Persept Documentation

Release 0.4.4

Ronak Rathore, Vikrant Rathore

Jun 29, 2025

Getting Started

1 ✨ Key Features 3

2 🎯 Supported AI Providers 5

3 📋 Persppt 7
  3.1 Introduction to Persppt . . . 7
  3.2 Getting Started . . . 10
  3.3 Installation Guide . . . 21
  3.4 Configuration Guide . . . 31
  3.5 User Guide . . . 42
  3.6 Developer Guide . . . 103
  3.7 API Reference . . . 220
  3.8 Changelog . . . 286
  3.9 License . . . 293
  3.10 Acknowledgments . . . 297

4 🔗 Quick Links 301

5 Indices and Tables 303

Perspt (pronounced "perspect," short for **Per**sonal **S**pectrum **P**ertaining **T**houghts) is a high-performance command-line interface (CLI) application that gives you a peek into the mind of Large Language Models (LLMs). Built with Rust for maximum speed and reliability, it allows you to chat with the latest AI models from multiple providers directly in your terminal using a modern, unified interface powered by the cutting-edge `genai` crate.

Documentation Navigation

- 🚀 **Quick Start**: Get up and running with Perspt in minutes. Install, configure, and start chatting with AI models. (See chapter: getting-started)
- 📚 **User Guide**: Complete guide to using Perspt effectively, from basic chat to advanced features. (See chapter: *User Guide*)
- 🛠 **Developer Guide**: Deep dive into Perspt's architecture, contribute to the project, and extend functionality. (See chapter: *Developer Guide*)
- 📖 **API Reference**: Comprehensive API documentation generated from source code comments. (See chapter: *API Reference*)

Chapter 1

✨ Key Features

🤖	**Zero-Config Startup:** Automatic provider detection from environment variables - just set your API key and run `perspt`!
🎨	**Interactive Chat Interface:** A colorful and responsive chat interface powered by Ratatui
□□	**Simple CLI Mode:** NEW! Minimal command-line mode for direct Q&A, scripting, and accessibility - perfect for Unix workflows
⚡	**Streaming Responses:** Real-time streaming of LLM responses for an interactive experience
🔀	**Multiple Provider Support:** Seamlessly switch between OpenAI, Anthropic, Google, Groq, Cohere, XAI, DeepSeek, and Ollama
🚀	**Dynamic Model Discovery:** Automatically discovers available models without manual updates
⚙️□	**Configurable:** Flexible configuration via JSON files or command-line arguments
🔄	**Input Queuing:** Type new questions while AI is responding - inputs are queued and processed sequentially
□	**Save Conversations:** Export chat sessions to text files with `/save` command for archival and reference
📜	**Markdown Rendering:** Beautiful markdown support directly in the terminal
🛡️□	**Graceful Error Handling:** Robust handling of network issues, API errors, and edge cases

Chapter 2

🎯 Supported AI Providers

OpenAI

- **GPT-4.1** - Latest and most advanced model
- **GPT-4o series** - GPT-4o, GPT-4o-mini for fast responses
- **o1 reasoning models** - o1-preview, o1-mini, o3-mini
- **GPT-4 series** - GPT-4-turbo, GPT-4 for complex tasks
- Latest model variants automatically supported

Anthropic

- Claude 3.5 (latest Sonnet, Haiku)
- Claude 3 (Opus, Sonnet, Haiku)
- Latest Claude models

Google

- **Gemini 2.5 Pro** - Latest multimodal model
- Gemini Pro, Gemini 1.5 Pro/Flash
- PaLM models

Ollama (Local)

- **Llama 3.2** - Latest Meta model
- **CodeLlama** - Code-specialized models
- **Mistral** - Fast and capable models
- **Qwen** - Multilingual models
- All popular open-source models

Cloud Providers

- **Groq**: Ultra-fast Llama 3.x inference
- **Cohere**: Command R/R+ models
- **XAI**: Grok models
- **DeepSeek**: Advanced reasoning models

> ℹ **Note**
>
> Perspt leverages the powerful genai crate for unified LLM access, ensuring automatic support for new models and providers with cutting-edge features like reasoning model support.

Chapter 3

📋 Persept

3.1 Introduction to Persept

3.1.1 What is Persept?

Persept (pronounced “perspect,” short for **Per**sonal **S**pectrum **P**ertaining **T**houghts) represents a paradigm shift in how developers and AI enthusiasts interact with Large Language Models. Born from the need for a unified, fast, and beautiful terminal-based interface to the AI world, Persept bridges the gap between raw API calls and user-friendly AI interaction. Built on the modern `genai` crate, it provides cutting-edge support for the latest reasoning models like GPT-4.1, o1-preview, o3-mini, and Gemini 2.5 Pro.

3.1.2 Philosophy & Vision

> *The keyboard hums, the screen aglow,*
> *AI’s wisdom, a steady flow.*
> *Will robots take over, it’s quite the fright,*
> *Or just provide insights, day and night?*
> *We ponder and chat, with code as our guide,*
> *Is AI our helper or our human pride?*
>
> -The Persept Manifesto

In an era where artificial intelligence is rapidly transforming how we work, learn, and create, Persept embodies the belief that the most powerful tools should be accessible, fast, and delightful to use. We envision a world where interacting with AI is as natural as opening a terminal and starting a conversation.

3.1.3 Why Persept?

The Modern Developer’s Dilemma

Today’s developers face several challenges when working with AI:

🔧 Tool Fragmentation Multiple providers, different APIs, inconsistent interfaces. Switching between OpenAI, Anthropic, Google, and others requires learning different tools and maintaining separate configurations.

🐌 Performance Issues Web-based interfaces can be slow, unreliable, and resource-heavy. Developers need something that matches the speed of their terminal workflow.

🎨 Poor Terminal Integration Most AI tools don’t integrate well with terminal-based workflows, forcing context switches that break concentration and productivity.

🔓 Vendor Lock-in Many tools tie you to specific providers or models, making it difficult to experiment with different AI capabilities or switch providers based on use case.

The Perspt Solution

Perspt addresses these challenges through:

Unified Interface
A single, consistent interface for all major LLM providers. Switch between GPT-4, Claude, Gemini, and others without changing your workflow.

Terminal-Native Design
Built specifically for terminal users who value speed, keyboard shortcuts, and seamless integration with existing development workflows.

Performance First
Written in Rust for maximum performance. Streaming responses, efficient memory usage, and instant startup times.

Provider Agnostic
Leverages the modern genai crate for automatic support of new models and providers, including cutting-edge reasoning models and ultra-fast inference platforms.

Beautiful UX
Rich markdown rendering, syntax highlighting, and a responsive interface powered by Ratatui make AI interaction delightful.

3.1.4 Core Principles

Simplicity

Perspt follows the Unix philosophy: do one thing and do it well. It's designed to be a straightforward, powerful chat interface without unnecessary complexity.

```
# Simple as it gets
perspt
# Start chatting immediately
```

Performance

Every design decision prioritizes speed and efficiency:

- **Rust foundation** for memory safety and performance
- **Streaming responses** for immediate feedback
- **Minimal resource usage** - runs efficiently even on modest hardware
- **Fast startup** - be chatting within seconds

Extensibility

Built with the future in mind:

- **Plugin architecture** ready for extensions
- **Provider abstraction** makes adding new AI services trivial
- **Configuration flexibility** adapts to any workflow
- **Open source** encourages community contributions

Developer Experience

Created by developers, for developers:

- **Terminal-first design** respects your workflow
- **Keyboard-driven** interface for maximum efficiency

- **Comprehensive error handling** with helpful messages
- **Detailed documentation** and examples

3.1.5 Use Cases

Perspt excels in various scenarios:

Development

- **Code review and analysis**
- **Architecture discussions**
- **Bug troubleshooting**
- **Documentation generation**
- **Learning new technologies**

Research

- **Literature reviews**
- **Concept exploration**
- **Data analysis discussions**
- **Hypothesis testing**
- **Academic writing assistance**

Creative Work

- **Content brainstorming**
- **Writing assistance**
- **Creative problem solving**
- **Idea validation**
- **Story development**

Daily Tasks

- **Quick questions**
- **Email drafting**
- **Decision making**
- **Learning and tutorials**
- **General assistance**

3.1.6 The Technology Stack

Perspt is built on a foundation of cutting-edge technologies:

Rust Core
Memory-safe, performant, and reliable. Rust ensures Perspt is fast, secure, and maintainable.

Ratatui TUI Framework
Rich terminal user interfaces with responsive design, smooth animations, and beautiful rendering.

genai Crate Integration
Unified access to multiple LLM providers through a single, modern Rust API with support for cutting-edge reasoning models.

Tokio Async Runtime
Efficient handling of concurrent operations, streaming responses, and network communication.

Serde JSON
Robust configuration management and API communication with excellent error handling.

3.1.7 Community & Philosophy

Perspt is more than just a tool—it's a community of developers, researchers, and AI enthusiasts who believe in the power of accessible, high-quality tools. We embrace:

Open Source Values
Transparency, collaboration, and shared ownership of the tools we use daily.

Inclusive Design
Tools should work for everyone, regardless of technical background or accessibility needs.

Continuous Learning
The AI landscape evolves rapidly, and our tools should evolve with it.

Quality Over Quantity
Better to have fewer features that work exceptionally well than many features that work poorly.

3.1.8 What's Next?

Ready to dive in? Here's your path forward:

1. **Installation**: Follow our *Installation Guide* guide to get Perspt running on your system
2. **Quick Start**: Jump into the *Getting Started* tutorial for your first AI conversation
3. **Configuration**: Learn about *Configuration Guide* options to customize your experience
4. **User Guide**: Explore the complete *User Guide* for advanced features
5. **Development**: Interested in contributing? Check out our *Developer Guide*

Note

Perspt is actively developed and maintained. Join our community to stay updated on new features, share feedback, and contribute to the project's evolution.

See also

- *Getting Started* - Get up and running in minutes
- *Installation Guide* - Detailed installation instructions
- *User Guide* - Complete user documentation
- *Developer Guide* - Developer and contributor resources

3.2 Getting Started

Welcome to Perspt! This guide will get you up and running with your first AI conversation in just a few minutes.

3.2.1 Prerequisites

Before installing Perspt, ensure you have the following:

System Requirements

Component	Requirement
Operating System	Linux, macOS, or Windows
Rust Toolchain	Rust 1.82.0 or later
Terminal	Any modern terminal emulator
Internet Connection	Required for AI provider API calls

API Keys

You'll need an API key from at least one AI provider:

OpenAI

1. Visit OpenAI Platform
2. Sign up or log in to your account
3. Navigate to API Keys section
4. Create a new API key
5. Copy and save it securely

```
export OPENAI_API_KEY="sk-your-openai-api-key-here"
```

Anthropic

1. Visit Anthropic Console
2. Sign up or log in
3. Go to API Keys
4. Generate a new key
5. Save it securely

```
export ANTHROPIC_API_KEY="your-anthropic-api-key-here"
```

Google

1. Visit Google AI Studio
2. Create or select a project
3. Generate API key
4. Configure authentication

```
export GOOGLE_API_KEY="your-google-api-key-here"
```

Ollama (Local)

1. Install Ollama from ollama.ai
2. Pull a model
3. Start Ollama service

```
ollama pull llama3.2
# Ollama service starts automatically
```

3.2.2 Quick Installation

Method 1: From Source (Recommended)

```
# Clone the repository
git clone https://github.com/eonseed/perspt.git
cd perspt

# Build the project
cargo build --release

# Install to your PATH (optional)
cargo install --path .

# Or run directly
./target/release/perspt
```

Method 2: Using Cargo

```
# Install from crates.io (when published)
cargo install perspt

# Run Perspt
perspt
```

Method 3: Download Binary

```
# Download the latest release (replace with actual URL)
curl -L https://github.com/eonseed/perspt/releases/latest/download/perspt-linux-x86_64.tar.
 ↪gz | tar xz

# Make executable and move to PATH
chmod +x perspt
sudo mv perspt /usr/local/bin/
```

3.2.3 Your First Conversation

Let's start your first AI conversation with Perspt! You can choose between two interface modes:

1. **Interactive TUI Mode** - Rich terminal interface with markdown rendering (default)
2. **Simple CLI Mode** - Minimal command-line interface for scripting and accessibility (NEW!)

Zero-Config Quick Start

NEW! Perspt now features intelligent automatic provider detection. Simply set an environment variable for any supported provider, and Perspt will automatically detect and use it - no additional configuration needed!

> ℹ **Note**
>
> **Automatic Provider Detection Priority:**
>
> 1. OpenAI (`OPENAI_API_KEY`)
> 2. Anthropic (`ANTHROPIC_API_KEY`)

3. Google Gemini (`GEMINI_API_KEY`)
4. Groq (`GROQ_API_KEY`)
5. Cohere (`COHERE_API_KEY`)
6. XAI (`XAI_API_KEY`)
7. DeepSeek (`DEEPSEEK_API_KEY`)
8. Ollama (no API key needed - auto-detected if running)

Interactive TUI Mode (Default)

Rich terminal interface with markdown rendering and scrollable history:

```
# Set your API key
export OPENAI_API_KEY="sk-your-actual-api-key-here"

# Launch Perspt in TUI mode (default)
perspt
# Automatically uses OpenAI with gpt-4o-mini
```

Simple CLI Mode (NEW!)

Minimal command-line interface perfect for scripting and accessibility:

```
# Set your API key
export OPENAI_API_KEY="sk-your-actual-api-key-here"

# Launch Perspt in simple CLI mode
perspt --simple-cli
# Unix-style prompt with streaming responses
```

Anthropic Claude

```
# Set your API key
export ANTHROPIC_API_KEY="sk-ant-your-key"

# TUI mode (default)
perspt

# Simple CLI mode
perspt --simple-cli
# Automatically uses Anthropic with claude-3-5-sonnet-20241022
```

Google Gemini

```
# Set your API key
export GEMINI_API_KEY="your-gemini-key"

# TUI mode (default)
perspt

# Simple CLI mode with logging
perspt --simple-cli --log-file gemini-session.txt
# Automatically uses Gemini with gemini-1.5-flash
```

Ollama (Local)

```
# Just make sure Ollama is running
ollama serve

# TUI mode (default)
perspt

# Simple CLI mode for scripting
perspt --simple-cli
# Auto-detects Ollama if no other providers found
```

Step 1: Set Your API Key (Manual Configuration)

If you prefer manual configuration or want to override automatic detection:

```
# For OpenAI (most common)
export OPENAI_API_KEY="sk-your-actual-api-key-here"

# Verify it's set
echo $OPENAI_API_KEY
```

Step 2: Launch Perspt

Choose between TUI mode (rich interface) or Simple CLI mode (minimal interface):

TUI Mode (Default)

```
# Start with automatic detection (recommended)
perspt

# Or specify provider manually
perspt --provider openai --model gpt-4o-mini
```

You should see a welcome screen like this:

```
┌──────────────────────────────────────────────────────────┐
│                     Welcome to Perspt!                   │
│              Your Terminal's Window to AI                │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  Provider: OpenAI                                        │
│  Model: gpt-4o-mini                                      │
│  Status: Ready                                           │
│                                                          │
│  Type your message and press Enter to start chatting!    │
│  Press Ctrl+C to exit                                    │
│                                                          │
└──────────────────────────────────────────────────────────┘

You:
```

Simple CLI Mode (NEW!)

```
# Start simple CLI mode
perspt --simple-cli

# With session logging
perspt --simple-cli --log-file my-session.txt
```

You should see a minimal interface like this:

```
Perspt Simple CLI Mode
Model: gpt-4o-mini
Type 'exit' or press Ctrl+D to quit.

>
```

The simple CLI mode is perfect for:

- **Scripting and automation**
- **Accessibility (screen readers)**
- **Unix-style workflows**
- **Session logging**
- **Lightweight usage**

Step 3: Start Chatting

Type your first message and press Enter:

```
You: Hello! Can you explain what Rust is in simple terms?

Assistant: Hello! Rust is a modern programming language that's designed to be both
fast and safe. Here are the key things that make Rust special:

**Speed**: Rust programs run as fast as C and C++ programs because it compiles
directly to machine code.

**Safety**: Unlike C/C++, Rust prevents common programming errors like accessing
invalid memory or data races in concurrent programs.

**No Garbage Collector**: Rust manages memory automatically without needing a
garbage collector, which keeps programs fast and predictable.

**Growing Ecosystem**: It's increasingly used for web backends, system programming,
blockchain, and even WebAssembly applications.

Think of Rust as giving you the performance of low-level languages like C, but
with the safety and ergonomics of higher-level languages like Python or Java.

You:
```

Congratulations! 🎉 You've successfully started your first conversation with Perspt.

3.2.4 Basic Commands

While chatting, you can use these keyboard shortcuts:

Shortcut	Action
Enter	Send your message
Ctrl+C	Exit Perspt
↑/↓ Arrow Keys	Scroll through chat history
Page Up/Down	Scroll chat quickly
Ctrl+L	Clear the screen

3.2.5 Switching Models

You can easily switch between different AI models and providers:

OpenAI Models

```
# Use GPT-4
perspt --model-name gpt-4

# Use GPT-4 Turbo
perspt --model-name gpt-4-turbo-preview

# Use GPT-4o Mini (recommended for most use cases)
perspt --model-name gpt-4o-mini

# Use latest GPT-4.1
perspt --model-name gpt-4.1
```

Other Providers

```
# Use Anthropic Claude
perspt --provider-type anthropic --model-name claude-3-sonnet-20240229

# Use Google Gemini
perspt --provider-type google --model-name gemini-pro

# Use Ollama (Local)
perspt --provider-type ollama --model-name llama3.2
```

List Available Models

```
# See all available models for your provider
perspt --list-models
```

3.2.6 Basic Configuration

For frequent use, create a configuration file to set your preferences:

Create Config File

```
# Create a config.json file
touch config.json
```

Add your configuration:

```
{
  "api_key": "your-api-key-here",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai",
  "provider_type": "openai"
}
```

Use Config File

```
# Use your configuration file
perspt --config config.json

# Or place config.json in the same directory as perspt
perspt
```

3.2.7 Common First-Time Issues

Issue: "API key not found"

Solution: Make sure your API key is properly set:

```
# Check if the key is set
echo $OPENAI_API_KEY

# If empty, set it again
export OPENAI_API_KEY="sk-your-key-here"
```

Issue: "Model not available"

Solution: Check available models for your provider:

```
# List available models
perspt --list-models

# Use a specific model that's available
perspt --model-name gpt-4o-mini
```

Issue: "Network connection failed"

Solution: Check your internet connection and API key permissions:

```
# Test with a simple curl command
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     "https://api.openai.com/v1/models"
```

Issue: Terminal display problems

Solution: Ensure your terminal supports modern features:

```
# Try a different terminal emulator if needed
# Update your terminal if it's very old
# Check terminal size
echo $COLUMNS x $LINES
```

3.2.8 Next Steps

Now that you have Perspt running, explore these topics:

📖 User Guide Learn about advanced features, keyboard shortcuts, and productivity tips.

User Guide ⚙️ Configuration Customize Perspt to fit your workflow perfectly.

Configuration Guide 🔀 Multiple Providers Set up and switch between different AI providers.

AI Providers 🛠️ Advanced Features Discover powerful features like input queuing and markdown rendering.

Advanced Features

3.2.9 Example Conversations

Here are some example conversations to inspire your Perspt usage:

Code Review

~~~
You: Can you review this Rust function?

```rust
fn calculate_fibonacci(n: u32) -> u32 {
    if n <= 1 {
        return n;
    }
    calculate_fibonacci(n - 1) + calculate_fibonacci(n - 2)
}
~~~

Assistant: I see you've implemented a recursive Fibonacci function. Here's my review:

Issues:
1. **Performance**: This has exponential time complexity O(2^n) due to repeated␣
 ↪calculations
2. **Stack overflow**: Large values of `n` will cause stack overflow
3. **Type limitations**: `u32` limits the range of Fibonacci numbers you can calculate

Improved version:
```rust
fn fibonacci_iterative(n: u32) -> u64 {
    if n <= 1 {
        return n as u64;
    }

    let (mut a, mut b) = (0u64, 1u64);
```

(continues on next page)

(continued from previous page)

~~~
    for _ in 2..=n {
        let temp = a + b;
        a = b;
        b = temp;
    }
    b
}
```

This iterative version is O(n) time and O(1) space, much more efficient!
~~~

Technical Explanation

~~~
You: Explain how async/await works in Rust

Assistant: Async/await in Rust is a powerful feature for handling concurrent operations.
Here's how it works:

## The Basics

**Async Functions**: Return a `Future` instead of executing immediately
```rust
async fn fetch_data() -> String {
    // This doesn't run until awaited
    "data".to_string()
}
```

**Await**: Actually executes the future
```rust
let result = fetch_data().await;
```

## Under the Hood

1. **State Machine**: The compiler transforms async functions into state machines
2. **Zero-cost**: No heap allocations for simple futures
3. **Cooperative**: Tasks yield control at await points

## Key Concepts

- **Futures are lazy**: They don't do anything until polled
- **Executors**: Runtime systems (like Tokio) that drive futures to completion
- **Non-blocking**: Awaiting doesn't block the thread

This makes Rust excellent for high-performance concurrent applications!
~~~

3.2.10 Troubleshooting

No Provider Detected

If you see an error message like this when launching Perspt:

```
▯ No LLM provider configured!

To get started, either:
  1. Set an environment variable for a supported provider:
     • OPENAI_API_KEY=sk-your-key
     • ANTHROPIC_API_KEY=sk-ant-your-key
     • GEMINI_API_KEY=your-key
     # ... (shows all supported providers)

  2. Use command line arguments:
     perspt --provider openai --api-key sk-your-key
```

Solution: Set at least one API key environment variable:

```
# Quick fix - set any supported provider
export OPENAI_API_KEY="sk-your-actual-key"
perspt  # Should now auto-detect and start
```

Provider Priority

If you have multiple API keys set and want to use a specific provider:

```
# Override automatic detection
perspt --provider anthropic  # Forces Anthropic even if OpenAI key is set

# Or unset other providers temporarily
unset OPENAI_API_KEY
export ANTHROPIC_API_KEY="your-key"
perspt  # Now auto-detects Anthropic
```

Connection Issues

If Perspt detects your provider but can't connect:

1. **Check your API key**: Ensure it's valid and has sufficient credits
2. **Test your connection**: Try a simple curl request to the provider's API
3. **Check firewall**: Ensure your network allows HTTPS connections
4. **Try Ollama**: For offline usage, install Ollama for local models

```
# Test OpenAI connection
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     https://api.openai.com/v1/models
```

3.2.11 Tips for Success

1. **Start Simple**: Begin with basic conversations before exploring advanced features
2. **Experiment**: Try different models and providers to find what works best for your use case
3. **Use Configuration**: Set up a config file for your most common settings
4. **Join the Community**: Connect with other Perspt users for tips and support

5. **Stay Updated**: Check for updates regularly to get new features and improvements

See also

- *Installation Guide* - Detailed installation instructions
- *Configuration Guide* - Complete configuration guide
- *Basic Usage* - Everyday usage patterns
- *Troubleshooting* - Common issues and solutions

3.3 Installation Guide

This comprehensive guide covers all the ways to install Perspt on your system, from simple binary downloads to building from source.

3.3.1 System Requirements

Minimum Requirements

Component	Requirement
Operating System	Linux, macOS, Windows 10+
Architecture	x86_64, ARM64 (Apple Silicon)
Memory	50MB RAM minimum
Storage	10MB disk space
Terminal	Any terminal with UTF-8 support
Network	Internet connection for AI API calls

Recommended Requirements

Component	Recommendation
Terminal	Modern terminal with 256+ colors and Unicode support
Font	Monospace font with good Unicode coverage (e.g., Fira Code, JetBrains Mono)
Shell	Bash, Zsh, Fish, or PowerShell
Memory	100MB+ RAM for optimal performance

3.3.2 Quick Install

Choose your preferred installation method:

📦 Binary Download (Fastest)

Download pre-built binaries for immediate use:

Linux x86_64:

```
curl -L https://github.com/eonseed/perspt/releases/latest/download/perspt-linux-x86_64.tar.
↪gz | tar xz
chmod +x perspt
sudo mv perspt /usr/local/bin/
```

macOS (Intel):

```
curl -L https://github.com/eonseed/perspt/releases/latest/download/perspt-darwin-x86_64.tar.
↪gz | tar xz
chmod +x perspt
sudo mv perspt /usr/local/bin/
```

macOS (Apple Silicon):

```
curl -L https://github.com/eonseed/perspt/releases/latest/download/perspt-darwin-arm64.tar.
↪gz | tar xz
chmod +x perspt
sudo mv perspt /usr/local/bin/
```

Windows:

```
# Download from GitHub releases page
# Extract perspt.exe and add to PATH
```

🦀 Cargo Install

Install using Rust's package manager:

```
# Install from crates.io
cargo install perspt

# Or install the latest development version
cargo install --git https://github.com/eonseed/perspt
```

🏗️ Build from Source

Build the latest version from source:

```
# Clone repository
git clone https://github.com/eonseed/perspt.git
cd perspt

# Build release version
cargo build --release

# Install to cargo bin
cargo install --path .
```

3.3.3 Package Managers

Homebrew (macOS/Linux)

```
# Add tap (when available)
brew tap eonseed/perspt

# Install
brew install perspt

# Update
brew upgrade perspt
```

Scoop (Windows)

```
# Add bucket (when available)
scoop bucket add perspt https://github.com/eonseed/scoop-perspt

# Install
scoop install perspt

# Update
scoop update perspt
```

Chocolatey (Windows)

```
# Install (when available)
choco install perspt

# Update
choco upgrade perspt
```

APT (Debian/Ubuntu)

```
# Add repository (when available)
curl -fsSL https://releases.perspt.dev/gpg | sudo gpg --dearmor -o /usr/share/keyrings/
↪perspt.gpg
echo "deb [signed-by=/usr/share/keyrings/perspt.gpg] https://releases.perspt.dev/apt␣
↪stable main" | sudo tee /etc/apt/sources.list.d/perspt.list

# Install
sudo apt update
sudo apt install perspt
```

RPM (Red Hat/Fedora)

```
# Add repository (when available)
sudo dnf config-manager --add-repo https://releases.perspt.dev/rpm/perspt.repo

# Install
sudo dnf install perspt
```

3.3.4 Building from Source

Prerequisites

```
# Install Rust (if not already installed)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env

# Verify installation
rustc --version
cargo --version
```

Clone and Build

```
# Clone the repository
git clone https://github.com/eonseed/perspt.git
cd perspt

# Build in release mode
cargo build --release

# The binary will be in target/release/perspt
./target/release/perspt --version
```

Install System-Wide

```
# Option 1: Using cargo install
cargo install --path .

# Option 2: Manual installation
sudo cp target/release/perspt /usr/local/bin/
sudo chmod +x /usr/local/bin/perspt

# Option 3: User-local installation
mkdir -p ~/.local/bin
cp target/release/perspt ~/.local/bin/
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Development Build

For development and testing:

```
# Clone with all development tools
git clone https://github.com/eonseed/perspt.git
cd perspt

# Install development dependencies
cargo install cargo-watch cargo-edit

# Build in debug mode
cargo build

# Run tests
cargo test

# Run with hot reload during development
cargo watch -x run
```

3.3.5 Docker Installation

Run Perspt in a Docker container:

Basic Usage

```
# Pull the image
docker pull ghcr.io/eonseed/perspt:latest

# Run interactively
docker run -it --rm \
  -e OPENAI_API_KEY="$OPENAI_API_KEY" \
  ghcr.io/eonseed/perspt:latest
```

With Configuration

```
# Create a config directory
mkdir -p ~/.config/perspt

# Create your config.json
cat > ~/.config/perspt/config.json << EOF
{
  "api_key": "your-api-key-here",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai"
}
EOF

# Run with mounted config
docker run -it --rm \
  -v ~/.config/perspt:/app/config \
  ghcr.io/eonseed/perspt:latest \
  --config /app/config/config.json
```

Docker Compose

Create a *docker-compose.yml* file:

```
version: '3.8'
services:
  perspt:
    image: ghcr.io/eonseed/perspt:latest
    stdin_open: true
    tty: true
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    volumes:
      - ./config:/app/config
    command: ["--config", "/app/config/config.json"]
```

Run with:

```
docker-compose run --rm perspt
```

3.3.6 Platform-Specific Instructions

Linux

Ubuntu/Debian:

```
# Update package list
sudo apt update

# Install dependencies for building (if building from source)
sudo apt install build-essential curl git

# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env

# Install Perspt
cargo install perspt
```

Arch Linux:

```
# Install from AUR (when available)
yay -S perspt

# Or build from source
sudo pacman -S rust git
git clone https://github.com/eonseed/perspt.git
cd perspt
cargo build --release
```

CentOS/RHEL/Fedora:

```
# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env

# Install development tools
sudo dnf groupinstall "Development Tools"
sudo dnf install git

# Install Perspt
cargo install perspt
```

macOS

Using Homebrew (Recommended):

```
# Install Homebrew if not already installed
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.
↪sh)"

# Install Rust
brew install rust
```

(continues on next page)

(continued from previous page)

```
# Install Perspt
cargo install perspt
```

Using MacPorts:

```
# Install Rust
sudo port install rust

# Install Perspt
cargo install perspt
```

Manual Installation:

```
# Install Xcode command line tools
xcode-select --install

# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env

# Install Perspt
cargo install perspt
```

Windows

Using Chocolatey:

```
# Install Chocolatey
Set-ExecutionPolicy Bypass -Scope Process -Force
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.
↪ServicePointManager]::SecurityProtocol -bor 3072
iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/
↪install.ps1'))

# Install Rust
choco install rust

# Install Perspt
cargo install perspt
```

Using Scoop:

```
# Install Scoop
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
irm get.scoop.sh | iex

# Install Rust
scoop install rust

# Install Perspt
cargo install perspt
```

Manual Installation:

1. Download and install Rust from rustup.rs
2. Open Command Prompt or PowerShell
3. Run: `cargo install perspt`

3.3.7 Verification

After installation, verify that Perspt is working correctly:

```
# Check version
perspt --version

# Check help
perspt --help

# Test basic functionality (requires API key)
export OPENAI_API_KEY="your-key-here"
perspt --model-name gpt-4o-mini
```

You should see output similar to:

```
perspt 0.4.0
Your Terminal's Window to the AI World
```

3.3.8 Updating Perspt

Cargo Installation

```
# Update to latest version
cargo install perspt --force

# Or update all cargo packages
cargo install-update -a
```

Binary Installation

```
# Download and replace binary
curl -L https://github.com/eonseed/perspt/releases/latest/download/perspt-linux-x86_64.tar.
↪gz | tar xz
sudo mv perspt /usr/local/bin/
```

Package Managers

```
# Homebrew
brew upgrade perspt

# APT
sudo apt update && sudo apt upgrade perspt

# DNF
sudo dnf upgrade perspt

# Chocolatey
```

(continues on next page)

(continued from previous page)

```
choco upgrade perspt

# Scoop
scoop update perspt
```

3.3.9 Uninstallation

Cargo Installation

```
# Uninstall using cargo
cargo uninstall perspt
```

Manual Binary

```
# Remove binary
sudo rm /usr/local/bin/perspt

# Remove configuration (optional)
rm -rf ~/.config/perspt
```

Package Managers

```
# Homebrew
brew uninstall perspt

# APT
sudo apt remove perspt

# DNF
sudo dnf remove perspt

# Chocolatey
choco uninstall perspt

# Scoop
scoop uninstall perspt
```

3.3.10 Troubleshooting

Common Issues

"Command not found" error:

```
# Check if cargo bin is in PATH
echo $PATH | grep -q "$HOME/.cargo/bin" && echo "Cargo bin in PATH" || echo "Cargo bin NOT in PATH"

# Add to PATH if missing
echo 'export PATH="$HOME/.cargo/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Permission denied:

```
# Make sure the binary is executable
chmod +x /usr/local/bin/perspt

# Or use without sudo
mkdir -p ~/.local/bin
cp perspt ~/.local/bin/
export PATH="$HOME/.local/bin:$PATH"
```

Build failures:

```
# Update Rust toolchain
rustup update

# Clear cargo cache
cargo clean

# Rebuild
cargo build --release
```

Missing dependencies on Linux:

```
# Ubuntu/Debian
sudo apt install build-essential pkg-config libssl-dev

# CentOS/RHEL/Fedora
sudo dnf groupinstall "Development Tools"
sudo dnf install pkgconfig openssl-devel
```

Getting Help

If you encounter issues during installation:

1. **Check the GitHub Issues**: Issues Page
2. **Join the Discussion**: GitHub Discussions
3. **Read the FAQ**: *Troubleshooting*
4. **Contact Support**: Create a new issue with: - Your operating system and version - Rust version (`rustc --version`) - Installation method used - Complete error message

3.3.11 Next Steps

After successful installation:

1. **Set up API keys**: *Configuration Guide*
2. **Learn basic usage**: *Getting Started*
3. **Explore features**: *User Guide*
4. **Join the community**: GitHub Discussions

See also

- *Getting Started* - Your first conversation
- *Configuration Guide* - Setting up API keys and preferences
- *Basic Usage* - Everyday usage patterns

- *Troubleshooting* - Common issues and solutions

3.4 Configuration Guide

Perspt offers flexible configuration options to customize your AI chat experience. This guide covers all configuration methods, from zero-config automatic provider detection to advanced JSON configurations.

3.4.1 Automatic Provider Detection (Zero-Config)

NEW! Perspt now features intelligent automatic provider detection that makes getting started as simple as setting an environment variable.

How It Works

When you launch Perspt without specifying a provider, it automatically scans your environment variables for supported provider API keys and selects the first one found based on this priority order:

1. **OpenAI** (`OPENAI_API_KEY`) - *Default model: gpt-4o-mini*
2. **Anthropic** (`ANTHROPIC_API_KEY`) - *Default model: claude-3-5-sonnet-20241022*
3. **Google Gemini** (`GEMINI_API_KEY`) - *Default model: gemini-1.5-flash*
4. **Groq** (`GROQ_API_KEY`) - *Default model: llama-3.1-70b-versatile*
5. **Cohere** (`COHERE_API_KEY`) - *Default model: command-r-plus*
6. **XAI** (`XAI_API_KEY`) - *Default model: grok-beta*
7. **DeepSeek** (`DEEPSEEK_API_KEY`) - *Default model: deepseek-chat*
8. **Ollama** (auto-detected if running) - *Default model: llama3.2*

Quick Examples

```
# Option 1: OpenAI (highest priority)
export OPENAI_API_KEY="sk-your-key"
perspt  # Auto-detects OpenAI, uses gpt-4o-mini

# Option 2: Multiple providers - OpenAI takes priority
export OPENAI_API_KEY="sk-your-openai-key"
export ANTHROPIC_API_KEY="sk-ant-your-anthropic-key"
perspt  # Uses OpenAI (higher priority)

# Option 3: Force a specific provider
perspt --provider anthropic  # Uses Anthropic even if OpenAI key exists

# Option 4: Ollama (no API key needed)
# Just ensure Ollama is running: ollama serve
perspt  # Auto-detects Ollama if no other keys found
```

What Happens When No Providers Are Found

If no API keys are detected, Perspt displays helpful setup instructions:

```
□ No LLM provider configured!

To get started, either:
  1. Set an environment variable for a supported provider:
```

(continues on next page)

(continued from previous page)

```
      • OPENAI_API_KEY=sk-your-key
      • ANTHROPIC_API_KEY=sk-ant-your-key
      • GEMINI_API_KEY=your-key
      # ... (shows all supported providers)

   2. Use command line arguments:
      perspt --provider openai --api-key sk-your-key
```

3.4.2 Manual Configuration Methods

For more control or advanced setups, Perspt supports traditional configuration methods with the following priority order (highest to lowest):

1. **Command-line arguments** (highest priority)
2. **Configuration file** (`config.json`)
3. **Environment variables**
4. **Automatic provider detection**
5. **Default values** (lowest priority)

This means command-line arguments will override config file settings, which override environment variables, and so on.

3.4.3 Environment Variables

Environment variables are the simplest way to configure Perspt and enable automatic provider detection.

API Keys (Auto-Detection Enabled)

Setting any of these environment variables enables automatic provider detection:

```
# OpenAI (Priority 1 - will be auto-selected first)
export OPENAI_API_KEY="sk-your-openai-api-key-here"

# Anthropic (Priority 2)
export ANTHROPIC_API_KEY="your-anthropic-api-key-here"

# Google Gemini (Priority 3)
export GEMINI_API_KEY="your-google-api-key-here"

# Groq (Priority 4)
export GROQ_API_KEY="your-groq-api-key-here"

# Cohere (Priority 5)
export COHERE_API_KEY="your-cohere-api-key-here"

# XAI (Priority 6)
export XAI_API_KEY="your-xai-api-key-here"

# DeepSeek (Priority 7)
export DEEPSEEK_API_KEY="your-deepseek-api-key-here"

# Ollama (Priority 8 - no API key needed, auto-detected if service is running)
# Just run: ollama serve
```

Note

Automatic Detection: Simply set any of these environment variables and run `perspt` with no arguments. Perspt will automatically detect and use the highest priority provider available.

