter.CTkFrame:
   - Displays folder tree structure at top
   - Shows conversation list below
   - Search bar at top of sidebar

2. Components:
   - CTkEntry for search/filter
   - CTkScrollableFrame for folder tree
   - CTkScrollableFrame for conversation list
   - "New Conversation" button at bottom
   - "New Folder" button next to search

3. Methods:
   - __init__(master, conversation_storage: ConversationStorage)
   - load_folders() -> None (populate folder tree)
   - load_conversations(folder_path: str = "") -> None
   - on_conversation_select(conversation_id: str) -> None
   - on_folder_select(folder_path: str) -> None
   - refresh() -> None
   - filter_conversations(query: str) -> None

4. Display each conversation as:
   - Title (truncated to fit)
   - Last message preview (1 line)
   - Timestamp (relative, e.g., "2 hours ago")

5. Right-click context menu for conversations:
   - Rename, Move to Folder, Delete, Export to PDF

6. Use callback pattern to notify parent window of selection changes

Include icons for folders and conversations using Unicode symbols initially.
```

Day 5: Chat Interface

Work Package 5.1: Chat Panel - Message Display

- **Duration**: 5 hours
- **Deliverables**:
    - `gui/chat_panel.py` with message rendering
    - Auto-scroll functionality
    - Message timestamps and model indicators

LLM Prompt:

```
Create a chat panel component (gui/chat_panel.py) using CustomTkinter:
```

```
1. Implement a ChatPanel class inheriting from customtkinter.CTkFrame:
   - Displays conversation messages in a scrollable area
   - Shows user messages right-aligned, assistant left-aligned
   - Input area at bottom with send button and attachment button

2. Layout:
   - Top: Conversation title and model selector dropdown
   - Middle: CTkScrollableFrame for messages (80% height)
   - Bottom: CTkTextbox for input (3 rows) + buttons (20% height)

3. Methods:
   - __init__(master, langchain_handler: LangChainHandler)
   - load_conversation(conversation: Conversation) -> None
   - display_message(message: Message) -> None
     * Renders with proper alignment and styling
     * User messages: light background, right-aligned
     * Assistant messages: darker background, left-aligned
     * Shows timestamp and model used for assistant messages
   - send_message() -> None (handles user input)
   - stream_response(response_generator) -> None
     * Updates UI with streaming chunks
   - clear_input() -> None
   - enable_input() / disable_input() -> None

4. Message styling:
   - Use CTkFrame for each message bubble
   - CTkLabel for message content (wraplength for text wrapping)
   - Different colors for user vs assistant
   - Monospace font option for code blocks

5. Input area:
   - CTkTextbox for multi-line input
   - CTkButton "Send" (or Enter key binding)
   - CTkButton "Attach File" with file icon
   - Character/token counter below input

6. Auto-scroll to bottom on new messages
7. Show "typing..." indicator during streaming

Use CustomTkinter widgets and ensure responsive layout.
```

Work Package 5.2: Message Rendering & Markdown Support

- **Duration**: 3 hours
- **Deliverables**:
    - Enhanced message display with basic markdown
    - Code block highlighting
    - Copyable code blocks

LLM Prompt:

```
Enhance the ChatPanel class (gui/chat_panel.py) with markdown rendering:

1. Add a method render_markdown(text: str) -> None that:
   - Detects code blocks (```language\ncode\n```)
   - Renders code in CTkTextbox with monospace font
   - Adds a "Copy" button next to code blocks
   - Detects bold (**text**) and italic (*text*)
   - Renders lists (-, *, 1.)
   - Handles line breaks properly
```

```
2. For code blocks:
   - Use a different background color
   - Monospace font (Courier or Consolas)
   - Horizontal scrollbar if needed
   - Copy button using tkinter.Button with icon

3. For inline formatting:
   - **bold**: use CTkLabel with bold font
   - *italic*: use CTkLabel with italic font
   - Links: make clickable (open in browser)

4. Keep it simple - don't implement full markdown spec
5. Focus on what LLMs commonly output (code, lists, emphasis)

The render_markdown method should return a CTkFrame containing the formatted
content that can be inserted into the chat scroll area.

Include comments explaining the rendering logic.
```

Day 6: Settings & Preferences

Work Package 6.1: Settings Window

- **Duration**: 4 hours
- **Deliverables**:
  - `gui/settings_window.py` with tabbed preferences
  - API key management interface
  - Theme and default model settings

LLM Prompt:

```
Create a settings window (gui/settings_window.py) using CustomTkinter:

1. Implement a SettingsWindow class inheriting from customtkinter.CTkToplevel:
   - Modal dialog (disable main window while open)
   - Size: 600x500
   - Tabbed interface with 3 tabs: API, Preferences, Advanced

2. API Tab:
   - CTkLabel: "OpenRouter API Key"
   - CTkEntry for API key (password mode with show/hide toggle)
   - CTkButton: "Test Connection" (validates key)
   - CTkLabel: Connection status (success/failure message)
   - Link to OpenRouter.ai for getting API key

3. Preferences Tab:
   - CTkOptionMenu: Default model selection
   - CTkSlider: Default temperature (0.0 - 2.0)
   - CTkSwitch: Dark/Light theme toggle
   - CTkEntry: Conversations directory path
   - CTkButton: "Browse" for directory selection

4. Advanced Tab:
   - CTkSwitch: Enable streaming responses
   - CTkEntry: Custom API endpoint (optional override)
   - CTkSlider: Max tokens per request
   - CTkButton: "Reset to Defaults"

5. Methods:
   - __init__(master, settings: Settings)
   - load_current_settings() -> None
   - save_settings() -> None
   - test_api_connection() -> bool
   - apply_theme(theme: str) -> None
   - on_save() -> None
   - on_cancel() -> None

6. Buttons at bottom:
   - "Save" (saves and closes)
   - "Apply" (saves without closing)
   - "Cancel" (closes without saving)

Use CustomTkinter's CTkTabview for tabs. Include validation for API key format.
```

Work Package 6.2: Prompt Manager UI

- **Duration**: 4 hours
- **Deliverables**:
  - `gui/prompt_manager.py` for system prompt management
  - Prompt library with saved templates
  - Folder organization for prompts

LLM Prompt:

```
Create a prompt manager component (gui/prompt_manager.py) using CustomTkinter:

1. Implement a PromptManager class inheriting from customtkinter.CTkFrame:
   - Can be embedded in right panel or opened as separate window
   - Shows current system prompt
   - Allows editing and saving prompts
   - Displays saved prompt library

2. Layout:
   - Top: CTkLabel "System Prompt"
   - Middle: CTkTextbox for editing current prompt (60% height)
   - Below: "Apply to Conversation" button
   - Bottom: CTkScrollableFrame showing saved prompts (40% height)

3. Saved prompts section:
   - Each saved prompt shows: title, preview (first 50 chars)
   - Click to load into editor
   - Right-click menu: Edit, Delete, Duplicate
   - Organized in folders (similar to conversation folders)