Legacy Provider Settings (Manual Override)

These variables override automatic detection and force manual configuration:

```
# Default provider
export PERSPT_PROVIDER="openai"

# Default model
export PERSPT_MODEL="gpt-4o-mini"

# Custom API base URL
export PERSPT_API_BASE="https://api.openai.com/v1"
```

3.4.4 Configuration File

For persistent settings, create a `config.json` file:

Basic Configuration

```
{
  "api_key": "your-api-key-here",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai",
  "provider_type": "openai"
}
```

Complete Configuration

```
{
  "api_key": "sk-your-openai-api-key",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai",
  "provider_type": "openai",
  "providers": {
    "openai": "https://api.openai.com/v1",
    "anthropic": "https://api.anthropic.com",
    "google": "https://generativelanguage.googleapis.com/v1beta",
    "azure": "https://your-resource.openai.azure.com/"
  },
  "ui": {
    "theme": "dark",
    "show_timestamps": true,
    "markdown_rendering": true,
    "auto_scroll": true
  },
  "behavior": {
```

(continues on next page)

(continued from previous page)

```
    "stream_responses": true,
    "input_queuing": true,
    "auto_save_history": false,
    "max_history_length": 1000
  },
  "advanced": {
    "request_timeout": 30,
    "retry_attempts": 3,
    "retry_delay": 1.0,
    "concurrent_requests": 1
  }
}
```

Configuration File Locations

Perspt searches for configuration files in this order:

1. **Specified path**: `perspt --config /path/to/config.json`
2. **Current directory**: `./config.json`
3. **User config directory**: - Linux: `~/.config/perspt/config.json` - macOS: `~/Library/Application Support/perspt/config.json` - Windows: `%APPDATA%/perspt/config.json`

3.4.5 Provider Configuration

OpenAI

Environment Variables

```
export OPENAI_API_KEY="sk-your-key-here"
export PERSPT_PROVIDER="openai"
export PERSPT_MODEL="gpt-4o-mini"
```

Config File

```
{
  "api_key": "sk-your-key-here",
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "providers": {
    "openai": "https://api.openai.com/v1"
  }
}
```

Command Line

```
perspt --provider-type openai \
       --model-name gpt-4o-mini \
       --api-key "sk-your-key-here"
```

Available Models: - `gpt-4.1` - Latest and most advanced GPT model - `gpt-4o` - Latest GPT-4 Omni model - `gpt-4o-mini` - Faster, cost-effective GPT-4 Omni - `o1-preview` - Advanced reasoning model - `o1-mini` - Efficient reasoning model - `o3-mini` - Next-generation reasoning model - `gpt-4-turbo` - Latest GPT-4 Turbo - `gpt-4` - Standard GPT-4

Anthropic

Environment Variables

```
export ANTHROPIC_API_KEY="your-key-here"
export PERSPT_PROVIDER="anthropic"
export PERSPT_MODEL="claude-3-sonnet-20240229"
```

Config File

```
{
  "api_key": "your-key-here",
  "provider_type": "anthropic",
  "default_model": "claude-3-sonnet-20240229",
  "providers": {
    "anthropic": "https://api.anthropic.com"
  }
}
```

Command Line

```
perspt --provider-type anthropic \
       --model-name claude-3-sonnet-20240229 \
       --api-key "your-key-here"
```

Available Models: - `claude-3-opus-20240229` - Most capable Claude model - `claude-3-sonnet-20240229` - Balanced performance and speed - `claude-3-haiku-20240307` - Fastest Claude model

Google (Gemini)

Environment Variables

```
export GOOGLE_API_KEY="your-key-here"
export PERSPT_PROVIDER="google"
export PERSPT_MODEL="gemini-pro"
```

Config File

```
{
  "api_key": "your-key-here",
  "provider_type": "google",
  "default_model": "gemini-pro",
  "providers": {
    "google": "https://generativelanguage.googleapis.com/v1beta"
  }
}
```

Command Line

```
perspt --provider-type google \
       --model-name gemini-pro \
       --api-key "your-key-here"
```

Available Models: - `gemini-pro` - Google's most capable model - `gemini-pro-vision` - Multimodal capabilities

3.4.6 Command-Line Options

Perspt supports extensive command-line configuration:

Basic Options

```
perspt [OPTIONS]
```

Option	Description
`--config <PATH>`	Path to configuration file
`--provider-type <TYPE>`	AI provider (openai, anthropic, google, groq, cohere, xai, deepseek, ollama)
`--model-name <MODEL>`	Specific model to use
`--api-key <KEY>`	API key for authentication
`--list-models`	List available models for provider
`--help`	Show help information
`--version`	Show version information

Advanced Options

```
# Custom API endpoint
perspt --api-base "https://your-custom-endpoint.com/v1"

# Increase request timeout
perspt --timeout 60

# Disable streaming responses
perspt --no-stream

# Set maximum retries
perspt --max-retries 5

# Custom user agent
perspt --user-agent "MyApp/1.0"
```

Examples

```
# Use specific OpenAI model
perspt --provider-type openai --model-name gpt-4

# Use Anthropic with custom timeout
perspt --provider-type anthropic \
       --model-name claude-3-sonnet-20240229 \
       --timeout 45

# Use custom configuration file
perspt --config ~/.perspt/work-config.json

# List available models
perspt --provider-type openai --list-models
```

3.4.7 UI Customization

Interface Settings

Configure the terminal interface appearance:

```
{
  "ui": {
    "theme": "dark",
    "show_timestamps": true,
    "timestamp_format": "%H:%M",
    "markdown_rendering": true,
    "syntax_highlighting": true,
    "auto_scroll": true,
    "scroll_buffer": 1000,
    "word_wrap": true,
    "show_token_count": false
  }
}
```

Color Themes

Customize colors for different message types:

```
{
  "ui": {
    "colors": {
      "user_message": "#60a5fa",
      "assistant_message": "#10b981",
      "error_message": "#ef4444",
      "warning_message": "#f59e0b",
      "info_message": "#8b5cf6",
      "timestamp": "#6b7280",
      "border": "#374151",
      "background": "#111827"
    }
  }
}
```

3.4.8 Behavior Settings

Streaming and Responses

```
{
  "behavior": {
    "stream_responses": true,
    "input_queuing": true,
    "auto_retry_on_error": true,
    "show_thinking_indicator": true,
    "preserve_context": true
  }
}
```

History Management

```
{
  "behavior": {
    "auto_save_history": true,
    "history_file": "~/.perspt/chat_history.json",
    "max_history_length": 1000,
    "history_compression": true,
    "clear_history_on_exit": false
  }
}
```

3.4.9 Advanced Configuration

Network Settings

```
{
  "advanced": {
    "request_timeout": 30,
    "connect_timeout": 10,
    "retry_attempts": 3,
    "retry_delay": 1.0,
    "retry_exponential_backoff": true,
    "max_concurrent_requests": 1,
    "user_agent": "Perspt/0.4.0",
    "proxy": {
      "http": "http://proxy:8080",
      "https": "https://proxy:8080"
    }
  }
}
```

Security Settings

```
{
  "security": {
    "verify_ssl": true,
    "api_key_masking": true,
    "log_requests": false,
    "log_responses": false,
    "encrypt_history": false
  }
}
```

Performance Tuning

```
{
  "performance": {
    "buffer_size": 8192,
    "chunk_size": 1024,
    "memory_limit": "100MB",
    "cache_responses": false,
```

(continues on next page)

(continued from previous page)

```
    "preload_models": false
  }
}
```

3.4.10 Multiple Configurations

Work vs Personal

Create separate configurations for different contexts:

work-config.json:

```
{
  "api_key": "sk-work-key-here",
  "provider_type": "openai",
  "default_model": "gpt-4",
  "ui": {
    "theme": "professional",
    "show_timestamps": true
  },
  "behavior": {
    "auto_save_history": true,
    "history_file": "~/.perspt/work_history.json"
  }
}
```

personal-config.json:

```
{
  "api_key": "sk-personal-key-here",
  "provider_type": "anthropic",
  "default_model": "claude-3-sonnet-20240229",
  "ui": {
    "theme": "vibrant",
    "show_timestamps": false
  },
  "behavior": {
    "auto_save_history": false
  }
}
```

Usage:

```
# Work configuration
perspt --config work-config.json

# Personal configuration
perspt --config personal-config.json

# Create aliases for convenience
alias work-ai="perspt --config ~/.perspt/work-config.json"
alias personal-ai="perspt --config ~/.perspt/personal-config.json"
```

3.4.11 Configuration Validation

Perspt validates your configuration and provides helpful error messages:

```
# Validate configuration without starting
perspt --config config.json --validate

# Check configuration and list available models
perspt --config config.json --list-models
```

Common validation errors:

- **Invalid API key format**: Ensure your API key follows the correct format
- **Missing required fields**: Some providers require specific configuration
- **Invalid model names**: Use `--list-models` to see available options
- **Network connectivity**: Check internet connection and proxy settings

3.4.12 Configuration Templates

Generate template configurations for different use cases:

```
# Generate basic template
perspt --generate-config basic > config.json

# Generate advanced template
perspt --generate-config advanced > advanced-config.json

# Generate provider-specific template
perspt --generate-config openai > openai-config.json
```

3.4.13 Migration and Import

From Other Tools

Import configurations from similar tools:

```
# Import from environment variables
perspt --import-env > config.json

# Import from ChatGPT CLI config
perspt --import chatgpt-cli ~/.chatgpt-cli/config.yaml

# Import from OpenAI CLI config
perspt --import openai-cli ~/.openai/config.json
```

Backup and Restore

```
# Backup current configuration
cp ~/.config/perspt/config.json ~/.config/perspt/config.backup.json

# Restore from backup
cp ~/.config/perspt/config.backup.json ~/.config/perspt/config.json

# Export configuration with history
perspt --export-config --include-history > full-backup.json
```

3.4.14 Best Practices

Security

1. **Never commit API keys** to version control
2. **Use environment variables** for sensitive data
3. **Rotate API keys** regularly
4. **Use separate keys** for different projects
5. **Enable API key masking** in logs

Organization

1. **Use descriptive config names** (`work-config.json`, `research-config.json`)
2. **Create aliases** for frequently used configurations
3. **Document your configurations** with comments (where supported)
4. **Use version control** for non-sensitive configuration parts
5. **Regular backups** of important configurations

Performance

1. **Set appropriate timeouts** based on your network
2. **Configure retry settings** for reliability
3. **Use streaming** for better user experience
4. **Limit history length** to prevent memory issues
5. **Enable compression** for large chat histories

3.4.15 Troubleshooting

Common Issues

Configuration not found:

```
# Check current working directory
ls -la config.json

# Check user config directory
ls -la ~/.config/perspt/

# Use absolute path
perspt --config /full/path/to/config.json
```

Invalid JSON format:

```
# Validate JSON syntax
cat config.json | python -m json.tool

# Or use jq
jq . config.json
```

API key not working:

```
# Test API key directly
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     "https://api.openai.com/v1/models"
```

(continues on next page)

(continued from previous page)

```
# Check environment variable
echo $OPENAI_API_KEY
```

Provider connection issues:

```
# Test network connectivity
ping api.openai.com

# Check proxy settings
echo $HTTP_PROXY $HTTPS_PROXY

# Test with verbose output
perspt --config config.json --verbose
```

Getting Help

If you need assistance with configuration:

1. **Check the examples** in this guide
2. **Use the validation commands** to check your config
3. **Review the error messages** - they often contain helpful hints
4. **Ask the community** on GitHub Discussions
5. **File an issue** if you find a bug in configuration handling

See also

- *Getting Started* - Basic setup and first run
- *AI Providers* - Provider-specific guides
- *Troubleshooting* - Common issues and solutions
- *Advanced Features* - Advanced usage patterns

3.5 User Guide

This comprehensive user guide covers everything you need to know to use Perspt effectively, from basic conversations to advanced productivity techniques.

3.5.1 Basic Usage

This guide covers the fundamental usage patterns of Perspt, from starting your first conversation to understanding the CLI commands and streaming features powered by the modern genai crate.

Starting Perspt

Perspt uses the latest genai crate (v0.3.5) for unified LLM access with enhanced capabilities. You can start it with various configuration options:

Basic Usage

```
# Start with default configuration (OpenAI gpt-4o-mini)
perspt
```

Provider Selection

```
# Use Anthropic with Claude 3.5 Sonnet
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022

# Use Google Gemini
perspt --provider-type gemini --model gemini-1.5-flash

# Use latest reasoning models
perspt --provider-type openai --model o1-mini
```

Configuration Files

```
# Use custom configuration file
perspt --config /path/to/your/config.json

# Override API key from command line
perspt --api-key your-api-key-here
```

Model Discovery

```
# List all available models for current provider
perspt --list-models

# List models for specific provider
perspt --provider-type anthropic --list-models
```

Your First Conversation

When Perspt starts, you'll see a clean interface with model validation and streaming capabilities:

```
Perspt v0.4.0 - Performance LLM Chat CLI
Provider: OpenAI | Model: gpt-4o-mini | Status: Connected ✓
Enhanced streaming with genai crate v0.3.5

Type your message and press Enter to start a conversation.
Use Ctrl+C to exit gracefully.

>
```

Simply type your message or question and press Enter. Perspt will validate the model connection before starting:

```
> Hello, can you explain quantum computing?
```

Enhanced Streaming Experience

With the genai crate integration, responses stream in real-time with proper event handling:

- **Reasoning Models**: See thinking process with reasoning chunks for o1-series models
- **Regular Models**: Smooth token-by-token streaming for immediate feedback
- **Error Recovery**: Robust error handling with terminal restoration

The AI maintains context throughout the session and provides rich, formatted responses with markdown support.

CLI Arguments and Options

Perspt supports comprehensive command-line arguments that actually work with the genai crate integration:

Core Arguments

```
# Configuration
perspt --config|-c FILE          # Custom configuration file path

# Authentication
perspt --api-key|-k KEY          # Override API key

# Model Selection
perspt --model|-m MODEL          # Specific model name
perspt --provider-type|-p TYPE   # Provider type
perspt --provider PROFILE        # Provider profile from config

# Discovery
perspt --list-models|-l          # List available models

# Interface Modes
perspt --simple-cli              # Enable simple CLI mode
perspt --log-file FILE           # Session logging (requires --simple-cli)
```

Supported Provider Types

```
openai          # OpenAI GPT models (default)
anthropic       # Anthropic Claude models
google          # Google Gemini models
groq            # Groq ultra-fast inference
cohere          # Cohere Command models
xai             # XAI Grok models
deepseek        # DeepSeek models
ollama          # Local Ollama models
```

Example Usage Patterns

```
# TUI mode with specific provider
perspt -p openai -m gpt-4o-mini

# Simple CLI mode with logging
perspt --simple-cli --log-file session.txt

# Creative writing with Claude in simple CLI
perspt --simple-cli -p anthropic -m claude-3-5-sonnet-20241022

# Fast local inference in simple CLI
perspt --simple-cli -p ollama -m llama3.2

# Validate model before starting
perspt -p google -m gemini-2.0-flash-exp --list-models
```

Interactive Commands

Once in the chat interface, you can use keyboard shortcuts and built-in commands for efficient interaction:

Built-in Chat Commands

Command	Description
`/save`	Save conversation with timestamped filename (e.g., conversation_1735123456.txt)
`/save filename.txt`	Save conversation with custom filename

Conversation Export Examples

```
# Save with automatic timestamped filename
> /save
□ Conversation saved to: conversation_1735123456.txt

# Save with custom filename for organization
> /save python_debugging_session.txt
□ Conversation saved to: python_debugging_session.txt

# Attempt to save empty conversation
> /save
□ No conversation to save
```

Export File Format

The saved conversations are exported as plain text files with the following structure:

```
Perspt Conversation
==================
[2024-01-01 12:00:00] User: Hello, can you explain quantum computing?
[2024-01-01 12:00:01] Assistant: Quantum computing is a revolutionary approach...

[2024-01-01 12:02:15] User: What are the main applications?
[2024-01-01 12:02:16] Assistant: The main applications of quantum computing include...
```

Navigation Shortcuts

Shortcut	Action
Enter	Send message (validated before transmission)
Ctrl+C	Exit gracefully with terminal restoration
↑/↓ Keys	Scroll through chat history
Page Up/Down	Fast scroll through long conversations
Ctrl+L	Clear screen (preserves context)

Input Management

- **Multi-line Input**: Natural line breaks supported
- **Input Queuing**: Type new messages while AI responds
- **Context Preservation**: Full conversation history maintained
- **Markdown Rendering**: Rich text formatting in responses

Managing Conversations

Enhanced Context Management

With the genai crate integration, Perspt provides superior context handling:

Context Awareness - Full conversation history maintained per session - Automatic context window management for each provider - Smart truncation when approaching token limits - Provider-specific optimizations

Streaming and Responsiveness - Real-time token streaming for immediate feedback - Reasoning chunk display for o1-series models - Background processing while you type new queries - Robust error recovery with terminal restoration

Example of enhanced conversation flow:

```
> I'm working on a Rust project with async/await
[Streaming...] I'd be happy to help with your Rust async project!
Rust's async/await provides excellent performance for concurrent operations...

> How do I handle multiple futures concurrently?
[Streaming...] For handling multiple futures concurrently in your Rust project,
you have several powerful options with tokio...

> Show me an example with tokio::join!
[Reasoning...] Let me provide a practical example using tokio::join!
for your async Rust project...
```

Advanced Conversation Features

- **Input Queuing**: Continue typing while AI generates responses
- **Context Preservation**: Seamless topic transitions within sessions
- **Error Recovery**: Automatic reconnection and state restoration
- **Model Validation**: Pre-flight checks ensure model availability

Message Formatting and Rendering

Enhanced Markdown Support

Perspt includes a custom markdown parser optimized for terminal rendering:

Supported Formatting

````
**Bold text** and *italic text*
`inline code` and ```code blocks```

# Headers and ## Subheaders

- Bullet points
- With proper indentation

1. Numbered lists
2. With automatic formatting
````

Code Block Rendering

Share code with syntax highlighting hints:

```
> Can you help optimize this Rust function?
```

(continues on next page)

(continued from previous page)

~~~
```rust
async fn process_data(data: Vec<String>) -> Result<Vec<String>, Error> {
    // Your code here
}
```
~~~

Long Message Handling

- Automatic text wrapping for terminal width
- Proper paragraph breaks and spacing
- Smooth scrolling through long responses
- Visual indicators for streaming progress

Testing Local Models with Ollama

Ollama provides an excellent way to test local models without API keys or internet connectivity. This section walks through setting up and testing Ollama with Perspt.

Prerequisites

Install and Start Ollama

```
# macOS
brew install ollama

# Linux
curl -fsSL https://ollama.ai/install.sh | sh

# Start Ollama service
ollama serve
```

Download Test Models

```
# Download Llama 3.2 (3B) - fast and efficient
ollama pull llama3.2

# Download Code Llama - for coding tasks
ollama pull codellama

# Verify models are available
ollama list
```

Basic Ollama Testing

Start with Simple Conversations

```
# Test basic functionality
perspt --provider-type ollama --model llama3.2
```

Example conversation flow:

```
Perspt v0.4.0 - Performance LLM Chat CLI
Provider: Ollama | Model: llama3.2 | Status: Connected ✓
```

(continues on next page)

(continued from previous page)

```
Local model hosting - no API key required

> Hello! Can you help me understand how LLMs work?

[Assistant responds with explanation of language models...]

> That's helpful! Now explain it like I'm 5 years old.

[Assistant provides simplified explanation...]
```

Test Different Model Types

```
# General conversation
perspt --provider-type ollama --model llama3.2

# Coding assistance
perspt --provider-type ollama --model codellama

# Larger model for complex tasks (if you have enough RAM)
perspt --provider-type ollama --model llama3.1:8b
```

Performance Testing

Model Comparison

Test different model sizes to find the right balance for your system:

Model	Size	RAM Required	Best For
`llama3.2`	3B	~4GB	Quick responses, chat
`llama3.1:8b`	8B	~8GB	Better reasoning, longer context
`codellama`	7B	~7GB	Code generation, technical tasks
`mistral`	7B	~7GB	Balanced performance

Speed Testing

```
# Time how long responses take
time perspt --provider-type ollama --model llama3.2

# Compare with cloud providers
time perspt --provider-type openai --model gpt-4o-mini
```

Practical Test Scenarios

```
# Test 1: Basic Knowledge
> What is the capital of France?

# Test 2: Reasoning
> If a train travels 60 mph for 2.5 hours, how far does it go?

# Test 3: Creative Writing
> Write a short story about a robot learning to paint.
```

(continues on next page)

(continued from previous page)

```
# Test 4: Code Generation (with codellama)
> Write a Python function to calculate fibonacci numbers.
```

Troubleshooting Ollama

Common Issues

```
# Check if Ollama is running
curl http://localhost:11434/api/tags

# If connection fails
ollama serve

# List available models
perspt --provider-type ollama --list-models

# Pull missing models
ollama pull llama3.2
```

Performance Issues

- **Slow responses**: Try smaller models (llama3.2 vs llama3.1:8b)
- **Out of memory**: Close other applications or use lighter models
- **Model not found**: Ensure you've pulled the model with `ollama pull`

Configuration for Regular Use

Create a config file for easy Ollama usage:

```
{
  "provider_type": "ollama",
  "default_model": "llama3.2",
  "providers": {
    "ollama": "http://localhost:11434/v1"
  },
  "api_key": "not-required"
}
```

```
# Save as ollama_config.json and use
perspt --config ollama_config.json
```

Benefits of Local Testing

- **Privacy**: All data stays on your machine
- **Cost**: No API fees or usage limits
- **Offline**: Works without internet after initial setup
- **Experimentation**: Try different models and settings freely
- **Learning**: Understand model capabilities and limitations

Best Practices for Effective Usage

Communication Strategies

Optimized for GenAI Crate Integration

1. **Model-Specific Approaches**: - **Reasoning Models (o1-series)**: Provide complex problems and let them work through the logic - **Fast Models (gpt-4o-mini, claude-3-haiku)**: Use for quick questions and iterations - **Large Context Models (claude-3-5-sonnet)**: Share entire codebases or documents
2. **Provider Strengths**: - **OpenAI**: Latest reasoning capabilities, coding assistance - **Anthropic**: Safety-focused, analytical reasoning, constitutional AI - **Google**: Multimodal capabilities, large context windows - **Groq**: Ultra-fast inference for real-time conversations

Effective Prompting Techniques

```
# Instead of vague requests:
> Help me with my code

# Be specific with context:
> I'm working on a Rust HTTP server using tokio and warp. The server
compiles but panics when handling concurrent requests. Here's the
relevant code: [paste code]. Can you help me identify the race condition?
```

Session Management Strategies

- **Single-Topic Sessions**: Keep related discussions in one session for better context
- **Model Switching**: Use *perspt –list-models* to explore optimal models for different tasks
- **Configuration Profiles**: Set up different configs for work, creative, and development tasks

Troubleshooting Common Issues

Connection and Model Issues

Model Validation Failures

```
# Check if model exists for provider
perspt --provider-type openai --list-models | grep o1-mini

# Test connection with basic model
perspt --provider-type openai --model gpt-3.5-turbo
```

API Key Problems

```
# Test API key directly
perspt --api-key your-key --provider-type openai --list-models

# Use environment variables (recommended)
export OPENAI_API_KEY="your-key"
perspt
```

Streaming Issues

If streaming responses seem slow or interrupted:

1. **Network Check**: Ensure stable internet connection
2. **Provider Status**: Check provider service status pages
3. **Model Selection**: Try faster models like gpt-4o-mini
4. **Terminal Compatibility**: Ensure terminal supports ANSI colors and UTF-8

Performance Optimization

Memory and Speed

- **Local Models**: Use Ollama for privacy and reduced latency
- **Model Selection**: Choose appropriate model size for your task
- **Context Management**: Clear context for unrelated new topics

Cost Optimization

- **Model Tiers**: Use cheaper models (gpt-3.5-turbo) for simple queries
- **Streaming Benefits**: Stop generation early if you have enough information
- **Batch Questions**: Ask related questions in single sessions to share context

Next Steps

Once you're comfortable with basic usage:

- **Advanced Features**: Learn about configuration profiles and system prompts in *Advanced Features*
- **Provider Deep-Dive**: Explore specific provider capabilities in *AI Providers*
- **Troubleshooting**: Get help with specific issues in *Troubleshooting*
- **Configuration**: Set up custom configurations in *Configuration Guide*

Simple CLI Mode - Direct Q&A Interface

NEW in v0.4.5: Perspt now includes a minimal command-line interface mode for direct question-and-answer interaction without the TUI overlay. This mode follows the Unix philosophy of simple, composable tools and is perfect for scripting, accessibility needs, or users who prefer command-line interfaces.

When to Use Simple CLI Mode

The simple CLI mode is ideal for:

- 🤖 **Scripting & Automation**: Integrate Perspt into shell scripts, CI/CD pipelines, or automated workflows
- □ **Accessibility**: Simple, scrolling console output for users with screen readers or accessibility needs
- □ **Logging & Documentation**: Built-in session logging for keeping detailed records of AI interactions
- ⚡ **Quick Queries**: Lightweight interface for fast, one-off questions without UI overhead
- □ **Unix Philosophy**: Clean, composable tool that works well with pipes, redirects, and other command-line tools

Basic Simple CLI Usage

Starting Simple CLI Mode

```
# Basic simple CLI mode (uses auto-detected provider)
perspt --simple-cli

# With specific provider and model
perspt --simple-cli --provider-type openai --model gpt-4o-mini

# With Gemini
perspt --simple-cli --provider-type gemini --model gemini-1.5-flash

# With local Ollama (no API key needed)
perspt --simple-cli --provider-type ollama --model llama3.2
```

Interactive Session Example

```
$ perspt --simple-cli --provider-type openai --model gpt-4o-mini
Persp Simple CLI Mode
Model: gpt-4o-mini
Type 'exit' or press Ctrl+D to quit.

> What is the capital of France?
Paris is the capital and largest city of France. It's located in the
north-central part of the country on the Seine River...

> How many people live there?
The city of Paris proper has a population of approximately 2.1 million
people as of recent estimates. However, the Greater Paris metropolitan
area (Île-de-France region) has a much larger population...

> exit
Goodbye!
```

Session Logging

One of the key features of simple CLI mode is built-in session logging:

Basic Logging

```
# Log entire session to a file
perspt --simple-cli --log-file my-session.txt

# Use timestamped filenames for organization
perspt --simple-cli --log-file "$(date +%Y%m%d_%H%M%S)_ai_session.txt"

# Combined with specific provider
perspt --simple-cli --provider-type anthropic --model claude-3-5-sonnet-20241022 --log-
↪file claude-session.txt
```

Log File Format

The log files contain both user input and AI responses in a clean, readable format:

```
> What is machine learning?
Machine learning is a subset of artificial intelligence (AI) that involves
training algorithms to recognize patterns in data and make predictions or
decisions without being explicitly programmed for each specific task...

> Give me 3 practical examples
Here are three practical examples of machine learning in everyday use:

1. **Email Spam Detection**: Email services like Gmail use machine learning...
2. **Recommendation Systems**: Platforms like Netflix, Spotify, and Amazon...
3. **Voice Assistants**: Siri, Alexa, and Google Assistant use machine learning...

>
```

Scripting and Automation

The simple CLI mode excels at scripting and automation scenarios:

Direct Input via Pipes

```
# Pipe a single question
echo "What is quantum computing?" | perspt --simple-cli

# Use in shell scripts
#!/bin/bash
question="Explain the difference between REST and GraphQL APIs"
echo "$question" | perspt --simple-cli --log-file api-explanation.txt
```

Multiple Questions

```
# Chain multiple questions with automatic exit
{
  echo "What is Docker?"
  echo "How is it different from virtual machines?"
  echo "Give me a simple Docker example"
  echo "exit"
} | perspt --simple-cli --log-file docker-tutorial.txt
```

Environment Integration

```
# Set up environment for regular use
export OPENAI_API_KEY="your-key"
alias ai="perspt --simple-cli"
alias ai-log="perspt --simple-cli --log-file"

# Now use anywhere
ai
ai-log research-session.txt

# Add to your .bashrc or .zshrc for permanent setup
echo 'alias ai="perspt --simple-cli"' >> ~/.bashrc
```

Advanced Simple CLI Features

Error Handling

Unlike the TUI mode, simple CLI mode is designed to be resilient for scripting:

```
> This might cause an error
Error: Rate limit exceeded. Please try again in a few moments.

> This question works fine
[Normal response continues...]

> exit
Goodbye!
```

Individual request errors don't terminate the session, making it suitable for long-running scripts.

Exit Methods

Simple CLI mode supports multiple exit methods for different use cases:

```
# Method 1: Type 'exit' command
> exit

# Method 2: Send EOF (Ctrl+D on Unix, Ctrl+Z on Windows)
> ^D

# Method 3: Interrupt signal (Ctrl+C)
> ^C
```

Configuration Files

Create dedicated configuration files for simple CLI use:

```
{
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "api_key": "your-api-key"
}
```

```
# Save as simple-cli-config.json and use
persp --simple-cli --config simple-cli-config.json
```

Simple CLI vs TUI Mode Comparison

Feature	Simple CLI Mode	TUI Mode
Interface Style	Minimal Unix prompt	Rich terminal UI
Scrolling	Natural terminal scrolling	Built-in history navigation
Markdown Rendering	Raw text output	Formatted rendering
Session Management	Built-in logging option	Manual `/save` command
Scripting Support	Excellent (pipes, redirects)	Not suitable
Accessibility	High (screen reader friendly)	Moderate
Resource Usage	Minimal overhead	Moderate (UI rendering)
Background Operation	Foreground only	Visual feedback
Multi-line Input	Line-by-line	Rich text editing

Use Case Examples

Documentation Generation

```
# Generate documentation for a project
{
  echo "Explain the architecture of a REST API"
  echo "What are the best practices for REST API design?"
  echo "How do you handle authentication in REST APIs?"
  echo "exit"
} | persp --simple-cli --log-file rest-api-docs.txt
```

Code Review Assistant

```
# Review code with AI assistance
{
  echo "Review this Python function for potential issues:"
  cat my_function.py
  echo "exit"
} | perspt --simple-cli --provider-type openai --model gpt-4o --log-file code-review.txt
```

Learning and Research

```
# Research session with logging
perspt --simple-cli --provider-type anthropic --model claude-3-5-sonnet-20241022 \
      --log-file "$(date +%Y%m%d)_learning_session.txt"
```

Quick Consultations

```
# Quick question without UI overhead
echo "What's the best way to optimize PostgreSQL queries?" | perspt --simple-cli
```

Troubleshooting Simple CLI Mode

Common Issues

```
# Test if simple CLI mode works
perspt --simple-cli --provider-type openai --list-models

# Verify logging permissions
touch test-log.txt && rm test-log.txt

# Check if provider is properly configured
perspt --simple-cli --provider-type your-provider --model your-model
```

Performance Tips

- Use faster models like `gpt-4o-mini` or `gemini-1.5-flash` for quick queries
- For local usage, `ollama` with `llama3.2` provides excellent performance
- Log files are appended to, so you can continue sessions across multiple runs

Integration with Other Tools

```
# Use with jq for structured output (if AI returns JSON)
echo "Return the top 3 programming languages as JSON" | \
perspt --simple-cli | jq '.languages[]'

# Use with grep for filtering
echo "List 10 Linux commands" | perspt --simple-cli | grep -E "^[0-9]+"

# Combine with watch for monitoring
watch -n 300 'echo "What is the current status of the Python package index?" | perspt --
↪simple-cli'
```

3.5.2 Advanced Features

This guide covers Perspt's advanced features powered by the modern genai crate (v0.3.5), enabling sophisticated AI interactions, enhanced streaming capabilities, and productivity workflows.

Configuration Profiles and Multi-Provider Setup

GenAI-Powered Provider Management

With the genai crate integration, Perspt supports seamless switching between providers and models:

```
# Work profile with reasoning models
perspt --config ~/.config/perspt/work.json

# Creative profile with latest models
perspt --config ~/.config/perspt/creative.json

# Development profile with coding-focused models
perspt --config ~/.config/perspt/dev.json

# Research profile with large context models
perspt --config ~/.config/perspt/research.json
```

Example profile configurations:

Work Profile (`work.json`):

```
{
  "provider_type": "anthropic",
  "default_model": "claude-3-5-sonnet-20241022",
  "api_key": "${ANTHROPIC_API_KEY}",
  "providers": {
    "anthropic": "https://api.anthropic.com",
    "openai": "https://api.openai.com/v1"
  }
}
```

Creative Profile (`creative.json`):

```
{
  "provider_type": "openai",
  "default_model": "gpt-4.1",
  "api_key": "${OPENAI_API_KEY}",
  "providers": {
    "openai": "https://api.openai.com/v1",
    "xai": "https://api.x.ai/v1"
  }
}
```

Development Profile (`dev.json`):

```
{
  "provider_type": "openai",
  "default_model": "o1-mini",
  "api_key": "${OPENAI_API_KEY}",
```

(continues on next page)

(continued from previous page)

```
  "providers": {
    "openai": "https://api.openai.com/v1",
    "groq": "https://api.groq.com/openai/v1"
  }
}
```

Research Profile (`research.json`):

```
{
  "provider_type": "google",
  "default_model": "gemini-1.5-pro",
  "api_key": "${GOOGLE_API_KEY}",
  "providers": {
    "google": "https://generativelanguage.googleapis.com",
    "anthropic": "https://api.anthropic.com"
  }
}
```

Enhanced Streaming and Real-time Features

GenAI Crate Streaming Capabilities

The genai crate provides sophisticated streaming with multiple event types:

Standard Streaming - Token-by-token streaming for immediate feedback - Smooth rendering with buffer management - Context-aware response building

Reasoning Model Streaming - `ChatStreamEvent::Start`: Beginning of response - `ChatStreamEvent::Chunk`: Regular content tokens - `ChatStreamEvent::ReasoningChunk`: Thinking process (o1-series) - `ChatStreamEvent::End`: Response completion

Advanced Streaming Features

```
# Example with reasoning model (o1-mini)
> Solve this complex math problem: ...

[Reasoning] Let me think through this step by step...
[Reasoning] First, I'll identify the key variables...
[Reasoning] Now I'll apply the quadratic formula...
[Streaming] Based on my analysis, the solution is...
```

Real-time Model Switching

Switch between models during conversations while maintaining context:

```
# Start with fast model for exploration
perspt --provider-type groq --model llama-3.1-8b-instant

# Switch to reasoning model for complex analysis
# (Context maintained across switch)
perspt --provider-type openai --model o1-mini
```

Model Validation and Discovery

Pre-flight model validation ensures reliable connections:

```
# Validate model before starting conversation
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022 --list-models

# Discover available models for provider
perspt --provider-type google --list-models | grep gemini-2
```

Automatic Fallback Configuration

Configure automatic fallbacks for reliability:

```
{
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "fallback_providers": [
    {
      "provider_type": "anthropic",
      "model": "claude-3-5-haiku-20241022"
    },
    {
      "provider_type": "groq",
      "model": "llama-3.1-70b-versatile"
    }
  ]
}
```

Advanced Conversation Patterns

Multi-Model Collaborative Workflows

Leverage different models for their strengths within single sessions:

Research and Analysis Workflow

```
# 1. Start with fast model for initial exploration
perspt --provider-type groq --model llama-3.1-8b-instant

# 2. Switch to reasoning model for deep analysis
perspt --provider-type openai --model o1-mini

# 3. Use large context model for comprehensive review
perspt --provider-type google --model gemini-1.5-pro
```

Code Development Workflow

```
# Use reasoning model for architecture planning
> Design a microservices architecture for an e-commerce platform
[Using o1-mini for complex reasoning]

# Switch to coding-focused model for implementation
> Now implement the user authentication service
[Using claude-3-5-sonnet for code generation]
```

(continues on next page)

(continued from previous page)

```

# Use fast model for quick iterations and testing
> Review this code for potential bugs
[Using llama-3.1-70b for rapid feedback]
```

Provider-Specific Optimizations

OpenAI Reasoning Models - Best for: Complex problem-solving, mathematical reasoning, logic puzzles - Features: Step-by-step thinking process, enhanced accuracy - Usage: Allow extra time for reasoning, provide complex multi-step problems

Anthropic Constitutional AI - Best for: Safety-critical applications, ethical reasoning, content moderation - Features: Built-in safety guardrails, nuanced understanding - Usage: Ideal for sensitive topics, business communications

Google Multimodal Capabilities - Best for: Document analysis, image understanding, large context processing - Features: 2M token context, multimodal input support - Usage: Large document analysis, comprehensive research

Groq Ultra-Fast Inference - Best for: Real-time chat, rapid prototyping, interactive sessions - Features: Sub-second response times, consistent performance - Usage: Brainstorming sessions, quick iterations

Local Model Privacy Features

Enhanced privacy with local Ollama integration:

Private Development Environment

```json
{
  "provider_type": "ollama",
  "default_model": "llama3.2:8b",
  "privacy_mode": true,
  "data_retention": "none",
  "providers": {
    "ollama": "http://localhost:11434"
  }
}
```

Sensitive Data Processing

```
# Use local models for proprietary code review
perspt --provider-type ollama --model qwen2.5:14b

# Process confidential documents offline
perspt --provider-type ollama --model llama3.2:8b
```

Terminal UI Enhancements

Advanced Markdown Rendering - Syntax highlighting for code blocks - Proper table formatting and alignment - Mathematical equation rendering - Nested list and quote support

Streaming Visual Indicators - Real-time token streaming animations - Reasoning process visualization for o1-models - Connection status and model information - Error recovery visual feedback

Keyboard Shortcuts and Navigation - Input queuing while AI responds - Seamless scrolling through long conversations - Context-aware copy/paste operations - Quick model switching hotkeys

Domain Expert Prompts

Software Development:

```
{
  "system_prompt": "You are a senior software engineer with expertise in multiple␣
↪programming languages, system design, and best practices. Provide detailed, practical␣
↪advice with code examples when helpful. Focus on maintainability, performance, and␣
↪security."
}
```

Academic Research:

```
{
  "system_prompt": "You are an academic research assistant with expertise in methodology,␣
↪citation practices, and critical analysis. Provide well-researched, evidence-based␣
↪responses with appropriate academic tone and references when possible."
}
```

Creative Writing:

```
{
  "system_prompt": "You are a creative writing mentor with expertise in storytelling,␣
↪character development, and various literary forms. Help develop ideas, provide␣
↪constructive feedback, and suggest techniques to improve writing craft."
}
```

Productivity and Session Management

Conversation Export and Archival

Perspt includes built-in conversation export functionality for productivity workflows:

```
# Save conversation with timestamped filename
> /save
▯ Conversation saved to: conversation_1735123456.txt

# Save with custom filename for organization
> /save python_debugging_session.txt
▯ Conversation saved to: python_debugging_session.txt
```

Export Features: - Raw text format without terminal styling - Chronological message order with timestamps - User and assistant messages (system messages excluded) - Automatic filename generation with Unix timestamps - Custom filename support for organized archives

Typical Export Format:

```
Perspt Conversation
==================
[2024-01-01 12:00:00] User: How do I optimize this Python function?
[2024-01-01 12:00:01] Assistant: I can help you optimize that function...

[2024-01-01 12:02:15] User: What about memory usage?
[2024-01-01 12:02:16] Assistant: For memory optimization, consider...
```

Workflow Integration:

```
# Research session workflow
perspt --provider-type openai --model o1-mini
# Conduct research conversation
# /save research_quantum_computing_2024.txt

# Code review session
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022
# Review and discuss code
# /save code_review_auth_service.txt

# Learning session
perspt --provider-type google --model gemini-1.5-pro
# Educational conversation
# /save learning_session_rust_async.txt
```

Session Management Best Practices: - Use descriptive filenames with dates and topics - Export before switching contexts or models - Organize exported conversations in project folders - Archive important problem-solving sessions - Keep transcripts of architectural discussions

Context-Aware Prompts

Dynamic system prompts based on context:

```
{
  "system_prompt": "You are assisting with a ${PROJECT_TYPE} project. The user is working␣
↪in ${LANGUAGE} and prefers ${STYLE} coding style. Adapt your responses accordingly and␣
↪provide relevant examples."
}
```

Session Persistence

Save and Resume Conversations

Perspt can maintain conversation history across sessions:

```
{
  "conversation_history": {
    "enabled": true,
    "max_sessions": 10,
    "auto_save": true,
    "storage_path": "~/.config/perspt/history/"
  }
}
```

Session Commands

Manage conversation sessions:

```
> /save session_name        # Save current conversation
> /load session_name        # Load saved conversation
> /list sessions           # List all saved sessions
> /delete session_name     # Delete a saved session
```

Export Conversations

Export conversations in various formats:

```
> /export markdown conversation.md
> /export json conversation.json
> /export html conversation.html
```

Multi-Model Conversations

Model Comparison

Compare responses from different models:

```
> /compare "Explain quantum computing" gpt-4o claude-3-5-sonnet-20241022
```

This sends the same prompt to multiple models and displays responses side by side.

Model Switching

Switch models mid-conversation while maintaining context:

```
> We've been discussing Python optimization
AI: Yes, we covered several techniques including caching and algorithmic improvements.

> /model claude-3-5-sonnet-20241022
Model switched to claude-3-5-sonnet-20241022

> Can you continue with memory optimization techniques?
AI: Continuing our Python optimization discussion, let's explore memory optimization...
```

Plugin System

Perspt supports plugins for extended functionality:

Code Analysis Plugin

Analyze code quality and suggest improvements:

```
{
  "plugins": {
    "code_analysis": {
      "enabled": true,
      "languages": ["python", "javascript", "rust"],
      "features": ["linting", "security", "performance"]
    }
  }
}
```

Usage:

~~~
> /analyze-code
```python
def inefficient_function(data):
~~~

(continues on next page)

(continued from previous page)

```
    result = []
    for item in data:
        if item > 0:
            result.append(item * 2)
    return result
```
```

Document Processing Plugin

Process and analyze documents:

```
{
  "plugins": {
    "document_processor": {
      "enabled": true,
      "supported_formats": ["pdf", "docx", "txt", "md"],
      "max_file_size": "10MB"
    }
  }
}
```

Usage:

```
> /process-document /path/to/document.pdf
> Summarize this document and highlight key points
```

Web Integration Plugin

Fetch and analyze web content:

```
{
  "plugins": {
    "web_integration": {
      "enabled": true,
      "allowed_domains": ["github.com", "stackoverflow.com", "docs.python.org"],
      "max_content_length": 50000
    }
  }
}
```

Usage:

```
> /fetch-url https://docs.python.org/3/library/asyncio.html
> Explain the key concepts from this documentation
```

Advanced Conversation Patterns

Role-Playing Scenarios

Set up specific roles for focused assistance:
```

```
> /role code_reviewer
AI: I'm now acting as a code reviewer. Please share your code for detailed analysis.

> /role system_architect
AI: I'm now acting as a system architect. Let's discuss your system design requirements.
```

Collaborative Problem Solving

Break down complex problems into manageable parts:

```
> /problem-solving mode
AI: I'm now in problem-solving mode. Let's break down your challenge systematically.

> I need to design a scalable microservices architecture
AI: Great! Let's approach this systematically:
    1. First, let's identify your core business domains
    2. Then we'll determine service boundaries
    3. Next, we'll design the communication patterns
    4. Finally, we'll address scalability and deployment

    Let's start with step 1: What are your main business domains?
```

Iterative Refinement

Continuously improve solutions through iteration:

```
> /iterative mode
AI: I'm now in iterative mode. I'll help you refine solutions step by step.

> Here's my initial algorithm implementation
AI: I see several areas for improvement. Let's iterate:
    Version 1: Your current implementation
    Version 2: Optimized algorithm complexity
    Version 3: Added error handling
    Version 4: Improved readability and maintainability

    Which aspect would you like to focus on first?
```

Automation and Scripting

Command Scripting

Create scripts for common workflows:

development_workflow.perspt:

```
/model gpt-4
/role senior_developer
/context "Working on a ${PROJECT_NAME} project in ${LANGUAGE}"

Ready for development assistance!
```

Run with:

```
perspt --script development_workflow.perspt
```

Batch Processing

Process multiple queries in batch:

```
> /batch process_queries.txt
```

Where `process_queries.txt` contains:

```
Explain the benefits of microservices
---
Compare REST vs GraphQL APIs
---
Best practices for database design
```

Configuration Validation

Validate your configuration setup:

```
> /validate-config
```

This checks:

- API key validity
- Model availability
- Configuration syntax
- Plugin compatibility
- Network connectivity

Performance Optimization

Response Caching

Cache responses for repeated queries:

```
{
  "cache": {
    "enabled": true,
    "ttl": 3600,
    "max_size": "100MB",
    "strategy": "lru"
  }
}
```

Parallel Processing

Process multiple requests simultaneously:

```
{
  "parallel_processing": {
    "enabled": true,
    "max_concurrent": 3,
```

(continues on next page)

(continued from previous page)

```
    "timeout": 30
  }
}
```

Custom Integrations

IDE Integration

Integrate Perspt with your development environment:

VS Code Extension:

```
{
  "vscode": {
    "enabled": true,
    "keybindings": {
      "ask_perspt": "Ctrl+Shift+P",
      "explain_code": "Ctrl+Shift+E"
    }
  }
}
```

Vim Plugin:

```
" Add to .vimrc
nnoremap <leader>p :!perspt --query "<C-R><C-W>"<CR>
```

API Integration

Use Perspt programmatically:

```
import requests

def ask_perspt(question):
    response = requests.post('http://localhost:8080/api/chat', {
        'message': question,
        'model': 'gpt-4'
    })
    return response.json()['response']
```

Next Steps

Explore more advanced topics:

- *AI Providers* - Deep dive into AI provider capabilities
- *Troubleshooting* - Advanced troubleshooting techniques
- *Extending Perspt* - Create custom plugins and extensions
- *API Reference* - API reference for programmatic usage

3.5.3 AI Providers

This comprehensive guide covers all supported AI providers in Perspt powered by the modern genai crate (v0.3.5), their latest capabilities, configuration options, and best practices for optimal performance.

Overview

Persp leverages the unified genai crate to provide seamless access to multiple AI providers with consistent APIs and enhanced features:

OpenAI	Latest GPT models including reasoning models (o1-series), GPT-4.1, and optimized variants
Anthropic	Claude 3.5 family with constitutional AI and safety-focused design
Google AI	Gemini 2.5 Pro and multimodal capabilities with large context windows
Groq	Ultra-fast inference with Llama and Mixtral models
Cohere	Command R+ models optimized for business and RAG applications
XAI	Grok models with real-time web access and humor
Ollama	Local model hosting with privacy and offline capabilities
XAI	Grok models for advanced reasoning and conversation

OpenAI

OpenAI provides cutting-edge language models including the latest reasoning capabilities through the genai crate integration.

Supported Models

Model	Context Length	Best For	Notes
`gpt-4.1`	128K tokens	Enhanced reasoning, latest capabilities	Most advanced GPT-4 variant (2025)
`o1-preview`	128K tokens	Complex reasoning, problem solving	Advanced reasoning with step-by-step thinking
`o1-mini`	128K tokens	Fast reasoning, coding tasks	Efficient reasoning model
`o3-mini`	128K tokens	Latest reasoning capabilities	Newest reasoning model (2025)
`gpt-4o`	128K tokens	Multimodal, fast performance	Optimized for speed and quality
`gpt-4o-mini`	128K tokens	Fast, cost-effective (default)	Efficient version of GPT-4o
`gpt-4-turbo`	128K tokens	Complex reasoning, analysis	Previous generation flagship
`gpt-3.5-turbo`	16K tokens	Fast, cost-effective	Good for simple tasks

Configuration

Basic OpenAI configuration with genai crate:

```
{
  "provider_type": "openai",
```

(continues on next page)

(continued from previous page)

```
  "api_key": "sk-your-openai-api-key",
  "default_model": "gpt-4o-mini",
  "providers": {
    "openai": "https://api.openai.com/v1"
  }
}
```

CLI Usage

```
# Use latest reasoning model
perspt --provider-type openai --model o1-mini

# Use fastest model (default)
perspt --provider-type openai --model gpt-4o-mini

# List all available OpenAI models
perspt --provider-type openai --list-models
```

Reasoning Model Features

O1-series models provide enhanced reasoning with visual feedback:

```
> Solve this logic puzzle: There are 5 houses in a row...

[Reasoning...] Let me work through this step by step:
1. Setting up the constraints...
2. Analyzing the color clues...
3. Cross-referencing with pet information...
[Streaming...] Based on my analysis, here's the solution...
```

Environment Variables

```
export OPENAI_API_KEY="sk-your-key-here"
export OPENAI_ORG_ID="org-your-org-id"  # Optional
```

Anthropic (Claude)

Anthropic's Claude models excel at safety, reasoning, and nuanced understanding through constitutional AI principles.

Supported Models

Model	Context Length	Best For	Notes
`claude-3-5-sonnet-20241022`	200K tokens	Balanced performance, latest version	Recommended default
`claude-3-5-sonnet-20240620`	200K tokens	Previous Sonnet version	Stable and reliable
`claude-3-5-haiku-20241022`	200K tokens	Fast responses, cost-effective	Good for simple tasks
`claude-3-opus-20240229`	200K tokens	Most capable, complex reasoning	Highest quality responses

Configuration

```
{
  "provider_type": "anthropic",
  "api_key": "sk-ant-your-anthropic-key",
  "default_model": "claude-3-5-sonnet-20241022",
  "providers": {
    "anthropic": "https://api.anthropic.com"
  }
}
```

CLI Usage

```
# Use latest Claude model
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022

# Use fastest Claude model
perspt --provider-type anthropic --model claude-3-5-haiku-20241022

# List available Anthropic models
perspt --provider-type anthropic --list-models
```

Environment Variables

```
export ANTHROPIC_API_KEY="sk-ant-your-key-here"
```

Google AI (Gemini)

Google's Gemini models offer multimodal capabilities and large context windows with competitive performance.

Supported Models

Model	Context Length	Best For	Notes
`gemini-2.0-flash-exp`	1M tokens	Latest experimental model	Cutting-edge capabilities (2025)
`gemini-1.5-pro`	2M tokens	Large documents, complex analysis	Largest context window
`gemini-1.5-flash`	1M tokens	Fast responses, good balance	Recommended default
`gemini-pro`	32K tokens	General purpose tasks	Stable and reliable

Configuration

```
{
  "provider_type": "google",
  "api_key": "your-google-api-key",
  "default_model": "gemini-1.5-flash",
  "providers": {
    "google": "https://generativelanguage.googleapis.com"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

CLI Usage

```
# Use latest Gemini model
perspt --provider-type google --model gemini-2.0-flash-exp

# Use model with largest context
perspt --provider-type google --model gemini-1.5-pro

# List available Google models
perspt --provider-type google --list-models
```

Environment Variables

```
export GOOGLE_API_KEY="your-key-here"
# or
export GEMINI_API_KEY="your-key-here"
    "User-Agent": "Perspt/1.0"
  }
}
```

Best Practices

1. **Model Selection**: - Use `gpt-4-turbo` for complex reasoning tasks - Use `gpt-3.5-turbo` for simple queries to save costs - Use `gpt-4-vision-preview` when working with images
2. **Token Management**: - Monitor usage with longer conversations - Use appropriate `max_tokens` limits - Consider conversation history truncation
3. **Rate Limits**: - Implement retry logic for rate limit errors - Consider upgrading to higher tier plans for increased limits

Anthropic (Claude)

Anthropic's Claude models are known for their helpfulness, harmlessness, and honesty.

Supported Models

Model	Context Length	Best For	Notes
`claude-3-opus-20240229`	200K tokens	Complex reasoning, creative tasks	Most capable Claude model
`claude-3-sonnet-20240229`	200K tokens	Balanced performance/speed	Good general-purpose model
`claude-3-haiku-2024030`	200K tokens	Fast responses, simple tasks	Most cost-effective
`claude-2.1`	200K tokens	Legacy support	Deprecated, use Claude-3

Configuration

Basic Anthropic configuration:

```
{
  "provider": "anthropic",
  "api_key": "your-anthropic-api-key",
  "model": "claude-3-opus-20240229",
  "base_url": "https://api.anthropic.com",
  "version": "2023-06-01",
  "max_tokens": 4000,
  "temperature": 0.7,
  "top_p": 1.0,
  "top_k": 40,
  "stop_sequences": ["\\n\\nHuman:", "\\n\\nAssistant:"]
}
```

Advanced Configuration

System Messages:

```
{
  "provider": "anthropic",
  "model": "claude-3-opus-20240229",
  "system_message": "You are a helpful assistant specialized in software development.␣
↪Provide detailed, accurate responses with code examples when appropriate."
}
```

Content Filtering:

```
{
  "provider": "anthropic",
  "content_filtering": {
    "enabled": true,
    "strictness": "moderate"
  }
}
```

Best Practices

1. **Model Selection**: - Use `claude-3-opus` for complex analysis and creative work - Use `claude-3-sonnet` for balanced general-purpose tasks - Use `claude-3-haiku` for quick questions and simple tasks
2. **Prompt Engineering**: - Claude responds well to clear, structured prompts - Use explicit instructions and examples - Leverage Claude's strong reasoning capabilities
3. **Long Conversations**: - Take advantage of the large context window - Maintain conversation flow without frequent truncation

Google AI (Gemini)

Google's Gemini models offer strong reasoning and multimodal capabilities.

Supported Models

Model	Context Length	Best For	Notes
`gemini-2.5-pro`	2M tokens	Advanced reasoning, analysis	Latest and most capable
`gemini-2.0-flash`	1M tokens	Fast, efficient performance	Optimized for speed
`gemini-1.5-pro`	2M tokens	Complex reasoning, long context	High-capability model
`gemini-1.5-flash`	1M tokens	Fast responses, good quality	Balanced speed and capability
`gemini-pro`	32K tokens	General reasoning	Legacy model
`gemini-pro-vision`	16K tokens	Multimodal tasks	Supports images and text

Configuration

Basic Google AI configuration:

```json
{
  "provider": "google",
  "api_key": "your-google-api-key",
  "model": "gemini-pro",
  "base_url": "https://generativelanguage.googleapis.com/v1",
  "safety_settings": {
    "harassment": "BLOCK_MEDIUM_AND_ABOVE",
    "hate_speech": "BLOCK_MEDIUM_AND_ABOVE",
    "sexually_explicit": "BLOCK_MEDIUM_AND_ABOVE",
    "dangerous_content": "BLOCK_MEDIUM_AND_ABOVE"
  },
  "generation_config": {
    "temperature": 0.7,
    "top_p": 1.0,
    "top_k": 40,
    "max_output_tokens": 4000
  }
}
```

Multimodal Configuration

For image analysis with Gemini Vision:

```json
{
  "provider": "google",
  "model": "gemini-pro-vision",
  "multimodal": {
    "enabled": true,
    "supported_formats": ["png", "jpg", "jpeg", "webp", "gif"],
    "max_image_size": "20MB"
  }
}
```

Best Practices

1. **Safety Settings**: - Configure appropriate safety levels for your use case - Consider more permissive settings for creative tasks
2. **Multimodal Usage**: - Use Gemini Vision for image analysis and understanding - Combine text and images for richer interactions

Local Models

Perspt supports various local inference solutions for privacy and offline usage.

Ollama

Configuration for Ollama local models:

```
{
  "provider": "ollama",
  "base_url": "http://localhost:11434",
  "model": "llama2:7b",
  "stream": true,
  "options": {
    "temperature": 0.7,
    "top_p": 0.9,
    "top_k": 40,
    "repeat_penalty": 1.1,
    "seed": -1,
    "num_ctx": 4096
  }
}
```

Popular Ollama Models:

```
# Install popular models
ollama pull llama2:7b          # General purpose
ollama pull codellama:7b       # Code generation
ollama pull mistral:7b         # Fast and capable
ollama pull neural-chat:7b     # Conversational
```

LM Studio

Configuration for LM Studio:

```
{
  "provider": "lm_studio",
  "base_url": "http://localhost:1234/v1",
  "model": "local-model",
  "stream": true,
  "context_length": 4096,
  "gpu_layers": 35
}
```

OpenAI-Compatible Servers

For other OpenAI-compatible local servers:

```
{
  "provider": "openai_compatible",
  "base_url": "http://localhost:8000/v1",
  "api_key": "not-needed",
  "model": "local-model-name",
  "stream": true
}
```

Provider Comparison

Provider	Speed	Quality	Cost	Privacy	Context	Multi-modal
OpenAI	Fast	Excellent	Medium	Cloud	128K	Yes
Anthropic	Medium	Excellent	Medium	Cloud	200K	No
Google AI	Fast	Very Good	Low	Cloud	32K	Yes
Groq	Ultra-Fast	Excellent	Low	Cloud	32K	No
Local (Ollama)	Variable	Good	Free	Local	Variable	Limited

Multi-Provider Setup

Configure multiple providers for different use cases:

```
{
  "providers": {
    "primary": {
      "provider": "openai",
      "model": "gpt-4-turbo",
      "api_key": "your-openai-key"
    },
    "coding": {
      "provider": "anthropic",
      "model": "claude-3-opus-20240229",
      "api_key": "your-anthropic-key"
    },
    "local": {
      "provider": "ollama",
      "model": "codellama:7b",
      "base_url": "http://localhost:11434"
    }
  },
  "default_provider": "primary"
}
```

Switch between providers during conversation:

```
> /provider coding
Switched to coding provider (Claude-3 Opus)
```

(continues on next page)

(continued from previous page)

```
> /provider local
Switched to local provider (CodeLlama)
```

Fallback Configuration

Set up automatic fallbacks:

```
{
  "fallback_chain": [
    {
      "provider": "openai",
      "model": "gpt-4-turbo"
    },
    {
      "provider": "anthropic",
      "model": "claude-3-sonnet-20240229"
    },
    {
      "provider": "ollama",
      "model": "llama2:7b"
    }
  ],
  "fallback_conditions": [
    "rate_limit_exceeded",
    "api_error",
    "timeout"
  ]
}
```

Troubleshooting

Common Issues

API Key Issues:

```
> /validate-key
Checking API key validity...
✓ OpenAI key: Valid
✗ Anthropic key: Invalid or expired
```

Connection Problems:

```
# Test connectivity
curl -H "Authorization: Bearer your-api-key" \\
     https://api.openai.com/v1/models
```

Rate Limiting:

```
{
  "rate_limiting": {
    "requests_per_minute": 60,
```

(continues on next page)

(continued from previous page)

```
    "tokens_per_minute": 40000,
    "retry_strategy": "exponential_backoff",
    "max_retries": 3
  }
}
```

Performance Optimization

Request Optimization:

```
{
  "optimization": {
    "batch_requests": true,
    "compress_requests": true,
    "connection_pooling": true,
    "timeout": 30
  }
}
```

Caching:

```
{
  "cache": {
    "enabled": true,
    "provider_specific": true,
    "ttl": 3600,
    "max_size": "100MB"
  }
}
```

Next Steps

- *Troubleshooting* - Detailed troubleshooting for provider-specific issues
- *Advanced Features* - Advanced features that work with different providers
- *Configuration Guide* - Complete configuration reference
- *Extending Perspt* - Create custom provider integrations

Groq

Groq provides ultra-fast inference speeds with popular open-source models, optimized for real-time conversations.

Supported Models

Model	Context Length	Best For	Notes
`llama-3.1-405b-reasoning`	128K tokens	Complex reasoning, analysis	Largest Llama model
`llama-3.1-70b-versatile`	128K tokens	Balanced performance	Good general purpose model
`llama-3.1-8b-instant`	128K tokens	Ultra-fast responses	Best for speed
`mixtral-8x7b-32768`	32K tokens	Mixture of experts	Strong coding capabilities

Configuration

```
{
  "provider_type": "groq",
  "api_key": "your-groq-api-key",
  "default_model": "llama-3.1-70b-versatile",
  "providers": {
    "groq": "https://api.groq.com/openai/v1"
  }
}
```

CLI Usage

```
# Ultra-fast responses
persp t --provider-type groq --model llama-3.1-8b-instant

# Balanced performance
perspt --provider-type groq --model llama-3.1-70b-versatile
```

Environment Variables

```
export GROQ_API_KEY="your-key-here"
```

Cohere

Cohere specializes in enterprise-focused models with strong RAG (Retrieval-Augmented Generation) capabilities.

Supported Models

Model	Context Length	Best For	Notes
`command-r-plus`	128K tokens	RAG, business applications	Most capable Cohere model
`command-r`	128K tokens	General purpose, fast	Good balance of speed and quality
`command`	4K tokens	Simple tasks, cost-effective	Basic model

Configuration

```
{
  "provider_type": "cohere",
  "api_key": "your-cohere-api-key",
  "default_model": "command-r-plus",
  "providers": {
    "cohere": "https://api.cohere.ai"
  }
}
```

Environment Variables

```
export COHERE_API_KEY="your-key-here"
```

XAI (Grok)

XAI’s Grok models provide real-time web access and are known for their humor and current knowledge.

Supported Models

Model	Context Length	Best For	Notes
`grok-beta`	128K tokens	Current events, humor	Latest Grok model
`grok-vision-beta`	128K tokens	Multimodal analysis	Image understanding

Configuration

```
{
  "provider_type": "xai",
  "api_key": "your-xai-api-key",
  "default_model": "grok-beta",
  "providers": {
    "xai": "https://api.x.ai/v1"
  }
}
```

Environment Variables

```
export XAI_API_KEY="your-key-here"
```

Ollama (Local Models)

Ollama provides local model hosting for privacy, offline usage, and cost control with the genai crate integration. Perfect for testing, development, and privacy-conscious users.

Supported Models

Popular models available through Ollama:

Model	Size	RAM Required	Best Use Cases
`llama3.2`	3B	~4GB	General chat, quick responses, testing
`llama3.1:8b`	8B	~8GB	Better reasoning, longer conversations
`llama3.1:70b`	70B	~40GB	Complex reasoning, professional tasks
`codellama`	7B	~7GB	Code generation, debugging, technical docs
`mistral`	7B	~7GB	Balanced performance, multilingual
`phi3`	3.8B	~4GB	Efficient, resource-constrained systems
`qwen2.5:7b`	7B	~7GB	Strong reasoning, mathematics

```
# Large models (requires significant RAM)
llama3.1:70b     # Most capable local model
qwen2.5:72b      # Alibaba's flagship model
```

(continues on next page)

(continued from previous page)

```

# Medium models (good balance)
llama3.1:8b      # Recommended for most users
mistral-nemo:12b # Mistral's latest
codellama        # Specialized for coding

# Small models (fast, low resource)
llama3.2         # Latest efficient model (default)
phi3             # Microsoft's compact model
qwen2.5:7b       # Compact but capable
```

Setup and Configuration

1. **Install Ollama**:

```
# macOS
brew install ollama

# Linux
curl -fsSL https://ollama.com/install.sh | sh
```

2. **Download Models**:

```
# Download recommended starter models
ollama pull llama3.2        # General purpose (3B)
ollama pull codellama       # Code assistance (7B)
ollama pull mistral         # Balanced performance (7B)

# Optional: Download larger models if you have RAM
ollama pull llama3.1:8b     # Better reasoning (8B)
ollama pull qwen2.5:7b      # Strong at math/logic (7B)

# Check what's available
ollama list
```

3. **Start Ollama Service**:

```
# Start the service (runs on http://localhost:11434)
ollama serve

# Or run in background
nohup ollama serve > ollama.log 2>&1 &
```

4. **Configure Perspt**:

```
{
  "provider_type": "ollama",
  "default_model": "llama3.2",
  "providers": {
    "ollama": "http://localhost:11434/v1"
  },
  "api_key": "not-required"
}
```

CLI Usage

```
# Basic usage (no API key needed!)
perspt --provider-type ollama --model llama3.2

# Use specific models for different tasks
perspt --provider-type ollama --model codellama    # For coding
perspt --provider-type ollama --model mistral      # General purpose
perspt --provider-type ollama --model llama3.1:8b  # Better reasoning

# List installed Ollama models
perspt --provider-type ollama --list-models

# Test connection and performance
perspt --provider-type ollama --model llama3.2 --config ollama_config.json
```

Testing Different Models

```
# Quick test with small model
echo "Explain quantum computing in simple terms" | \
perspt --provider-type ollama --model llama3.2

# Coding test with Code Llama
echo "Write a Python function to sort a list" | \
perspt --provider-type ollama --model codellama

# Reasoning test with larger model
echo "Solve this logic puzzle: ..." | \
perspt --provider-type ollama --model llama3.1:8b
```

Performance Monitoring

```
# Monitor resource usage
htop  # Check CPU/Memory while running

# Time responses
time perspt --provider-type ollama --model llama3.2

# Compare model speeds
for model in llama3.2 mistral codellama; do
  echo "Testing $model..."
  time echo "What is 2+2?" | perspt --provider-type ollama --model $model
done
```

Benefits of Local Models

- **Privacy**: Data stays on your machine
- **Offline Usage**: No internet required after setup
- **Cost Control**: No per-token charges
- **Customization**: Fine-tune models for specific tasks

Environment Variables

```
export OLLAMA_HOST="http://localhost:11434"
```

3.5.4 Troubleshooting

This comprehensive troubleshooting guide helps you diagnose and resolve issues with Perspt's genai crate integration, provider connectivity, and advanced features.

Quick Diagnostics

Start with these diagnostic commands to check system status:

```
# Check provider connectivity and model availability
perspt --provider-type openai --list-models

# Validate specific model
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022 --list-models

# Test with minimal configuration
perspt --api-key your-key --provider-type openai --model gpt-3.5-turbo
```

Environment Variable Check

```
# Check if API keys are set
echo $OPENAI_API_KEY
echo $ANTHROPIC_API_KEY
echo $GOOGLE_API_KEY

# Verify genai crate can access providers
export RUST_LOG=debug
perspt --provider-type openai --list-models
```

Common Issues

GenAI Crate Integration Issues

Provider Authentication Failures

```
Error: Authentication failed for provider 'openai'
Caused by: Invalid API key
```

Solutions:

1. **Verify API key format**:

   ```
   # OpenAI keys start with 'sk-'
   echo $OPENAI_API_KEY | head -c 5  # Should show 'sk-'

   # Anthropic keys start with 'sk-ant-'
   echo $ANTHROPIC_API_KEY | head -c 7  # Should show 'sk-ant-'
   ```

2. **Test API key directly**:

   ```
   # Test OpenAI API key
   curl -H "Authorization: Bearer $OPENAI_API_KEY" \
        https://api.openai.com/v1/models

   # Test Anthropic API key
   ```

(continues on next page)

(continued from previous page)

```
curl -H "x-api-key: $ANTHROPIC_API_KEY" \
     https://api.anthropic.com/v1/models
```

3. **Check API key permissions and billing**: - Ensure API key has model access permissions - Verify account has sufficient credits/billing set up - Check for rate limiting or usage quotas

Model Validation Failures

```
Error: Model 'gpt-4.1' not available for provider 'openai'
Available models: gpt-3.5-turbo, gpt-4, gpt-4-turbo...
```

Solutions:

1. **Check model availability**:

```
# List all available models for provider
perspt --provider-type openai --list-models

# Search for specific model
perspt --provider-type openai --list-models | grep gpt-4
```

2. **Use correct model names**:

```
# Correct model names (case-sensitive)
perspt --provider-type openai --model gpt-4o-mini       # ✓ Correct
perspt --provider-type openai --model GPT-4O-Mini       # ▯ Wrong case
perspt --provider-type openai --model gpt4o-mini        # ▯ Missing hyphen
```

3. **Check provider-specific model access**: - Some models require special access (e.g., GPT-4, Claude Opus) - Verify your account tier supports the requested model - Check if model is in beta/preview status

Streaming Connection Issues

```
Error: Streaming connection interrupted
Caused by: Connection reset by peer
```

Solutions:

1. **Network connectivity check**:

```
# Test basic connectivity
ping api.openai.com
ping api.anthropic.com

# Check for proxy/firewall issues
curl -I https://api.openai.com/v1/models
```

2. **Provider service status**: - Check OpenAI Status: https://status.openai.com - Check Anthropic Status: https://status.anthropic.com - Check Google AI Status: https://status.google.com
3. **Adjust streaming settings**:

```
{
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "stream_timeout": 30,
  "retry_attempts": 3,
  "buffer_size": 1024
}
```

Common syntax errors:

```json
{
  "provider": "openai",  // □ Comments not allowed in JSON
  "api_key": "sk-...",   // □ Trailing comma
}
```

Correct format:

```json
{
  "provider": "openai",
  "api_key": "sk-..."
}
```

2. **Missing Required Fields**:

```json
{
  "provider": "openai"
  // □ Missing api_key
}
```

Solution: Ensure all required fields are present:

```json
{
  "provider": "openai",
  "api_key": "your-api-key",
  "model": "gpt-4"
}
```

Configuration File Not Found

```
Error: Configuration file not found at ~/.config/perspt/config.json
```

Solutions:

1. Create the configuration directory:

```bash
mkdir -p ~/.config/perspt
```

2. Create a basic configuration file:

```bash
cat > ~/.config/perspt/config.json << EOF
{
  "provider": "openai",
  "api_key": "your-api-key",
  "model": "gpt-4"
}
EOF
```

3. Specify a custom configuration path:

```
perspt --config /path/to/your/config.json
```

API Connection Issues

Invalid API Key

```
Error: Authentication failed - Invalid API key
```

Solutions:

1. **Verify API key format**:

   ```
   # OpenAI keys start with 'sk-'
   # Anthropic keys start with 'sk-ant-'
   # Check your provider's documentation
   ```

2. **Test API key manually**:

   ```
   # OpenAI
   curl -H "Authorization: Bearer YOUR_API_KEY" \\
        https://api.openai.com/v1/models

   # Anthropic
   curl -H "x-api-key: YOUR_API_KEY" \\
        -H "anthropic-version: 2023-06-01" \\
        https://api.anthropic.com/v1/messages
   ```

3. **Check API key permissions**: - Ensure the key has necessary permissions - Check if the key is associated with the correct organization - Verify the key hasn't expired

Network Connectivity Issues

```
Error: Failed to connect to API endpoint
```

Solutions:

1. **Check internet connectivity**:

   ```
   ping google.com
   curl -I https://api.openai.com
   ```

2. **Verify firewall/proxy settings**:

   ```
   # Check if behind corporate firewall
   echo $HTTP_PROXY
   echo $HTTPS_PROXY
   ```

3. **Test with different endpoints**:

   ```
   # Try different base URLs
   curl https://api.openai.com/v1/models
   curl https://api.anthropic.com/v1/models
   ```

4. **Configure proxy if needed**:

   ```
   {
     "provider": "openai",
     "proxy": {
   ```

(continues on next page)

(continued from previous page)

```
    "http": "http://proxy.company.com:8080",
    "https": "https://proxy.company.com:8080"
  }
}
```

Rate Limiting

```
Error: Rate limit exceeded
```

Solutions:

1. **Wait and retry**: - Most rate limits reset within minutes - Implement exponential backoff
2. **Check rate limits**:

   ```
   # Check OpenAI rate limits
   curl -H "Authorization: Bearer YOUR_API_KEY" \\
        https://api.openai.com/v1/usage
   ```

3. **Optimize requests**:

   ```
   {
     "rate_limiting": {
       "requests_per_minute": 50,
       "delay_between_requests": 1.2,
       "max_retries": 3
     }
   }
   ```

4. **Upgrade API plan**: - Consider higher-tier plans for increased limits - Contact provider support for enterprise limits

Model and Response Issues

Model Not Available

```
Error: Model 'gpt-5' not found
```

Solutions:

1. **Check available models**:

   ```
   > /list-models
   ```

2. **Verify model name spelling**:

   ```
   {
     "model": "gpt-4-turbo",  // ✓ Correct
     "model": "gpt-4-turob"   // □ Typo
   }
   ```

3. **Check provider model availability**: - Some models may be region-specific - Newer models might not be available to all users

Slow Responses

Causes and solutions:

1. **Large context windows**:

```
{
  "max_tokens": 1000,        // ✓ Reasonable
  "conversation_history_limit": 20  // ✓ Limit history
}
```

2. **Network latency**:

```
# Test latency to provider
ping api.openai.com
```

3. **Provider server load**: - Check provider status pages - Try different models or regions

Unexpected Responses

```
AI responses seem off-topic or inappropriate
```

Solutions:

1. **Review system prompt**:

```
{
  "system_prompt": "You are a helpful assistant..."  // Clear instructions
}
```

2. **Adjust model parameters**:

```
{
  "temperature": 0.3,     // Lower for more focused responses
  "top_p": 0.8,          // Reduce randomness
  "frequency_penalty": 0.2  // Reduce repetition
}
```

3. **Clear conversation history**:

```
> /clear
```

Local Model Issues

Ollama Connection Failed

```
Error: Failed to connect to Ollama at localhost:11434
```

Solutions:

1. **Check if Ollama is running**:

```
# Start Ollama
ollama serve

# Check if running
curl http://localhost:11434/api/tags
```

2. **Verify model is installed**:

```
ollama list
ollama pull llama2:7b  # Install if missing
```

3. **Check port configuration**:

```
{
  "provider": "ollama",
  "base_url": "http://localhost:11434"  // Correct port
}
```

Insufficient Memory/GPU

```
Error: Out of memory when loading model
```

Solutions:

1. **Use smaller models**:

   ```
   # Instead of 13B model, use 7B
   ollama pull llama2:7b
   ollama pull mistral:7b
   ```

2. **Adjust GPU layers**:

   ```
   {
     "provider": "ollama",
     "options": {
       "num_gpu": 0,     // Use CPU only
       "num_thread": 4   // Limit CPU threads
     }
   }
   ```

3. **Monitor system resources**:

   ```
   # Check memory usage
   htop
   nvidia-smi  # For GPU usage
   ```

Platform-Specific Issues

macOS Issues

Gatekeeper Blocking Execution

```
"perspt" cannot be opened because it is from an unidentified developer
```

Solution:

```
sudo xattr -rd com.apple.quarantine /path/to/perspt
```

Homebrew Installation Issues

```
# Update Homebrew
brew update
brew upgrade

# Clear caches
brew cleanup

# Reinstall if needed
```

(continues on next page)

(continued from previous page)

```
brew uninstall perspt
brew install perspt
```

Linux Issues

Missing Shared Libraries

```
error while loading shared libraries: libssl.so.1.1
```

Solutions:

```
# Ubuntu/Debian
sudo apt update
sudo apt install libssl1.1 libssl-dev

# Fedora/RHEL
sudo dnf install openssl-libs openssl-devel

# Check library dependencies
ldd /path/to/perspt
```

Permission Issues

```
# Make executable
chmod +x perspt

# Install system-wide
sudo cp perspt /usr/local/bin/
```

Windows Issues

PowerShell Execution Policy

```
# Check current policy
Get-ExecutionPolicy

# Set policy to allow local scripts
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Windows Defender False Positive

1. Add Perspt to Windows Defender exclusions
2. Download from official sources only
3. Verify file hashes if available

Advanced Troubleshooting

Debug Mode

Enable detailed logging:

```
{
  "debug": {
```

(continues on next page)

(continued from previous page)

```
    "enabled": true,
    "log_level": "trace",
    "log_file": "~/.config/perspt/debug.log"
  }
}
```

Run with verbose output:

```
perspt --verbose --debug
```

Log Analysis

Check log files for detailed error information:

```
# View recent logs
tail -f ~/.config/perspt/perspt.log

# Search for specific errors
grep -i "error" ~/.config/perspt/perspt.log

# Analyze API calls
grep -i "api" ~/.config/perspt/debug.log
```

Network Debugging

Capture network traffic:

```
# Using tcpdump (Linux/macOS)
sudo tcpdump -i any -n host api.openai.com

# Using netstat
netstat -an | grep :443
```

Test with curl:

```
# Test OpenAI API
curl -v -H "Authorization: Bearer YOUR_API_KEY" \\
     -H "Content-Type: application/json" \\
     -d '{"model":"gpt-4o-mini","messages":[{"role":"user","content":"Hello"}]}' \\
     https://api.openai.com/v1/chat/completions
```

Configuration Debugging

Validate configuration:

```
# Check JSON syntax
python -c "import json; print(json.load(open('config.json')))"

# Validate with Perspt
perspt --validate-config
```

Test minimal configuration:

```
{
  "provider": "openai",
  "api_key": "your-key",
  "model": "gpt-4o-mini"
}
```

Performance Debugging

Monitor resource usage:

```
# Monitor CPU and memory
top -p $(pgrep perspt)

# Monitor disk I/O
iotop -p $(pgrep perspt)
```

Profile network usage:

```
# Monitor bandwidth usage
netlimit -p $(pgrep perspt)
```

Recovery Procedures

Reset Configuration

1. **Backup current configuration**:

   ```
   cp ~/.config/perspt/config.json ~/.config/perspt/config.json.backup
   ```

2. **Reset to defaults**:

   ```
   rm ~/.config/perspt/config.json
   perspt --create-config
   ```

3. **Restore from backup if needed**:

   ```
   cp ~/.config/perspt/config.json.backup ~/.config/perspt/config.json
   ```

Clear Cache and Data

```
# Clear conversation history
rm -rf ~/.config/perspt/history/

# Clear cache
rm -rf ~/.config/perspt/cache/

# Clear temporary files
rm -rf /tmp/perspt*
```

Complete Reinstallation

```
# Remove all Perspt data
rm -rf ~/.config/perspt/
rm -rf ~/.local/share/perspt/

# Uninstall and reinstall
# (method depends on installation method)
```

Getting Help

Community Support

- **GitHub Issues**: Report bugs and feature requests
- **Discussions**: Ask questions and share tips
- **Discord/Slack**: Real-time community support

Reporting Issues

When reporting issues, include:

1. **System information**:

   ```
   perspt --version
   uname -a  # or systeminfo on Windows
   ```

2. **Configuration** (sanitized):

   ```
   {
     "provider": "openai",
     "model": "gpt-4",
     "api_key": "sk-***redacted***"
   }
   ```

3. **Error messages** (full text)
4. **Steps to reproduce**
5. **Expected vs actual behavior**

Professional Support

For enterprise users:

- **Priority support tickets**
- **Direct communication channels**
- **Custom configuration assistance**
- **Integration consulting**

Provider-Specific Troubleshooting

OpenAI Provider Issues

Authentication and API Key Problems

```
Error: Invalid API key for OpenAI
Error: Rate limit exceeded for model gpt-4
```

Solutions:

1. **API Key Validation**:

```
# Verify OpenAI API key format (should start with 'sk-')
echo $OPENAI_API_KEY | head -c 3  # Should show 'sk-'

# Test API key with curl
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     https://api.openai.com/v1/models
```

2. **Rate Limiting Management**:

```
# Use tier-appropriate models
perspt --provider-type openai --model gpt-3.5-turbo  # Lower tier
perspt --provider-type openai --model gpt-4o-mini    # Tier 1+
perspt --provider-type openai --model gpt-4          # Tier 3+
```

3. **Quota and Billing Issues**: - Check OpenAI dashboard for usage limits - Verify payment method is valid - Monitor usage to avoid unexpected charges

Model Access Issues

```
Error: Model 'o1-preview' not available
Error: Insufficient quota for GPT-4
```

Solutions:

1. **Model Tier Requirements**:

```
# Tier 1 models (widely available)
perspt --provider-type openai --model gpt-3.5-turbo
perspt --provider-type openai --model gpt-4o-mini

# Tier 2+ models (higher usage requirements)
perspt --provider-type openai --model gpt-4
perspt --provider-type openai --model gpt-4-turbo

# Special access models (invitation/waitlist)
perspt --provider-type openai --model o1-preview
perspt --provider-type openai --model o1-mini
```

2. **Reasoning Model Limitations**: - o1 models have special usage patterns - Higher latency expected for reasoning - May have stricter rate limits

Anthropic Provider Issues

Claude Model Access

```
Error: Model 'claude-3-opus-20240229' not available
Error: Anthropic API key authentication failed
```

Solutions:

1. **API Key Format**:

```
# Anthropic keys start with 'sk-ant-'
echo $ANTHROPIC_API_KEY | head -c 7  # Should show 'sk-ant-'

```

(continues on next page)

(continued from previous page)

```
# Test with curl
curl -H "x-api-key: $ANTHROPIC_API_KEY" \
     -H "anthropic-version: 2023-06-01" \
     https://api.anthropic.com/v1/models
```

2. **Model Availability**:

```
# Generally available models
perspt --provider-type anthropic --model claude-3-5-sonnet-20241022
perspt --provider-type anthropic --model claude-3-5-haiku-20241022

# Request access for Opus through Anthropic Console
perspt --provider-type anthropic --model claude-3-opus-20240229
```

3. **Rate Limiting**: - Anthropic has strict rate limits for new accounts - Build up usage history for higher limits - Use Haiku model for testing and development

Google AI (Gemini) Provider Issues

API Key and Setup Problems

```
Error: Google AI API key not valid
Error: Gemini model access denied
```

Solutions:

1. **API Key Configuration**:

```
# Get API key from Google AI Studio
export GOOGLE_API_KEY="your-api-key"
# Alternative environment variable
export GEMINI_API_KEY="your-api-key"

# Test API access
curl -H "Content-Type: application/json" \
     "https://generativelanguage.googleapis.com/v1beta/models?key=$GOOGLE_API_KEY"
```

2. **Model Selection**:

```
# Recommended models
perspt --provider-type google --model gemini-1.5-flash     # Fast, cost-effective
perspt --provider-type google --model gemini-1.5-pro      # Balanced capability
perspt --provider-type google --model gemini-1.5-pro-exp  # Experimental features
```

3. **Geographic Restrictions**: - Some Gemini models have geographic limitations - Check Google AI availability in your region - Use VPN if necessary and allowed by Google's terms

Groq Provider Issues

Service Availability