4. Methods:
   - __init__(master)
   - load_prompt(prompt_id: str) -> None
   - save_prompt(title: str, content: str, folder: str) -> None
   - delete_prompt(prompt_id: str) -> None
   - apply_prompt_to_conversation(conversation_id: str) -> None
   - list_prompts(folder: str = "") -> List[Dict]
   - create_folder(folder_name: str) -> None

5. Buttons:
   - "Save as Template" (saves current prompt to library)
   - "Clear" (removes prompt from current conversation)
   - "New Folder" (creates prompt folder)

6. Prompt persistence:
   - Save prompts as JSON files in data/prompts/
   - Structure: {id, title, content, folder, created_at}
```

```
Include placeholder prompts: "Helpful Assistant", "Code Expert", "Creative
Writer"
```

Phase 3: Core Functionality Integration (Days 7-9)

Day 7: Message Flow & Conversation Management

Work Package 7.1: Send Message Pipeline

- **Duration**: 4 hours
- **Deliverables**:
    - Complete message sending flow from GUI to LangChain
    - Streaming response handling with UI updates
    - Error handling and retry logic

LLM Prompt:

```
Integrate the message sending pipeline in the main application:

1. In main_window.py, create method:
   - async_send_message(content: str, attachments: List[str] = None) -> None
     * Called when user clicks Send or presses Enter
     * Validates input is not empty
     * Creates Message object for user message
     * Adds message to current conversation
     * Disables input area
     * Calls LangChainHandler.send_message() with streaming=True
     * Handles streaming response chunks
     * Updates ChatPanel in real-time
     * Creates Message object for assistant response
     * Adds to conversation
     * Saves conversation to disk
     * Enables input area
     * Handles errors gracefully (show error message in chat)

2. Threading strategy:
   - Use threading.Thread to prevent UI blocking
   - Use queue.Queue to pass chunks from thread to main GUI thread
   - Update UI using after() callback for thread safety

3. Error handling:
   - Network errors: Show retry button
   - API errors: Display error message with details
   - Invalid API key: Prompt to open settings
   - Rate limits: Show wait time and retry automatically

4. Add loading indicator:
   - Show "..." animation in chat during response
   - Update with each streamed chunk
   - Show token count when complete

5. Update conversation timestamp and save to disk after each exchange

Include comprehensive error messages and logging.
```

Work Package 7.2: Model Switching During Conversation

- **Duration**: 3 hours

- **Deliverables**:
    - Model selector in chat header
    - Mid-conversation model switching
    - Model history tracking

LLM Prompt:

```
Implement model switching functionality:

1. In chat_panel.py, add:
   - CTkOptionMenu in header for model selection
   - Populated from ModelManager.get_all_models()
   - Current model highlighted
   - Grouped by provider (use separator or nested menu)

2. Create method on_model_change(new_model: str) -> None:
   - Updates conversation.current_model
   - Saves conversation
   - Shows notification in chat: "Switched to {model_name}"
   - Validates model supports required features (if using tools)
   - Updates LangChainHandler configuration

3. Display model used for each assistant message:
   - Add small badge/label showing model name
   - Different color per provider (OpenAI=blue, Anthropic=orange, etc.)

4. Add model comparison mode (optional enhancement):
   - Button to send same message to multiple models
   - Display responses side-by-side

5. Track model usage statistics:
   - Count messages per model
   - Display in conversation metadata

Ensure model changes take effect immediately for next message.
```

Work Package 7.3: Temperature Control & Advanced Parameters

- **Duration**: 1 hour
- **Deliverables**:
    - Temperature slider in right panel
    - Real-time parameter updates
    - Parameter presets

LLM Prompt:

```
Add temperature control to the application:

1. In the right panel of main_window.py, add:
   - CTkLabel: "Temperature"
   - CTkSlider: Range 0.0 to 2.0, default from settings
   - CTkLabel: Current value display (updates as slider moves)
   - Preset buttons: "Precise (0.3)", "Balanced (0.7)", "Creative (1.2)"

2. Create method update_temperature(value: float) -> None:
   - Updates conversation.current_temperature
   - Saves conversation
   - Takes effect on next message (no notification needed)

3. Add tooltip explaining temperature:
   - Hover over label shows explanation
   - "Lower = more focused and deterministic"
   - "Higher = more creative and random"

4. Optional: Add more parameters in collapsible section:
   - Top-p (nucleus sampling)
   - Max tokens
   - Frequency penalty
   - Presence penalty

5. Save parameter changes to conversation immediately

Keep UI clean - only show temperature by default, hide advanced in collapsible.
```

Day 8: File Handling & Tools

Work Package 8.1: File Upload Integration

- **Duration**: 4 hours
- **Deliverables**:
    - File attachment button functionality
    - File preview in chat input area
    - File handling in API requests

LLM Prompt:

```
Implement file upload functionality:

1. In chat_panel.py, enhance the attach file button:
   - Opens file dialog (using customtkinter.filedialog)
   - Supports multiple file types: .txt, .pdf, .docx, .py, .json, .csv, .md
   - Max size: 10MB per file
   - Shows selected file in input area with remove button

2. File preview in input area:
   - Create frame above textbox showing attached files
   - Each file displayed as: [Icon] filename.ext (123 KB) [X]
   - Click X to remove attachment
   - Different icons per file type

3. Integrate with FileHandler:
   - Validate file before attaching
   - Read file content and encode if needed
   - Include in Message object attachments field

4. Send attachments with message:
   - For text files: Include content directly in message
   - Format: "User uploaded {filename}:\n\n{content}\n\n{user_message}"
   - For binary files: Describe file and include base64 in metadata

5. Display attachments in message history:
   - Show file icon and name in message bubble
   - Click to open/view file locally
   - Include file size and type

6. Error handling:
   - File too large: Show error message
   - Unsupported type: Warn but allow attachment
   - File not found: Handle gracefully
   - Read permission denied: Show appropriate error
```

```
Use customtkinter widgets and include progress indication for large files.
```

Work Package 8.2: Tool Integration for Function Calling

- **Duration**: 3 hours
- **Deliverables**:
    - Basic tool definition framework
    - Function calling for supported models
    - Tool result display in chat

LLM Prompt:

```
Add tool/function calling support for models that support it:

1. Create core/tools.py with basic tool definitions:
   - Define tools as LangChain Tool objects
   - Start with simple tools:
     * get_current_time() -> str
     * calculate(expression: str) -> float
     * read_file(path: str) -> str (for uploaded files)

2. In langchain_handler.py, enhance send_message():
   - Check if model supports function calling
   - If yes, include tools parameter
   - Handle function call responses
   - Execute requested function
   - Send result back to model
   - Display in chat: "Used tool: {tool_name}"

3. Tool result display:
   - Show tool calls in chat with special formatting
   - Different background color
   - Format: "[🛠️ Tool] {tool_name}({args}) → {result}"

4. Safety considerations:
   - Whitelist allowed functions
   - Validate file paths (prevent directory traversal)
   - Limit calculation complexity
   - Add user confirmation for sensitive operations (optional)

5. Model capability checking:
   - Only enable tools for compatible models
   - Show tooltip if user tries to use tools with unsupported model

Keep tool set minimal for MVP. Focus on file operations since that's the main
requirement.
```

Work Package 8.3: Drag-and-Drop File Support

- **Duration**: 1 hour
- **Deliverables**:
    - Drag-and-drop file attachment
    - Visual feedback during drag

LLM Prompt:

```
Add drag-and-drop file support to the chat panel:

1. In chat_panel.py, implement drag-and-drop:
   - Enable drop target on the chat input CTkTextbox
   - Use tkinter DND events: <<Drop>>, <<DragEnter>>, <<DragLeave>>

2. Visual feedback:
   - Change input border color on drag enter (highlight)
   - Show "Drop files here" overlay
   - Reset on drag leave or drop

3. Handle dropped files:
   - Get file paths from drop event
   - Validate each file using FileHandler
   - Add valid files to attachment list
   - Show error for invalid files

4. Support multiple files:
   - Allow dropping multiple files at once
   - Add all valid files to attachment list

5. Integration:
   - Reuse existing file attachment logic
   - Same preview and removal functionality

Keep implementation simple using tkinter's built-in DND support.
```

Day 9: Conversation Management Features

Work Package 9.1: Conversation CRUD Operations

- **Duration**: 3 hours
- **Deliverables**:
    - New conversation creation
    - Conversation renaming
    - Conversation deletion with confirmation
    - Move to folder functionality

LLM Prompt:

```
Implement conversation management operations:

1. In main_window.py, add methods:

   - new_conversation() -> None:
     * Creates new Conversation object with UUID
     * Default title: "New Conversation"
     * Opens in ChatPanel
     * Adds to sidebar
     * Focuses input area

   - rename_conversation(conversation_id: str, new_title: str) -> None:
     * Updates conversation.title
     * Saves to disk
     * Updates sidebar display
     * Shows confirmation toast

   - delete_conversation(conversation_id: str) -> None:
     * Shows confirmation dialog with CTkInputDialog
     * "Are you sure? This cannot be undone."
     * Deletes JSON file
     * Removes from sidebar
     * If current conversation, opens new blank conversation
   - move_conversation(conversation_id: str, target_folder: str) -> None:
     * Updates conversation.folder_path
     * Moves JSON file to new location
     * Updates sidebar tree
     * Saves conversation

2. Add dialog for rename:
   - Use CTkInputDialog
   - Pre-fill with current title
   - Validate not empty
   - Cancel option

3. Context menu integration:
   - Connect right-click menu in sidebar to these methods
   - Add keyboard shortcuts: Ctrl+N (new), F2 (rename), Del (delete)

4. Auto-save:
   - Save conversation after every message
   - Debounce title changes (wait 1 second after typing stops)

Include proper error handling and user feedback for all operations.
```

Work Package 9.2: Folder Operations & Organization

- **Duration**: 3 hours
- **Deliverables**:
  - Folder creation UI
  - Folder rename and delete
  - Drag-and-drop conversation to folder
  - Nested folder support

LLM Prompt:

```
Implement folder management UI and operations:

1. In sidebar.py, add folder operations:

   - create_folder_dialog() -> None:
     * Opens CTkInputDialog for folder name
     * Supports nested paths: "project/subfolder"
     * Validates folder name (alphanumeric, -, _)
     * Creates folder structure
     * Refreshes sidebar tree

   - rename_folder_dialog(folder_path: str) -> None:
     * Input dialog with current name
     * Updates all conversations in folder
     * Handles nested folders correctly

   - delete_folder_dialog(folder_path: str) -> None:
     * Shows confirmation with count of conversations
     * Options: "Delete conversations" or "Move to root"
     * Executes deletion via FolderManager
     * Refreshes sidebar

2. Folder tree visualization:
   - Use expandable/collapsible frames
   - Indent nested folders
   - Show conversation count per folder
   - Folder icons: 📁 (closed) 📂 (open)
```

```
3. Drag-and-drop conversations:
   - Allow dragging conversation items
   - Drop on folders to move
   - Visual feedback during drag (show target folder highlighted)
   - Update immediately on drop

4. Right-click context menu for folders:
   - New Subfolder
   - Rename
   - Delete
   - New Conversation (creates conversation in that folder)

5. Folder sorting:
   - Alphabetical by default
   - Folders before conversations
   - Recent conversations option (sort by updated_at)

Use CustomTkinter's scrollable frame and standard tkinter drag-drop events.
```

Work Package 9.3: Search & Filter

- **Duration**: 2 hours
- **Deliverables**:
    - Full-text conversation search
    - Filter by folder, date, model
    - Search results highlighting

LLM Prompt:

```
Implement search and filtering functionality:

1. In sidebar.py, enhance search bar:
   - CTkEntry with search icon
   - Real-time search (300ms debounce)
   - Search in: conversation titles, message content
   - Case-insensitive matching

2. Add filter options below search:
   - CTkOptionMenu: "All Folders" / specific folder
   - CTkOptionMenu: "All Dates" / Today / This Week / This Month
   - CTkOptionMenu: "All Models" / specific model

3. Create method apply_filters() -> None:
   - Combines search query + filters
   - Uses ConversationStorage.search_conversations()
   - Updates conversation list display
   - Shows count: "23 conversations found"

4. Search results display:
   - Highlight matching text in conversation titles
   - Show snippet of matching message (with context)
   - Display match count per conversation

5. Clear filters button:
   - Resets all filters
   - Shows all conversations
   - Clears search input

6. Keyboard shortcut:
   - Ctrl+F focuses search bar
   - Escape clears search
```

```
Keep search simple but functional. Use string matching, not complex indexing for
MVP.
```

Phase 4: Export & Polish (Days 10-12)

Day 10: PDF Export Functionality

Work Package 10.1: PDF Generation with ReportLab

- **Duration**: 5 hours
- **Deliverables**:
    - `export/pdf_exporter.py` with PDF generation
    - High-quality typography with serif fonts
    - Proper message formatting and layout

LLM Prompt:

```
Create a PDF export system (export/pdf_exporter.py) using ReportLab:

1. Implement a PDFExporter class with method:
   - export_conversation(conversation: Conversation, output_path: str) -> bool

2. PDF structure:
   - Title page:
     * Conversation title (24pt, bold serif)
     * Date range (first to last message)
     * Total messages count
     * Model(s) used

   - Each message:
     * Role indicator (User: / Assistant:) in bold
     * Timestamp (small, gray)
     * Message content (11pt serif, justified)
     * Model name for assistant messages
     * Spacing between messages

   - Footer on each page:
     * Page number
     * Generated date
     * "Exported from AI Gateway"

3. Typography:
   - Use high-quality serif fonts:
     * Title: Times-Bold or Palatino
     * Body: Times-Roman or Georgia
     * Code blocks: Courier
   - Line spacing: 1.4
   - Margins: 1 inch all sides
   - Professional color scheme (black text, light gray accents)