```
Error: Groq service temporarily unavailable
Error: Model inference timeout
```

Solutions:

1. **Service Reliability**: - Groq prioritizes speed over availability - Configure fallback providers for production use - Monitor Groq status page for outages
2. **Model Selection**:

   ```
   # Fast inference models
   perspt --provider-type groq --model llama-3.1-8b-instant
   perspt --provider-type groq --model mixtral-8x7b-32768
   perspt --provider-type groq --model gemma-7b-it
   ```

3. **Timeout Configuration**:

   ```
   {
     "provider_type": "groq",
     "timeout": 30,
     "retry_attempts": 2,
     "fallback_provider": "openai"
   }
   ```

Cohere Provider Issues

API Integration Problems

```
Error: Cohere API authentication failed
Error: Model 'command-r-plus' not accessible
```

Solutions:

1. **API Key Setup**:

   ```
   export COHERE_API_KEY="your-api-key"

   # Test API access
   curl -H "Authorization: Bearer $COHERE_API_KEY" \
        https://api.cohere.ai/v1/models
   ```

2. **Model Access**:

   ```
   # Available Cohere models
   perspt --provider-type cohere --model command-r
   perspt --provider-type cohere --model command-r-plus
   perspt --provider-type cohere --model command-light
   ```

XAI (Grok) Provider Issues

Grok Model Access

```
Error: XAI API key invalid
Error: Grok model not available
```

Solutions:

1. **API Configuration**:

   ```
   export XAI_API_KEY="your-api-key"
   ```

(continues on next page)

(continued from previous page)

```
# Check available models
perspt --provider-type xai --list-models
```

2. **Model Selection**:

```
# Available Grok models
perspt --provider-type xai --model grok-beta
```

Ollama (Local) Provider Issues

Service Connection Problems

```
Error: Could not connect to Ollama server
Error: Model not found in Ollama
```

Solutions:

1. **Ollama Service Management**:

```
# Check if Ollama is running
curl http://localhost:11434/api/tags

# Start Ollama service
ollama serve

# Start as background service (macOS)
brew services start ollama
```

2. **Model Management**:

```
# List installed models
ollama list

# Install popular models
ollama pull llama3.2:8b
ollama pull mistral:7b
ollama pull codellama:7b

# Remove unused models to save space
ollama rm unused-model
```

3. **Resource Optimization**:

```
# Check system resources
htop
nvidia-smi  # For GPU users

# Use smaller models for limited resources
ollama pull llama3.2:3b      # 3B parameters
ollama pull phi3:mini        # Microsoft Phi-3 Mini
```

4. **Configuration Tuning**:

```
{
  "provider_type": "ollama",
```

(continues on next page)

(continued from previous page)

```
  "base_url": "http://localhost:11434",
  "options": {
    "num_gpu": 1,           // Number of GPU layers
    "num_thread": 8,        // CPU threads
    "num_ctx": 4096,        // Context window
    "temperature": 0.7,
    "top_p": 0.9
  }
}
```

Performance Optimization

Response Time Optimization

Model Selection for Speed

```
# Fastest models by provider
perspt --provider-type groq --model llama-3.1-8b-instant     # Groq (fastest)
perspt --provider-type openai --model gpt-4o-mini            # OpenAI (fast)
perspt --provider-type google --model gemini-1.5-flash       # Google (fast)
perspt --provider-type anthropic --model claude-3-5-haiku-20241022  # Anthropic (fast)
```

Configuration Tuning

```
{
  "performance": {
    "max_tokens": 1000,           // Limit response length
    "stream": true,               // Enable streaming
    "timeout": 15,                // Shorter timeout
    "parallel_requests": 2,       // Multiple requests
    "cache_responses": true       // Cache similar queries
  }
}
```

Memory and Resource Management

System Resource Monitoring

```
# Monitor CPU and memory usage
top -p $(pgrep perspt)

# Monitor network usage
iftop -i any -f "host api.openai.com"

# Check disk usage for logs and cache
du -sh ~/.config/perspt/
```

Resource Optimization

```
{
  "resource_limits": {
    "max_history_size": 50,       // Limit conversation history
```

(continues on next page)

(continued from previous page)

```
    "cache_size_mb": 100,          // Limit cache size
    "log_rotation_size": "10MB",   // Rotate logs
    "cleanup_interval": "24h"      // Regular cleanup
  }
}
```

Network Performance

Connection Optimization

```
{
  "network": {
    "keep_alive": true,            // Reuse connections
    "connection_pool_size": 5,     // Pool connections
    "dns_cache": true,             // Cache DNS lookups
    "compression": true            // Enable compression
  }
}
```

Regional Configuration

```
{
  "provider_endpoints": {
    "openai": "https://api.openai.com",            // US
    "anthropic": "https://api.anthropic.com",      // US
    "google": "https://generativelanguage.googleapis.com"  // Global
  }
}
```

Advanced Recovery Procedures

Complete System Reset

Full Configuration Reset

```
# Backup current configuration
cp -r ~/.config/perspt ~/.config/perspt.backup.$(date +%Y%m%d)

# Remove all Perspt data
rm -rf ~/.config/perspt/
rm -rf ~/.local/share/perspt/
rm -rf ~/.cache/perspt/

# Clear temporary files
rm -rf /tmp/perspt*

# Recreate default configuration
perspt --create-default-config
```

Selective Reset Options

```
# Reset only configuration
rm ~/.config/perspt/config.json
perspt --setup

# Clear only cache
rm -rf ~/.config/perspt/cache/

# Clear only conversation history
rm -rf ~/.config/perspt/history/

# Reset only logs
rm ~/.config/perspt/*.log
```

Emergency Fallback Procedures

Provider Fallback Chain

```
{
  "fallback_chain": [
    {
      "provider_type": "openai",
      "model": "gpt-4o-mini",
      "on_failure": "next"
    },
    {
      "provider_type": "anthropic",
      "model": "claude-3-5-haiku-20241022",
      "on_failure": "next"
    },
    {
      "provider_type": "ollama",
      "model": "llama3.2:8b",
      "on_failure": "fail"
    }
  ]
}
```

Manual Override Mode

```
# Force specific provider regardless of config
perspt --force-provider openai --force-model gpt-3.5-turbo

# Use minimal configuration
perspt --no-config --api-key sk-... --provider-type openai

# Debug mode with maximum verbosity
perspt --debug --verbose --log-level trace
```

Data Recovery

Conversation History Recovery

```
# Check for backup files
ls ~/.config/perspt/history/*.backup

# Restore from backup
cp ~/.config/perspt/history/conversation.backup \
   ~/.config/perspt/history/conversation.json

# Export conversations before reset
perspt --export-history ~/perspt-backup.json
```

Configuration Recovery

```
# Restore from automatic backup
cp ~/.config/perspt/config.json.backup ~/.config/perspt/config.json

# Recreate from environment variables
perspt --config-from-env

# Interactive configuration rebuild
perspt --reconfigure
```

Version Migration Issues

Upgrading from allms to genai

```
# Backup old configuration
cp ~/.config/perspt/config.json ~/.config/perspt/config.allms.backup

# Run migration script
perspt --migrate-config

# Manual migration if needed
perspt --validate-config --fix-issues
```

Downgrade Procedures

```
# Install specific version
cargo install perspt --version 0.2.0

# Use version-specific configuration
cp ~/.config/perspt/config.v0.2.0.json ~/.config/perspt/config.json
```

Emergency Contact and Support

Critical Issue Escalation

For production-critical issues:

1. **Immediate Workarounds**: - Switch to backup providers - Use local models (Ollama) for offline capability - Enable debug logging for detailed diagnosis

2. **Community Support Channels**: - GitHub Issues: https://github.com/eonseed/perspt/issues - Discord Community: [Link to Discord] - Reddit: r/perspt
3. **Enterprise Support**: - Priority ticket system - Direct developer contact - Custom configuration assistance

Issue Documentation Template

When reporting issues, include this information:

```
**Environment Information:**
- OS: [macOS 14.1 / Ubuntu 22.04 / Windows 11]
- Perspt Version: [perspt --version]
- Installation Method: [cargo / brew / binary]

**Configuration:**
- Provider: [openai / anthropic / google / etc.]
- Model: [gpt-4o-mini / claude-3-5-sonnet / etc.]
- Config file: [attach sanitized config.json]

**Error Details:**
- Full error message: [exact text]
- Error code: [if available]
- Stack trace: [if available]

**Reproduction Steps:**
1. [Step 1]
2. [Step 2]
3. [Error occurs]

**Expected vs Actual Behavior:**
- Expected: [what should happen]
- Actual: [what actually happens]

**Additional Context:**
- Network environment: [corporate / home / proxy]
- Recent changes: [configuration / system updates]
- Workarounds attempted: [list what you've tried]
```

Recovery Verification

After resolving issues, verify system health:

```
# Test basic functionality
perspt --provider-type openai --model gpt-3.5-turbo --test-connection

# Verify configuration
perspt --validate-config

# Test streaming
echo "Hello" | perspt --provider-type openai --model gpt-4o-mini --stream

# Check all providers
for provider in openai anthropic google groq; do
  echo "Testing $provider..."
```

(continues on next page)

(continued from previous page)

```
  perspt --provider-type $provider --list-models
done
```

Related Documentation

For additional help:

- *AI Providers* - Provider-specific configuration and features
- *Advanced Features* - Advanced usage patterns and optimization
- *Configuration Guide* - Complete configuration reference
- *Developer Guide* - Development and API documentation
- *API Reference* - API reference and integration guides

3.5.5 Overview

Perspt is a high-performance terminal-based chat application built with Rust that provides a unified interface for interacting with multiple Large Language Model (LLM) providers. Using the modern *genai* crate (v0.3.5), Perspt offers seamless access to the latest AI models with enhanced streaming capabilities, robust error handling, and intuitive CLI functionality.

Key Features:

- **Multi-Provider Support**: OpenAI, Anthropic, Google, Groq, Cohere, XAI, and Ollama
- **Latest Models**: Support for reasoning models (o1-mini, o1-preview, o3-mini), GPT-4.1, Claude 3.5, Gemini 2.5 Pro
- **Real-time Streaming**: Enhanced streaming with proper reasoning chunk handling
- **Robust CLI**: Working command-line arguments with model validation
- **Beautiful UI**: Responsive terminal interface with markdown rendering
- **Conversation Saving**: Export chat sessions to text files with `/save` command

🚀 Basic Usage Learn the fundamentals of chatting with AI models, keyboard shortcuts, and everyday usage patterns.

Basic Usage ⚡ Advanced Features Discover powerful features like input queuing, markdown rendering, and productivity workflows.

Advanced Features 🔀 Provider Guide Complete guide to all supported AI providers, their models, and specific configuration options.

AI Providers 🛠️ Troubleshooting Solutions to common issues, error messages, and optimization tips.

Troubleshooting

3.5.6 Quick Reference

Essential Keyboard Shortcuts

Shortcut	Action
Enter	Send message
Ctrl+C	Exit application
↑/↓ Keys	Scroll chat history
Page Up/Down	Fast scroll
Ctrl+L	Clear screen

Common Commands

```
# Start with default settings (gpt-4o-mini)
persp t

# Use specific model with validation
persp t --model gpt-4.1

# Switch provider and model
persp t --provider-type anthropic --model claude-3-5-sonnet-20241022

# List available models for current provider
persp t --list-models

# Use custom configuration file
persp t --config my-config.json

# Override API key from command line
persp t --api-key your-api-key

# Use provider profile from config
persp t --provider work-profile
```

3.5.7 Typical Workflows

Daily Development

1. **Code Review**: Paste code and ask for feedback
2. **Documentation**: Generate or improve documentation
3. **Debugging**: Discuss error messages and solutions
4. **Learning**: Ask about new technologies or concepts

Research and Writing

1. **Information Gathering**: Ask questions about topics
2. **Content Creation**: Get help with writing and editing
3. **Brainstorming**: Generate ideas and explore concepts
4. **Fact Checking**: Verify information and get references

3.5.8 Getting the Most from Persp t

Best Practices

- **Be Specific**: Clear, detailed questions get better answers
- **Provide Context**: Include relevant background information
- **Iterate**: Build on previous responses for deeper understanding
- **Experiment**: Try different models for different types of tasks

Productivity Tips

- **Use Configuration Files**: Set up profiles for different use cases
- **Learn Keyboard Shortcuts**: Speed up your workflow
- **Leverage Streaming**: Keep typing while AI responds
- **Save Important Conversations**: Copy valuable responses

3.5.9 What's Next?

Choose your path based on your experience level:

New Users: Start with *Basic Usage* to learn the fundamentals.

Experienced Users: Jump to *Advanced Features* for productivity techniques.

Multi-Provider Users: Check out *AI Providers* for provider-specific tips.

Having Issues?: Visit *Troubleshooting* for solutions.

See also

- *Getting Started* - Initial setup and first conversation
- *Configuration Guide* - Customizing Perspt for your workflow
- *Developer Guide* - Contributing and extending Perspt

3.6 Developer Guide

Welcome to the Perspt developer guide! This section is for developers who want to understand Perspt's architecture, contribute to the project, or extend its functionality.

3.6.1 Architecture

This document provides a comprehensive overview of Perspt's architecture, design principles, and internal structure.

Overview

Perspt is built as a modular, extensible command-line application written in Rust. The architecture emphasizes:

- **Modularity**: Clear separation of concerns with well-defined interfaces
- **Extensibility**: Plugin-based architecture for adding new providers and features
- **Performance**: Efficient resource usage and fast response times
- **Reliability**: Robust error handling and graceful degradation
- **Security**: Safe handling of API keys and user data

High-Level Architecture

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   User Input    │    │  Configuration  │    │   AI Providers  │
│   (CLI/Simple)  │    │    Manager      │    │   (OpenAI,etc.) │
└─────────┬───────┘    └─────────┬───────┘    └─────────┬───────┘
          │                      │                      │
          v                      v                      v
┌─────────────────────────────────────────────────────────────────┐
│                    Core Application                             │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐              │
│  │ UI Manager  │  │ LLM Bridge  │  │ Config Mgr  │              │
│  │   / CLI     │  │             │  │             │              │
│  └─────────────┘  └─────────────┘  └─────────────┘              │
└─────────────────────────────────────────────────────────────────┘
          │                      │                      │
          v                      v                      v
```

(continues on next page)

(continued from previous page)

```
┌──────────────────┐    ┌──────────────────┐    ┌──────────────────┐
│  Conversation    │    │    Provider      │    │    Storage       │
│    Manager       │    │    Registry      │    │    Layer         │
└──────────────────┘    └──────────────────┘    └──────────────────┘
```

Core Components

main.rs

The application entry point and orchestration layer.

Responsibilities:

- Command-line argument parsing with clap derive macros
- Application initialization and mode selection (TUI vs Simple CLI)
- Main event loop coordination for both interface modes
- Graceful shutdown handling with terminal restoration
- Comprehensive panic handling with contextual error messages

Key Functions:

```rust
#[tokio::main]
async fn main() -> Result<()> {
    // Set up panic hook before anything else
    setup_panic_hook();

    // Initialize logging
    env_logger::Builder::from_default_env()
        .filter_level(LevelFilter::Error)
        .init();

    // Parse CLI arguments with clap
    let matches = Command::new("Perspt - Performance LLM Chat CLI")
        .version("0.4.5")
        .author("Vikrant Rathore")
        .about("A performant CLI for talking to LLMs using the genai crate")
        .arg(Arg::new("simple-cli")
            .long("simple-cli")
            .help("Use simple command-line interface instead of TUI")
            .action(ArgAction::SetTrue))
        .arg(Arg::new("log-file")
            .long("log-file")
            .help("Log conversation to file (Simple CLI mode only)")
            .value_name("FILE")
            .action(ArgAction::Set))
        // ... other argument definitions
        .get_matches();

    // Load configuration and create provider
    let config = config::load_config(config_path).await?;
    let provider = Arc::new(GenAIProvider::new_with_config(
        config.provider_type.as_deref(),
        config.api_key.as_deref()
    )?);
```

(continues on next page)

(continued from previous page)

```rust

    // Route to appropriate interface mode
    if matches.get_flag("simple-cli") {
        // Simple CLI mode - minimal Unix-style interface
        let log_file = matches.get_one::<String>("log-file").cloned();
        cli::run_simple_cli(config, model_name, api_key, provider, log_file).await?;
    } else {
        // TUI mode - rich terminal interface
        let mut terminal = initialize_terminal()?;
        run_ui(&mut terminal, config, model_name, api_key, provider).await?;
        cleanup_terminal()?;
    }

    Ok(())
}

fn setup_panic_hook() {
    panic::set_hook(Box::new(move |panic_info| {
        // Force terminal restoration immediately
        let _ = disable_raw_mode();
        let _ = execute!(io::stdout(), LeaveAlternateScreen);

        // Provide contextual error messages and recovery tips
        let panic_str = format!("{}", panic_info);
        if panic_str.contains("PROJECT_ID") {
            eprintln!("□ Tip: Set PROJECT_ID environment variable for Google Gemini");
        } else if panic_str.contains("API key") {
            eprintln!("□ Tip: Set your provider's API key environment variable");
        } else if panic_str.contains("model") {
            eprintln!("□ Tip: Use --list-models to see available models");
        }
        // ... more context-specific help
    }));
}
```

config.rs

Configuration management and validation.

Responsibilities:

- Configuration file parsing (JSON)
- Environment variable integration
- Configuration validation and defaults
- Provider inference and API key management

Key Structures:

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Config {
    pub provider: String,
    pub api_key: Option<String>,
    pub model: Option<String>,
```

(continues on next page)

(continued from previous page)

```rust
    pub temperature: Option<f32>,
    pub max_tokens: Option<u32>,
    pub timeout_seconds: Option<u64>,
}

impl Config {
    pub fn load() -> Result<Self, ConfigError> {
        // Load from file, environment, or defaults
    }

    pub fn infer_provider_from_key(api_key: &str) -> String {
        // Smart provider inference from API key format
    }

    pub fn get_effective_model(&self) -> String {
        // Get model with provider-specific defaults
    }
}
```

llm_provider.rs

LLM provider abstraction using the genai crate for unified API access.

Responsibilities:

- Multi-provider LLM integration (OpenAI, Anthropic, Gemini, Groq, Cohere, XAI, DeepSeek, Ollama)
- Streaming response handling with real-time updates
- Error handling and retry logic
- Message formatting and conversation management

Key Functions:

```rust
use genai::chat::{ChatMessage, ChatRequest, ChatRequestOptions, ChatResponse};
use genai::Client;

pub async fn send_message(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // Create GenAI client with provider configuration
    let client = Client::default();

    // Build chat request with streaming enabled
    let chat_req = ChatRequest::new(vec![
        ChatMessage::system("You are a helpful assistant."),
        ChatMessage::user(message),
    ]);

    // Configure request options
    let options = ChatRequestOptions {
        model: Some(config.get_effective_model()),
        temperature: config.temperature,
```

(continues on next page)

(continued from previous page)

```
        max_tokens: config.max_tokens,
        stream: Some(true),
        ..Default::default()
    };

    // Execute streaming request
    let stream = client.exec_stream(&chat_req, &options).await?;

    // Process streaming response
    while let Some(chunk) = stream.next().await {
        match chunk {
            Ok(response) => {
                if let Some(content) = response.content_text_as_str() {
                    tx.send(content.to_string())?;
                }
            }
            Err(e) => return Err(e.into()),
        }
    }

    Ok(())
}
```

Provider Support:

The GenAI crate provides unified access to:

- **OpenAI**: GPT-4, GPT-3.5, GPT-4o, o1-mini, o1-preview, o3-mini, o4-mini models
- **Anthropic**: Claude 3 (Opus, Sonnet, Haiku), Claude 3.5 models
- **Google**: Gemini Pro, Gemini 1.5 Pro/Flash, Gemini 2.0 models
- **Groq**: Llama 3.x models with ultra-fast inference
- **Cohere**: Command R/R+ models
- **XAI**: Grok models (grok-3-beta, grok-3-fast-beta, etc.)
- **DeepSeek**: DeepSeek chat and reasoning models
- **Ollama**: Local model hosting (requires local setup)

Streaming Architecture:

The streaming implementation uses Tokio channels for real-time communication:

```
// Channel for streaming content to UI
let (tx, mut rx) = tokio::sync::mpsc::unbounded_channel::<String>();

// Spawn streaming task
let stream_task = tokio::spawn(async move {
    send_message(&config, &message, tx).await
});

// Handle streaming updates in UI thread
while let Some(content) = rx.recv().await {
    // Update UI with new content
    update_ui_content(content);
}
```

ui.rs

Terminal UI management using Ratatui for responsive user interaction.

Responsibilities:

- Real-time terminal UI rendering with Ratatui
- Cross-platform input handling with Crossterm
- Streaming content display with immediate updates
- Markdown rendering with pulldown-cmark
- Conversation history management

Key Functions:

```rust
use ratatui::{
    backend::CrosstermBackend,
    layout::{Constraint, Direction, Layout},
    style::{Color, Modifier, Style},
    text::{Line, Span, Text},
    widgets::{Block, Borders, Clear, List, ListItem, Paragraph, Wrap},
    Frame, Terminal
};
use crossterm::{
    event::{self, DisableMouseCapture, EnableMouseCapture, Event, KeyCode},
    execute,
    terminal::{disable_raw_mode, enable_raw_mode, EnterAlternateScreen, LeaveAlternateScreen},
};

pub async fn run_ui(
    terminal: &mut Terminal<CrosstermBackend<std::io::Stdout>>,
    config: Config,
    model_name: String,
    api_key: String,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut app = App::new(config, model_name, api_key);

    loop {
        // Render UI frame
        terminal.draw(|f| ui(f, &app))?;

        // Handle events with timeout for responsiveness
        if event::poll(Duration::from_millis(50))? {
            if let Event::Key(key) = event::read()? {
                match app.handle_key_event(key).await {
                    Ok(should_quit) => {
                        if should_quit { break; }
                    }
                    Err(e) => app.set_error(format!("Error: {}", e)),
                }
            }
        }

        // Handle streaming updates
        app.process_streaming_updates();
```

(continues on next page)

(continued from previous page)

```
    }

    Ok(())
}

fn ui(f: &mut Frame, app: &App) {
    // Create responsive layout
    let chunks = Layout::default()
        .direction(Direction::Vertical)
        .constraints([
            Constraint::Min(3),     // Messages area
            Constraint::Length(3),  // Input area
            Constraint::Length(1),  // Status bar
        ])
        .split(f.size());

    // Render conversation messages
    render_messages(f, app, chunks[0]);

    // Render input area with prompt
    render_input_area(f, app, chunks[1]);

    // Render status bar with model info
    render_status_bar(f, app, chunks[2]);
}
```

Real-time Streaming:

The UI handles streaming responses with immediate display updates:

```
impl App {
    pub fn process_streaming_updates(&mut self) {
        // Non-blocking check for new streaming content
        while let Ok(content) = self.stream_receiver.try_recv() {
            if let Some(last_message) = self.messages.last_mut() {
                last_message.content.push_str(&content);
                self.scroll_to_bottom = true;
            }
        }
    }

    pub fn start_streaming_response(&mut self, user_message: String) {
        // Add user message to conversation
        self.add_message(Message::user(user_message.clone()));

        // Add placeholder for assistant response
        self.add_message(Message::assistant(String::new()));

        // Start streaming task
        let config = self.config.clone();
        let tx = self.stream_sender.clone();

        tokio::spawn(async move {
```

(continues on next page)

(continued from previous page)

```
            if let Err(e) = send_message(&config, &user_message, tx).await {
                // Handle streaming errors
                eprintln!("Streaming error: {}", e);
            }
        });
    }
}
```

Markdown Rendering:

Conversation messages support rich markdown formatting:

```
use pulldown_cmark::{Event, Options, Parser, Tag};

fn render_markdown_to_text(markdown: &str) -> Text {
    let parser = Parser::new_ext(markdown, Options::all());
    let mut spans = Vec::new();

    for event in parser {
        match event {
            Event::Text(text) => {
                spans.push(Span::raw(text.to_string()));
            }
            Event::Code(code) => {
                spans.push(Span::styled(
                    code.to_string(),
                    Style::default().fg(Color::Yellow).add_modifier(Modifier::BOLD)
                ));
            }
            Event::Start(Tag::Strong) => {
                // Handle bold text styling
            }
            // ... other markdown elements
            _ => {}
        }
    }

    Text::from(Line::from(spans))
}
```

Enhanced Scroll Handling:

Recent improvements to the scroll system ensure accurate display of long responses:

```
impl App {
    /// Calculate maximum scroll position with text wrapping awareness
    pub fn max_scroll(&self) -> usize {
        // Calculate visible height for the chat area
        let chat_area_height = self.terminal_height.saturating_sub(11).max(1);
        let visible_height = chat_area_height.saturating_sub(2).max(1);

        // Calculate terminal width for text wrapping calculations
        let chat_width = self.input_width.saturating_sub(4).max(20);
```

(continues on next page)

(continued from previous page)

```
        // Calculate actual rendered lines accounting for text wrapping
        let total_rendered_lines: usize = self.chat_history
            .iter()
            .map(|msg| {
                let mut lines = 1; // Header line

                // Content lines - account for text wrapping
                for line in &msg.content {
                    let line_text = line.spans.iter()
                        .map(|span| span.content.as_ref())
                        .collect::<String>();

                    if line_text.trim().is_empty() {
                        lines += 1; // Empty lines
                    } else {
                        // Character-based text wrapping calculation
                        let display_width = line_text.chars().count();
                        if display_width <= chat_width {
                            lines += 1;
                        } else {
                            let wrapped_lines = (display_width + chat_width - 1) / chat_
↪width;
                            lines += wrapped_lines.max(1);
                        }
                    }
                }

                lines += 1; // Separator line after each message
                lines
            })
            .sum();

        // Conservative scroll calculation to prevent content cutoff
        if total_rendered_lines > visible_height {
            let max_scroll = total_rendered_lines.saturating_sub(visible_height);
            max_scroll.saturating_sub(1) // Buffer to ensure last lines are visible
        } else {
            0
        }
    }

    /// Update scroll state with accurate content length calculation
    pub fn update_scroll_state(&mut self) {
        // Uses same logic as max_scroll() for consistency
        let chat_width = self.input_width.saturating_sub(4).max(20);
        let total_rendered_lines = /* same calculation as above */;

        self.scroll_state = self.scroll_state
            .content_length(total_rendered_lines.max(1))
            .position(self.scroll_position);
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Key Scroll Improvements:

- **Text Wrapping Awareness**: Uses character count (*.chars().count()*) instead of byte length for accurate Unicode text measurement
- **Conservative Buffering**: Reduces max scroll by 1 position to prevent content cutoff at bottom
- **Consistent Separator Handling**: Always includes separator lines after each message for uniform spacing
- **Terminal Width Adaptive**: Properly calculates available chat area excluding UI borders and padding
- **Synchronized State**: Both *max_scroll()* and *update_scroll_state()* use identical line counting logic

These improvements ensure that all lines of long LLM responses are visible and properly scrollable, especially when viewing the bottom of the conversation.

Data Flow

Real-time Message Processing Pipeline

TUI Mode (Terminal User Interface):

1. **User Input Capture**:

   ```
   Terminal keypress → Crossterm event → Ratatui input handler → Message validation
   ```

2. **Message Processing**:

   ```
   User message → Conversation context → GenAI chat request → Provider routing
   ```

3. **LLM Provider Interaction**:

   ```
   GenAI client → HTTP streaming request → Real-time response chunks → Channel␣
    ↪transmission
   ```

4. **Response Display**:

   ```
   Streaming chunks → UI update → Markdown rendering → Terminal display
   ```

Simple CLI Mode (NEW in v0.4.5):

1. **Input Processing**:

   ```
   stdin readline → Input validation → Command processing → LLM request
   ```

2. **Streaming Response**:

   ```
   LLM response chunks → Real-time stdout display → Session logging (optional)
   ```

3. **Session Management**:

   ```
   User input → Log timestamp → AI response → Log timestamp → File flush
   ```

Streaming Response Flow

The application uses Tokio channels for real-time streaming:

```rust
async fn message_flow_example() {
    // 1. User input received
    let user_message = "Explain quantum computing";
```

(continues on next page)

(continued from previous page)

```

    // 2. Create streaming channel
    let (tx, mut rx) = tokio::sync::mpsc::unbounded_channel::<String>();

    // 3. Start streaming task
    let config = app.config.clone();
    tokio::spawn(async move {
        send_message(&config, &user_message, tx).await
    });

    // 4. Process streaming updates in real-time
    while let Some(chunk) = rx.recv().await {
        app.append_to_current_response(chunk);
        app.trigger_ui_refresh();
    }
}
```

Command Processing Pipeline

Added in v0.4.3 - Built-in command system for productivity features:

1. **Command Detection**:

```
User input → Command prefix check ('/') → Command parsing → Action dispatch
```

2. **Save Command Flow**:

```
/save [filename] → Conversation validation → File generation → User feedback
```

3. **Command Implementation**:

```
impl App {
    pub fn handle_input(&mut self, input: String) -> Result<()> {
        if input.starts_with('/') {
            // Handle built-in commands
            self.process_command(input)?;
        } else {
            // Handle regular chat message
            self.send_user_message(input).await?;
        }
        Ok(())
    }

    fn process_command(&mut self, input: String) -> Result<()> {
        let parts: Vec<&str> = input.splitn(2, ' ').collect();
        match parts[0] {
            "/save" => {
                let filename = parts.get(1).map(|s| s.to_string());
                match self.save_conversation(filename) {
                    Ok(saved_filename) => {
                        self.add_system_message(
                            format!("□ Conversation saved to: {}", saved_filename)
                        );
                    }
```

(continues on next page)

(continued from previous page)

```
                    Err(e) => {
                        self.add_system_message(
                            format!("□ Error saving conversation: {}", e)
                        );
                    }
                }
            }
            _ => {
                self.add_system_message(
                    format!("□ Unknown command: {}", parts[0])
                );
            }
        }
        Ok(())
    }
}
```

Error Handling Strategy

Comprehensive Error Management

Perspt uses Rust's robust error handling with custom error types:

```
use anyhow::{Context, Result};
use thiserror::Error;

#[derive(Error, Debug)]
pub enum PersptError {
    #[error("Configuration error: {0}")]
    Config(#[from] ConfigError),

    #[error("LLM provider error: {0}")]
    Provider(#[from] genai::Error),

    #[error("UI error: {0}")]
    UI(#[from] std::io::Error),

    #[error("Network error: {0}")]
    Network(String),

    #[error("Streaming error: {0}")]
    Streaming(String),
}

// Graceful error recovery in main application loop
pub async fn handle_error_with_recovery(error: PersptError) -> bool {
    match error {
        PersptError::Network(_) => {
            // Show retry dialog, attempt reconnection
            show_retry_dialog();
            true // Continue running
        }
```

(continues on next page)

(continued from previous page)

```
        PersptError::Provider(_) => {
            // Try fallback provider if available
            attempt_provider_fallback();
            true
        }
        PersptError::UI(_) => {
            // Terminal issues - attempt recovery
            attempt_terminal_recovery();
            false // May need to exit
        }
        _ => {
            // Log error and continue
            log::error!("Application error: {}", error);
            true
        }
    }
}
```

Memory Management

Efficient Message Storage

Perspt manages conversation history efficiently in memory:

```
#[derive(Debug, Clone)]
pub struct Message {
    pub role: MessageRole,
    pub content: String,
    pub timestamp: std::time::SystemTime,
}

#[derive(Debug, Clone)]
pub enum MessageRole {
    User,
    Assistant,
    System,
}

impl Message {
    pub fn user(content: String) -> Self {
        Self {
            role: MessageRole::User,
            content,
            timestamp: std::time::SystemTime::now(),
        }
    }

    pub fn assistant(content: String) -> Self {
        Self {
            role: MessageRole::Assistant,
            content,
            timestamp: std::time::SystemTime::now(),
```

(continues on next page)

(continued from previous page)

```
        }
    }
}

// Conversation management with memory optimization
pub struct App {
    messages: Vec<Message>,
    max_history: usize,
    // ... other fields
}

impl App {
    pub fn add_message(&mut self, message: Message) {
        self.messages.push(message);

        // Limit memory usage by keeping only recent messages
        if self.messages.len() > self.max_history {
            self.messages.drain(0..self.messages.len() - self.max_history);
        }
    }

    /// Save current conversation to a text file
    /// Added in v0.4.3 for productivity workflows
    pub fn save_conversation(&self, filename: Option<String>) -> Result<String> {
        use std::fs;
        use std::time::{SystemTime, UNIX_EPOCH};

        // Filter out system messages for export
        let conversation_messages: Vec<_> = self.messages
            .iter()
            .filter(|msg| matches!(msg.role, MessageRole::User | MessageRole::Assistant))
            .collect();

        if conversation_messages.is_empty() {
            return Err(anyhow::anyhow!("No conversation to save"));
        }

        // Generate filename with timestamp if not provided
        let filename = filename.unwrap_or_else(|| {
            let timestamp = SystemTime::now()
                .duration_since(UNIX_EPOCH)
                .unwrap()
                .as_secs();
            format!("conversation_{}.txt", timestamp)
        });

        // Format conversation content
        let mut content = String::new();
        content.push_str("Perspt Conversation\n");
        content.push_str(&"=".repeat(18));
        content.push('\n');
```

(continues on next page)

(continued from previous page)

```
        for message in conversation_messages {
            let role = match message.role {
                MessageRole::User => "User",
                MessageRole::Assistant => "Assistant",
                MessageRole::System => continue, // Skip system messages
            };
            content.push_str(&format!("[{}] {}: {}\n\n",
                message.timestamp.format("%Y-%m-%d %H:%M:%S"),
                role,
                message.content
            ));
        }

        // Write to file
        fs::write(&filename, content)?;
        Ok(filename)
    }
}
```

Streaming Buffer Management

For streaming responses, Perspt uses efficient buffering:

```
impl App {
    pub fn append_to_current_response(&mut self, content: String) {
        if let Some(last_message) = self.messages.last_mut() {
            match last_message.role {
                MessageRole::Assistant => {
                    last_message.content.push_str(&content);
                }
                _ => {
                    // Create new assistant message if needed
                    self.add_message(Message::assistant(content));
                }
            }
        }
    }
}
```

Concurrency Model

Async Architecture with Tokio

Persppt uses Tokio for efficient asynchronous operations:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // Initialize panic handler
    setup_panic_hook();

    // Parse CLI arguments
    let args = Args::parse();
```

(continues on next page)

(continued from previous page)

```
    // Load configuration
    let config = Config::load()?;

    // Setup terminal
    enable_raw_mode()?;
    let mut stdout = io::stdout();
    execute!(stdout, EnterAlternateScreen, EnableMouseCapture)?;
    let backend = CrosstermBackend::new(stdout);
    let mut terminal = Terminal::new(backend)?;

    // Run main UI loop
    let result = run_ui(&mut terminal, config, args.model, args.api_key).await;

    // Cleanup
    disable_raw_mode()?;
    execute!(
        terminal.backend_mut(),
        LeaveAlternateScreen,
        DisableMouseCapture
    )?;
    terminal.show_cursor()?;

    result
}
```

Task Management

The application manages multiple concurrent tasks:

```
pub struct TaskManager {
    streaming_tasks: Vec<tokio::task::JoinHandle<()>>,
    ui_refresh_task: Option<tokio::task::JoinHandle<()>>,
}

impl App {
    pub async fn handle_user_input(&mut self, input: String) {
        // Spawn streaming task for LLM communication
        let config = self.config.clone();
        let tx = self.stream_sender.clone();

        let handle = tokio::spawn(async move {
            if let Err(e) = send_message(&config, &input, tx).await {
                log::error!("Streaming error: {}", e);
            }
        });

        self.task_manager.streaming_tasks.push(handle);

        // Cleanup completed tasks
        self.cleanup_completed_tasks();
    }
```

(continues on next page)

(continued from previous page)

```
    fn cleanup_completed_tasks(&mut self) {
        self.task_manager.streaming_tasks.retain(|handle| !handle.is_finished());
    }
}
```

Real-time Event Processing

The UI event loop handles multiple event sources concurrently:

```
pub async fn run_ui(
    terminal: &mut Terminal<CrosstermBackend<std::io::Stdout>>,
    config: Config,
    model_name: String,
    api_key: String,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut app = App::new(config, model_name, api_key);

    loop {
        // Render UI
        terminal.draw(|f| ui(f, &app))?;

        // Handle multiple event sources
        tokio::select! {
            // Terminal input events
            event = async {
                if event::poll(Duration::from_millis(50))? {
                    Some(event::read()?)
                } else {
                    None
                }
            } => {
                if let Some(Event::Key(key)) = event {
                    if app.handle_key_event(key).await? {
                        break;
                    }
                }
            }

            // Streaming content updates
            content = app.stream_receiver.recv() => {
                if let Some(content) = content {
                    app.append_to_current_response(content);
                }
            }

            // Periodic UI refresh
            _ = tokio::time::sleep(Duration::from_millis(16)) => {
                // 60 FPS refresh rate for smooth UI
            }
        }
    }
```

(continues on next page)

(continued from previous page)

```
    Ok(())
}
        let id = RequestId::new();
        let handle = tokio::spawn(async move {
            tokio::time::timeout(self.request_timeout, process_request(request)).await
        });
        self.active_requests.insert(id, handle);
        id
    }
}
```

Security Considerations

API Key Management

Persept handles API keys securely through environment variables and configuration:

```
impl Config {
    pub fn load() -> Result<Self, ConfigError> {
        // Try environment variable first (most secure)
        let api_key = env::var("OPENAI_API_KEY")
            .or_else(|_| env::var("ANTHROPIC_API_KEY"))
            .or_else(|_| env::var("GEMINI_API_KEY"))
            .or_else(|_| env::var("GROQ_API_KEY"))
            .or_else(|_| env::var("COHERE_API_KEY"))
            .or_else(|_| env::var("XAI_API_KEY"))
            .or_else(|_| env::var("DEEPSEEK_API_KEY"))
            .ok();

        // Load from config file as fallback
        let mut config = Self::load_from_file().unwrap_or_default();

        // Environment variables take precedence
        if let Some(key) = api_key {
            config.api_key = Some(key);
            config.provider = Self::infer_provider_from_key(&key);
        }

        Ok(config)
    }

    pub fn infer_provider_from_key(api_key: &str) -> String {
        match api_key {
            key if key.starts_with("sk-") => "openai".to_string(),
            key if key.starts_with("claude-") => "anthropic".to_string(),
            key if key.starts_with("AIza") => "gemini".to_string(),
            key if key.starts_with("gsk_") => "groq".to_string(),
            key if key.starts_with("xai-") => "xai".to_string(),
            key if key.starts_with("ds-") => "deepseek".to_string(),
            _ => "openai".to_string(), // Default fallback
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Input Validation and Sanitization

User input is validated before processing:

```
impl App {
    pub fn validate_user_input(&self, input: &str) -> Result<String, ValidationError> {
        // Check input length limits
        if input.len() > MAX_MESSAGE_LENGTH {
            return Err(ValidationError::TooLong);
        }

        // Remove control characters
        let sanitized = input
            .chars()
            .filter(|c| !c.is_control() || *c == '\n' || *c == '\t')
            .collect::<String>();

        // Trim whitespace
        let sanitized = sanitized.trim().to_string();

        if sanitized.is_empty() {
            return Err(ValidationError::Empty);
        }

        Ok(sanitized)
    }
}
```