4. Formatting:
   - Detect code blocks and use monospace font with background
   - Preserve line breaks and paragraphs
   - Handle long messages across pages
   - Wrap long URLs

5. Metadata:
   - PDF title: conversation title
   - Author: "AI Gateway"
   - Subject: "AI Conversation Export"
   - Keywords: model names used

6. Error handling:
   - Handle write permission errors
   - Validate output path
   - Fallback for missing fonts

Use reportlab.lib.pagesizes for Letter size, reportlab.platypus for layout.
Include progress indication for long conversations (optional).
```

Work Package 10.2: PDF Export UI Integration

- **Duration**: 2 hours
- **Deliverables**:
    - Export menu item and dialog
    - File save location picker
    - Export progress indication

LLM Prompt:

```
Integrate PDF export into the GUI:

1. Add menu item:
   - File → Export to PDF (Ctrl+E)
   - Only enabled when conversation is open
   - Opens file save dialog

2. Export dialog:
   - Use customtkinter.filedialog.asksaveasfilename()
   - Default filename: "{conversation_title}_{date}.pdf"
   - Default location: ~/Documents/AI_Gateway_Exports/
   - Filter: PDF files (*.pdf)

3. In main_window.py, add method:
   - export_current_conversation() -> None:
     * Gets current conversation
     * Validates conversation has messages
     * Opens save dialog
     * Calls PDFExporter in background thread
     * Shows progress dialog (if > 50 messages)
     * Shows success/error message

4. Progress indication:
   - For long conversations, show modal progress dialog
   - "Exporting conversation... 45/120 messages"
   - Cancel button (optional)

5. Right-click menu:
   - Add "Export to PDF" to conversation context menu in sidebar
   - Allows exporting without opening conversation

6. Success notification:
   - Toast notification: "Conversation exported successfully"
   - Button to open PDF location in file manager
   - Button to open PDF directly

Use threading to prevent UI freezing during export.
```

Work Package 10.3: Batch Export

- **Duration**: 1 hour
- **Deliverables**:
    - Export multiple conversations
    - Export entire folder

LLM Prompt:

```
Add batch export functionality:

1. In sidebar.py, add right-click menu for folders:
   - "Export Folder to PDF" option
   - Shows dialog: "Export {count} conversations?"

2. Batch export method:
   - export_folder(folder_path: str, output_directory: str) -> None:
     * Gets all conversations in folder
     * Creates output directory if needed
     * Exports each conversation with filename: {folder}_{title}_{date}.pdf
     * Shows progress bar: "Exporting 3/15 conversations..."
     * Logs any failures

3. Multi-select conversations:
   - Allow Ctrl+Click to select multiple conversations in sidebar
   - Right-click → "Export Selected ({count})"
   - Same batch process

4. Output structure:
   - If exporting folder: creates subfolder with folder name
   - Maintains folder hierarchy in filenames

5. Error handling:
   - Skip conversations that fail to export
   - Show summary: "Exported 14/15 conversations. 1 failed."
   - Option to view error log

Keep UI responsive during batch operations using threading.
```

Day 11: Prompt Library & Management

Work Package 11.1: Prompt Storage & Persistence

- **Duration**: 3 hours
- **Deliverables**:
    - Prompt data model and JSON storage
    - Prompt CRUD operations
    - Folder organization for prompts

LLM Prompt:

```
Implement prompt storage system:

1. Create storage/prompt_storage.py with:

   - Prompt data model (dataclass or dict):
     * prompt_id (UUID)
     * title (string)
     * content (string)
     * folder_path (string)
     * created_at (datetime)
     * last_used (datetime)
     * use_count (int)
     * tags (list of strings)

   - PromptStorage class with methods:
     * save_prompt(prompt: Dict) -> str
       - Saves to data/prompts/{folder_path}/{prompt_id}.json
       - Returns prompt_id

     * load_prompt(prompt_id: str) -> Dict
       - Loads and deserializes JSON

     * list_prompts(folder_path: str = "") -> List[Dict]
       - Returns prompt metadata sorted by last_used

     * delete_prompt(prompt_id: str) -> bool

     * search_prompts(query: str) -> List[Dict]
       - Search in title, content, tags

     * update_usage(prompt_id: str) -> None
       - Increments use_count, updates last_used

2. Folder management:
   - Reuse FolderManager pattern
   - Support nested folders
   - Create default folders: "General", "Coding", "Writing", "Analysis"

3. Default prompts:
   - Create 5-10 starter prompts on first run
   - Examples: "Helpful Assistant", "Python Expert", "Technical Writer"
   - Save in "General" folder

Include comprehensive error handling and atomic writes.
```

Work Package 11.2: Prompt Manager UI Enhancement

- **Duration**: 3 hours
- **Deliverables**:
  - Complete prompt library interface
  - Quick-apply functionality
  - Prompt editing and versioning

LLM Prompt:

```
Complete the PromptManager UI (gui/prompt_manager.py):

1. Enhance layout with tabs:
   - Tab 1: Current Prompt (editor)
   - Tab 2: Library (saved prompts)
   - Tab 3: History (recently used)

2. Current Prompt tab:
   - CTkTextbox for editing (300px height)
   - Buttons: Apply, Save as Template, Clear, Undo
   - Character counter
   - Template variables support: {topic}, {style}, {language}
   - Preview button (shows rendered prompt with sample values)

3. Library tab:
   - Folder tree on left (30% width)
   - Prompt list on right (70% width)
   - Each prompt shows: title, preview, use count, last used
   - Quick apply button (small icon on each prompt)
   - Search bar at top

4. History tab:
   - List of recently used prompts (last 20)
   - Timestamp and conversation where used
   - Quick apply button
   - "Save to Library" button for unsaved prompts

5. Prompt item display:
   - CTkFrame with hover effect
   - Title (bold, 14pt)
   - Content preview (2 lines, ellipsis)
   - Metadata: folder, use count, date
   - Icons: ⭐ (favorite), 📝 (edit), 🗑️ (delete)

6. Context menu for prompts:
   - Edit, Duplicate, Move to Folder, Delete, Add to Favorites

7. Variables feature:
   - Support {variable_name} syntax in prompts
   - When applying, show dialog to fill variables
   - Save filled values for next use

Include drag-and-drop to reorder favorites.
```

Work Package 11.3: Prompt Templates & Snippets

- **Duration**: 2 hours
- **Deliverables**:
  - Pre-built prompt templates
  - Snippet insertion functionality
  - Template categories

LLM Prompt:

```
Add prompt templates and snippets:

1. Create config/prompt_templates.json with categories:
   - General Purpose (5 templates)
   - Code & Development (5 templates)
   - Writing & Content (5 templates)
   - Data Analysis (5 templates)
   - Creative & Brainstorming (5 templates)

2. Each template includes:
   - name, description, content, category, variables
   - Example: {
       "name": "Code Review Expert",
       "description": "Reviews code for best practices",
       "content": "You are an expert code reviewer...",
       "category": "Code & Development",
       "variables": ["language", "focus_area"]
     }

3. In PromptManager, add:
   - "Insert Template" dropdown menu
   - Grouped by category
   - Click to insert template into editor
   - Prompts for variable values if needed
```

```
4. Snippet system:
   - Common prompt fragments
   - Examples: "Be concise", "Step by step", "With examples"
   - Insert at cursor position
   - Keyboard shortcuts: Ctrl+1, Ctrl+2, etc. for top 10

5. Import/Export:
   - Button to export prompts as JSON
   - Import prompts from JSON file
   - Share prompt collections

Include 25 high-quality templates in prompt_templates.json.
```

Day 12: Polish, Testing & Documentation

Work Package 12.1: Error Handling & User Feedback

- **Duration**: 3 hours
- **Deliverables**:
  - Comprehensive error handling throughout app
  - Toast notifications for actions
  - Error logging system

LLM Prompt:

```
Implement comprehensive error handling and user feedback:

1. Create utils/error_handler.py with:
   - ErrorHandler class
   - Centralized error logging to file (logs/errors.log)
   - Error categorization: Network, API, File, Validation
   - User-friendly error messages mapping

2. Error message strategy:
   - Technical errors → User-friendly messages
   - Example: "ConnectionError" → "Can't connect to AI service. Check your
internet."
   - Include recovery suggestions
   - Log full technical details

3. Toast notification system:
   - Create utils/toast.py
   - Show temporary notifications (3 seconds)
   - Types: success (green), error (red), info (blue), warning (yellow)
   - Position: bottom-right corner
   - Auto-dismiss with fade animation
   - Queue multiple toasts

4. Add toasts for all user actions:
   - "Conversation saved"
   - "Model switched to GPT-4"
   - "Prompt applied"
   - "File attached successfully"
   - "Exported to PDF"

5. Error recovery:
   - Auto-retry for network errors (3 attempts)
   - Offer manual retry button
   - Save draft messages if send fails
   - Graceful degradation (disable features if API unavailable)

6. Loading states:
   - Show spinner for long operations
   - Disable buttons during processing
   - Progress bars for batch operations

7. Validation feedback:
   - Real-time input validation
   - Red border for invalid fields
   - Error message below field
   - Examples: empty API key, invalid folder name

Include logging with rotating file handler (max 10MB, 5 backups).
```

Work Package 12.2: Performance Optimization

- **Duration**: 2 hours
- **Deliverables**:
    - Lazy loading for conversations
    - Caching for model metadata
    - UI responsiveness improvements

LLM Prompt:

```
Optimize application performance:

1. Lazy loading conversations:
   - Don't load full conversation content in sidebar
   - Load only metadata: title, preview, timestamp
   - Load full messages only when conversation opened
   - Paginate message history (load 50 at a time, "Load More" button)

2. Caching strategy:
   - Cache ModelManager data in memory
   - Cache frequently used prompts
   - Cache folder structure (refresh only on changes)
   - Implement simple LRU cache for conversations

3. UI responsiveness:
   - Use after() for smooth animations
   - Debounce search input (300ms)
   - Throttle folder tree updates
   - Virtual scrolling for long conversation lists (if >100 items)

4. Background tasks:
   - Move file operations to background threads
   - Use thread pool (max 3 threads)
   - Queue for API requests
   - Async save operations

5. Startup optimization:
   - Load UI first, then data
   - Show splash screen during initialization
   - Lazy import heavy libraries
   - Cache last opened conversation

6. Memory management:
   - Unload old conversations from memory
   - Clear message history cache when switching conversations
   - Limit in-memory conversation count (max 10)
```

```
7. Profile and optimize:
   - Identify slow operations
   - Add timing logs for development
   - Optimize JSON serialization (use ujson if available)

Keep optimizations simple - focus on perceived performance.
```

Work Package 12.3: Final Polish & UX Improvements

- **Duration**: 3 hours
- **Deliverables**:
    - Keyboard shortcuts reference
    - UI animations and transitions
    - Welcome screen for first-time users
    - About dialog

LLM Prompt:

```
Add final polish and UX improvements:

1. Keyboard shortcuts:
   - Create utils/keyboard_shortcuts.py
   - Implement global shortcuts:
     * Ctrl+N: New conversation
     * Ctrl+F: Focus search
     * Ctrl+E: Export to PDF
     * Ctrl+,: Open settings
     * Ctrl+K: Focus model selector
     * Ctrl+/: Show keyboard shortcuts help
     * Escape: Clear selection/close dialogs
   - Show shortcuts dialog (Ctrl+/ or Help menu)

2. UI animations:
   - Fade in/out for dialogs
   - Smooth sidebar collapse/expand
   - Message appear animation (slide in)
   - Button hover effects
   - Loading spinner animation

3. Welcome screen (first launch):
   - Modal window with 3 steps:
     1. Welcome message and app overview
     2. API key setup (link to OpenRouter)
     3. Quick tour (highlight main features)
   - "Don't show again" checkbox
   - Skip button

4. About dialog:
   - App name and version
   - Description
   - Links: GitHub, Documentation, OpenRouter
   - Credits and license
   - System info: Python version, OS, dependencies

5. Empty states:
   - New user: "Create your first conversation"
   - No search results: "No conversations found. Try different keywords."
   - Empty folder: "This folder is empty. Create a conversation or move one
here."
   - Include helpful icons and suggestions
```

```
6. Onboarding tooltips:
   - First time using features: show tooltip
   - Examples: "Click here to attach files", "Change model mid-conversation"
   - Dismissible, don't show again option

7. Theme refinement:
   - Ensure consistent colors throughout
   - Proper contrast ratios
   - Dark mode and light mode fully supported
   - Smooth theme switching

8. Icons and visual elements:
   - Use Unicode emojis or simple icons
   - Consistent icon style
   - Loading indicators
   - Status indicators (online/offline)

Make the app feel polished and professional.
```

Phase 5: Testing & Documentation (Days 13-15)

Day 13: Integration Testing

Work Package 13.1: Core Functionality Testing

- **Duration**: 4 hours
- **Deliverables**:
  - Test suite for core features
  - Manual test checklist
  - Bug tracking document

LLM Prompt:

```
Create a comprehensive test suite and testing documentation:

1. Create tests/test_core.py with unit tests for:
   - Conversation data model serialization/deserialization
   - Message handling and formatting
   - File attachment validation
   - Folder operations (create, rename, delete, move)
   - Prompt storage and retrieval
   - Settings management
   - Model manager functionality

2. Create tests/test_integration.py for:
   - Full message sending flow
   - Model switching during conversation
   - File upload and attachment
   - Conversation save/load cycle
   - PDF export generation
   - Search and filter functionality

3. Manual test checklist (tests/manual_tests.md):
   - UI responsiveness tests
   - Error handling scenarios
   - Edge cases (empty inputs, long text, special characters)
   - Performance tests (100+ conversations, long messages)
   - Theme switching
   - All keyboard shortcuts
   - Context menus
   - Drag and drop