Secure Error Handling

Error messages are sanitized to prevent information leakage:

```
pub fn sanitize_error_message(error: &dyn std::error::Error) -> String {
    match error.to_string() {
        msg if msg.contains("API key") => "Authentication error".to_string(),
        msg if msg.contains("token") => "Authentication error".to_string(),
        msg => {
            // Remove potentially sensitive information
            msg.lines()
                .filter(|line| !line.contains("Bearer") && !line.contains("Authorization"))
                .collect::<Vec<_>>()
                .join("\n")
        }
    }
}
```

Testing Architecture

Unit Testing Strategy

Persipt includes comprehensive unit tests for each module:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_config_loading() {
        let config = Config::load().unwrap();
        assert!(!config.provider.is_empty());
    }

    #[test]
    fn test_provider_inference() {
        assert_eq!(Config::infer_provider_from_key("sk-test"), "openai");
        assert_eq!(Config::infer_provider_from_key("claude-test"), "anthropic");
        assert_eq!(Config::infer_provider_from_key("AIza-test"), "gemini");
        assert_eq!(Config::infer_provider_from_key("gsk_test"), "groq");
        assert_eq!(Config::infer_provider_from_key("xai-test"), "xai");
        assert_eq!(Config::infer_provider_from_key("ds-test"), "deepseek");
    }

    #[test]
    fn test_message_creation() {
        let msg = Message::user("Hello".to_string());
        assert!(matches!(msg.role, MessageRole::User));
        assert_eq!(msg.content, "Hello");
    }

    #[test]
    fn test_input_validation() {
        let app = App::default();

        // Valid input
        assert!(app.validate_user_input("Hello world").is_ok());

        // Empty input
        assert!(app.validate_user_input("").is_err());

        // Too long input
        let long_input = "a".repeat(10000);
        assert!(app.validate_user_input(&long_input).is_err());
    }
}
```

Integration Testing

Integration tests verify the complete application flow:

```rust
// tests/integration_tests.rs
use perspt::*;
use std::env;

#[tokio::test]
async fn test_full_conversation_flow() {
    // Skip if no API key available
    if env::var("OPENAI_API_KEY").is_err() {
        return;
    }

    let config = Config {
        provider: "openai".to_string(),
        api_key: env::var("OPENAI_API_KEY").ok(),
        model: Some("gpt-3.5-turbo".to_string()),
        temperature: Some(0.7),
        max_tokens: Some(100),
        timeout_seconds: Some(30),
    };

    let (tx, mut rx) = tokio::sync::mpsc::unbounded_channel();

    // Test streaming response
    let result = send_message(&config, "Hello, how are you?", tx).await;
    assert!(result.is_ok());

    // Verify we receive streaming content
    let mut received_content = String::new();
    while let Ok(content) = rx.try_recv() {
        received_content.push_str(&content);
    }
    assert!(!received_content.is_empty());
}

#[test]
fn test_config_loading_hierarchy() {
    // Test config loading from different sources
    let config = Config::load().unwrap();
    assert!(!config.provider.is_empty());
}
```

Performance Considerations

Optimization Strategies

Perspt is optimized for performance through several key strategies:

1. **Streaming Responses**: Immediate display of LLM responses as they arrive
2. **Efficient Memory Management**: Limited conversation history with automatic cleanup
3. **Async/Await Architecture**: Non-blocking operations with Tokio
4. **Minimal Dependencies**: Fast compilation and small binary size
5. **Zero-Copy Operations**: Efficient string handling where possible

Real-time Performance Metrics:

```rust
impl App {
    pub fn get_performance_stats(&self) -> PerformanceStats {
        PerformanceStats {
            messages_per_second: self.calculate_message_rate(),
            memory_usage_mb: self.get_memory_usage(),
            ui_refresh_rate: 60.0, // Target 60 FPS
            streaming_latency_ms: self.get_average_streaming_latency(),
        }
    }

    fn calculate_message_rate(&self) -> f64 {
        let recent_messages = self.messages.iter()
            .filter(|m| m.timestamp.elapsed().unwrap().as_secs() < 60)
            .count();
        recent_messages as f64 / 60.0
    }
}
```

Memory Optimization

```rust
const MAX_HISTORY_MESSAGES: usize = 100;
const MAX_MESSAGE_LENGTH: usize = 8192;

impl App {
    pub fn optimize_memory(&mut self) {
        // Remove old messages if exceeding limit
        if self.messages.len() > MAX_HISTORY_MESSAGES {
            let keep_from = self.messages.len() - MAX_HISTORY_MESSAGES;
            self.messages.drain(0..keep_from);
        }

        // Compact long messages
        for message in &mut self.messages {
            if message.content.len() > MAX_MESSAGE_LENGTH {
                message.content.truncate(MAX_MESSAGE_LENGTH);
                message.content.push_str("... [truncated]");
            }
        }
    }
}
```

Future Architecture Considerations

Planned Enhancements

Based on the current GenAI-powered architecture, future enhancements include:

1. **Multi-Provider Streaming**: Simultaneous requests to multiple providers with fastest response wins
2. **Enhanced Conversation Context**: Intelligent context window management for long conversations
3. **Plugin Architecture**: Extensible plugin system for custom commands and integrations
4. **Advanced UI Components**: Rich markdown rendering, syntax highlighting, and interactive elements
5. **Offline Mode**: Local model support for privacy-sensitive scenarios

Implementation Roadmap:

```
// Future: Multi-provider streaming
pub async fn stream_from_multiple_providers(
    providers: &[String],
    message: &str,
) -> Result<impl Stream<Item = String>, Error> {
    let streams = providers.iter().map(|provider| {
        let config = Config::for_provider(provider);
        send_message_stream(&config, message)
    });

    // Return the fastest responding stream
    futures::stream::select_all(streams)
}

// Future: Plugin system
pub trait Plugin: Send + Sync {
    async fn execute(&self, command: &str, args: &[String]) -> PluginResult;
    fn commands(&self) -> Vec<String>;
}
```

Migration Strategies

For evolutionary architecture changes:

1. **GenAI Provider Expansion**: Easy addition of new providers through the genai crate
2. **Configuration Evolution**: Backward-compatible config format changes
3. **UI Component Modularity**: Incremental UI improvements without breaking changes
4. **Streaming Protocol Evolution**: Enhanced streaming with metadata and typing indicators

Next Steps

For developers looking to contribute or extend Perspt:

- *Contributing* - Contribution guidelines and development setup
- *Extending Perspt* - Creating custom providers and plugins
- *Testing* - Testing strategies and guidelines
- *API Reference* - API reference and integration guides

The architecture is designed to be extensible and maintainable, making it easy to add new features while preserving the core performance and reliability characteristics.

cli.rs

NEW in v0.4.5 - Minimal command-line interface for Unix-style interactions.

Responsibilities:

- Unix-style prompt interface (>) for direct Q&A
- Session logging with timestamped conversations
- Scriptable interface for automation and workflows
- Accessibility-friendly text-only output
- Integration with existing provider and configuration systems

Key Functions:

```
pub async fn run_simple_cli(
    config: AppConfig,
    model_name: String,
    api_key: String,
    provider: Arc<GenAIProvider>,
    log_file: Option<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut session_log = if let Some(log_path) = log_file {
        Some(SessionLogger::new(log_path)?)
    } else {
        None
    };

    println!("Perspt Simple CLI Mode");
    println!("Model: {}", model_name);
    println!("Type 'exit' or press Ctrl+D to quit.\n");

    loop {
        // Display prompt
        print!("> ");
        io::stdout().flush()?;

        // Read user input
        let mut input = String::new();
        match io::stdin().read_line(&mut input) {
            Ok(0) => break, // EOF (Ctrl+D)
            Ok(_) => {
                let input = input.trim();
                if input.is_empty() { continue; }
                if input == "exit" { break; }

                // Log user input
                if let Some(ref mut logger) = session_log {
                    logger.log_user_input(input)?;
                }

                // Process with LLM
                match process_simple_request(input, &model_name, &provider).await {
                    Ok(response) => {
                        println!("{}", response);
                        if let Some(ref mut logger) = session_log {
                            logger.log_ai_response(&response)?;
                        }
                    }
                    Err(e) => {
                        eprintln!("Error: {}", e);
                    }
                }
                println!(); // Add spacing between exchanges
            }
            Err(e) => {
                eprintln!("Input error: {}", e);
                break;
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }

    println!("Goodbye!");
    Ok(())
}

async fn process_simple_request(
    input: &str,
    model: &str,
    provider: &GenAIProvider,
) -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    let (tx, mut rx) = tokio::sync::mpsc::unbounded_channel::<String>();

    // Start streaming request
    provider.generate_response_stream_to_channel(model, input, tx).await?;

    // Collect streaming response
    let mut full_response = String::new();
    while let Some(chunk) = rx.recv().await {
        if chunk == "<<EOT>>" { break; }
        print!("{}", chunk);
        io::stdout().flush()?;
        full_response.push_str(&chunk);
    }

    Ok(full_response)
}
```

Session Logging Implementation:

```
struct SessionLogger {
    file: File,
}

impl SessionLogger {
    pub fn new(log_path: String) -> Result<Self, std::io::Error> {
        let file = OpenOptions::new()
            .create(true)
            .append(true)
            .open(log_path)?;
        Ok(Self { file })
    }

    pub fn log_user_input(&mut self, input: &str) -> Result<(), std::io::Error> {
        let timestamp = chrono::Local::now().format("%Y-%m-%d %H:%M:%S");
        writeln!(self.file, "[{}] User: {}", timestamp, input)?;
        self.file.flush()?;
        Ok(())
    }

```

(continues on next page)

(continued from previous page)

```
    pub fn log_ai_response(&mut self, response: &str) -> Result<(), std::io::Error> {
        let timestamp = chrono::Local::now().format("%Y-%m-%d %H:%M:%S");
        writeln!(self.file, "[{}] Assistant: {}", timestamp, response)?;
        writeln!(self.file)?; // Add spacing
        self.file.flush()?;
        Ok(())
    }
}
```

Design Rationale:

- **Unix Philosophy**: Simple, composable tool that follows Unix conventions
- **Streaming Support**: Real-time response display using the same streaming infrastructure as TUI mode
- **Scriptable**: Perfect for automation, shell integration, and batch processing
- **Accessibility**: Text-only output that works well with screen readers and accessibility tools
- **Session Logging**: Built-in conversation logging for documentation and audit trails

Usage Patterns:

```
# Basic simple CLI mode
perspt --simple-cli

# With session logging
perspt --simple-cli --log-file session.txt

# Scripting integration
echo "What is quantum computing?" | perspt --simple-cli

# Chained queries
{
  echo "What is machine learning?"
  echo "Give me 3 examples"
  echo "exit"
} | perspt --simple-cli --log-file ml-explanation.txt
```

3.6.2 Contributing

Welcome to the Perspt project! This guide will help you get started with contributing to Perspt, whether you're fixing bugs, adding features, or improving documentation.

Getting Started

Prerequisites

Before contributing, ensure you have:

- **Rust** (latest stable version)
- **Git** for version control
- **A GitHub account** for pull requests
- **Code editor** with Rust support (VS Code with rust-analyzer recommended)

Development Environment Setup

1. **Fork and Clone**:

```
# Fork the repository on GitHub, then:
git clone https://github.com/YOUR_USERNAME/perspt.git
cd perspt
```

2. **Set up the development environment**:

```
# Install Rust if not already installed
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# Install additional components
rustup component add clippy rustfmt

# Install development dependencies (optional but recommended)
cargo install cargo-watch cargo-nextest
```

3. **Set up API keys for testing**:

```
# Copy example config
cp config.json.example config.json

# Edit config.json with your API keys (optional for basic development)
# Or set environment variables:
export OPENAI_API_KEY="your-key-here"
export ANTHROPIC_API_KEY="your-key-here"
```

4. **Verify the setup**:

```
# Build the project
cargo build

# Run tests (some may be skipped without API keys)
cargo test

# Check formatting and linting
cargo fmt --check
cargo clippy -- -D warnings

# Test the application
cargo run -- "Hello, can you help me?"
```

Development Workflow

Branch Strategy

We follow a simplified Git flow:

- **main**: Stable, production-ready code
- **develop**: Integration branch for new features
- **feature/**: Feature development branches
- **fix/**: Bug fix branches
- **docs/**: Documentation improvement branches

Creating a Feature Branch

```
# Ensure you're on the latest develop branch
git checkout develop
git pull origin develop

# Create a new feature branch
git checkout -b feature/your-feature-name

# Make your changes
# ...

# Commit your changes
git add .
git commit -m "feat: add your feature description"

# Push to your fork
git push origin feature/your-feature-name
```

Code Style and Standards

Rust Style Guide

We follow the official Rust style guide with these additions:

Formatting:

```
# Auto-format your code
cargo fmt
```

Linting:

```
# Check for common issues
cargo clippy -- -D warnings
```

Documentation:

```
/// Brief description of the function.
///
/// More detailed explanation if needed.
///
/// # Arguments
///
/// * `param1` - Description of parameter
/// * `param2` - Description of parameter
///
/// # Returns
///
/// Description of return value
///
/// # Errors
///
/// Description of possible errors
///
```

(continues on next page)

(continued from previous page)

```
/// # Examples
///
/// ```
/// let result = function_name(arg1, arg2);
/// assert_eq!(result, expected);
/// ```
pub fn function_name(param1: Type1, param2: Type2) -> Result<ReturnType, Error> {
    // Implementation
}
```

Naming Conventions

- **Functions and variables**: *snake_case*
- **Types and traits**: *PascalCase*
- **Constants**: *SCREAMING_SNAKE_CASE*
- **Modules**: *snake_case*

```
// Good
pub struct LlmProvider;
pub trait ConfigManager;
pub fn process_message() -> Result<String, Error>;
pub const DEFAULT_TIMEOUT: Duration = Duration::from_secs(30);

// Avoid
pub struct llmProvider;
pub trait configManager;
pub fn ProcessMessage() -> Result<String, Error>;
```

Error Handling

Use the *thiserror* crate for error definitions:

```
use thiserror::Error;

#[derive(Error, Debug)]
pub enum ConfigError {
    #[error("Configuration file not found: {path}")]
    FileNotFound { path: String },

    #[error("Invalid configuration: {reason}")]
    Invalid { reason: String },

    #[error("IO error: {0}")]
    Io(#[from] std::io::Error),
}
```

Testing Guidelines

Test Structure

Organize tests in the same file as the code they test:

```
pub struct MessageProcessor {
    config: Config,
}

impl MessageProcessor {
    pub fn new(config: Config) -> Self {
        Self { config }
    }

    pub async fn process(&self, input: &str) -> Result<String, ProcessError> {
        // Implementation using GenAI crate
        validate_message(input)?;
        let response = send_message(&self.config, input, tx).await?;
        Ok(response)
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use tokio::sync::mpsc;

    #[test]
    fn test_message_validation() {
        let processor = MessageProcessor::new(Config::default());
        assert!(processor.validate_message("valid message").is_ok());
        assert!(processor.validate_message("").is_err());
    }

    #[tokio::test]
    async fn test_async_processing() {
        // Skip if no API key available
        if std::env::var("OPENAI_API_KEY").is_err() {
            return;
        }

        let config = Config {
            provider: "openai".to_string(),
            api_key: std::env::var("OPENAI_API_KEY").ok(),
            model: Some("gpt-3.5-turbo".to_string()),
            ..Default::default()
        };

        let (tx, mut rx) = mpsc::unbounded_channel();
        let result = send_message(&config, "test", tx).await;
        assert!(result.is_ok());
    }
}
```

Integration Tests

Place integration tests in the *tests/* directory:

```rust
// tests/integration_test.rs
use perspt::config::Config;
use perspt::llm_provider::send_message;
use std::env;
use tokio::sync::mpsc;

#[tokio::test]
async fn test_full_conversation_flow() {
    // Skip if no API keys available
    if env::var("OPENAI_API_KEY").is_err() {
        return;
    }

    let config = Config {
        provider: "openai".to_string(),
        api_key: env::var("OPENAI_API_KEY").ok(),
        model: Some("gpt-3.5-turbo".to_string()),
        temperature: Some(0.7),
        max_tokens: Some(100),
        timeout_seconds: Some(30),
    };

    let (tx, mut rx) = mpsc::unbounded_channel();

    // Test streaming response
    let result = send_message(&config, "Hello, how are you?", tx).await;
    assert!(result.is_ok());

    // Verify we receive streaming content
    let mut received_content = String::new();
    while let Ok(content) = rx.try_recv() {
        received_content.push_str(&content);
    }
    assert!(!received_content.is_empty());
}

#[test]
fn test_config_loading_hierarchy() {
    // Test config loading from different sources
    let config = Config::load();
    assert!(config.is_ok());
}
```

Test Categories

We have several categories of tests:

1. **Unit Tests**: Test individual functions and methods

```
# Run only unit tests
cargo test --lib
```

2. **Integration Tests**: Test module interactions

```
# Run integration tests
cargo test --test '*'
```

3. **API Tests**: Test against real APIs (require API keys)

```
# Run with API keys set
OPENAI_API_KEY=xxx ANTHROPIC_API_KEY=yyy cargo test
```

4. **UI Tests**: Test terminal UI components

```
# Run UI tests (may require TTY)
cargo test ui::tests
```

Test Utilities

Use these utilities for consistent testing:

```
// Test configuration helper
impl Config {
    pub fn test_config() -> Self {
        Config {
            provider: "test".to_string(),
            api_key: Some("test-key".to_string()),
            model: Some("test-model".to_string()),
            temperature: Some(0.7),
            max_tokens: Some(100),
            timeout_seconds: Some(30),
        }
    }
}

// Mock message sender for testing
pub async fn mock_send_message(
    _config: &Config,
    message: &str,
    tx: tokio::sync::mpsc::UnboundedSender<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    tx.send(format!("Mock response to: {}", message))?;
    Ok(())
}
```

Running Tests

```
# Run all tests
cargo test

# Run tests with output
cargo test -- --nocapture

```

(continues on next page)

(continued from previous page)

```
# Run specific test
cargo test test_name

# Run tests with coverage (requires cargo-tarpaulin)
cargo install cargo-tarpaulin
cargo tarpaulin --out Html
```

Pull Request Process

Before Submitting

1. **Ensure tests pass**:

   ```
   cargo test
   cargo clippy -- -D warnings
   cargo fmt --check
   ```

2. **Update documentation** if needed
3. **Add tests** for new functionality
4. **Update changelog** if applicable

PR Description Template

When creating a pull request, use this template:

```
## Description
Brief description of changes made.

## Type of Change
- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing functionality to not work␣
↪as expected)
- [ ] Documentation update

## Testing
- [ ] Unit tests added/updated
- [ ] Integration tests added/updated
- [ ] Manual testing performed

## Checklist
- [ ] Code follows the project's style guidelines
- [ ] Self-review completed
- [ ] Comments added to hard-to-understand areas
- [ ] Documentation updated
- [ ] No new warnings introduced
```

Review Process

1. **Automated checks** must pass (CI/CD pipeline)
2. **Code review** by at least one maintainer
3. **Testing** in development environment
4. **Final approval** and merge

Areas for Contribution

Good First Issues

Look for issues labeled *good first issue*:

- Documentation improvements and typo fixes
- Configuration validation enhancements
- Error message improvements
- Test coverage improvements
- Code formatting and cleanup
- Example configurations for new providers

Feature Development

Major areas where contributions are welcome:

New AI Provider Support:

```
// Add support for new providers in llm_provider.rs
pub async fn send_message_custom_provider(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // Use the GenAI crate to add new provider support
    let client = genai::Client::builder()
        .with_api_key(&config.api_key.unwrap_or_default())
        .build()?;

    let chat_req = genai::chat::ChatRequest::new(vec![
        genai::chat::ChatMessage::user(message)
    ]);

    let stream = client.exec_stream(&config.model.clone().unwrap_or_default(), chat_req).
↪await?;

    // Handle streaming response
    // Implementation details...

    Ok(())
}
```

UI Component Enhancements:

```
// Add new Ratatui components in ui.rs
pub struct CustomWidget {
    content: String,
    scroll_offset: u16,
}

impl CustomWidget {
    pub fn render(&self, area: Rect, buf: &mut Buffer) {
        let block = Block::default()
            .borders(Borders::ALL)
```

(continues on next page)

(continued from previous page)

```
            .title("Custom Feature");

        let inner = block.inner(area);
        block.render(area, buf);

        // Custom rendering logic using Ratatui
        self.render_content(inner, buf);
    }
}
```

Configuration System Extensions:

```rust
// Extend Config struct in config.rs
#[derive(Debug, Deserialize, Serialize, Clone)]
pub struct ExtendedConfig {
    #[serde(flatten)]
    pub base: Config,

    // New configuration options
    pub custom_endpoints: Option<HashMap<String, String>>,
    pub retry_config: Option<RetryConfig>,
    pub logging_config: Option<LoggingConfig>,
}
```

Performance and Reliability:

- Streaming response optimizations
- Better error handling and recovery
- Configuration validation improvements
- Memory usage optimizations for large conversations
- Connection pooling and retry logic

Developer Experience:

- Better debugging tools and logging
- Enhanced error messages with suggestions
- Configuration validation with helpful feedback
- Developer-friendly CLI options

Bug Reports and Issues

Filing Bug Reports

When filing a bug report, include:

1. **Clear description** of the issue
2. **Steps to reproduce** the problem
3. **Expected behavior** vs actual behavior
4. **Environment information**:

   ```
   - OS: [e.g., macOS 12.0, Ubuntu 20.04]
   - Perspt version: [e.g., 1.0.0]
   - Rust version: [e.g., 1.70.0]
   - Provider: [e.g., OpenAI GPT-4]
   ```

5. **Configuration** (sanitized):

```
{
  "provider": "openai",
  "model": "gpt-4",
  "api_key": "[REDACTED]"
}
```

6. **Error messages** (full text)
7. **Log files** if available

Feature Requests

For feature requests, provide:

1. **Clear description** of the desired feature
2. **Use case** and motivation
3. **Proposed implementation** (if you have ideas)
4. **Alternatives considered**
5. **Additional context** or examples

Documentation Contributions

Types of Documentation

- **API documentation**: Rust doc comments in source code
- **Developer guides**: Sphinx documentation in *docs/perspt_book/*
- **README**: Project overview and quick start
- **Examples**: Sample configurations and use cases
- **Changelog**: Version history and migration guides

Documentation Standards

- Use clear, concise language
- Include working code examples that match current implementation
- Keep examples up-to-date with current API and dependencies
- Cross-reference related sections using Sphinx references
- Follow reStructuredText formatting for Sphinx docs

Building Documentation

Rust API Documentation:

```
# Generate and open Rust documentation
cargo doc --open --no-deps --all-features
```

Sphinx Documentation:

```
# Build HTML documentation
cd docs/perspt_book
uv run make html

# Build PDF documentation
uv run make latexpdf

# Clean and rebuild everything
uv run make clean && uv run make html && uv run make latexpdf
```

Watch Mode for Development:

```
# Auto-rebuild on changes
cd docs/perspt_book
uv run sphinx-autobuild source build/html
```

Available VS Code Tasks:

You can also use the VS Code tasks for documentation:

- "Build Sphinx HTML Documentation"
- "Build Sphinx PDF Documentation"
- "Watch and Auto-build HTML Documentation"
- "Open Sphinx HTML Documentation"
- "Validate Documentation Links"

Writing Documentation

Code Examples: Ensure all code examples compile and work:

```rust
// Good: Complete, working example
use perspt::config::Config;
use tokio::sync::mpsc;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let config = Config::load()?;
    let (tx, mut rx) = mpsc::unbounded_channel();

    perspt::llm_provider::send_message(&config, "Hello", tx).await?;

    while let Some(response) = rx.recv().await {
        println!("{}", response);
    }

    Ok(())
}
```

Configuration Examples: Use realistic, sanitized configs:

```json
{
  "provider": "openai",
  "api_key": "${OPENAI_API_KEY}",
  "model": "gpt-4",
  "temperature": 0.7,
  "max_tokens": 2000,
  "timeout_seconds": 30
}
```

Community Guidelines

Code of Conduct

We follow the Rust Code of Conduct. In summary:

- Be friendly and patient

- Be welcoming
- Be considerate
- Be respectful
- Be careful in word choice
- When we disagree, try to understand why

Communication Channels

- **GitHub Issues**: Bug reports and feature requests
- **GitHub Discussions**: General questions and ideas
- **Discord/Slack**: Real-time community chat
- **Email**: Direct contact with maintainers

Recognition

Contributors are recognized in:

- **CONTRIBUTORS.md**: List of all contributors
- **Release notes**: Major contributions highlighted
- **Documentation**: Author attribution where appropriate
- **Community highlights**: Regular contributor spotlights

Release Process

Version Numbering

We follow Semantic Versioning (SemVer):

- **MAJOR**: Breaking changes
- **MINOR**: New features (backward compatible)
- **PATCH**: Bug fixes (backward compatible)

Release Cycle

- **Major releases**: Every 6-12 months
- **Minor releases**: Every 1-3 months
- **Patch releases**: As needed for critical fixes

Next Steps

See the following documentation for more detailed information:

- *Architecture* - Understanding Perspt's internal architecture
- *Extending Perspt* - How to extend Perspt with new features
- *Testing* - Testing strategies and best practices
- *User Guide* - User guide for understanding the application
- *API Reference* - API reference documentation

Development Workflow Tips

Using VS Code Tasks

The project includes several VS Code tasks for common development activities:

```
# Available tasks (use Ctrl+Shift+P -> "Tasks: Run Task"):
- "Generate Documentation" (cargo doc)
- "Build Sphinx HTML Documentation"
- "Build Sphinx PDF Documentation"
- "Watch and Auto-build HTML Documentation"
- "Clean and Build All Documentation"
- "Validate Documentation Links"
```

Hot Reloading During Development

For faster development cycles:

```
# Watch for changes and rebuild
cargo install cargo-watch
cargo watch -x 'build'

# Watch and run tests
cargo watch -x 'test'

# Watch and run with sample input
cargo watch -x 'run -- "test message"'
```

Debugging

Enable Debug Logging:

```
# Set environment variable for detailed logs
export RUST_LOG=debug
cargo run -- "your message"
```

Debug Streaming Issues:

The project includes debug scripts:

```
# Debug long responses and streaming
./debug-long-response.sh
```

Use Rust Debugger:

```
// Add debug prints in your code
eprintln!("Debug: config = {:?}", config);

// Use dbg! macro for quick debugging
let result = dbg!(some_function());
```

Project Structure Understanding

Key files and their purposes:

- `src/main.rs`: CLI entry point, panic handling, terminal setup
- `src/config.rs`: Configuration loading and validation
- `src/llm_provider.rs`: GenAI integration and streaming
- `src/ui.rs`: Ratatui terminal UI components
- `Cargo.toml`: Dependencies and project metadata

- `config.json.example`: Sample configuration file
- `docs/perspt_book/`: Sphinx documentation source
- `tests/`: Integration tests
- `validate-docs.sh`: Documentation validation script

Common Development Patterns

Error Handling Pattern:

```rust
use anyhow::{Context, Result};
use thiserror::Error;

#[derive(Error, Debug)]
pub enum MyError {
    #[error("Configuration error: {0}")]
    Config(String),
    #[error("Network error")]
    Network(#[from] reqwest::Error),
}

pub fn example_function() -> Result<String> {
    let config = load_config()
        .context("Failed to load configuration")?;

    process_config(&config)
        .context("Failed to process configuration")
}
```

Async/Await Pattern:

```rust
use tokio::sync::mpsc::{UnboundedSender, UnboundedReceiver};

pub async fn stream_handler(
    mut rx: UnboundedReceiver<String>,
    tx: UnboundedSender<String>,
) -> Result<()> {
    while let Some(message) = rx.recv().await {
        let processed = process_message(&message).await?;
        tx.send(processed).context("Failed to send processed message")?;
    }
    Ok(())
}
```

Configuration Pattern:

```rust
use serde::{Deserialize, Serialize};

#[derive(Debug, Deserialize, Serialize, Clone)]
pub struct ModuleConfig {
    pub enabled: bool,
    pub timeout: Option<u64>,
    #[serde(default)]
    pub advanced_options: AdvancedOptions,
}
```

(continues on next page)

(continued from previous page)

```
impl Default for ModuleConfig {
    fn default() -> Self {
        Self {
            enabled: true,
            timeout: Some(30),
            advanced_options: AdvancedOptions::default(),
        }
    }
}
```

Dependency Management

Adding New Dependencies:

```
# Add a new dependency
cargo add serde --features derive

# Add a development dependency
cargo add --dev mockall

# Add an optional dependency
cargo add optional-dep --optional
```

Dependency Guidelines:

1. **Minimize dependencies**: Only add what's necessary
2. **Use well-maintained crates**: Check recent updates and issues
3. **Consider security**: Use `cargo audit` to check for vulnerabilities
4. **Version pinning**: Be specific about versions in Cargo.toml

```
# Good: Specific versions
serde = { version = "1.0.196", features = ["derive"] }
tokio = { version = "1.36.0", features = ["full"] }

# Avoid: Wildcard versions
serde = "*"
```

Security Auditing:

```
# Install cargo-audit
cargo install cargo-audit

# Run security audit
cargo audit

# Update advisories database
cargo audit --update
```

Release Process

Version Bumping:

```
# Update version in Cargo.toml
# Update CHANGELOG.md with changes
# Create release notes

# Tag the release
git tag -a v1.2.0 -m "Release version 1.2.0"
git push origin v1.2.0
```

Pre-release Checklist:

1. All tests pass: `cargo test`
2. Documentation builds: `cargo doc`
3. No clippy warnings: `cargo clippy -- -D warnings`
4. Code formatted: `cargo fmt --check`
5. CHANGELOG.md updated
6. Version bumped in Cargo.toml
7. Security audit clean: `cargo audit`

Release Notes Template:

```
## Version X.Y.Z - YYYY-MM-DD

### Added
- New features and enhancements

### Changed
- Breaking changes and modifications

### Fixed
- Bug fixes and issue resolutions

### Security
- Security-related changes

### Dependencies
- Updated dependencies
```

Performance Profiling

CPU Profiling:

```
# Install profiling tools
cargo install cargo-flamegraph

# Profile your application
cargo flamegraph --bin perspt -- "test message"
```

Memory Profiling:

```
# Use valgrind (Linux/macOS)
cargo build
valgrind --tool=massif target/debug/perspt "test message"
```

Benchmarking:

```
// Add to benches/benchmark.rs
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn benchmark_message_processing(c: &mut Criterion) {
    c.bench_function("process_message", |b| {
        b.iter(|| {
            let result = process_message(black_box("test input"));
            result
        })
    });
}

criterion_group!(benches, benchmark_message_processing);
criterion_main!(benches);
```

Troubleshooting Common Issues

Build Failures:

```
# Clean build artifacts
cargo clean

# Update toolchain
rustup update

# Rebuild dependencies
cargo build
```

Test Failures:

```
# Run tests with output
cargo test -- --nocapture

# Run a specific test
cargo test test_name -- --exact

# Run ignored tests
cargo test -- --ignored
```

API Key Issues:

```
# Check environment variables
env | grep -i api

# Verify config file
cat ~/.config/perspt/config.json

```

(continues on next page)

(continued from previous page)

```
# Test with explicit config
echo '{"provider":"openai","api_key":"test"}' | cargo run
```

Documentation Build Issues:

```
# Check Python/uv installation
uv --version

# Reinstall dependencies
cd docs/perspt_book
uv sync

# Clean and rebuild
uv run make clean && uv run make html
```

Getting Help

If you encounter issues or need guidance:

1. **Check existing issues** on GitHub
2. **Search the documentation** for similar problems
3. **Ask in discussions** for general questions
4. **Create a detailed issue** for bugs or feature requests
5. **Join the community** chat for real-time help

When asking for help, include:

- Your operating system and version
- Rust version (`rustc --version`)
- Perspt version or commit hash
- Full error messages
- Steps to reproduce the issue
- Your configuration (sanitized)

Final Notes

Code Quality:

- Write self-documenting code with clear variable names
- Add comments for complex logic
- Keep functions small and focused
- Use meaningful error messages
- Follow Rust idioms and best practices

Testing Philosophy:

- Test behavior, not implementation
- Write tests before fixing bugs (TDD when possible)
- Cover edge cases and error conditions
- Use descriptive test names
- Keep tests fast and reliable

Documentation Philosophy:

- Document the “why”, not just the “what”
- Keep examples current and working
- Use real-world scenarios in examples

- Cross-reference related concepts
- Update docs with code changes

Ready to contribute? Here's your next steps:

1. **Fork the repository** and set up your environment
2. **Find an issue** to work on or propose a new feature
3. **Read the codebase** to understand the current patterns
4. **Start small** with documentation or simple fixes
5. **Ask questions** early and often
6. **Submit your PR** with tests and documentation

Welcome to the Perspt development community! 🎉

Contributing to Simple CLI Mode

NEW in v0.4.6 - The Simple CLI mode offers several areas for contribution:

Areas for Enhancement:

1. **Command Extensions**: - Additional built-in commands (`/history`, `/config`, `/provider`) - Command auto-completion - Command history navigation
2. **Session Management**: - Enhanced logging formats (JSON, CSV, Markdown) - Session resumption capabilities - Multi-session management
3. **Accessibility Improvements**: - Screen reader optimizations - High contrast mode support - Keyboard navigation enhancements
4. **Scripting Integration**: - Better shell integration - Environment variable templating - Batch processing improvements

Development Guidelines for CLI Features:

```rust
// Example: Adding a new CLI command
// In src/cli.rs

async fn process_cli_command(
    command: &str,
    session_log: &mut Option<SessionLogger>,
    app_state: &mut CliAppState, // New state management
) -> Result<bool, Box<dyn std::error::Error + Send + Sync>> {
    let parts: Vec<&str> = command.splitn(2, ' ').collect();

    match parts[0] {
        // Existing commands...

        "/history" => {
            // NEW: Show conversation history
            display_conversation_history(&app_state.conversation_history);
            Ok(true)
        }
        "/config" => {
            // NEW: Show current configuration
            display_current_config(&app_state.config);
            Ok(true)
        }
        "/provider" => {
            // NEW: Switch providers dynamically
```

(continues on next page)

(continued from previous page)

```
            if let Some(provider_name) = parts.get(1) {
                switch_provider(provider_name, app_state).await?;
                println!("Switched to provider: {}", provider_name);
            } else {
                println!("Current provider: {}", app_state.current_provider());
            }
            Ok(true)
        }
        _ => {
            println!("Unknown command: {}. Type /help for available commands.", parts[0]);
            Ok(true)
        }
    }
}

// State management for CLI mode
pub struct CliAppState {
    pub config: AppConfig,
    pub conversation_history: Vec<ConversationEntry>,
    pub current_provider: String,
    pub session_stats: SessionStats,
}

impl CliAppState {
    pub fn new(config: AppConfig) -> Self {
        Self {
            current_provider: config.provider_type.clone().unwrap_or_default(),
            config,
            conversation_history: Vec::new(),
            session_stats: SessionStats::new(),
        }
    }

    pub fn add_conversation_entry(&mut self, user_input: String, ai_response: String) {
        self.conversation_history.push(ConversationEntry {
            timestamp: SystemTime::now(),
            user_input,
            ai_response,
        });
        self.session_stats.increment_exchange_count();
    }
}
```

Testing Requirements for CLI Contributions:

```
// All CLI contributions should include tests
#[cfg(test)]
mod cli_contribution_tests {
    use super::*;

    #[test]
    fn test_new_command_parsing() {
```

(continues on next page)

(continued from previous page)

```
        // Test new command syntax
        assert!(matches!(parse_command("/newcommand arg"), Ok(Command::NewCommand(_))));
    }

    #[tokio::test]
    async fn test_new_command_execution() {
        let mut app_state = CliAppState::new(test_config());
        let result = process_cli_command("/newcommand test", &mut None, &mut app_state).
↪await;
        assert!(result.is_ok());
    }

    #[test]
    fn test_accessibility_compliance() {
        // Ensure new features maintain accessibility
        let output = execute_command_with_screen_reader_simulation("/newcommand");
        assert!(is_screen_reader_friendly(&output));
    }
}
```

Code Style for CLI Contributions:

```
// Follow these patterns for CLI code

// 1. Clear error messages with helpful context
fn handle_cli_error(error: &dyn std::error::Error) -> String {
    match error.to_string().as_str() {
        msg if msg.contains("network") => {
            "□ Network error. Check your internet connection and try again.".to_string()
        }
        msg if msg.contains("api key") => {
            "□ API key error. Verify your API key is set correctly.".to_string()
        }
        _ => format!("□ Error: {}", error),
    }
}

// 2. Consistent prompt and output formatting
fn display_cli_prompt(provider: &str, model: &str) {
    println!("Perspt Simple CLI Mode");
    println!("Provider: {} | Model: {}", provider, model);
    println!("Type 'exit' or press Ctrl+D to quit.\n");
}

// 3. Graceful handling of edge cases
fn sanitize_cli_input(input: &str) -> Result<String, CliError> {
    let trimmed = input.trim();

    if trimmed.is_empty() {
        return Err(CliError::EmptyInput);
    }
```

(continues on next page)

(continued from previous page)

```
    if trimmed.len() > MAX_INPUT_LENGTH {
        return Err(CliError::InputTooLong(trimmed.len()));
    }

    // Remove potentially problematic characters while preserving meaning
    let sanitized = trimmed
        .chars()
        .filter(|c| c.is_ascii() && (!c.is_control() || *c == '\n' || *c == '\t'))
        .collect::<String>();

    Ok(sanitized)
}
```

Documentation Requirements:

When contributing CLI features, please include:

1. **Inline Documentation**:

```
/// Processes user commands in Simple CLI mode
///
/// # Arguments
///
/// * `command` - The command string starting with '/'
/// * `session_log` - Optional session logger for recording commands
/// * `app_state` - Mutable reference to CLI application state
///
/// # Returns
///
/// * `Ok(true)` - Continue CLI session
/// * `Ok(false)` - Exit CLI session
/// * `Err(e)` - Command processing error
///
/// # Examples
///
/// ```rust
/// let result = process_cli_command("/help", &mut None, &mut app_state).await?;
/// assert_eq!(result, Ok(true));
/// ```
pub async fn process_cli_command(
    command: &str,
    session_log: &mut Option<SessionLogger>,
    app_state: &mut CliAppState,
) -> Result<bool, Box<dyn std::error::Error + Send + Sync>>
```

2. **User Documentation**: Update the user guide with new commands and usage examples
3. **Integration Examples**: Provide shell script examples showing how to use new features

Pull Request Guidelines for CLI Features:

1. **Feature Description**: Clearly describe the CLI enhancement and its use cases
2. **Backward Compatibility**: Ensure changes don’t break existing CLI workflows
3. **Accessibility Testing**: Verify features work with screen readers and accessibility tools
4. **Performance Testing**: Test with large inputs and long sessions
5. **Cross-Platform Testing**: Verify functionality on Windows, macOS, and Linux

Example Pull Request Template for CLI Features:

```
## CLI Feature: [Feature Name]

### Description
Brief description of the new CLI feature or enhancement.

### Use Cases
- [ ] Scripting and automation workflows
- [ ] Accessibility improvements
- [ ] Developer productivity enhancements
- [ ] Integration with external tools

### Testing
- [ ] Unit tests for new functionality
- [ ] Integration tests with sample scripts
- [ ] Accessibility testing with screen reader simulation
- [ ] Cross-platform compatibility testing

### Documentation
- [ ] Updated inline documentation
- [ ] Updated user guide with examples
- [ ] Shell script examples in `/examples/` directory