4. Test each user workflow:
   - Complete conversation flow: new → message → save → export
   - Model switching workflow
   - Folder organization workflow
   - Prompt management workflow
   - Settings changes workflow

5. Bug tracking template (tests/bugs.md):
   - Priority, description, steps to reproduce, expected vs actual
   - Status tracking (open, in progress, fixed, verified)

6. Create run_tests.sh script:
   - Runs all unit tests
   - Generates coverage report
   - Runs integration tests
   - Outputs summary

Use pytest framework. Include fixtures for test data.
```

Work Package 13.2: API Integration Testing

- **Duration**: 2 hours
- **Deliverables**:
  - OpenRouter API integration tests
  - Mock API for offline testing
  - Error scenario handling tests

LLM Prompt:

```
Create API integration tests:

1. Create tests/test_api.py with:
   - Test OpenRouter authentication
   - Test message sending (non-streaming)
   - Test streaming responses
   - Test model switching
   - Test function calling (if enabled)
   - Test error handling (invalid API key, rate limits, network errors)
   - Test different model parameters (temperature, max tokens)

2. Mock API for testing:
   - Create tests/mock_api.py
   - Simulates OpenRouter responses
   - Supports streaming
   - Can simulate errors
   - Faster testing without real API calls
   - No API key required for unit tests

3. Test scenarios:
   - Valid API calls
   - Invalid API key
   - Network timeout
   - Rate limit exceeded
   - Model not found
   - Malformed requests
   - Large responses (10k+ tokens)

4. Environment setup:
   - Use .env.test with test API key (or mock)
   - Separate test data directory
   - Clean up after tests

5. Rate limit testing:
   - Verify retry logic
   - Test exponential backoff
   - Maximum retry limit

Run these tests with pytest. Use responses library for HTTP mocking.
```

Work Package 13.3: User Acceptance Testing Preparation

- **Duration**: 2 hours
- **Deliverables**:
  - UAT test scenarios
  - Feedback collection template
  - User guide for testers

LLM Prompt:

```
Prepare user acceptance testing materials:

1. Create tests/uat_scenarios.md with 10 realistic scenarios:
   - Scenario 1: First-time user setup
   - Scenario 2: Daily usage (multiple conversations)
   - Scenario 3: Research project (folder organization)
   - Scenario 4: Model comparison workflow
   - Scenario 5: File analysis task
   - Scenario 6: Prompt experimentation
   - Scenario 7: Conversation export and sharing
   - Scenario 8: System prompt management
   - Scenario 9: Search and retrieve old conversations
   - Scenario 10: Settings configuration

2. Each scenario includes:
   - Goal
   - Step-by-step instructions
   - Expected outcomes
   - Success criteria
   - Time estimate

3. Feedback template (tests/feedback_template.md):
   - What worked well
   - What was confusing
   - What didn't work
   - Feature requests
   - Usability rating (1-5)
   - Performance rating (1-5)

4. Quick start guide for testers:
   - Installation steps
   - API key setup
   - Brief feature overview
   - Where to report issues

5. Testing environment setup:
   - Fresh install instructions
   - Sample data package (5 pre-made conversations)
   - Test API key setup

Make scenarios realistic and cover all major features.
```

Day 14: Documentation

Work Package 14.1: User Documentation

- **Duration**: 4 hours
- **Deliverables**:
    - Complete user manual
    - Quick start guide
    - FAQ document
    - Troubleshooting guide

LLM Prompt:

```
Create comprehensive user documentation:

1. docs/USER_MANUAL.md with sections:
   - Introduction and overview
   - Installation instructions (Ubuntu 24.04 specific)
   - First-time setup (API key configuration)
   - Main features walkthrough:
     * Starting a conversation
     * Changing models
     * Adjusting temperature
     * Attaching files
     * Using system prompts
     * Organizing conversations
     * Exporting to PDF
   - Advanced features:
     * Model switching mid-conversation
     * Function calling/tools
     * Prompt management
     * Batch operations
   - Settings reference
   - Keyboard shortcuts reference
   - Tips and best practices

2. docs/QUICKSTART.md:
   - 5-minute getting started guide
   - Essential steps only
   - Screenshots or ASCII diagrams
   - First conversation walkthrough

3. docs/FAQ.md with common questions:
   - How do I get an OpenRouter API key?
   - Which models are supported?
   - How much does it cost?
   - Where are conversations stored?
   - How do I backup my data?
   - Can I use custom models?
   - How do I change themes?
   - Why is my message failing?
   - How do I update the application?
   - How do I report bugs?

4. docs/TROUBLESHOOTING.md:
   - Common issues and solutions
   - Error messages explained
   - Network connectivity issues
   - API authentication problems
   - File permission errors
   - Application won't start
   - UI rendering issues
   - Performance problems
   - Data recovery procedures

5. Include:
   - Table of contents for each document
   - Clear headings and formatting
   - Examples and code snippets where relevant
   - Links between related sections

Write in clear, beginner-friendly language. Use markdown formatting.
```

Work Package 14.2: Developer Documentation

- **Duration**: 3 hours
- **Deliverables**:
  - Architecture documentation
  - API reference
  - Contributing guide
  - Development setup guide

LLM Prompt:

```
Create developer documentation:

1. docs/ARCHITECTURE.md:
   - System overview and design decisions
   - Component diagram (ASCII or description)
   - Data flow diagrams
   - Module descriptions:
     * GUI components
     * Core logic
     * Storage layer
     * API integration
   - Design patterns used
   - Tech stack justification
   - Future extensibility considerations

2. docs/API_REFERENCE.md:
   - Core classes and methods
   - Function signatures with type hints
   - Usage examples for key APIs
   - Conversation data model
   - Storage interface
   - LangChain integration details
   - Extension points

3. docs/CONTRIBUTING.md:
   - How to contribute
   - Code style guidelines (PEP 8)
   - Git workflow (branching, commits)
   - Pull request process
   - Testing requirements
   - Documentation standards
   - Issue reporting guidelines

4. docs/DEVELOPMENT.md:
   - Development environment setup
   - Dependencies installation
   - Running in development mode
   - Debugging tips
   - Hot reload setup
   - Building from source
   - Common development tasks
   - IDE recommendations (VS Code, PyCharm)

5. Code comments and docstrings:
   - Ensure all public methods have docstrings
   - Complex logic has inline comments
   - Type hints throughout codebase
   - Examples in docstrings where helpful

Use clear technical language. Include code examples and diagrams.
```

Work Package 14.3: Configuration & Deployment Documentation

- **Duration**: 1 hour
- **Deliverables**:
  - Deployment guide
  - Configuration reference
  - Backup and restore guide

LLM Prompt:

```
Create deployment and configuration documentation:

1. docs/DEPLOYMENT.md:
   - System requirements (Ubuntu 24.04, Python 3.11+)
   - Installation methods:
     * From source
     * Using setup script
     * Virtual environment setup
   - Post-installation configuration
   - Updating to new versions
   - Uninstallation procedures
   - Running on other Linux distros (brief notes)

2. docs/CONFIGURATION.md:
   - All configuration options explained
   - .env file format and variables
   - settings.json structure
   - models.json customization
   - Default values and valid ranges
   - Environment variables reference
   - Advanced configuration options

3. docs/BACKUP_RESTORE.md:
   - What data to backup (conversations, prompts, settings)
   - Backup procedures:
     * Manual backup (copy directories)
     * Automated backup script
   - Restore procedures
   - Data migration between versions
   - Export/import workflows

4. Create backup script (scripts/backup.sh):
   - Creates timestamped backup archive
   - Includes all user data
   - Excludes cache and logs
   - Compression options

5. Create restore script (scripts/restore.sh):
   - Validates backup integrity
   - Restores to specified location
   - Preserves existing data option

Include examples and common use cases for each document.
```

Day 15: Final Review & Package

Work Package 15.1: Code Review & Cleanup

- **Duration**: 3 hours
- **Deliverables**:
  - Code cleanup and refactoring
  - Consistent formatting
  - Removed unused code
  - Final bug fixes

LLM Prompt:

```
Perform final code review and cleanup:

1. Code quality checklist:
   - Run pylint or flake8 on entire codebase
   - Fix all critical issues
   - Address code style violations
   - Ensure consistent formatting (use black formatter)
   - Remove debug print statements
   - Remove commented-out code
   - Remove unused imports
   - Check for TODO comments

2. Refactoring priorities:
   - Identify duplicated code and extract to functions
   - Long methods (>50 lines) - consider splitting
   - Deep nesting - flatten where possible
   - Magic numbers - convert to named constants
   - Hardcoded strings - move to config

3. Documentation review:
   - All public methods have docstrings
   - Docstrings follow consistent format (Google style)
   - Type hints are complete and accurate
   - README.md is up to date
   - CHANGELOG.md documents all features

4. Security review:
   - API keys never logged or displayed
   - File paths validated (no directory traversal)
   - User inputs sanitized
   - Dependencies checked for vulnerabilities (use safety)

5. Performance review:
   - Profile slow operations
   - Optimize identified bottlenecks
   - Check memory usage
   - Verify no memory leaks

6. Create final_review_checklist.md:
   - All features working
   - All tests passing
   - Documentation complete
   - No critical bugs
   - Performance acceptable
   - Code style consistent

Run automated tools where possible. Document any remaining issues.
```

Work Package 15.2: Build & Package

- **Duration**: 2 hours
- **Deliverables**:
    - Installable package
    - Setup script
    - Release notes
    - Version tagging

LLM Prompt:

```
Create installation package and release materials:

1. Create setup.py for pip installation:
   - Package metadata (name, version, author, description)
   - Dependencies list
   - Entry point for running application
   - Include data files (models.json, templates)
   - Long description from README

2. Create install.sh script:
   - Checks Python version (3.11+)
   - Creates virtual environment
   - Installs dependencies
   - Sets up data directories
   - Creates desktop launcher (Ubuntu .desktop file)
   - Instructions for adding to PATH

3. Desktop launcher (ai-gateway.desktop):
   - Name, description, icon
   - Exec command
   - Categories (Development;Utility)
   - Terminal=false

4. Release notes (RELEASE_NOTES.md):
   - Version number and date
   - New features list
   - Bug fixes
   - Known issues
   - Breaking changes (if any)
   - Upgrade instructions

5. Version management:
   - Update version in all files
   - Create version.py with version constant
   - Git tag for release

6. Distribution package:
   - Create dist/ directory with all files
   - Include README, LICENSE, docs/
   - Zip or tar.gz archive
   - Calculate checksums
```

```
7. Update README.md with:
   - Installation instructions using new installer
   - Quick start guide
   - Screenshots (placeholders or ASCII art)
   - Features list
   - License information
   - Contribution guidelines link

Make installation as simple as possible - ideally one command.
```

Work Package 15.3: Final Testing & Validation

- **Duration**: 3 hours
- **Deliverables**:
    - End-to-end testing
    - Fresh install validation
    - Performance benchmarks
    - Release approval checklist

LLM Prompt:

```
Perform final validation and testing:

1. Fresh installation test:
   - Clean Ubuntu 24.04 VM or container
   - Run install.sh script
   - Verify all dependencies installed
   - Check directory structure created
   - Test first-run experience
   - Complete one full conversation workflow

2. End-to-end test scenarios:
   - Complete user journey: install → setup → use → export
   - Test all major features in sequence
   - Verify data persistence
   - Test error recovery
   - Stress test (100+ conversations, long messages)

3. Performance benchmarks:
   - Application startup time (target < 3 seconds)
   - Conversation load time (target < 1 second)
   - Message send time (excluding API latency)
   - Search performance (1000+ conversations)
   - PDF export time (100 message conversation)
   - Memory usage (idle and under load)

4. Compatibility testing:
   - Ubuntu 24.04 (primary target)
   - Ubuntu 22.04 (should work)
   - Test on different Python versions (3.11, 3.12)
   - Different screen resolutions
   - Dark and light themes

5. Release approval checklist:
   - [ ] All planned features implemented
   - [ ] All tests passing
   - [ ] No critical bugs
   - [ ] Documentation complete
   - [ ] Performance acceptable
   - [ ] Security review completed
   - [ ] Installation tested on clean system
   - [ ] User feedback incorporated
   - [ ] Release notes finalized
   - [ ] Version tagged in git

6. Create performance_report.md:
   - Benchmark results
   - Resource usage statistics
   - Identified bottlenecks
   - Optimization recommendations for future

Document all test results. Create video demo of key workflows (optional).
```

Implementation Best Practices

Daily Workflow

Each day should follow this pattern:

1. **Morning**: Review roadmap for the day, clarify any questions with LLM
2. **Implementation**: Work through packages sequentially, commit after each completed package
3. **Testing**: Test newly implemented features interactively
4. **Documentation**: Update relevant docs as you build
5. **Evening**: Review day's progress, prepare prompts for next day

LLM Prompting Strategy

Effective Prompt Structure:

```
[Context] I'm building [feature] as part of [larger system]

[Task] Create [specific component] that:
1. [Requirement 1]
2. [Requirement 2]
...