### Backward Compatibility
- [ ] Existing CLI workflows continue to work
- [ ] Configuration file compatibility maintained
- [ ] Command line argument compatibility preserved
```

3.6.3 Extending Perspt

This guide covers how to extend Perspt with custom providers, plugins, and integrations based on the current GenAI-powered architecture.

Extension Overview

Perspt's architecture allows several extension points:

- **Custom LLM Providers**: Add new providers through the GenAI crate
- **UI Components**: Enhance the Ratatui-based terminal interface
- **Configuration Extensions**: Add custom configuration options
- **Command Extensions**: Implement custom slash commands
- **Streaming Enhancements**: Custom streaming response processing

Working with GenAI Providers

Adding New Providers

Perspt uses the *genai* crate which supports multiple providers out of the box. To add support for a new provider:

```rust
// In config.rs - Add provider support
impl Config {
    pub fn infer_provider_from_key(api_key: &str) -> String {
        match api_key {
```

(continues on next page)

(continued from previous page)

```
            key if key.starts_with("sk-") => "openai".to_string(),
            key if key.starts_with("claude-") => "anthropic".to_string(),
            key if key.starts_with("AIza") => "gemini".to_string(),
            key if key.starts_with("gsk_") => "groq".to_string(),
            key if key.starts_with("xai-") => "xai".to_string(),
            key if key.starts_with("ds-") => "deepseek".to_string(),
            key if key.starts_with("hf_") => "huggingface".to_string(), // New provider
            key if key.starts_with("co_") => "cohere".to_string(),       // New provider
            _ => "openai".to_string(),
        }
    }

    pub fn get_effective_model(&self) -> String {
        match self.model {
            Some(ref model) => model.clone(),
            None => match self.provider.as_str() {
                "openai" => "gpt-4o-mini".to_string(),
                "anthropic" => "claude-3-5-sonnet-20241022".to_string(),
                "gemini" => "gemini-1.5-flash".to_string(),
                "groq" => "llama-3.1-70b-versatile".to_string(),
                "cohere" => "command-r-plus".to_string(),
                "xai" => "grok-3-beta".to_string(),
                "deepseek" => "deepseek-chat".to_string(),
                "ollama" => "llama3.2".to_string(),
                "huggingface" => "microsoft/DialoGPT-medium".to_string(), // New default
                _ => "gpt-4o-mini".to_string(),
            }
        }
    }
}
```

Custom Provider Implementation

For providers not supported by GenAI, you can extend the message handling:

```
// In llm_provider.rs - Custom provider wrapper
pub async fn send_message_custom_provider(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    match config.provider.as_str() {
        "custom_provider" => {
            send_message_to_custom_api(config, message, tx).await
        }
        _ => {
            // Use standard GenAI implementation
            send_message(config, message, tx).await
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
async fn send_message_to_custom_api(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // Custom HTTP client implementation
    let client = reqwest::Client::new();

    let payload = serde_json::json!({
        "prompt": message,
        "max_tokens": config.max_tokens.unwrap_or(1000),
        "temperature": config.temperature.unwrap_or(0.7),
        "stream": true
    });

    let response = client
        .post("https://api.custom-provider.com/v1/chat")
        .header("Authorization", format!("Bearer {}", config.api_key.as_ref().unwrap()))
        .json(&payload)
        .send()
        .await?;

    // Handle streaming response
    let mut stream = response.bytes_stream();
    while let Some(chunk) = stream.next().await {
        let chunk = chunk?;
        if let Ok(text) = String::from_utf8(chunk.to_vec()) {
            tx.send(text)?;
        }
    }

    Ok(())
}
```

Extending UI Components

Custom Terminal UI Elements

You can extend the Ratatui-based UI with custom components:

```
// In ui.rs - Custom rendering components
use ratatui::{
    prelude::*,
    widgets::{Block, Borders, Paragraph, Wrap},
};

pub fn render_custom_status_bar(f: &mut Frame, app: &App, area: Rect) {
    let status_text = format!(
        "Provider: {} | Model: {} | Messages: {} | Memory: {:.1}MB",
        app.config.provider,
        app.config.get_effective_model(),
        app.messages.len(),
```

(continues on next page)

(continued from previous page)

```
        app.get_memory_usage_mb()
    );

    let status_paragraph = Paragraph::new(status_text)
        .style(Style::default().fg(Color::Yellow))
        .block(Block::default().borders(Borders::TOP));

    f.render_widget(status_paragraph, area);
}

pub fn render_typing_indicator(f: &mut Frame, area: Rect, is_typing: bool) {
    if is_typing {
        let indicator = Paragraph::new("AI is typing...")
            .style(Style::default().fg(Color::Cyan).add_modifier(Modifier::ITALIC))
            .wrap(Wrap { trim: true });

        f.render_widget(indicator, area);
    }
}

// Custom markdown rendering enhancements
pub fn render_enhanced_markdown(content: &str) -> Text {
    use pulldown_cmark::{Event, Parser, Tag};

    let parser = Parser::new(content);
    let mut spans = Vec::new();
    let mut current_style = Style::default();

    for event in parser {
        match event {
            Event::Start(Tag::Emphasis) => {
                current_style = current_style.add_modifier(Modifier::ITALIC);
            }
            Event::Start(Tag::Strong) => {
                current_style = current_style.add_modifier(Modifier::BOLD);
            }
            Event::Start(Tag::CodeBlock(_)) => {
                current_style = Style::default()
                    .fg(Color::Green)
                    .bg(Color::Black);
            }
            Event::Text(text) => {
                spans.push(Span::styled(text.to_string(), current_style));
            }
            Event::End(_) => {
                current_style = Style::default();
            }
            _ => {}
        }
    }

    Text::from(Line::from(spans))
```

(continues on next page)

(continued from previous page)

```
}
```

Enhanced Scroll Handling

Recent improvements to Perspt's scroll system demonstrate best practices for handling long content in terminal UIs:

```
// Custom scroll handling for terminal applications
impl App {
    /// Advanced scroll calculation accounting for text wrapping
    pub fn calculate_content_height(&self, content: &[ChatMessage], terminal_width: usize) -
↪> usize {
        let chat_width = terminal_width.saturating_sub(4).max(20); // Account for borders

        content.iter().map(|msg| {
            let mut lines = 1; // Header line

            // Calculate wrapped content lines
            for line in &msg.content {
                let line_text = line.spans.iter()
                    .map(|span| span.content.as_ref())
                    .collect::<String>();

                if line_text.trim().is_empty() {
                    lines += 1;
                } else {
                    // Character-aware text wrapping (important for Unicode)
                    let display_width = line_text.chars().count();
                    if display_width <= chat_width {
                        lines += 1;
                    } else {
                        let wrapped_lines = (display_width + chat_width - 1) / chat_width;
                        lines += wrapped_lines.max(1);
                    }
                }
            }

            lines += 1; // Separator line
            lines
        }).sum()
    }

    /// Conservative scroll bounds to prevent content cutoff
    pub fn calculate_max_scroll(&self, content_height: usize, visible_height: usize) ->␣
↪usize {
        if content_height > visible_height {
            let max_scroll = content_height.saturating_sub(visible_height);
            // Conservative buffer to ensure bottom content is always visible
            max_scroll.saturating_sub(1)
        } else {
            0
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Key Extension Points for Scroll Handling:

- **Text Wrapping Logic**: Customize how text wraps based on content type or user preferences
- **Scroll Animation**: Add smooth scrolling animations for better user experience
- **Auto-scroll Behavior**: Implement smart auto-scrolling that respects user navigation intent
- **Content-aware Scrolling**: Different scroll behavior for code blocks, lists, or other content types
- **Accessibility Features**: Add scroll indicators, position feedback, or keyboard shortcuts

Best Practices for Terminal UI Scrolling:

1. **Character-based calculations**: Always use *.chars().count()* for Unicode-safe text measurement
2. **Conservative buffering**: Leave small buffers to prevent content cutoff at boundaries
3. **Consistent state**: Keep scroll calculation logic identical across all scroll methods
4. **Terminal adaptation**: Account for borders, padding, and other UI elements in calculations
5. **User feedback**: Provide visual indicators (scrollbars, position info) for scroll state

Configuration Extensions

Adding Custom Configuration Options

You can extend the configuration system to support custom options:

```
// Extended configuration structure
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ExtendedConfig {
    #[serde(flatten)]
    pub base: Config,

    // Custom extensions
    pub custom_theme: Option<String>,
    pub auto_save: Option<bool>,
    pub custom_commands: Option<HashMap<String, String>>,
    pub ui_preferences: Option<UiPreferences>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct UiPreferences {
    pub show_timestamps: bool,
    pub message_limit: usize,
    pub enable_syntax_highlighting: bool,
    pub custom_colors: Option<ColorScheme>,
}

impl ExtendedConfig {
    pub fn load_extended() -> Result<Self, ConfigError> {
        // Try to load extended config first
        if let Ok(config_str) = fs::read_to_string("config.extended.json") {
            return serde_json::from_str(&config_str)
                .map_err(|e| ConfigError::ParseError(e.to_string()));
        }

        // Fallback to base config
```

(continues on next page)

(continued from previous page)

```
        let base_config = Config::load()?;
        Ok(ExtendedConfig {
            base: base_config,
            custom_theme: None,
            auto_save: Some(true),
            custom_commands: None,
            ui_preferences: Some(UiPreferences::default()),
        })
    }
}
```

Custom Command System

Implement custom slash commands for enhanced functionality:

```
// In main.rs or ui.rs - Command processing
pub enum CustomCommand {
    SaveConversation(String),
    LoadConversation(String),
    SetTheme(String),
    ShowStats,
    ClearHistory,
    ExportMarkdown(String),
}

impl CustomCommand {
    pub fn parse(input: &str) -> Option<Self> {
        let parts: Vec<&str> = input.trim_start_matches('/').split_whitespace().collect();

        match parts.get(0)? {
            "save" => Some(CustomCommand::SaveConversation(
                parts.get(1).unwrap_or("conversation").to_string()
            )),
            "load" => Some(CustomCommand::LoadConversation(
                parts.get(1).unwrap_or("conversation").to_string()
            )),
            "theme" => Some(CustomCommand::SetTheme(
                parts.get(1).unwrap_or("default").to_string()
            )),
            "stats" => Some(CustomCommand::ShowStats),
            "clear" => Some(CustomCommand::ClearHistory),
            "export" => Some(CustomCommand::ExportMarkdown(
                parts.get(1).unwrap_or("conversation.md").to_string()
            )),
            _ => None,
        }
    }

    pub async fn execute(&self, app: &mut App) -> Result<String, Box<dyn std::error::Error>
 ↪> {
        match self {
            CustomCommand::SaveConversation(name) => {
```

(continues on next page)

(continued from previous page)

```
                app.save_conversation(name).await?;
                Ok(format!("Conversation saved as '{}'", name))
            }
            CustomCommand::LoadConversation(name) => {
                app.load_conversation(name).await?;
                Ok(format!("Conversation '{}' loaded", name))
            }
            CustomCommand::SetTheme(theme) => {
                app.set_theme(theme);
                Ok(format!("Theme changed to '{}'", theme))
            }
            CustomCommand::ShowStats => {
                let stats = app.get_conversation_stats();
                Ok(format!(
                    "Messages: {}, Total characters: {}, Session time: {}min",
                    stats.message_count,
                    stats.total_characters,
                    stats.session_time_minutes
                ))
            }
            CustomCommand::ClearHistory => {
                app.clear_conversation_history();
                Ok("Conversation history cleared".to_string())
            }
            CustomCommand::ExportMarkdown(filename) => {
                app.export_to_markdown(filename).await?;
                Ok(format!("Conversation exported to '{}'", filename))
            }
        }
    }
}
    pub timeout: Option<u64>,
}

pub struct CustomProvider {
    client: reqwest::Client,
    config: CustomProviderConfig,
}

impl CustomProvider {
    pub fn new(config: CustomProviderConfig) -> Self {
        let client = reqwest::Client::builder()
            .timeout(std::time::Duration::from_secs(config.timeout.unwrap_or(30)))
            .build()
            .expect("Failed to create HTTP client");

        Self { client, config }
    }
}

#[async_trait]
impl LLMProvider for CustomProvider {
```

(continues on next page)

(continued from previous page)

```
    async fn chat_completion(
        &self,
        messages: &[Message],
        options: &ChatOptions,
    ) -> Result<ChatResponse, LLMError> {
        let request_body = self.build_request(messages, options)?;

        let response = self.client
            .post(&format!("{}/chat/completions", self.config.base_url))
            .header("Authorization", format!("Bearer {}", self.config.api_key))
            .header("Content-Type", "application/json")
            .json(&request_body)
            .send()
            .await
            .map_err(|e| LLMError::NetworkError(e.to_string()))?;

        let response_body: CustomResponse = response
            .json()
            .await
            .map_err(|e| LLMError::ParseError(e.to_string()))?;

        Ok(self.parse_response(response_body)?)
    }

    async fn stream_completion(
        &self,
        messages: &[Message],
        options: &ChatOptions,
    ) -> Result<Pin<Box<dyn Stream<Item = Result<ChatChunk, LLMError>>>>, LLMError> {
        // Implement streaming response handling
        todo!("Implement streaming for your provider")
    }

    fn validate_config(&self, config: &ProviderConfig) -> Result<(), LLMError> {
        // Validate provider-specific configuration
        if self.config.api_key.is_empty() {
            return Err(LLMError::ConfigurationError("API key is required".to_string()));
        }
        Ok(())
    }
}
```

Advanced Provider Features

Function Calling Support:

```
impl CustomProvider {
    fn build_request_with_functions(
        &self,
        messages: &[Message],
        options: &ChatOptions,
        functions: &[Function],
```

(continues on next page)

(continued from previous page)

```
    ) -> Result<CustomRequest, LLMError> {
        CustomRequest {
            model: self.config.model.clone(),
            messages: self.convert_messages(messages),
            functions: Some(functions.iter().map(|f| f.into()).collect()),
            function_call: options.function_call.clone(),
            // ... other fields
        }
    }
}
```

Multimodal Support:

```
#[async_trait]
impl MultimodalProvider for CustomProvider {
    async fn chat_completion_with_images(
        &self,
        messages: &[Message],
        images: &[ImageData],
        options: &ChatOptions,
    ) -> Result<ChatResponse, LLMError> {
        let request = self.build_multimodal_request(messages, images, options)?;
        // Implementation
    }
}
```

Creating Custom Commands

Built-in Command System

Added in v0.4.3 - Perspt includes a built-in command system for productivity features. Commands are prefixed with `/` and processed before regular chat messages.

Save Conversation Command Implementation:

```
impl App {
    /// Handle built-in commands like /save
    pub fn handle_command(&mut self, input: String) -> Result<bool, String> {
        if !input.starts_with('/') {
            return Ok(false); // Not a command
        }

        let parts: Vec<&str> = input.splitn(2, ' ').collect();
        let command = parts[0];

        match command {
            "/save" => {
                let filename = parts.get(1).map(|s| s.to_string());
                self.execute_save_command(filename)
            }
            "/help" => {
                self.show_help_command();
                Ok(true)
```

(continues on next page)

(continued from previous page)

```
            }
            _ => {
                self.add_system_message(format!("□ Unknown command: {}", command));
                Ok(true)
            }
        }
    }

    fn execute_save_command(&mut self, filename: Option<String>) -> Result<bool, String> {
        match self.save_conversation(filename) {
            Ok(saved_filename) => {
                self.add_system_message(format!("□ Conversation saved to: {}", saved_
↪filename));
                Ok(true)
            }
            Err(e) => {
                self.add_system_message(format!("□ Error saving conversation: {}", e));
                Ok(true) // Command was handled, even with error
            }
        }
    }

    /// Save conversation to text file with proper formatting
    pub fn save_conversation(&self, filename: Option<String>) -> Result<String> {
        use std::fs;
        use std::time::{SystemTime, UNIX_EPOCH};

        // Filter conversation messages (exclude system messages)
        let conversation_messages: Vec<_> = self.chat_history
            .iter()
            .filter(|msg| matches!(msg.message_type, MessageType::User |␣
↪MessageType::Assistant))
            .collect();

        if conversation_messages.is_empty() {
            return Err(anyhow::anyhow!("No conversation to save"));
        }

        // Generate timestamped filename if not provided
        let filename = filename.unwrap_or_else(|| {
            let timestamp = SystemTime::now()
                .duration_since(UNIX_EPOCH)
                .unwrap()
                .as_secs();
            format!("conversation_{}.txt", timestamp)
        });

        // Format conversation content
        let mut content = String::new();
        content.push_str("Perspt Conversation\n");
        content.push_str(&"=".repeat(18));
        content.push('\n');
```

(continues on next page)

(continued from previous page)

```
        for message in conversation_messages {
            let role = match message.message_type {
                MessageType::User => "User",
                MessageType::Assistant => "Assistant",
                _ => continue, // Skip other message types
            };

            content.push_str(&format!("[{}] {}: {}\n\n",
                message.timestamp,
                role,
                message.raw_content
            ));
        }

        // Write to file
        fs::write(&filename, content)?;
        Ok(filename)
    }
}
```

Command Registration System:

```
pub struct CommandRegistry {
    commands: HashMap<String, Box<dyn Command>>,
}

pub trait Command: Send + Sync {
    fn name(&self) -> &str;
    fn description(&self) -> &str;
    fn execute(&self, app: &mut App, args: Vec<&str>) -> Result<(), String>;
}

impl CommandRegistry {
    pub fn new() -> Self {
        let mut registry = Self {
            commands: HashMap::new(),
        };

        // Register built-in commands
        registry.register(Box::new(SaveCommand));
        registry.register(Box::new(HelpCommand));

        registry
    }

    pub fn register(&mut self, command: Box<dyn Command>) {
        self.commands.insert(command.name().to_string(), command);
    }

    pub fn execute(&self, app: &mut App, input: String) -> Result<bool, String> {
        let parts: Vec<&str> = input.splitn(2, ' ').collect();
```

(continues on next page)

(continued from previous page)

```
        let command_name = parts[0].trim_start_matches('/');

        if let Some(command) = self.commands.get(command_name) {
            let args = if parts.len() > 1 {
                parts[1].split_whitespace().collect()
            } else {
                vec![]
            };

            command.execute(app, args)?;
            Ok(true)
        } else {
            Ok(false) // Command not found
        }
    }
}
```

Adding Custom Commands

You can extend the command system with your own commands:

```
pub struct ExportMarkdownCommand;

impl Command for ExportMarkdownCommand {
    fn name(&self) -> &str {
        "export-md"
    }

    fn description(&self) -> &str {
        "Export conversation as markdown with formatting preserved"
    }

    fn execute(&self, app: &mut App, args: Vec<&str>) -> Result<(), String> {
        let filename = args.get(0)
            .map(|s| s.to_string())
            .unwrap_or_else(|| format!("conversation_{}.md",
                SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs()));

        let markdown_content = self.format_as_markdown(&app.chat_history)?;
        std::fs::write(&filename, markdown_content)
            .map_err(|e| format!("Failed to write file: {}", e))?;

        app.add_system_message(format!("□ Conversation exported as markdown to: {}",␣
 ↪filename));
        Ok(())
    }
}

// Usage in main application
let mut command_registry = CommandRegistry::new();
command_registry.register(Box::new(ExportMarkdownCommand));
```

File Processing Plugin Example:

Here’s a complete example of a plugin that adds file processing capabilities:

```
use async_trait::async_trait;
use perspt::{Plugin, PluginConfig, PluginResponse, PluginError};
use std::path::Path;
use tokio::fs;

pub struct FileProcessorPlugin {
    max_file_size: usize,
    supported_extensions: Vec<String>,
}

impl FileProcessorPlugin {
    pub fn new() -> Self {
        Self {
            max_file_size: 10 * 1024 * 1024, // 10MB
            supported_extensions: vec![
                "txt".to_string(),
                "md".to_string(),
                "rs".to_string(),
                "py".to_string(),
                "js".to_string(),
            ],
        }
    }

    async fn process_file(&self, file_path: &str) -> Result<String, PluginError> {
        let path = Path::new(file_path);

        // Validate file exists
        if !path.exists() {
            return Err(PluginError::InvalidInput(
                format!("File not found: {}", file_path)
            ));
        }

        // Check file size
        let metadata = fs::metadata(path).await
            .map_err(|e| PluginError::IOError(e.to_string()))?;

        if metadata.len() > self.max_file_size as u64 {
            return Err(PluginError::InvalidInput(
                "File too large".to_string()
            ));
        }

        // Check file extension
        if let Some(ext) = path.extension() {
            let ext_str = ext.to_str().unwrap_or("");
            if !self.supported_extensions.contains(&ext_str.to_string()) {
                return Err(PluginError::InvalidInput(
                    format!("Unsupported file type: {}", ext_str)
```

(continues on next page)

(continued from previous page)

```
                ));
            }
        }

        // Read file content
        let content = fs::read_to_string(path).await
            .map_err(|e| PluginError::IOError(e.to_string()))?;

        Ok(content)
    }
}

#[async_trait]
impl Plugin for FileProcessorPlugin {
    fn name(&self) -> &str {
        "file-processor"
    }

    fn version(&self) -> &str {
        "1.0.0"
    }

    fn description(&self) -> &str {
        "Process and analyze text files"
    }

    async fn initialize(&mut self, config: &PluginConfig) -> Result<(), PluginError> {
        if let Some(max_size) = config.get("max_file_size") {
            self.max_file_size = max_size.parse()
                .map_err(|_| PluginError::ConfigurationError(
                    "Invalid max_file_size".to_string()
                ))?;
        }

        if let Some(extensions) = config.get("supported_extensions") {
            self.supported_extensions = extensions
                .split(',')
                .map(|s| s.trim().to_string())
                .collect();
        }

        Ok(())
    }

    async fn shutdown(&mut self) -> Result<(), PluginError> {
        // Cleanup resources if needed
        Ok(())
    }

    async fn handle_command(
        &self,
        command: &str,
```

(continues on next page)

(continued from previous page)

```
        args: &[String],
    ) -> Result<PluginResponse, PluginError> {
        match command {
            "read-file" => {
                if args.is_empty() {
                    return Err(PluginError::InvalidInput(
                        "File path required".to_string()
                    ));
                }

                let content = self.process_file(&args[0]).await?;
                Ok(PluginResponse::Text(format!(
                    "File content ({}):
                     {}",
                    args[0], content
                )))
            }

            "analyze-file" => {
                if args.is_empty() {
                    return Err(PluginError::InvalidInput(
                        "File path required".to_string()
                    ));
                }

                let content = self.process_file(&args[0]).await?;
                let analysis = self.analyze_content(&content);

                Ok(PluginResponse::Structured(serde_json::json!({
                    "file": args[0],
                    "lines": content.lines().count(),
                    "characters": content.len(),
                    "words": content.split_whitespace().count(),
                    "analysis": analysis
                })))
            }

            _ => Err(PluginError::UnsupportedCommand(command.to_string()))
        }
    }

    fn supported_commands(&self) -> Vec<String> {
        vec!["read-file".to_string(), "analyze-file".to_string()]
    }
}

impl FileProcessorPlugin {
    fn analyze_content(&self, content: &str) -> serde_json::Value {
        // Simple content analysis
        let lines = content.lines().count();
        let words = content.split_whitespace().count();
        let chars = content.len();
```

(continues on next page)

(continued from previous page)

```
        serde_json::json!({
            "complexity": if lines > 100 { "high" } else if lines > 50 { "medium" } else {
↪"low" },
            "language": self.detect_language(content),
            "metrics": {
                "lines": lines,
                "words": words,
                "characters": chars
            }
        })
    }

    fn detect_language(&self, content: &str) -> &str {
        if content.contains("fn main()") && content.contains("println!") {
            "rust"
        } else if content.contains("def ") && content.contains("import ") {
            "python"
        } else if content.contains("function ") && content.contains("console.log") {
            "javascript"
        } else {
            "unknown"
        }
    }
}
```

Integration Plugin Example

Here's a plugin that integrates with external APIs:

```
pub struct WebSearchPlugin {
    api_key: String,
    client: reqwest::Client,
}

#[async_trait]
impl Plugin for WebSearchPlugin {
    fn name(&self) -> &str {
        "web-search"
    }

    fn version(&self) -> &str {
        "1.0.0"
    }

    fn description(&self) -> &str {
        "Search the web and return relevant results"
    }

    async fn initialize(&mut self, config: &PluginConfig) -> Result<(), PluginError> {
        self.api_key = config.get("api_key")
            .ok_or_else(|| PluginError::ConfigurationError(
```

(continues on next page)

(continued from previous page)

```
                "API key required for web search".to_string()
            ))?
            .to_string();

        Ok(())
    }

    async fn handle_command(
        &self,
        command: &str,
        args: &[String],
    ) -> Result<PluginResponse, PluginError> {
        match command {
            "search" => {
                if args.is_empty() {
                    return Err(PluginError::InvalidInput(
                        "Search query required".to_string()
                    ));
                }

                let query = args.join(" ");
                let results = self.search_web(&query).await?;

                Ok(PluginResponse::Structured(serde_json::json!({
                    "query": query,
                    "results": results
                })))
            }
            _ => Err(PluginError::UnsupportedCommand(command.to_string()))
        }
    }

    fn supported_commands(&self) -> Vec<String> {
        vec!["search".to_string()]
    }
}

impl WebSearchPlugin {
    async fn search_web(&self, query: &str) -> Result<Vec<SearchResult>, PluginError> {
        let url = format!("https://api.searchengine.com/search?q={}&key={}",
                         urlencoding::encode(query),
                         self.api_key);

        let response: SearchResponse = self.client
            .get(&url)
            .send()
            .await
            .map_err(|e| PluginError::NetworkError(e.to_string()))?
            .json()
            .await
            .map_err(|e| PluginError::ParseError(e.to_string()))?;
```

(continues on next page)

(continued from previous page)

```
        Ok(response.results)
    }
}
```

Plugin Configuration

Plugin Configuration Schema

```json
{
  "plugins": {
    "file-processor": {
      "enabled": true,
      "config": {
        "max_file_size": 10485760,
        "supported_extensions": "txt,md,rs,py,js,ts"
      }
    },
    "web-search": {
      "enabled": true,
      "config": {
        "api_key": "your-search-api-key",
        "max_results": 10
      }
    }
  }
}
```

Dynamic Plugin Loading

```rust
pub struct PluginManager {
    plugins: HashMap<String, Box<dyn Plugin>>,
    config: PluginManagerConfig,
}

impl PluginManager {
    pub async fn load_plugin_from_path(&mut self, path: &Path) -> Result<(), PluginError> {
        // Dynamic loading implementation
        let plugin = unsafe {
            self.load_dynamic_library(path)?
        };

        let plugin_name = plugin.name().to_string();
        self.plugins.insert(plugin_name, plugin);

        Ok(())
    }

    pub async fn execute_plugin_command(
        &self,
        plugin_name: &str,
        command: &str,
```

(continues on next page)

(continued from previous page)

```
        args: &[String],
    ) -> Result<PluginResponse, PluginError> {
        let plugin = self.plugins.get(plugin_name)
            .ok_or_else(|| PluginError::PluginNotFound(plugin_name.to_string()))?;

        plugin.handle_command(command, args).await
    }
}
```

Custom UI Components

Creating Custom Display Components

```
use perspt::ui::{DisplayComponent, RenderContext, UIError};

pub struct CustomProgressBar {
    progress: f32,
    width: usize,
    style: ProgressStyle,
}

impl DisplayComponent for CustomProgressBar {
    fn render(&self, context: &mut RenderContext) -> Result<(), UIError> {
        let filled = (self.progress * self.width as f32) as usize;
        let empty = self.width - filled;

        let bar = format!(
            "[{}{}] {:.1}%",
            "█".repeat(filled),
            "░".repeat(empty),
            self.progress * 100.0
        );

        context.write_line(&bar, &self.style.into())?;
        Ok(())
    }
}

pub struct CustomTable {
    headers: Vec<String>,
    rows: Vec<Vec<String>>,
    column_widths: Vec<usize>,
}

impl DisplayComponent for CustomTable {
    fn render(&self, context: &mut RenderContext) -> Result<(), UIError> {
        // Render table headers
        self.render_headers(context)?;

        // Render table rows
        for row in &self.rows {
            self.render_row(context, row)?;
```

(continues on next page)

(continued from previous page)

```
        }

        Ok(())
    }
}
```

Custom Command Processors

```
pub struct CustomCommandProcessor;

impl CommandProcessor for CustomCommandProcessor {
    fn process_command(
        &self,
        command: &str,
        args: &[String],
        context: &mut CommandContext,
    ) -> Result<CommandResult, CommandError> {
        match command {
            "custom-help" => {
                let help_text = self.generate_custom_help();
                Ok(CommandResult::Display(help_text))
            }

            "batch-process" => {
                if args.is_empty() {
                    return Err(CommandError::MissingArguments);
                }

                let results = self.process_batch(&args[0])?;
                Ok(CommandResult::Structured(results))
            }

            _ => Err(CommandError::UnknownCommand(command.to_string()))
        }
    }
}
```

Testing Plugins and Extensions

Unit Testing Plugins

```
#[cfg(test)]
mod tests {
    use super::*;
    use perspt::testing::{MockPluginConfig, MockContext};

    #[tokio::test]
    async fn test_file_processor_plugin() {
        let mut plugin = FileProcessorPlugin::new();
        let config = MockPluginConfig::new();

```

(continues on next page)

(continued from previous page)

```rust
        plugin.initialize(&config).await.unwrap();

        // Test file reading
        let response = plugin
            .handle_command("read-file", &["test.txt".to_string()])
            .await;

        assert!(response.is_ok());
    }

    #[tokio::test]
    async fn test_plugin_error_handling() {
        let plugin = FileProcessorPlugin::new();

        // Test error case
        let response = plugin
            .handle_command("read-file", &[])
            .await;

        assert!(matches!(response, Err(PluginError::InvalidInput(_))));
    }
}
```

Integration Testing

```rust
#[tokio::test]
async fn test_plugin_integration() {
    let mut app = TestApplication::new().await;

    // Load plugin
    app.load_plugin("file-processor", FileProcessorPlugin::new()).await.unwrap();

    // Test plugin command execution
    let response = app.execute_command("/read-file test.txt").await.unwrap();
    assert!(!response.is_empty());
}
```

Performance Testing

```rust
#[tokio::test]
async fn test_plugin_performance() {
    let plugin = WebSearchPlugin::new();
    let start = std::time::Instant::now();

    let _response = plugin
        .handle_command("search", &["rust programming".to_string()])
        .await
        .unwrap();

    let duration = start.elapsed();
```

(continues on next page)

(continued from previous page)

```
    assert!(duration.as_secs() < 5); // Should complete within 5 seconds
}
```

Distribution and Packaging

Plugin Distribution

Cargo Package:

```
# Cargo.toml for your plugin
[package]
name = "perspt-file-processor"
version = "1.0.0"
edition = "2021"

[dependencies]
perspt = "1.0"
async-trait = "0.1"
tokio = { version = "1.0", features = ["full"] }
serde = { version = "1.0", features = ["derive"] }
```

Plugin Manifest:

```
{
  "name": "file-processor",
  "version": "1.0.0",
  "description": "Process and analyze text files",
  "author": "Your Name",
  "license": "MIT",
  "min_perspt_version": "1.0.0",
  "dependencies": [],
  "commands": ["read-file", "analyze-file"],
  "configuration_schema": {
    "max_file_size": "integer",
    "supported_extensions": "string"
  }
}
```

Extension Deployment

Configuration-Based Extensions:

```
# Add custom provider configuration
echo '{
  "provider": "custom_openai",
  "api_key": "your-key",
  "model": "gpt-4",
  "base_url": "https://api.custom-provider.com/v1",
  "timeout_seconds": 60
}' > ~/.config/perspt/config.json
```

Code-Based Extensions:

```
# Fork and modify the main repository
git clone https://github.com/eonseed/perspt.git
cd perspt

# Add your custom provider logic
# Build and install
cargo build --release
cargo install --path .
```

Environment-Based Configuration:

```
# Set provider-specific environment variables
export OPENAI_API_KEY="your-openai-key"
export ANTHROPIC_API_KEY="your-anthropic-key"
export GEMINI_API_KEY="your-gemini-key"
export GROQ_API_KEY="your-groq-key"
export COHERE_API_KEY="your-cohere-key"
export XAI_API_KEY="your-xai-key"
export DEEPSEEK_API_KEY="your-deepseek-key"
export OLLAMA_API_BASE="http://localhost:11434"
export PERSPT_PROVIDER="openai"
export PERSPT_MODEL="gpt-4o-mini"
```

Best Practices

Provider Extension Development

1. **Error Handling**: Use comprehensive error types and meaningful messages

```
use anyhow::{Context, Result};
use thiserror::Error;

#[derive(Error, Debug)]
pub enum ProviderError {
    #[error("API key not provided for {provider}")]
    MissingApiKey { provider: String },
    #[error("Invalid model {model} for provider {provider}")]
    InvalidModel { model: String, provider: String },
    #[error("Request timeout after {seconds}s")]
    Timeout { seconds: u64 },
}
```

2. **Configuration Validation**: Implement robust config validation

```
impl Config {
    pub fn validate(&self) -> Result<()> {
        match self.provider.as_str() {
            "openai" => {
                if self.api_key.is_none() {
                    return Err(ProviderError::MissingApiKey {
                        provider: self.provider.clone()
                    }.into());
                }
            }
```

(continues on next page)

(continued from previous page)

```
            provider => {
                return Err(ProviderError::UnsupportedProvider {
                    provider: provider.to_string()
                }.into());
            }
        }
        Ok(())
    }
}
```

3. **Async/Await Patterns**: Follow proper async patterns with error handling

```
pub async fn send_custom_message(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<()> {
    let client = build_client(config).await
        .context("Failed to build HTTP client")?;

    let mut stream = create_stream(client, message).await
        .context("Failed to create response stream")?;

    while let Some(chunk) = stream.try_next().await
        .context("Error reading from stream")? {
        tx.send(chunk).context("Failed to send chunk")?;
    }

    Ok(())
}
```

4. **Testing**: Write comprehensive tests for all extension points

```
#[cfg(test)]
mod tests {
    use super::*;
    use tokio_test;

    #[tokio::test]
    async fn test_custom_provider_integration() {
        let config = Config {
            provider: "custom".to_string(),
            api_key: Some("test-key".to_string()),
            model: Some("test-model".to_string()),
            ..Default::default()
        };

        let (tx, mut rx) = tokio::sync::mpsc::unbounded_channel();

        // Test your custom provider logic
        let result = send_custom_message(&config, "test", tx).await;
        assert!(result.is_ok());
    }
}
```

UI Extension Development

1. **Component Modularity**: Keep UI components small and focused

```
pub struct CustomWidget {
    content: String,
    scroll_offset: u16,
}

impl CustomWidget {
    pub fn render(&self, area: Rect, buf: &mut Buffer) {
        let block = Block::default()
            .borders(Borders::ALL)
            .title("Custom Widget");

        let inner = block.inner(area);
        block.render(area, buf);

        // Custom rendering logic
        self.render_content(inner, buf);
    }
}
```

2. **Event Handling**: Implement responsive event handling

```
pub fn handle_custom_event(&mut self, event: Event) -> Result<bool> {
    match event {
        Event::Key(key) => {
            match key.code {
                KeyCode::Char('c') if key.modifiers.contains(KeyModifiers::CONTROL) =>
 ↪ {
                    // Custom control handling
                    return Ok(true); // Event consumed
                }
                _ => return Ok(false), // Event not handled
            }
        }
        _ => return Ok(false),
    }
}
```

Configuration Extension Development

1. **Schema Validation**: Define clear configuration schemas

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Deserialize, Serialize)]
pub struct ExtendedConfig {
    #[serde(flatten)]
    pub base: Config,
    pub custom_timeout: Option<u64>,
    pub retry_attempts: Option<u32>,
    pub custom_headers: Option<std::collections::HashMap<String, String>>,
```

(continues on next page)

(continued from previous page)

```
}
```

2. **Environment Integration**: Support environment variable overrides

```rust
impl ExtendedConfig {
    pub fn from_env() -> Result<Self> {
        let mut config = Config::load()?;

        if let Ok(timeout) = std::env::var("PERSPT_CUSTOM_TIMEOUT") {
            config.custom_timeout = Some(timeout.parse()?);
        }

        if let Ok(retries) = std::env::var("PERSPT_RETRY_ATTEMPTS") {
            config.retry_attempts = Some(retries.parse()?);
        }

        Ok(config)
    }
}
```

Performance Considerations

1. **Async Efficiency**: Use proper async patterns to avoid blocking

```rust
// Good: Non-blocking async operations
pub async fn efficient_processing(data: &[String]) -> Result<Vec<String>> {
    let tasks: Vec<_> = data.iter()
        .map(|item| process_item_async(item))
        .collect();

    let results = futures::future::try_join_all(tasks).await?;
    Ok(results)
}

// Avoid: Blocking operations in async context
pub async fn inefficient_processing(data: &[String]) -> Result<Vec<String>> {
    let mut results = Vec::new();
    for item in data {
        results.push(process_item_blocking(item)?); // Bad!
    }
    Ok(results)
}
```

2. **Memory Management**: Handle large responses efficiently

```rust
pub async fn stream_large_response(
    config: &Config,
    message: &str,
    tx: UnboundedSender<String>,
) -> Result<()> {
    const CHUNK_SIZE: usize = 1024;
    let mut buffer = String::with_capacity(CHUNK_SIZE);

```

(continues on next page)

(continued from previous page)

```
    // Process in chunks to avoid memory spikes
    let mut stream = create_response_stream(config, message).await?;

    while let Some(chunk) = stream.try_next().await? {
        buffer.push_str(&chunk);

        if buffer.len() >= CHUNK_SIZE {
            tx.send(buffer.clone())?;
            buffer.clear();
        }
    }

    if !buffer.is_empty() {
        tx.send(buffer)?;
    }

    Ok(())
}
```

Security Considerations

1. **API Key Management**: Secure handling of sensitive data

```
use secrecy::{ExposeSecret, Secret};

pub struct SecureConfig {
    pub provider: String,
    pub api_key: Option<Secret<String>>,
    pub model: Option<String>,
}

impl SecureConfig {
    pub fn load_secure() -> Result<Self> {
        let api_key = std::env::var("API_KEY")
            .map(Secret::new)
            .ok();

        Ok(SecureConfig {
            provider: "openai".to_string(),
            api_key,
            model: Some("gpt-4".to_string()),
        })
    }

    pub fn get_api_key(&self) -> Option<&str> {
        self.api_key.as_ref().map(|key| key.expose_secret())
    }
}
```

2. **Input Validation**: Sanitize and validate all inputs

```
pub fn validate_message(message: &str) -> Result<()> {
```

(continues on next page)

(continued from previous page)

```
    if message.is_empty() {
        return Err(anyhow::anyhow!("Message cannot be empty"));
    }

    if message.len() > 10_000 {
        return Err(anyhow::anyhow!("Message too long (max 10,000 characters)"));
    }

    // Check for potentially harmful content
    if message.contains("<script") || message.contains("javascript:") {
        return Err(anyhow::anyhow!("Message contains potentially harmful content"));
    }

    Ok(())
}
```

Next Steps

- *Testing* - Testing strategies for extensions
- *API Reference* - API reference for development
- *Contributing* - How to contribute your extensions
- *Architecture* - Understanding Perspt's internal architecture

Example Projects

For complete examples of extending Perspt, see:

- **Custom Provider Implementation**: Examples in the main repository showing how to add new LLM providers
- **UI Component Extensions**: Ratatui-based widgets for enhanced functionality
- **Configuration Extensions**: Advanced configuration patterns and validation
- **Testing Extensions**: Comprehensive test suites for extension development

To get started with your own extensions, we recommend:

1. Fork the main Perspt repository
2. Study the existing provider implementations in `src/llm_provider.rs`
3. Review the UI components in `src/ui.rs`
4. Examine the configuration system in `src/config.rs`
5. Run the test suite to understand the expected behavior
6. Start with small modifications and gradually build up complexity

Extending Simple CLI Mode

NEW in v0.4.5 - The Simple CLI mode can be extended with custom commands and enhanced functionality:

Adding Custom CLI Commands:

```
// In cli.rs - Extend command processing
pub async fn run_simple_cli_with_commands(
    config: AppConfig,
    model_name: String,
    api_key: String,
    provider: Arc<GenAIProvider>,
    log_file: Option<String>,
```

(continues on next page)

(continued from previous page)

```
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    // ... existing setup code ...

    loop {
        print!("> ");
        io::stdout().flush()?;

        let mut input = String::new();
        match io::stdin().read_line(&mut input) {
            Ok(0) => break,
            Ok(_) => {
                let input = input.trim();
                if input.is_empty() { continue; }
                if input == "exit" { break; }

                // Handle custom commands
                if input.starts_with('/') {
                    match process_cli_command(input, &mut session_log).await {
                        Ok(should_continue) => {
                            if !should_continue { break; }
                            continue;
                        }
                        Err(e) => {
                            eprintln!("Command error: {}", e);
                            continue;
                        }
                    }
                }

                // Handle regular conversation
                // ... existing processing code ...
            }
            Err(e) => break,
        }
    }

    Ok(())
}

async fn process_cli_command(
    command: &str,
    session_log: &mut Option<SessionLogger>,
) -> Result<bool, Box<dyn std::error::Error + Send + Sync>> {
    let parts: Vec<&str> = command.splitn(2, ' ').collect();

    match parts[0] {
        "/help" => {
            println!("Available commands:");
            println!("  /help     - Show this help");
            println!("  /clear    - Clear conversation history");
            println!("  /save     - Save current session to file");
            println!("  /model    - Show current model info");
```

(continues on next page)

(continued from previous page)

```
            println!("  /exit     - Exit the application");
            Ok(true)
        }
        "/clear" => {
            // Clear screen using ANSI escape codes
            print!("\x1B[2J\x1B[1;1H");
            io::stdout().flush()?;
            println!("Conversation cleared.");
            Ok(true)
        }
        "/save" => {
            let filename = parts.get(1)
                .map(|s| s.to_string())
                .unwrap_or_else(|| {
                    format!("session_{}.txt",
                        SystemTime::now()
                            .duration_since(UNIX_EPOCH)
                            .unwrap()
                            .as_secs())
                });

            if let Some(ref logger) = session_log {
                println!("Session saved to: {}", filename);
            } else {
                println!("No session log active. Use --log-file to enable logging.");
            }
            Ok(true)
        }
        "/model" => {
            println!("Current model: {}", /* current model info */);
            println!("Provider: {}", /* current provider */);
            Ok(true)
        }
        "/exit" => {
            println!("Goodbye!");
            Ok(false)
        }
        _ => {
            println!("Unknown command: {}. Type /help for available commands.", parts[0]);
            Ok(true)
        }
    }
}
```

Enhanced Session Logging:

```
// Enhanced session logger with metadata
pub struct EnhancedSessionLogger {
    file: File,
    session_start: SystemTime,
    command_count: u32,
}
```

(continues on next page)

(continued from previous page)

```rust

impl EnhancedSessionLogger {
    pub fn new(log_path: String) -> Result<Self, std::io::Error> {
        let mut file = OpenOptions::new()
            .create(true)
            .append(true)
            .open(&log_path)?;

        // Write session header
        let start_time = SystemTime::now();
        let timestamp = chrono::Local::now().format("%Y-%m-%d %H:%M:%S");
        writeln!(file, "=== Perspt Simple CLI Session Started: {} ===", timestamp)?;
        writeln!(file, "Log file: {}", log_path)?;
        writeln!(file)?;
        file.flush()?;

        Ok(Self {
            file,
            session_start: start_time,
            command_count: 0,
        })
    }

    pub fn log_command(&mut self, command: &str) -> Result<(), std::io::Error> {
        self.command_count += 1;
        let timestamp = chrono::Local::now().format("%Y-%m-%d %H:%M:%S");
        writeln!(self.file, "[{}] Command {}: {}", timestamp, self.command_count, command)?;
↪
        self.file.flush()?;
        Ok(())
    }

    pub fn log_session_stats(&mut self) -> Result<(), std::io::Error> {
        let duration = self.session_start.elapsed().unwrap_or_default();
        let timestamp = chrono::Local::now().format("%Y-%m-%d %H:%M:%S");

        writeln!(self.file)?;
        writeln!(self.file, "=== Session Ended: {} ===", timestamp)?;
        writeln!(self.file, "Duration: {:?}", duration)?;
        writeln!(self.file, "Total commands: {}", self.command_count)?;
        self.file.flush()?;
        Ok(())
    }
}
```

Scriptable Integration Examples:

```bash
# Custom script for batch AI queries
#!/bin/bash

QUESTIONS=(
    "What is machine learning?"
```

(continues on next page)

(continued from previous page)

```
    "Explain deep learning in simple terms"
    "What are neural networks?"
)

LOG_FILE="ai_learning_session_$(date +%Y%m%d_%H%M%S).txt"

for question in "${QUESTIONS[@]}"; do
    echo "Processing: $question"
    echo "$question" | perspt --simple-cli --log-file "$LOG_FILE"
    echo "---" >> "$LOG_FILE"
done

echo "Batch processing complete. Results in: $LOG_FILE"
```

Integration with External Tools:

```
// External tool integration example
pub async fn process_with_external_tool(
    input: &str,
    tool_name: &str,
) -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    match tool_name {
        "json_format" => {
            // Use jq or similar tool to format JSON responses
            let output = Command::new("jq")
                .arg(".")
                .stdin(Stdio::piped())
                .stdout(Stdio::piped())
                .spawn()?;

            // Process with external tool
            // ... implementation ...
            Ok(formatted_output)
        }
        "markdown_render" => {
            // Use pandoc or similar for markdown conversion
            let output = Command::new("pandoc")
                .arg("-f").arg("markdown")
                .arg("-t").arg("plain")
                .stdin(Stdio::piped())
                .stdout(Stdio::piped())
                .spawn()?;

            // ... implementation ...
            Ok(rendered_output)
        }
        _ => Err("Unknown tool".into())
    }
}
```

3.6.4 Testing

This guide covers testing strategies, tools, and best practices for Perspt development, including unit testing, integration testing, and end-to-end testing.

Testing Philosophy

Perspt follows a comprehensive testing approach:

- **Unit Tests**: Test individual components in isolation
- **Integration Tests**: Test component interactions
- **End-to-End Tests**: Test complete user workflows
- **Performance Tests**: Ensure performance requirements are met
- **Security Tests**: Validate security measures

Testing Structure

Test Organization

```
src/
├── main.rs          # Entry point with unit tests
├── config.rs        # Configuration with validation tests
├── llm_provider.rs  # GenAI integration with provider tests
└── ui.rs           # Ratatui UI components with widget tests

tests/
├── panic_handling_test.rs     # Panic handling integration tests
└── integration_tests/         # Additional integration tests
    ├── config_loading.rs
    ├── provider_streaming.rs
    └── ui_rendering.rs

benches/                       # Performance benchmarks
├── streaming_benchmarks.rs
└── config_benchmarks.rs
```

Current Test Structure

The project currently includes:

- **Unit tests**: Embedded in source files using `#[cfg(test)]`
- **Integration tests**: In the `tests/` directory
- **Panic handling tests**: Specialized tests for error recovery
- **Performance benchmarks**: For critical performance paths

Unit Testing

Testing Configuration Module

Tests for configuration loading, validation, and environment handling:

```rust
// src/config.rs
#[cfg(test)]
mod tests {
    use super::*;
```

(continues on next page)

(continued from previous page)

```
    use std::fs;
    use tempfile::TempDir;

    #[test]
    fn test_config_loading_from_file() {
        let temp_dir = TempDir::new().unwrap();
        let config_path = temp_dir.path().join("config.json");

        let config_content = r#"
        {
            "provider": "openai",
            "model": "gpt-4",
            "api_key": "test-key",
            "temperature": 0.7,
            "max_tokens": 2000,
            "timeout_seconds": 30
        }
        "#;

        fs::write(&config_path, config_content).unwrap();

        let config = Config::load_from_path(&config_path).unwrap();
        assert_eq!(config.provider, "openai");
        assert_eq!(config.model, Some("gpt-4".to_string()));
        assert_eq!(config.temperature, Some(0.7));
        assert_eq!(config.max_tokens, Some(2000));
    }

    #[test]
    fn test_provider_inference() {
        // Test automatic provider inference from API key environment
        std::env::set_var("OPENAI_API_KEY", "sk-test");
        let config = Config::with_inferred_provider().unwrap();
        assert_eq!(config.provider, "openai");

        std::env::remove_var("OPENAI_API_KEY");
        std::env::set_var("ANTHROPIC_API_KEY", "sk-ant-test");
        let config = Config::with_inferred_provider().unwrap();
        assert_eq!(config.provider, "anthropic");

        // Cleanup
        std::env::remove_var("ANTHROPIC_API_KEY");
    }

    #[test]
    fn test_config_validation() {
        let mut config = Config::default();
        config.provider = "openai".to_string();
        config.api_key = None; // Missing required API key

        let result = config.validate();
        assert!(result.is_err());
```

(continues on next page)

(continued from previous page)

```
        assert!(result.unwrap_err().to_string().contains("API key"));
    }

    #[test]
    fn test_config_defaults() {
        let config = Config::default();
        assert_eq!(config.provider, "openai");
        assert_eq!(config.model, Some("gpt-3.5-turbo".to_string()));
        assert_eq!(config.temperature, Some(0.7));
        assert_eq!(config.max_tokens, Some(4000));
        assert_eq!(config.timeout_seconds, Some(30));
    }
}
```

Testing LLM Provider Module

Tests for GenAI integration and streaming functionality:

```
// src/llm_provider.rs
#[cfg(test)]
mod tests {
    use super::*;
    use tokio::sync::mpsc;
    use std::time::Duration;

    #[tokio::test]
    async fn test_message_validation() {
        assert!(validate_message("Hello, world!").is_ok());
        assert!(validate_message("").is_err());
        assert!(validate_message(&"x".repeat(20_000)).is_err()); // Too long
    }

    #[tokio::test]
    async fn test_streaming_channel_communication() {
        let (tx, mut rx) = mpsc::unbounded_channel();

        // Simulate streaming response
        tokio::spawn(async move {
            for i in 0..5 {
                tx.send(format!("chunk_{}", i)).unwrap();
                tokio::time::sleep(Duration::from_millis(10)).await;
            }
        });

        let mut received = Vec::new();
        while let Ok(chunk) = tokio::time::timeout(
            Duration::from_millis(100),
            rx.recv()
        ).await {
            if let Some(chunk) = chunk {
                received.push(chunk);
            } else {
```

(continues on next page)

(continued from previous page)

```
                break;
            }
        }

        assert_eq!(received.len(), 5);
        assert_eq!(received[0], "chunk_0");
        assert_eq!(received[4], "chunk_4");
    }

    #[tokio::test]
    #[ignore] // Requires API key
    async fn test_real_provider_integration() {
        if std::env::var("OPENAI_API_KEY").is_err() {
            return; // Skip if no API key
        }

        let config = Config {
            provider: "openai".to_string(),
            api_key: std::env::var("OPENAI_API_KEY").ok(),
            model: Some("gpt-3.5-turbo".to_string()),
            temperature: Some(0.1), // Low temperature for predictable results
            max_tokens: Some(50),
            timeout_seconds: Some(30),
        };

        let (tx, mut rx) = mpsc::unbounded_channel();
        let result = send_message(&config, "Say 'Hello'", tx).await;

        assert!(result.is_ok());

        // Should receive at least some response
        let response = tokio::time::timeout(
            Duration::from_secs(10),
            rx.recv()
        ).await;
        assert!(response.is_ok());
    }

    #[test]
    fn test_config_preparation_for_genai() {
        let config = Config {
            provider: "openai".to_string(),
            api_key: Some("test-key".to_string()),
            model: Some("gpt-4".to_string()),
            temperature: Some(0.7),
            max_tokens: Some(1000),
            timeout_seconds: Some(60),
        };

        // Test that config can be converted to GenAI client format
        assert!(!config.api_key.unwrap().is_empty());
        assert!(config.model.unwrap().contains("gpt"));
```

(continues on next page)

(continued from previous page)

```
    }
}
            ) -> Result<String, HttpError>;
        }
    }

    #[tokio::test]
    async fn test_openai_chat_completion() {
        let mut mock_client = MockHttpClient::new();

        let expected_response = json!({
            "choices": [{
                "message": {
                    "content": "Hello! How can I help you today?"
                }
            }],
            "usage": {
                "total_tokens": 25
            }
        });

        mock_client
            .expect_post()
            .with(
                eq("https://api.openai.com/v1/chat/completions"),
                always(),
                contains("gpt-4")
            )
            .times(1)
            .returning(move |_, _, _| Ok(expected_response.to_string()));

        let config = OpenAIConfig {
            api_key: "test-key".to_string(),
            model: "gpt-4".to_string(),
            ..Default::default()
        };

        let provider = OpenAIProvider::new_with_client(config, Box::new(mock_client));

        let messages = vec![
            Message {
                role: "user".to_string(),
                content: "Hello".to_string(),
            }
        ];

        let options = ChatOptions::default();
        let response = provider.chat_completion(&messages, &options).await.unwrap();

        assert_eq!(response.content, "Hello! How can I help you today?");
        assert_eq!(response.tokens_used, Some(25));
    }
```

(continues on next page)

(continued from previous page)

```
    #[tokio::test]
    async fn test_provider_error_handling() {
        let mut mock_client = MockHttpClient::new();

        mock_client
            .expect_post()
            .returning(|_, _, _| Err(HttpError::NetworkError("Connection failed".to_
↪string())));

        let config = OpenAIConfig::default();
        let provider = OpenAIProvider::new_with_client(config, Box::new(mock_client));

        let messages = vec![Message::user("Test message")];
        let options = ChatOptions::default();

        let result = provider.chat_completion(&messages, &options).await;
        assert!(result.is_err());
        assert!(matches!(result.unwrap_err(), LLMError::NetworkError(_)));
    }

    #[tokio::test]
    async fn test_rate_limiting() {
        let mut mock_client = MockHttpClient::new();

        // First request succeeds
        mock_client
            .expect_post()
            .times(1)
            .returning(|_, _, _| Ok(r#"{"choices":[{"message":{"content":"Success"}}]}"#.
↪to_string()));

        // Second request hits rate limit
        mock_client
            .expect_post()
            .times(1)
            .returning(|_, _, _| Err(HttpError::RateLimit));

        let config = OpenAIConfig::default();
        let provider = OpenAIProvider::new_with_client(config, Box::new(mock_client));

        let messages = vec![Message::user("Test")];
        let options = ChatOptions::default();

        // First request should succeed
        let result1 = provider.chat_completion(&messages, &options).await;
        assert!(result1.is_ok());

        // Second request should fail with rate limit error
        let result2 = provider.chat_completion(&messages, &options).await;
        assert!(matches!(result2.unwrap_err(), LLMError::RateLimit));
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Testing UI Components

```
// src/ui.rs
#[cfg(test)]
mod tests {
    use super::*;
    use std::io::Cursor;

    #[test]
    fn test_message_formatting() {
        let formatter = MessageFormatter::new();

        let message = Message {
            role: "assistant".to_string(),
            content: "Here's some `code` and **bold** text.".to_string(),
        };

        let formatted = formatter.format_message(&message);
        assert!(formatted.contains("code"));
        assert!(formatted.contains("bold"));
    }

    #[test]
    fn test_input_parsing() {
        let parser = InputParser::new();

        // Test regular message
        let input = "Hello, world!";
        let parsed = parser.parse(input);
        assert!(matches!(parsed, ParsedInput::Message(_)));

        // Test command
        let input = "/help";
        let parsed = parser.parse(input);
        assert!(matches!(parsed, ParsedInput::Command { name: "help", .. }));

        // Test command with arguments
        let input = "/model gpt-4";
        let parsed = parser.parse(input);
        if let ParsedInput::Command { name, args } = parsed {
            assert_eq!(name, "model");
            assert_eq!(args, vec!["gpt-4"]);
        }
    }

    #[tokio::test]
    async fn test_ui_rendering() {
        let mut output = Cursor::new(Vec::new());
        let mut ui = UIManager::new_with_output(Box::new(output));
```

(continues on next page)

(continued from previous page)

```
        let message = Message::assistant("Test response");
        ui.render_message(&message).await.unwrap();

        let output_data = ui.get_output_data();
        let output_str = String::from_utf8(output_data).unwrap();
        assert!(output_str.contains("Test response"));
    }
}
```

Integration Testing

Provider Integration Tests

```
// tests/integration/provider_tests.rs
use perspt::*;
use std::env;

#[tokio::test]
#[ignore] // Requires API key
async fn test_openai_integration() {
    let api_key = env::var("OPENAI_API_KEY")
        .expect("OPENAI_API_KEY environment variable required for integration tests");

    let config = OpenAIConfig {
        api_key,
        model: "gpt-4o-mini".to_string(),
        ..Default::default()
    };

    let provider = OpenAIProvider::new(config);

    let messages = vec![
        Message::user("What is 2+2?")
    ];

    let options = ChatOptions {
        max_tokens: Some(50),
        temperature: Some(0.1),
        ..Default::default()
    };

    let response = provider.chat_completion(&messages, &options).await.unwrap();
    assert!(!response.content.is_empty());
    assert!(response.content.contains("4"));
}

#[tokio::test]
async fn test_provider_fallback() {
    let primary_config = OpenAIConfig {
        api_key: "invalid-key".to_string(),
        model: "gpt-4".to_string(),
```

(continues on next page)

(continued from previous page)

```rust
        ..Default::default()
    };

    let fallback_config = OllamaConfig {
        base_url: "http://localhost:11434".to_string(),
        model: "llama2".to_string(),
        ..Default::default()
    };

    let fallback_chain = FallbackChain::new(vec![
        Box::new(OpenAIProvider::new(primary_config)),
        Box::new(OllamaProvider::new(fallback_config)),
    ]);

    let messages = vec![Message::user("Hello")];
    let options = ChatOptions::default();

    // Should fallback to Ollama when OpenAI fails
    let response = fallback_chain.chat_completion(&messages, &options).await;
    assert!(response.is_ok() || response.is_err()); // Depends on Ollama availability
}
```

Configuration Integration Tests

```rust
// tests/integration/config_tests.rs
use perspt::*;
use tempfile::TempDir;
use std::fs;

#[test]
fn test_config_file_hierarchy() {
    let temp_dir = TempDir::new().unwrap();

    // Create multiple config files
    let global_config = temp_dir.path().join("global.json");
    let user_config = temp_dir.path().join("user.json");
    let local_config = temp_dir.path().join("local.json");

    fs::write(&global_config, r#"{"provider": "openai", "temperature": 0.5}"#).unwrap();
    fs::write(&user_config, r#"{"model": "gpt-4", "temperature": 0.7}"#).unwrap();
    fs::write(&local_config, r#"{"api_key": "local-key"}"#).unwrap();

    let mut config = Config::new();
    config.load_from_file(&global_config).unwrap();
    config.load_from_file(&user_config).unwrap();
    config.load_from_file(&local_config).unwrap();

    assert_eq!(config.provider, "openai");
    assert_eq!(config.model, "gpt-4");
    assert_eq!(config.api_key, Some("local-key".to_string()));
    assert_eq!(config.temperature, Some(0.7)); // user config overrides global
```

(continues on next page)

(continued from previous page)

```
}

#[tokio::test]
async fn test_config_validation_with_providers() {
    let config = Config {
        provider: "openai".to_string(),
        api_key: Some("sk-test123".to_string()),
        model: "gpt-4".to_string(),
        ..Default::default()
    };

    let provider_registry = ProviderRegistry::new();
    let validation_result = provider_registry.validate_config(&config).await;

    assert!(validation_result.is_ok());
}
```

End-to-End Testing

Full Conversation Flow

```
// tests/e2e/full_conversation_test.rs
use perspt::*;
use std::time::Duration;
use tokio::time::timeout;

#[tokio::test]
async fn test_complete_conversation_flow() {
    let config = Config::test_config();
    let mut app = Application::new(config).await.unwrap();

    // Start the application
    let app_handle = tokio::spawn(async move {
        app.run().await
    });

    // Simulate user input
    let mut client = TestClient::new("localhost:8080").await.unwrap();

    // Send first message
    let response1 = client.send_message("Hello, I'm testing Perspt").await.unwrap();
    assert!(!response1.is_empty());

    // Send follow-up message
    let response2 = client.send_message("Can you remember what I just said?").await.
↪unwrap();
    assert!(response2.to_lowercase().contains("testing") ||
            response2.to_lowercase().contains("perspt"));

    // Test command
    let response3 = client.send_command("/status").await.unwrap();
    assert!(response3.contains("Connected"));
```

(continues on next page)

(continued from previous page)

```rust
    // Cleanup
    client.send_command("/exit").await.unwrap();

    // Wait for app to shutdown
    timeout(Duration::from_secs(5), app_handle).await.unwrap().unwrap();
}

#[tokio::test]
async fn test_error_recovery() {
    let mut config = Config::test_config();
    config.api_key = Some("invalid-key".to_string());

    let mut app = Application::new(config).await.unwrap();
    let mut client = TestClient::new("localhost:8080").await.unwrap();

    // This should fail with invalid key
    let response = client.send_message("Hello").await;
    assert!(response.is_err());

    // Update config with valid key
    client.send_command("/config set api_key valid-key").await.unwrap();

    // This should now work
    let response = client.send_message("Hello").await.unwrap();
    assert!(!response.is_empty());
}
```

Plugin Integration Tests

```rust
// tests/e2e/plugin_integration_test.rs
use perspt::*;
use std::path::Path;

#[tokio::test]
async fn test_plugin_loading_and_execution() {
    let config = Config::test_config();
    let mut app = Application::new(config).await.unwrap();

    // Load a test plugin
    let plugin_path = Path::new("test_plugins/file_processor.so");
    if plugin_path.exists() {
        app.load_plugin(plugin_path).await.unwrap();

        let mut client = TestClient::new("localhost:8080").await.unwrap();

        // Test plugin command
        let response = client.send_command("/read-file test.txt").await.unwrap();
        assert!(response.contains("File content"));

        // Test plugin with invalid args
```

(continues on next page)

(continued from previous page)

```
        let response = client.send_command("/read-file").await;
        assert!(response.is_err());
    }
}
```

UI and Command Testing

Added in v0.4.3 - Testing user interface components and command functionality:

```
// src/ui.rs - Unit tests for UI components
#[cfg(test)]
mod tests {
    use super::*;
    use tempfile::TempDir;
    use std::fs;

    #[test]
    fn test_save_conversation_command() {
        let mut app = App::new_for_testing();

        // Add some test messages
        app.add_message(ChatMessage {
            message_type: MessageType::User,
            content: vec![Line::from("Hello, AI!")],
            timestamp: "2024-01-01 12:00:00".to_string(),
            raw_content: "Hello, AI!".to_string(),
        });

        app.add_message(ChatMessage {
            message_type: MessageType::Assistant,
            content: vec![Line::from("Hello! How can I help you?")],
            timestamp: "2024-01-01 12:00:01".to_string(),
            raw_content: "Hello! How can I help you?".to_string(),
        });

        // Test save with custom filename
        let temp_dir = TempDir::new().unwrap();
        let save_path = temp_dir.path().join("test_conversation.txt");
        let filename = save_path.to_string_lossy().to_string();

        let result = app.save_conversation(Some(filename.clone()));
        assert!(result.is_ok());
        assert_eq!(result.unwrap(), filename);

        // Verify file contents
        let content = fs::read_to_string(&save_path).unwrap();
        assert!(content.contains("Perspt Conversation"));
        assert!(content.contains("User: Hello, AI!"));
        assert!(content.contains("Assistant: Hello! How can I help you?"));
    }

    #[test]
```

(continues on next page)

(continued from previous page)

```
    fn test_command_handling() {
        let mut app = App::new_for_testing();

        // Add a test conversation
        app.add_message(ChatMessage {
            message_type: MessageType::User,
            content: vec![Line::from("Hello")],
            timestamp: "2024-01-01 12:00:00".to_string(),
            raw_content: "Hello".to_string(),
        });

        // Test /save command
        let result = app.handle_command("/save test.txt".to_string());
        assert!(result.is_ok());
        assert_eq!(result.unwrap(), true); // Command was handled

        // Clean up
        let _ = fs::remove_file("test.txt");
    }

    impl App {
        fn new_for_testing() -> Self {
            let config = crate::config::AppConfig {
                provider_type: Some("test".to_string()),
                api_key: Some("test-key".to_string()),
                default_model: "test-model".to_string(),
                ..Default::default()
            };
            Self::new(config)
        }
    }
}
```

Performance Testing

Benchmark Configuration

```
// benches/provider_benchmarks.rs
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use perspt::*;
use tokio::runtime::Runtime;

fn bench_openai_provider(c: &mut Criterion) {
    let rt = Runtime::new().unwrap();
    let config = OpenAIConfig::test_config();
    let provider = OpenAIProvider::new(config);

    c.bench_function("openai_chat_completion", |b| {
        b.to_async(&rt).iter(|| async {
            let messages = vec![Message::user("Hello")];
            let options = ChatOptions::default();

```

(continues on next page)

(continued from previous page)

```rust
            black_box(
                provider.chat_completion(&messages, &options).await.unwrap()
            )
        })
    });
}

fn bench_config_loading(c: &mut Criterion) {
    c.bench_function("config_load", |b| {
        b.iter(|| {
            let config = Config::load_from_string(black_box(r#"
                {
                    "provider": "openai",
                    "model": "gpt-4",
                    "api_key": "test-key"
                }
            "#)).unwrap();
            black_box(config)
        })
    });
}

criterion_group!(benches, bench_openai_provider, bench_config_loading);
criterion_main!(benches);
```

Memory and Resource Testing

```rust
#[tokio::test]
async fn test_memory_usage() {
    let initial_memory = get_memory_usage();

    let config = Config::test_config();
    let mut app = Application::new(config).await.unwrap();

    // Simulate long conversation
    for i in 0..1000 {
        let message = format!("Test message {}", i);
        app.process_message(&message).await.unwrap();
    }

    let final_memory = get_memory_usage();
    let memory_increase = final_memory - initial_memory;

    // Memory increase should be reasonable (less than 100MB for 1000 messages)
    assert!(memory_increase < 100 * 1024 * 1024);
}

fn get_memory_usage() -> usize {
    // Platform-specific memory measurement
    #[cfg(target_os = "linux")]
    {
```

(continues on next page)

(continued from previous page)

```
        use std::fs;
        let status = fs::read_to_string("/proc/self/status").unwrap();
        for line in status.lines() {
            if line.starts_with("VmRSS:") {
                let kb: usize = line.split_whitespace().nth(1).unwrap().parse().unwrap();
                return kb * 1024;
            }
        }
        0
    }

    #[cfg(not(target_os = "linux"))]
    {
        // Placeholder for other platforms
        0
    }
}
```

Security Testing

Input Validation Testing

```
#[tokio::test]
async fn test_input_sanitization() {
    let sanitizer = InputSanitizer::new();

    // Test potential XSS
    let malicious_input = "<script>alert('xss')</script>";
    let sanitized = sanitizer.sanitize(malicious_input);
    assert!(!sanitized.contains("<script>"));

    // Test SQL injection patterns
    let sql_injection = "'; DROP TABLE users; --";
    let sanitized = sanitizer.sanitize(sql_injection);
    assert!(!sanitized.contains("DROP TABLE"));

    // Test excessive length
    let long_input = "a".repeat(100_000);
    let sanitized = sanitizer.sanitize(&long_input);
    assert!(sanitized.len() <= 10_000); // Should be truncated
}

#[tokio::test]
async fn test_api_key_security() {
    let config = Config {
        api_key: Some("sk-super-secret-key".to_string()),
        ..Default::default()
    };

    // Ensure API key doesn't appear in logs
    let log_output = capture_logs(|| {
        log::info!("Config loaded: {:?}", config);
```

(continues on next page)

(continued from previous page)

```
    });

    assert!(!log_output.contains("sk-super-secret-key"));
    assert!(log_output.contains("[REDACTED]"));
}
```

Testing Utilities

Test Fixtures

```
// tests/common/fixtures.rs
pub struct TestFixtures;

impl TestFixtures {
    pub fn sample_config() -> Config {
        Config {
            provider: "test".to_string(),
            model: "test-model".to_string(),
            api_key: Some("test-key".to_string()),
            max_tokens: Some(100),
            temperature: Some(0.5),
            ..Default::default()
        }
    }

    pub fn sample_messages() -> Vec<Message> {
        vec![
            Message::user("Hello"),
            Message::assistant("Hi there! How can I help you?"),
            Message::user("What's the weather like?"),
        ]
    }

    pub fn sample_chat_response() -> ChatResponse {
        ChatResponse {
            content: "It's sunny today!".to_string(),
            tokens_used: Some(15),
            model: "test-model".to_string(),
            finish_reason: Some("stop".to_string()),
        }
    }
}
```

Mock Implementations

```
// tests/common/mocks.rs
pub struct MockLLMProvider {
    responses: Vec<String>,
    call_count: std::sync::Arc<std::sync::Mutex<usize>>,
}

```

(continues on next page)

(continued from previous page)

```rust
impl MockLLMProvider {
    pub fn new(responses: Vec<String>) -> Self {
        Self {
            responses,
            call_count: std::sync::Arc::new(std::sync::Mutex::new(0)),
        }
    }

    pub fn call_count(&self) -> usize {
        *self.call_count.lock().unwrap()
    }
}

#[async_trait]
impl LLMProvider for MockLLMProvider {
    async fn chat_completion(
        &self,
        _messages: &[Message],
        _options: &ChatOptions,
    ) -> Result<ChatResponse, LLMError> {
        let mut count = self.call_count.lock().unwrap();
        let response_index = *count % self.responses.len();
        *count += 1;

        Ok(ChatResponse {
            content: self.responses[response_index].clone(),
            tokens_used: Some(10),
            model: "mock".to_string(),
            finish_reason: Some("stop".to_string()),
        })
    }
}
```

Test Configuration

Cargo.toml Test Dependencies

```toml
[dev-dependencies]
tokio-test = "0.4"
mockall = "0.11"
criterion = "0.5"
tempfile = "3.0"
serde_json = "1.0"
env_logger = "0.10"

[[bench]]
name = "provider_benchmarks"
harness = false

[[bench]]
name = "ui_benchmarks"
harness = false
```

Running Tests

```
# Run all tests
cargo test

# Run unit tests only
cargo test --lib

# Run integration tests only
cargo test --test '*'

# Run specific test
cargo test test_openai_provider

# Run tests with output
cargo test -- --nocapture

# Run tests with specific thread count
cargo test -- --test-threads=1

# Run ignored tests (integration tests requiring API keys)
cargo test -- --ignored

# Run benchmarks
cargo bench

# Generate test coverage report
cargo tarpaulin --out Html
```

Continuous Integration

GitHub Actions Configuration

```
# .github/workflows/test.yml
name: Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Install Rust
      uses: actions-rs/toolchain@v1
      with:
        toolchain: stable
```

(continues on next page)

(continued from previous page)

```
        components: rustfmt, clippy

    - name: Check formatting
      run: cargo fmt --check

    - name: Run clippy
      run: cargo clippy -- -D warnings

    - name: Run unit tests
      run: cargo test --lib

    - name: Run integration tests
      run: cargo test --test '*'
      env:
        RUST_LOG: debug

    - name: Generate coverage report
      run: |
        cargo install cargo-tarpaulin
        cargo tarpaulin --out xml

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3
```

Best Practices

Testing Guidelines

1. **Test Isolation**: Each test should be independent
2. **Clear Naming**: Test names should describe what they verify
3. **Comprehensive Coverage**: Aim for high code coverage
4. **Fast Execution**: Unit tests should run quickly
5. **Reliable Results**: Tests should be deterministic
6. **Error Testing**: Test error conditions and edge cases

Performance Testing Guidelines

1. **Baseline Measurements**: Establish performance baselines
2. **Regression Detection**: Catch performance regressions early
3. **Resource Monitoring**: Monitor memory and CPU usage
4. **Load Testing**: Test under realistic load conditions

Next Steps

- *Contributing* - Contribution guidelines and development setup
- *Architecture* - Understanding the codebase for better testing
- *Extending Perspt* - Testing custom plugins and extensions
- *API Reference* - API reference for testing integration points

Testing Simple CLI Mode

NEW in v0.4.5 - Comprehensive testing for the Simple CLI mode requires specific strategies:

Unit Tests for CLI Module:

```
// In src/cli.rs - Unit tests
#[cfg(test)]
mod tests {
    use super::*;
    use std::io::{self, Cursor};
    use tokio::sync::mpsc;

    #[tokio::test]
    async fn test_simple_cli_input_processing() {
        let input = "What is Rust?";
        let (tx, mut rx) = mpsc::unbounded_channel();

        // Mock provider response
        let mock_provider = create_mock_provider();

        let result = process_simple_request(input, "test-model", &mock_provider).await;
        assert!(result.is_ok());

        // Verify streaming response collection
        let response = result.unwrap();
        assert!(!response.is_empty());
    }

    #[test]
    fn test_session_logger_creation() {
        let temp_file = std::env::temp_dir().join("test_session.txt");
        let logger = SessionLogger::new(temp_file.to_string_lossy().to_string());
        assert!(logger.is_ok());

        // Cleanup
        let _ = std::fs::remove_file(temp_file);
    }

    #[test]
    fn test_cli_command_parsing() {
        assert!(is_exit_command("exit"));
        assert!(is_exit_command("EXIT"));
        assert!(!is_exit_command("exit please"));
        assert!(!is_exit_command(""));
    }

    #[tokio::test]
    async fn test_streaming_response_collection() {
        let (tx, mut rx) = mpsc::unbounded_channel();

        // Simulate streaming chunks
        tx.send("Hello ".to_string()).unwrap();
        tx.send("world!".to_string()).unwrap();
```

(continues on next page)

(continued from previous page)

```
        tx.send("<<EOT>>".to_string()).unwrap();
        drop(tx);

        let mut response = String::new();
        while let Some(chunk) = rx.recv().await {
            if chunk == "<<EOT>>" { break; }
            response.push_str(&chunk);
        }

        assert_eq!(response, "Hello world!");
    }

    fn create_mock_provider() -> Arc<MockGenAIProvider> {
        // Create mock provider for testing
        Arc::new(MockGenAIProvider::new())
    }

    fn is_exit_command(input: &str) -> bool {
        input.trim().to_lowercase() == "exit"
    }
}
```

Integration Tests for CLI Workflows:

```
// tests/cli_integration_tests.rs
use perspt::cli::run_simple_cli;
use perspt::config::AppConfig;
use std::process::{Command, Stdio};
use std::io::Write;
use tempfile::NamedTempFile;

#[tokio::test]
async fn test_simple_cli_session_logging() {
    let log_file = NamedTempFile::new().unwrap();
    let log_path = log_file.path().to_string_lossy().to_string();

    // Create test configuration
    let config = AppConfig {
        provider_type: Some("openai".to_string()),
        api_key: Some("test-key".to_string()),
        default_model: Some("gpt-3.5-turbo".to_string()),
        // ... other fields
    };

    // Simulate CLI session with scripted input
    let script = "Hello\nexit\n";
    let mut child = Command::new("target/debug/perspt")
        .args(&["--simple-cli", "--log-file", &log_path])
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .stderr(Stdio::piped())
        .spawn()
```

(continues on next page)

(continued from previous page)

```
        .expect("Failed to start perspt");

    if let Some(stdin) = child.stdin.as_mut() {
        stdin.write_all(script.as_bytes()).unwrap();
    }

    let output = child.wait_with_output().unwrap();
    assert!(output.status.success());

    // Verify log file contents
    let log_contents = std::fs::read_to_string(&log_path).unwrap();
    assert!(log_contents.contains("User: Hello"));
    assert!(log_contents.contains("Assistant:"));
}

#[test]
fn test_cli_argument_parsing() {
    let output = Command::new("target/debug/perspt")
        .args(&["--simple-cli", "--help"])
        .output()
        .expect("Failed to execute perspt");

    assert!(output.status.success());
    let stdout = String::from_utf8(output.stdout).unwrap();
    assert!(stdout.contains("simple-cli"));
    assert!(stdout.contains("log-file"));
}

#[test]
fn test_exit_command_handling() {
    let output = Command::new("target/debug/perspt")
        .args(&["--simple-cli"])
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .stderr(Stdio::piped())
        .spawn()
        .expect("Failed to start perspt");

    // Send exit command
    if let Some(stdin) = output.stdin.as_mut() {
        stdin.write_all(b"exit\n").unwrap();
    }

    let result = output.wait_with_output().unwrap();
    assert!(result.status.success());

    let stdout = String::from_utf8(result.stdout).unwrap();
    assert!(stdout.contains("Goodbye!"));
}
```

Scripting Tests:

```
#!/bin/bash
# tests/test_cli_scripting.sh

set -e

echo "Testing Simple CLI scripting capabilities..."

# Test basic input/output
echo "What is 2+2?" | timeout 30s target/debug/perspt --simple-cli > /tmp/test_output.txt

if grep -q "4" /tmp/test_output.txt; then
    echo "▯ Basic math test passed"
else
    echo "▯ Basic math test failed"
    exit 1
fi

# Test session logging
LOG_FILE="/tmp/test_session_$(date +%s).txt"
echo -e "Hello\nexit" | timeout 30s target/debug/perspt --simple-cli --log-file "$LOG_FILE"

if [ -f "$LOG_FILE" ] && grep -q "User: Hello" "$LOG_FILE"; then
    echo "▯ Session logging test passed"
else
    echo "▯ Session logging test failed"
    exit 1
fi

# Test piping multiple questions
{
    echo "What is machine learning?"
    echo "Give a brief example"
    echo "exit"
} | timeout 60s target/debug/perspt --simple-cli --log-file "/tmp/multi_test.txt"

if grep -q "machine learning" /tmp/multi_test.txt; then
    echo "▯ Multi-question test passed"
else
    echo "▯ Multi-question test failed"
    exit 1
fi

echo "All CLI scripting tests passed!"
```

Accessibility Testing:

```
// tests/accessibility_tests.rs
use std::process::{Command, Stdio};
use std::io::Write;

#[test]
fn test_screen_reader_compatibility() {
    // Test that Simple CLI output is screen reader friendly
```

(continues on next page)

(continued from previous page)

```
    let mut child = Command::new("target/debug/perspt")
        .args(&["--simple-cli"])
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .stderr(Stdio::piped())
        .spawn()
        .expect("Failed to start perspt");

    if let Some(stdin) = child.stdin.as_mut() {
        stdin.write_all(b"Hello\nexit\n").unwrap();
    }

    let output = child.wait_with_output().unwrap();
    let stdout = String::from_utf8(output.stdout).unwrap();

    // Verify output contains clear prompt markers
    assert!(stdout.contains("> "));
    assert!(stdout.contains("Perspt Simple CLI Mode"));
    assert!(stdout.contains("Type 'exit' or press Ctrl+D to quit"));

    // Ensure no ANSI escape codes that might confuse screen readers
    assert!(!stdout.contains("\x1b["));
}

#[test]
fn test_keyboard_navigation() {
    // Test that common accessibility keyboard patterns work
    let test_inputs = vec![
        "\n",           // Empty input handling
        "   \n",        // Whitespace handling
        "\x04",         // Ctrl+D (EOF)
        "exit\n",       // Standard exit
    ];

    for input in test_inputs {
        let output = Command::new("target/debug/perspt")
            .args(&["--simple-cli"])
            .stdin(Stdio::piped())
            .stdout(Stdio::piped())
            .stderr(Stdio::piped())
            .spawn()
            .expect("Failed to start perspt");

        if let Some(stdin) = output.stdin.as_mut() {
            stdin.write_all(input.as_bytes()).unwrap();
        }

        let result = output.wait_with_output().unwrap();
        // Should handle all inputs gracefully without crashing
        assert!(result.status.success() || result.status.code() == Some(0));
    }
}
```

Performance Tests for CLI Mode:

```
// benches/cli_benchmarks.rs
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use perspt::cli::SessionLogger;
use std::time::Instant;

fn benchmark_session_logging(c: &mut Criterion) {
    c.bench_function("session_logging", |b| {
        let temp_file = std::env::temp_dir().join("bench_session.txt");
        let mut logger = SessionLogger::new(temp_file.to_string_lossy().to_string()).
↪unwrap();

        b.iter(|| {
            logger.log_user_input(black_box("Test input message")).unwrap();
            logger.log_ai_response(black_box("Test AI response")).unwrap();
        });

        let _ = std::fs::remove_file(temp_file);
    });
}

fn benchmark_input_processing(c: &mut Criterion) {
    c.bench_function("input_processing", |b| {
        b.iter(|| {
            let input = black_box("What is quantum computing?");
            // Benchmark input validation and sanitization
            let sanitized = input.trim().to_string();
            sanitized
        });
    });
}

criterion_group!(benches, benchmark_session_logging, benchmark_input_processing);
criterion_main!(benches);
```

Mock Provider for Testing:

```
// In src/llm_provider.rs - Mock provider for testing
#[cfg(test)]
pub struct MockGenAIProvider {
    responses: Vec<String>,
    current_index: std::sync::atomic::AtomicUsize,
}

#[cfg(test)]
impl MockGenAIProvider {
    pub fn new() -> Self {
        Self {
            responses: vec![
                "Hello! How can I help you?".to_string(),
                "That's a great question!".to_string(),
                "I'm happy to assist with that.".to_string(),
            ],
```

(continues on next page)

(continued from previous page)

```
            current_index: std::sync::atomic::AtomicUsize::new(0),
        }
    }

    pub async fn generate_response_stream_to_channel(
        &self,
        _model: &str,
        _prompt: &str,
        tx: tokio::sync::mpsc::UnboundedSender<String>,
    ) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
        let index = self.current_index.fetch_add(1, std::sync::atomic::Ordering::SeqCst);
        let response = &self.responses[index % self.responses.len()];

        // Simulate streaming by sending chunks
        for chunk in response.split_whitespace() {
            tx.send(format!("{} ", chunk))?;
            tokio::time::sleep(tokio::time::Duration::from_millis(10)).await;
        }

        tx.send("<<EOT>>".to_string())?;
        Ok(())
    }
}
```

Quick Navigation

- *Overview*
- *Project Structure*
- *Core Technologies*
  - *Technology Stack*
  - *Key Dependencies*
- *Design Principles*
  - *Performance First*
  - *Safety and Reliability*
  - *Modularity and Extensibility*
- *Development Environment Setup*
  - *Prerequisites*
  - *Clone and Setup*
  - *Development Workflow*
- *Code Organization*
  - *Module Structure*
  - *Design Patterns*
- *Testing Strategy*
  - *Unit Tests*
  - *Integration Tests*
  - *Performance Benchmarks*
- *Contributing Guidelines*
  - *Code Style*
  - *Git Workflow*
  - *Pull Request Process*

- *Release Process*
  - *Version Management*
  - *Release Checklist*
- *Documentation*
  - *Code Documentation*
  - *API Documentation*
- *Community and Support*
  - *Getting Help*
  - *Contributing Areas*
- *Next Steps*

3.6.5 Overview

Perspt is built with modern Rust practices, emphasizing performance, safety, and maintainability. The codebase is designed to be modular, testable, and easy to extend.