[Constraints]
- Must use [specific library/pattern]
- Should integrate with [existing component]
- Performance requirement: [specific metric]

[Output Format]
- File: [filename]
- Include [specific elements]
- Use [coding standards]
```

Iterative Refinement:

- Start with core functionality
- Test the generated code
- Request improvements: "Add error handling for X", "Optimize Y"
- Build incrementally rather than generating large files

Integration Tips:

- Generate one module at a time
- Test integration points immediately
- Use LLM for debugging: "This error occurs: [error]. The code is: [code]. How to fix?"

Git Workflow

Commit Strategy:

- Commit after each work package completion
- Use semantic commit messages: `feat:`, `fix:`, `docs:`, `refactor:`
- Tag major milestones: `v0.1-foundation`, `v0.2-gui`, etc.

Branching:

```
main (stable)
├── develop (integration)
    ├── feature/conversation-management
    ├── feature/pdf-export
    └── feature/prompt-library
```

Testing Strategy

Test as You Build:

- After Day 3: Test data models and storage
- After Day 6: Test full GUI navigation
- After Day 9: Test complete conversation workflows
- After Day 12: Test all features end-to-end

Manual Testing Checklist (run daily):

- [ ] Application launches without errors
- [ ] New features work as expected
- [ ] Existing features still work (regression)
- [ ] No console errors or warnings
- [ ] UI is responsive

Debugging Approach

Common Issues & Solutions:

1. **Import Errors**: Check virtual environment activation, verify package installation
2. **GUI Not Rendering**: Check CustomTkinter version, verify tkinter installed
3. **API Errors**: Validate API key, check network, inspect request/response
4. **File Permission Issues**: Check directory permissions, use absolute paths
5. **Threading Issues**: Ensure GUI updates only from main thread

Debugging Prompts:

```
"This error occurs when [action]: [full error traceback].
The relevant code is: [code snippet].
What's causing this and how do I fix it?"
```

Milestone Checklist

Week 1 End (Day 5)

- [ ] Core data models implemented and tested
- [ ] Basic GUI shell operational
- [ ] Can create and display conversations
- [ ] LangChain integration working
- [ ] Basic message sending functional

Week 2 End (Day 10)

- [ ] Full conversation management (CRUD)
- [ ] Model switching works
- [ ] File attachments functional
- [ ] PDF export operational
- [ ] Prompt management basic features

Week 3 End (Day 15)

- [ ] All features complete
- [ ] Documentation finished
- [ ] Testing completed
- [ ] Application packaged
- [ ] Ready for user testing

Success Metrics

Feature Completeness:

- ✅ All 11 primary features implemented
- ✅ No critical bugs
- ✅ Core workflows tested

Code Quality:

- ✅ <20% code duplication
- ✅ 80%+ test coverage (unit tests)
- ✅ All public APIs documented
- ✅ Passes linting (minimal warnings)

User Experience:

- ✅ App starts in <3 seconds
- ✅ UI is responsive (no freezing)
- ✅ Intuitive navigation (minimal docs needed)
- ✅ Helpful error messages

Documentation:

- ✅ User manual complete
- ✅ Installation guide tested
- ✅ API reference accurate
- ✅ Troubleshooting guide helpful

Risk Management

Technical Risks

Risk	Impact	Mitigation
LangChain API changes	High	Pin specific versions, check changelog
OpenRouter rate limits	Medium	Implement retry logic, show clear errors
CustomTkinter bugs	Medium	Test early, have PyQt6 as backup plan
File corruption	High	Atomic writes, auto-backup, validation
Memory leaks (long sessions)	Medium	Implement cache clearing, test long sessions

Project Risks

Risk	Impact	Mitigation
Scope creep	High	Stick to roadmap, defer enhancements
LLM hallucinations	Medium	Test all generated code, verify logic
Integration issues	High	Build incrementally, test continuously
Time overruns	Medium	Prioritize core features, cut nice-to-haves

---

Future Enhancements (Post-MVP)

Phase 6 Ideas (if time permits or future iterations):

1. **Advanced Features**:
   - Multi-model conversations (side-by-side comparison)
   - Voice input/output integration
   - Image generation support (DALL-E, Stable Diffusion)
   - RAG (Retrieval Augmented Generation) with local documents
   - Conversation branching (save different paths)
2. **Productivity**:
   - Keyboard-first navigation mode
   - Quick actions palette (Ctrl+K)
   - Conversation templates
   - Auto-tagging conversations
   - Smart search (semantic search)
3. **Collaboration**:
   - Export conversations as shareable links
   - Import conversations from others
   - Conversation annotations/comments

    - Team workspace support
4. **Analytics**:
    - Token usage tracking and visualization
    - Cost analysis dashboard
    - Model performance comparison
    - Usage patterns and insights
5. **Technical**:
    - Plugin system for extensions
    - Custom model integration
    - Cloud sync (optional)
    - Mobile companion app

Appendix A: Resource Links

Essential Documentation

- **LangChain**: https://python.langchain.com/docs/
- **OpenRouter API**: https://openrouter.ai/docs
- **CustomTkinter**: https://github.com/TomSchimansky/CustomTkinter
- **ReportLab**: https://www.reportlab.com/docs/reportlab-userguide.pdf
- **Python Type Hints**: https://docs.python.org/3/library/typing.html

Development Tools

- **VS Code Extensions**: Python, Pylance, GitLens
- **Testing**: pytest, pytest-cov, pytest-mock
- **Code Quality**: black, flake8, mypy
- **Git**: Git, GitHub Desktop (optional)

Learning Resources

- **CustomTkinter Tutorial**: Modern Python GUI course
- **Async Python**: Understanding threading and async
- **PDF Generation**: ReportLab tutorials
- **LangChain Cookbook**: Example implementations

Appendix B: Estimated Effort Breakdown

Phase	Days	Hours	% of Total
Foundation & Core	3	24	20%
GUI Foundation	3	24	20%
Core Functionality	3	24	20%
Export & Polish	3	24	20%
Testing & Docs	3	24	20%

Phase	Days	Hours	% of Total
Total	**15**	**120**	**100%**

Daily Breakdown: ~8 hours of focused work per day **Contingency**: Built-in buffer in each work package (~20%) **Flex Time**: Days 13-15 can absorb overruns from earlier phases

---

Appendix C: Tech Stack Decision Matrix

Criterion	CustomTkinter	PyQt6	Kivy	Winner
LLM-Friendly Code	⭐⭐⭐⭐⭐	⭐⭐⭐	⭐⭐	CustomTkinter
Ubuntu Support	⭐⭐⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	Tie
Learning Curve	⭐⭐⭐⭐⭐	⭐⭐	⭐⭐⭐	CustomTkinter
Modern UI	⭐⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐	PyQt6
Documentation	⭐⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐	PyQt6
Dependencies	⭐⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐⭐	CustomTkinter
Total Score	**27/30**	**22/30**	**19/30**	**CustomTkinter**

Reasoning: CustomTkinter wins due to simplicity for LLM-assisted development, minimal dependencies, and sufficient UI capabilities for project requirements. PyQt6 would be better for a more complex application or with experienced developers.

Final Notes

Philosophy: Build incrementally, test continuously, document as you go.

When Stuck:

1. Break the problem into smaller pieces
2. Ask LLM for specific guidance
3. Test each piece independently
4. Search documentation/examples
5. Simplify the approach if needed

Quality > Speed: Better to have 80% of features working well than 100% working poorly.

Enjoy the Process: Building with LLM assistance is a new skill. Learn, experiment, iterate.

Good Luck! 🚀 You have a solid roadmap. Execute it step by step, and you'll have a functional AI Gateway application in 15 days.