🏗□ Architecture Deep dive into Perspt's design patterns, module structure, and core principles.

Architecture □ Contributing Guidelines for contributing code, documentation, and reporting issues.

Contributing 🔧 Extending How to add new providers, features, and customize Perspt for your needs.

Extending Perspt □ Testing Testing strategies, test writing guidelines, and continuous integration.

Testing

3.6.6 Project Structure

```
perspt/
├── src/
│   ├── main.rs          # Entry point, CLI parsing, panic handling
│   ├── cli.rs           # Simple CLI mode implementation (NEW in v0.4.5)
│   ├── config.rs        # Configuration management and validation
│   ├── llm_provider.rs  # GenAI provider abstraction and implementation
│   └── ui.rs            # Terminal UI with Ratatui and real-time streaming
├── tests/
│   └── panic_handling_test.rs  # Integration tests
├── docs/
│   ├── perspt_book/     # Sphinx documentation
│   └── *.html           # Asset library and design system
├── Cargo.toml           # Dependencies and project metadata
└── config.json.example  # Sample configuration
```

3.6.7 Core Technologies

Technology Stack

Component	Technology & Purpose
LLM Integration	genai v0.3.5 - Unified interface for multiple LLM providers
Async Runtime	tokio v1.42 - High-performance async runtime
Terminal UI	ratatui v0.29 - Modern terminal user interface framework
Cross-platform Terminal	crossterm v0.28 - Cross-platform terminal manipulation
CLI Framework	clap v4.5 - Command line argument parser
Configuration	serde v1.0 + serde_json v1.0 - Serialization framework
Markdown Rendering	pulldown-cmark v0.12 - CommonMark markdown parser
Error Handling	anyhow v1.0 - Flexible error handling
Async Traits	async-trait v0.1.88 - Async functions in traits
Logging	log v0.4 + env_logger v0.11 - Structured logging
Streaming	futures v0.3 - Utilities for async programming

Key Dependencies

```toml
[dependencies]
# LLM unified interface - using genai for better model support
genai = "0.3.5"
futures = "0.3"

# Core async and traits
async-trait = "0.1.88"
tokio = { version = "1.42", features = ["full"] }

# CLI and configuration
clap = { version = "4.5", features = ["derive"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

# UI components
ratatui = "0.29"
crossterm = "0.28"
pulldown-cmark = "0.12"

# Logging
log = "0.4"
env_logger = "0.11"

# Utilities
anyhow = "1.0"

# CLI and configuration
clap = { version = "4.5", features = ["derive"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

# UI components
ratatui = "0.29"
```

(continues on next page)

(continued from previous page)

```
crossterm = "0.28"
pulldown-cmark = "0.12"

# Logging and error handling
log = "0.4"
env_logger = "0.11"
anyhow = "1.0"

[dependencies]
tokio = { version = "1.0", features = ["full"] }
ratatui = "0.26"
crossterm = "0.27"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
genai = "0.3.5"
clap = { version = "4.0", features = ["derive"] }
anyhow = "1.0"
thiserror = "1.0"
```

3.6.8 Design Principles

Performance First

Every design decision prioritizes performance:

- **Zero-copy operations** where possible
- **Efficient memory usage** with careful allocation
- **Streaming responses** for immediate user feedback
- **Minimal dependencies** to reduce compile time and binary size

Safety and Reliability

Rust’s type system ensures memory safety and prevents common errors:

- **No null pointer dereferences** through Option types
- **Thread safety** with Send and Sync traits
- **Error handling** with Result types throughout
- **Resource management** with RAII patterns

Modularity and Extensibility

The architecture supports easy extension and modification:

- **Trait-based abstractions** for provider independence
- **Configuration-driven behavior** without code changes
- **Plugin-ready architecture** for future extensions
- **Clear module boundaries** with well-defined interfaces

3.6.9 Development Environment Setup

Prerequisites

```
# Install Rust toolchain
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

(continues on next page)

(continued from previous page)

```
source $HOME/.cargo/env

# Install development tools
cargo install cargo-watch cargo-edit cargo-audit

# Install clippy and rustfmt
rustup component add clippy rustfmt

# Verify installation
rustc --version
cargo --version
```

Clone and Setup

```
# Clone repository
git clone https://github.com/eonseed/perspt.git
cd perspt

# Install dependencies
cargo build

# Run tests
cargo test

# Check code quality
cargo clippy
cargo fmt --check
```

Development Workflow

```
# Watch mode for development
cargo watch -x 'run -- --help'

# Run specific tests
cargo test test_config

# Run with debug output
RUST_LOG=debug cargo run

# Profile performance
cargo build --release
perf record target/release/perspt
perf report
```

3.6.10 Code Organization

Module Structure

Each module has a specific responsibility:

main.rs

- Application entry point and orchestration

- CLI argument parsing with clap derive macros
- Comprehensive panic handling with terminal restoration
- Terminal initialization and cleanup with crossterm
- Configuration loading and provider initialization
- Real-time event loop with enhanced responsiveness

config.rs

- JSON-based configuration system with serde
- Multi-provider support with intelligent defaults
- Automatic provider type inference
- Environment variable integration
- Configuration validation and fallbacks

llm_provider.rs

- GenAI crate integration for unified LLM access
- Support for OpenAI, Anthropic, Google (Gemini), Groq, Cohere, XAI, DeepSeek, Ollama
- Streaming response handling with proper event processing
- Model validation and discovery
- Comprehensive error categorization and recovery

ui.rs

- Ratatui-based terminal user interface
- Real-time markdown rendering with pulldown-cmark
- Responsive layout with scrollable chat history
- Enhanced keyboard input handling and cursor management
- Progress indicators and error display
- Help system with keyboard shortcuts

Design Patterns

GenAI Provider Architecture:

```rust
use genai::{Client, chat::{ChatRequest, ChatMessage}};
use futures::StreamExt;

pub struct GenAIProvider {
    client: Client,
}

impl GenAIProvider {
    pub fn new() -> Result<Self> {
        let client = Client::default();
        Ok(Self { client })
    }

    pub async fn generate_response_stream_to_channel(
        &self,
        model: &str,
        prompt: &str,
        tx: mpsc::UnboundedSender<String>
    ) -> Result<()> {
        let chat_req = ChatRequest::default()
            .append_message(ChatMessage::user(prompt));
```

(continues on next page)

(continued from previous page)

```
        let chat_res_stream = self.client
            .exec_chat_stream(model, chat_req, None)
            .await?;

        let mut stream = chat_res_stream.stream;
        while let Some(chunk_result) = stream.next().await {
            match chunk_result? {
                ChatStreamEvent::Chunk(chunk) => {
                    tx.send(chunk.content)?;
                }
                ChatStreamEvent::End(_) => break,
                _ => {}
            }
        }
        Ok(())
    }
}
```

Error Handling Strategy:

```
use anyhow::{Context, Result};
use thiserror::Error;

#[derive(Error, Debug)]
pub enum PersptError {
    #[error("Configuration error: {0}")]
    Config(String),

    #[error("Provider error: {0}")]
    Provider(#[from] genai::GenAIError),

    #[error("UI error: {0}")]
    Ui(String),

    #[error("Network error: {0}")]
    Network(String),
}

// Example error handling in main
fn setup_panic_hook() {
    panic::set_hook(Box::new(move |panic_info| {
        // Force terminal restoration immediately
        let _ = disable_raw_mode();
        let _ = execute!(io::stdout(), LeaveAlternateScreen);

        // Provide contextual error messages
        let panic_str = format!("{}", panic_info);
        if panic_str.contains("PROJECT_ID") {
            eprintln!("□ Tip: Set PROJECT_ID environment variable");
        }
        // ... more context-specific help
    }));
```

(continues on next page)

(continued from previous page)

```
}

#[derive(Error, Debug)]
pub enum ProviderError {
    #[error("Network error: {0}")]
    Network(#[from] reqwest::Error),

    #[error("API error: {message}")]
    Api { message: String },

    #[error("Configuration error: {0}")]
    Config(String),
}
```

Configuration Pattern:

```
use serde::Deserialize;
use std::collections::HashMap;

#[derive(Debug, Clone, Deserialize, PartialEq)]
pub struct AppConfig {
    pub providers: HashMap<String, String>,
    pub api_key: Option<String>,
    pub default_model: Option<String>,
    pub default_provider: Option<String>,
    pub provider_type: Option<String>,
}

pub async fn load_config(config_path: Option<&String>) -> Result<AppConfig> {
    let config: AppConfig = match config_path {
        Some(path) => {
            let config_str = fs::read_to_string(path)?;
            let initial_config: AppConfig = serde_json::from_str(&config_str)?;
            process_loaded_config(initial_config)
        }
        None => {
            // Comprehensive defaults with all supported providers
            let mut providers_map = HashMap::new();
            providers_map.insert("openai".to_string(),
                "https://api.openai.com/v1".to_string());
            providers_map.insert("anthropic".to_string(),
                "https://api.anthropic.com".to_string());
            // ... more providers

            AppConfig {
                providers: providers_map,
                api_key: None,
                default_model: Some("gpt-4o-mini".to_string()),
                default_provider: Some("openai".to_string()),
                provider_type: Some("openai".to_string()),
            }
        }
```

(continues on next page)

(continued from previous page)

```
    };
    Ok(config)
}

#[derive(Debug, Deserialize)]
pub struct AppConfig {
    #[serde(default)]
    pub api_key: Option<String>,

    #[serde(default = "default_model")]
    pub default_model: String,
}

fn default_model() -> String {
    "gpt-4o-mini".to_string()
}
```

3.6.11 Testing Strategy

Unit Tests

Each module includes comprehensive unit tests:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_config_parsing() {
        let json = r#"{"api_key": "test"}"#;
        let config: AppConfig = serde_json::from_str(json).unwrap();
        assert_eq!(config.api_key, Some("test".to_string()));
    }

    #[tokio::test]
    async fn test_provider_request() {
        // Mock provider tests
    }
}
```

Integration Tests

Full end-to-end testing in the *tests/* directory:

```
#[tokio::test]
async fn test_full_conversation_flow() {
    // Test complete conversation workflow
}
```

Performance Benchmarks

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn benchmark_config_parsing(c: &mut Criterion) {
    c.bench_function("config parsing", |b| {
        b.iter(|| {
            // Benchmark configuration parsing
        });
    });
}
```

3.6.12 Contributing Guidelines

Code Style

We follow standard Rust conventions:

```
# Format code
cargo fmt

# Check linting
cargo clippy -- -D warnings

# Check documentation
cargo doc --no-deps
```

Git Workflow

```
# Create feature branch
git checkout -b feature/new-provider

# Make changes and commit
git add .
git commit -m "feat: add support for new provider"

# Push and create PR
git push origin feature/new-provider
```

Pull Request Process

1. **Fork** the repository
2. **Create** a feature branch
3. **Write** tests for your changes
4. **Ensure** all tests pass
5. **Submit** a pull request with clear description

3.6.13 Release Process

Version Management

We use semantic versioning (SemVer):

- **MAJOR**: Breaking changes

- **MINOR**: New features, backward compatible
- **PATCH**: Bug fixes, backward compatible

Release Checklist

```
# Update version in Cargo.toml
# Update CHANGELOG.md
# Run full test suite
cargo test --all

# Build release
cargo build --release

# Create git tag
git tag v0.4.0
git push origin v0.4.0

# Publish to crates.io
cargo publish
```

3.6.14 Documentation

Code Documentation

Use Rust doc comments extensively:

```
/// Sends a chat request to the LLM provider.
///
/// # Arguments
///
/// * `input` - The user's message
/// * `model` - The model to use for the request
/// * `config` - Application configuration
/// * `tx` - Channel for streaming responses
///
/// # Returns
///
/// A `Result` indicating success or failure
///
/// # Errors
///
/// Returns `ProviderError` if the request fails
pub async fn send_chat_request(
    &self,
    input: &str,
    model: &str,
    config: &AppConfig,
    tx: &Sender<String>
) -> Result<()> {
    // Implementation
}
```

API Documentation

Generate documentation:

```
# Generate and open docs
cargo doc --open --no-deps

# Generate docs with private items
cargo doc --document-private-items
```

3.6.15 Community and Support

Getting Help

- **GitHub Issues**: Bug reports and feature requests
- **GitHub Discussions**: Questions and community chat
- **Discord**: Real-time development discussion
- **Documentation**: This guide and API docs

Contributing Areas

We welcome contributions in:

- **Code**: New features, bug fixes, optimizations
- **Documentation**: Guides, examples, API docs
- **Testing**: Unit tests, integration tests, benchmarks
- **Design**: UI/UX improvements, accessibility
- **Community**: Helping users, writing tutorials

3.6.16 Next Steps

Ready to dive deeper? Choose your path:

🏗□ Architecture Deep Dive Understand the internal design and implementation details.

Architecture □ Start Contributing Learn how to contribute code, documentation, or help the community.

Contributing 🔧 Extend Functionality Add new providers, features, or customize Perspt.

Extending Perspt □ Testing Guide Write tests, run benchmarks, and ensure quality.

Testing

See also

- *API Reference* - Complete API reference
- *User Guide* - User-focused documentation
- GitHub Repository - Source code and issues

3.7 API Reference

Complete API documentation for Perspt, automatically generated from source code comments and organized by module.

3.7.1 Module Overview

Perspt is organized into several focused modules, each handling specific aspects of the application. This page provides an overview of the module architecture and how they interact.

Architecture Overview

```
┌────────────────────────────────────────────────────────────────┐
│                       Perspt Architecture                      │
├────────────────────────────────────────────────────────────────┤
│                                                                │
│  ┌──────────────┐    ┌───────────────┐    ┌──────────────────┐ │
│  │    main      │────│    config     │────│   llm_provider   │ │
│  │ (Entry)      │    │ (Settings)    │    │   (AI APIs)      │ │
│  └──────────────┘    └───────────────┘    └──────────────────┘ │
│            │                                         │         │
│            ▼                                         ▼         │
│  ┌──────────────┐                         ┌──────────────────┐ │
│  │     ui       │◄────────────────────────│    External      │ │
│  │ (Interface)  │                         │     APIs         │ │
│  └──────────────┘                         └──────────────────┘ │
│                                                                │
└────────────────────────────────────────────────────────────────┘
```

Core Modules

main

Main Module

The `main` module serves as the entry point and orchestrator for the Perspt application, handling CLI argument parsing, application initialization, terminal management, and the main event loop coordination.

Overview

The main module is responsible for the complete application lifecycle, from startup to graceful shutdown. It implements comprehensive panic recovery, terminal state management, and coordinates between the UI, configuration, and LLM provider modules.

Key Responsibilities:

- **Application Bootstrap**: Initialize logging, parse CLI arguments, load configuration
- **Terminal Management**: Setup/cleanup terminal raw mode and alternate screen
- **Event Coordination**: Manage the main event loop and message passing between components
- **Error Recovery**: Comprehensive panic handling with terminal restoration
- **Resource Cleanup**: Ensure proper cleanup of terminal state and background tasks

Constants

EOT_SIGNAL

```rust
pub const EOT_SIGNAL: &str = "<<EOT>>";
```

End-of-transmission signal used throughout the application to indicate completion of streaming LLM responses.

Usage Pattern:

```
// LLM provider sends this signal when response is complete
tx.send(EOT_SIGNAL.to_string()).unwrap();

// UI receives and recognizes completion
if message == EOT_SIGNAL {
    app.finish_streaming();
}
```

Global State

TERMINAL_RAW_MODE

```
static TERMINAL_RAW_MODE: std::sync::Mutex<bool> = std::sync::Mutex::new(false);
```

Thread-safe global flag tracking terminal raw mode state for proper cleanup during panics and application crashes.

Safety Mechanism:

This global state ensures that even if the application panics or crashes unexpectedly, the panic handler can properly restore the terminal to a usable state, preventing terminal corruption for the user.

Core Functions

main()

```
#[tokio::main]
async fn main() -> Result<()>
```

Main application entry point that orchestrates the entire application lifecycle.

Returns:

- `Result<()>` - Success or application startup error

Application Lifecycle:

1. **Panic Hook Setup** - Configures comprehensive error recovery and terminal restoration
2. **Logging Initialization** - Sets up error-level logging for debugging
3. **CLI Argument Processing** - Parses command-line options with clap
4. **Configuration Management** - Loads config from file or generates intelligent defaults
5. **Provider Setup** - Initializes LLM provider with auto-configuration
6. **Model Discovery** - Optionally lists available models and exits early
7. **Terminal Initialization** - Sets up TUI with proper raw mode and alternate screen
8. **Event Loop Execution** - Runs the main UI loop with real-time responsiveness
9. **Graceful Cleanup** - Restores terminal state and releases resources

CLI Arguments Supported:

```
# Basic usage with auto-configuration
perspt

# Specify provider and model
perspt --provider-type anthropic --model-name claude-3-sonnet-20240229

# Use custom configuration file
```

(continues on next page)

(continued from previous page)

```
perspt --config /path/to/config.json

# List available models for current provider
perspt --list-models

# Override API key from command line
perspt --api-key sk-your-key-here
```

Error Scenarios:

- **Configuration Errors**: Invalid JSON, missing required fields
- **Provider Failures**: Invalid API keys, network connectivity issues
- **Terminal Issues**: Raw mode setup failures, insufficient permissions
- **Resource Constraints**: Memory limitations, file system errors

Terminal Management

setup_panic_hook()

```
fn setup_panic_hook()
```

Configures a comprehensive panic handler that ensures terminal integrity and provides helpful error messages with recovery guidance.

Recovery Actions:

1. **Immediate Terminal Restoration**: Disables raw mode and exits alternate screen
2. **Screen Cleanup**: Clears display and positions cursor appropriately
3. **Contextual Error Messages**: Provides specific guidance based on error type
4. **Clean Application Exit**: Prevents zombie processes and terminal corruption

Error Context Detection:

The panic hook intelligently detects common error scenarios:

- **Missing Environment Variables**: API keys, required configuration settings
- **Authentication Failures**: Invalid or expired API keys
- **Network Connectivity**: Connection timeouts, DNS resolution failures
- **Provider-Specific Issues**: Service outages, rate limiting

Example Error Output:

```
□ Application Error: External Library Panic
═══════════════════════════════════════════

□ Missing Google Cloud Configuration:
   Please set the PROJECT_ID environment variable
   Example: export PROJECT_ID=your-project-id

□ Troubleshooting Tips:
   - Check your provider configuration
   - Verify all required environment variables are set
   - Try a different provider (e.g., --provider-type openai)
```

set_raw_mode_flag()

```rust
fn set_raw_mode_flag(enabled: bool)
```

Thread-safe function to update the global terminal raw mode state flag.

Parameters:

- `enabled` - Whether raw mode is currently enabled

Thread Safety: Uses mutex protection to prevent race conditions during concurrent access.

initialize_terminal()

```rust
```

set_raw_mode_flag()

```rust
fn set_raw_mode_flag(enabled: bool)
```

Thread-safe function to update the global terminal raw mode state flag.

Parameters:

- `enabled` - Boolean indicating whether raw mode is currently enabled

Thread Safety:

Uses mutex protection to prevent race conditions during concurrent access. This function is called from multiple contexts:

- Main thread during terminal setup/cleanup
- Panic handler for emergency restoration
- Signal handlers for graceful shutdown

initialize_terminal()

```rust
fn initialize_terminal() -> Result<ratatui::Terminal<ratatui::backend::CrosstermBackend
↪<io::Stdout>>>
```

Initializes the terminal interface for TUI operation with comprehensive error handling and state tracking.

Returns:

- `Result<Terminal<...>>` - Configured terminal instance or initialization error

Initialization Sequence:

1. **Raw Mode Activation**: Enables character-by-character input without buffering
2. **Alternate Screen Entry**: Preserves user’s current terminal session
3. **Backend Creation**: Sets up crossterm backend for ratatui compatibility
4. **State Registration**: Updates global raw mode flag for panic recovery

Error Recovery:

If any step fails, the function automatically cleans up partial initialization to prevent terminal corruption.

cleanup_terminal()

```
fn cleanup_terminal() -> Result<()>
```

Performs comprehensive terminal cleanup and restoration to original state.

Returns:

- `Result<()>` - Success indication or cleanup error details

Restoration Process:

1. **State Flag Reset**: Updates global raw mode tracking to false
2. **Raw Mode Disable**: Restores normal terminal input behavior
3. **Alternate Screen Exit**: Returns to user's original terminal session
4. **Cursor Restoration**: Ensures cursor visibility and proper positioning

Fault Tolerance:

Each cleanup step is executed independently - if one fails, others continue to maximize terminal restoration.

Event Handling

handle_events()

```
pub async fn handle_events(
    app: &mut ui::App,
    tx_llm: &mpsc::UnboundedSender<String>,
    _api_key: &String,
    model_name: &String,
    provider: &Arc<GenAIProvider>,
) -> Option<AppEvent>
```

Processes terminal events and user input in the main application loop with real-time responsiveness.

Parameters:

- `app` - Mutable reference to application state for immediate updates
- `tx_llm` - Channel sender for LLM communication and streaming
- `_api_key` - API key for provider authentication (reserved)
- `model_name` - Current model identifier for requests
- `provider` - Arc reference to configured LLM provider

Returns:

- `Option<AppEvent>` - Some(event) for significant state changes, None for no-ops

Supported Keyboard Events:

Key Combination	Action
`Enter`	Send current input to LLM (queues if busy)
`Ctrl+C, Ctrl+Q`	Quit application gracefully
`F1`	Toggle help overlay display
`Esc`	Close help overlay or exit application
`↑/↓`	Scroll chat history up/down
`Page Up/Down`	Scroll chat history by 5 lines
`Home/End`	Jump to start/end of chat history
`Backspace`	Delete character before cursor
`Left/Right`	Move cursor in input field
`Printable chars`	Insert character at cursor position

Input Queuing System:

When the LLM is busy generating a response, user input is automatically queued and processed when the current response completes, ensuring no user input is lost.

Model and Provider Management

list_available_models()

```
async fn list_available_models(provider: &Arc<GenAIProvider>, _config: &AppConfig) ->␣
↪Result<()>
```

Discovers and displays all available models for the configured LLM provider, then exits the application.

Parameters:

- `provider` - Arc reference to the initialized LLM provider
- `_config` - Application configuration (reserved for filtering features)

Returns:

- `Result<()>` - Success or model discovery error

Output Format:

```
Available models for OpenAI:
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
✓ gpt-4o-mini                  Latest GPT-4 Optimized Mini
✓ gpt-4o                       GPT-4 Optimized
✓ gpt-4-turbo                  GPT-4 Turbo with Vision
✓ gpt-4                        Standard GPT-4
✓ gpt-3.5-turbo               GPT-3.5 Turbo
✓ o1-mini                      Reasoning Model Mini
✓ o1-preview                   Reasoning Model Preview
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
```

Provider Discovery:

Uses the `genai` crate's automatic model discovery to provide up-to-date model lists without manual maintenance.

- gpt-4o-mini
- o1-preview
- o1-mini
- o3-mini
- gpt-4-turbo

Example Usage:

```
perspt --list-models --provider-type anthropic
```

Event Handling

handle_events()

```
pub async fn handle_events(
    app: &mut ui::App,
    tx_llm: &mpsc::UnboundedSender<String>,
    _api_key: &String,
    model_name: &String,
    provider: &Arc<dyn LLMProvider + Send + Sync>,
) -> Option<AppEvent>
```

Handles terminal events and user input in the main application loop.

Parameters:

- `app` - Mutable reference to application state
- `tx_llm` - Channel sender for LLM communication
- `_api_key` - API key for LLM provider (currently unused)
- `model_name` - Name of current LLM model
- `provider` - Arc reference to LLM provider implementation

Returns:

- `Option<AppEvent>` - Some(AppEvent) for significant events, None otherwise

Supported Events:

Input	Action
`Enter`	Send current input to LLM (if not busy and input not empty)
`Escape`	Quit application or close help overlay
`F1`, `?`	Toggle help overlay display
`Ctrl+C`	Force quit application immediately
`Ctrl+L`	Clear chat history
`Arrow Up/Down`	Scroll chat history
`Page Up/Down`	Scroll chat history by page
`Home/End`	Scroll to top/bottom of chat
`Printable chars`	Add to input buffer
`Backspace`	Remove last character from input

Event Processing Flow:

1. Check for available terminal events
2. Process keyboard input through app.handle_input()
3. Handle specific application events (send message, quit, etc.)
4. Update UI state based on events
5. Return significant events to main loop

LLM Integration

initiate_llm_request()

```
async fn initiate_llm_request(
    app: &mut ui::App,
    input_to_send: String,
    provider: Arc<dyn LLMProvider + Send + Sync>,
    model_name: &str,
    tx_llm: &mpsc::UnboundedSender<String>,
)
```

Initiates an asynchronous LLM request with proper state management and user feedback.

Parameters:

- `app` - Mutable reference to application state
- `input_to_send` - User's message to send to the LLM
- `provider` - Arc reference to LLM provider implementation
- `model_name` - Name/identifier of the model to use
- `tx_llm` - Channel sender for streaming LLM responses

State Management:

1. **Pre-request State:** * Sets `is_llm_busy` to true * Sets `is_input_disabled` to true * Updates status message to show processing * Adds user message to chat history
2. **Request Processing:** * Spawns separate tokio task for LLM request * Maintains UI responsiveness during request * Handles provider-specific API calls
3. **Error Handling:** * Catches and displays network errors * Shows authentication failures * Handles rate limiting gracefully * Provides recovery suggestions
4. **Post-request State:** * Restores input availability * Updates status message * Adds response or error to chat history

Concurrency: Uses async/await and tokio tasks to prevent UI blocking during potentially slow LLM requests.

Utility Functions

truncate_message()

```
fn truncate_message(s: &str, max_chars: usize) -> String
```

Utility function to truncate messages for display in status areas and limited-width UI components.

Parameters:

- `s` - String to truncate
- `max_chars` - Maximum number of characters to include

Returns:

- `String` - Truncated string with "…" suffix if truncation occurred

Behavior:

- Returns original string if length ≤ max_chars
- Truncates to (max_chars - 3) and appends "…" if longer
- Handles Unicode characters properly
- Preserves word boundaries when possible

Example:

```
let short = truncate_message("Hello world", 5);
assert_eq!(short, "He...");

let unchanged = truncate_message("Hi", 10);
assert_eq!(unchanged, "Hi");
```

Error Handling

The main module implements comprehensive error handling across all application components:

Panic Recovery:

- Custom panic hook for terminal restoration
- User-friendly error messages with recovery suggestions
- Graceful degradation when possible

Runtime Error Handling:

- Configuration validation errors
- Provider authentication failures
- Network connectivity issues
- Terminal initialization failures
- LLM API errors and rate limiting

Error Display:

- Status bar error indicators
- Inline error messages in chat
- Detailed error information in logs
- Recovery action suggestions

Example Error Scenarios:

```
// Configuration error
if config.api_key.is_none() {
    return Err(anyhow!("API key not found. Please set your API key in config.json"));
}

// Provider error
match provider.validate_config(&config).await {
    Err(e) => {
        eprintln!("Provider configuration invalid: {}", e);
        std::process::exit(1);
    }
    Ok(()) => {}
}

// Terminal error
match initialize_terminal() {
    Err(e) => {
        eprintln!("Failed to initialize terminal: {}", e);
        eprintln!("Please ensure your terminal supports the required features.");
        std::process::exit(1);
    }
    Ok(terminal) => terminal
}
```

Application Lifecycle

The main function manages the complete application lifecycle:

Startup Phase:

1. Early panic hook setup for safety
2. Command-line argument processing
3. Configuration loading and validation
4. LLM provider initialization and validation
5. Terminal setup and UI initialization

Runtime Phase:

1. Main event loop with async event handling
2. Concurrent LLM request processing
3. Real-time UI updates and rendering
4. Error handling and recovery

Shutdown Phase:

1. Graceful termination signal handling
2. Terminal state restoration
3. Resource cleanup and deallocation
4. Exit with appropriate status code

Signals and Interrupts:

- `Ctrl+C` - Immediate termination with cleanup
- `SIGTERM` - Graceful shutdown (Unix systems)
- Panic conditions - Emergency terminal restoration

See Also

- *User Interface Module* - User interface implementation
- *Configuration Module* - Configuration management
- *LLM Provider Module* - LLM provider integration
- *Basic Usage* - Basic usage guide
- *Architecture* - Application architecture

The entry point and orchestrator of the application. Handles:

- **Application Lifecycle** - Startup, runtime, and shutdown management
- **CLI Processing** - Command-line argument parsing and validation
- **Event Loop** - Main application event handling and dispatching
- **Error Recovery** - Panic handling and terminal restoration
- **Resource Management** - Terminal initialization and cleanup

Key Components:

- Application initialization and configuration loading
- Terminal setup and TUI framework integration
- Event handling for user input and system events
- LLM request coordination and response management
- Graceful shutdown and error recovery

config

Configuration Module

The `config` module provides comprehensive configuration management for Perspt, supporting multiple LLM providers, flexible authentication, and intelligent defaults.

Core Structures

AppConfig

```
#[derive(Debug, Clone, Deserialize, PartialEq)]
pub struct AppConfig {
    pub providers: HashMap<String, String>,
    pub api_key: Option<String>,
    pub default_model: Option<String>,
    pub default_provider: Option<String>,
    pub provider_type: Option<String>,
}
```

The main configuration structure containing all configurable aspects of Perspt.

Fields:

- `providers` - Map of provider names to their API base URLs
- `api_key` - Universal API key for authentication
- `default_model` - Default model identifier for LLM requests
- `default_provider` - Name of default provider configuration
- `provider_type` - Provider type classification for API compatibility

Supported Provider Types:

Provider Type	Description
`openai`	OpenAI GPT models (GPT-4o, GPT-4o-mini, o1-preview, o1-mini, o3-mini, o4-mini)
`anthropic`	Anthropic Claude models (Claude 3.5 Sonnet, Claude 3 Opus/Sonnet/Haiku)
`gemini`	Google Gemini models (Gemini 1.5 Pro/Flash, Gemini 2.0 Flash)
`groq`	Groq ultra-fast inference (Llama 3.x models)
`cohere`	Cohere Command models (Command R, Command R+)
`xai`	XAI Grok models (grok-3-beta, grok-3-fast-beta)
`deepseek`	DeepSeek models (deepseek-chat, deepseek-reasoner)
`ollama`	Local model hosting via Ollama (requires local setup)

Example Configuration:

```
{
  "api_key": "sk-your-api-key",
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai",
  "providers": {
    "openai": "https://api.openai.com/v1",
    "anthropic": "https://api.anthropic.com",
    "gemini": "https://generativelanguage.googleapis.com/v1beta",
    "groq": "https://api.groq.com/openai/v1",
```

(continues on next page)

(continued from previous page)

```
    "cohere": "https://api.cohere.com/v1",
    "xai": "https://api.x.ai/v1",
    "deepseek": "https://api.deepseek.com/v1",
    "ollama": "http://localhost:11434/v1"
  }
}
```

Core Functions

process_loaded_config

```
pub fn process_loaded_config(mut config: AppConfig) -> AppConfig
```

Processes and validates loaded configuration, applying intelligent defaults and provider type inference.

Parameters:

- `config` - The configuration to process

Returns:

- `AppConfig` - Processed configuration with inferred values

Provider Type Inference Logic:

If `provider_type` is None, attempts inference from `default_provider`:

Default Provider	Inferred Type	Notes
`openai`	`openai`	Direct mapping
`anthropic`	`anthropic`	Direct mapping
`google`, `gemini`	`gemini`	Multiple aliases supported
`groq`	`groq`	Direct mapping
`cohere`	`cohere`	Direct mapping
`xai`	`xai`	Direct mapping
`deepseek`	`deepseek`	Direct mapping
`ollama`	`ollama`	Direct mapping
Unknown	`openai`	Fallback default

Example:

```
let mut config = AppConfig {
    providers: HashMap::new(),
    api_key: None,
    default_model: None,
    default_provider: Some("anthropic".to_string()),
    provider_type: None, // Will be inferred as "anthropic"
};

let processed = process_loaded_config(config);
assert_eq!(processed.provider_type, Some("anthropic".to_string()));
```

load_config

```
pub async fn load_config(config_path: Option<&String>) -> Result<AppConfig>
```

Loads application configuration from a file or provides comprehensive defaults.

Parameters:

- `config_path` - Optional path to JSON configuration file

Returns:

- `Result<AppConfig>` - Loaded configuration or error

Behavior:

With Configuration File (Some(path))

1. Reads JSON file from filesystem
2. Parses JSON into AppConfig structure
3. Processes configuration with `process_loaded_config()`
4. Returns processed configuration

Without Configuration File (None)

Creates default configuration with all supported provider endpoints pre-configured and OpenAI as default provider.

Default Provider Endpoints:

```
{
    "openai": "https://api.openai.com/v1",
    "anthropic": "https://api.anthropic.com",
    "gemini": "https://generativelanguage.googleapis.com/v1beta",
    "groq": "https://api.groq.com/openai/v1",
    "cohere": "https://api.cohere.com/v1",
    "xai": "https://api.x.ai/v1",
    "deepseek": "https://api.deepseek.com/v1",
    "ollama": "http://localhost:11434/v1"
}
```

Possible Errors:

- File system errors (file not found, permission denied)
- JSON parsing errors (invalid syntax, missing fields)
- I/O errors during file reading

Examples:

```
// Load from specific file
let config = load_config(Some(&"config.json".to_string())).await?;

// Use defaults
let default_config = load_config(None).await?;

// Error handling
match load_config(Some(&"missing.json".to_string())).await {
    Ok(config) => println!("Loaded: {:?}", config),
    Err(e) => eprintln!("Failed to load config: {}", e),
}
```

Configuration Examples

Basic OpenAI Configuration

```
{
  "api_key": "sk-your-openai-key",
  "provider_type": "openai",
  "default_model": "gpt-4o-mini"
}
```

Multi-Provider Configuration

```
{
  "api_key": "your-default-key",
  "provider_type": "anthropic",
  "default_model": "claude-3-5-sonnet-20241022",
  "default_provider": "anthropic",
  "providers": {
    "openai": "https://api.openai.com/v1",
    "anthropic": "https://api.anthropic.com",
    "gemini": "https://generativelanguage.googleapis.com/v1beta",
    "groq": "https://api.groq.com/openai/v1",
    "cohere": "https://api.cohere.com/v1",
    "xai": "https://api.x.ai/v1",
    "deepseek": "https://api.deepseek.com/v1",
    "ollama": "http://localhost:11434/v1"
  }
}
```

Minimal Configuration with Provider Inference

```
{
  "default_provider": "groq",
  "default_model": "llama-3.1-70b-versatile"
}
```

Provider type will be automatically inferred as "groq"

Development with Multiple Models

```
{
  "provider_type": "openai",
  "default_model": "gpt-4o-mini",
  "default_provider": "openai",
  "providers": {
    "openai": "https://api.openai.com/v1",
    "groq": "https://api.groq.com/openai/v1",
    "ollama": "http://localhost:11434/v1"
  }
}
```

See Also

- *Configuration Guide* - User configuration guide
- *LLM Provider Module* - LLM provider implementation
- *AI Providers* - Provider setup guide

Configuration management and provider setup. Handles:

- **Multi-Provider Support** - Configuration for all supported AI providers
- **Intelligent Defaults** - Automatic provider type inference and sensible defaults
- **Validation** - Configuration validation and error reporting
- **File Management** - JSON configuration file loading and processing

Key Components:

- `AppConfig` structure for comprehensive configuration
- Provider type inference and validation
- Default configuration generation for all supported providers
- Configuration processing and normalization

llm-provider

LLM Provider Module

The `llm_provider` module provides a modern, unified interface for integrating with multiple AI providers using the cutting-edge `genai` crate. This module enables real-time streaming responses, automatic model discovery, and consistent API behavior across different LLM services.

Core Philosophy

The module is designed around these principles:

1. **Modern GenAI Integration**: Built on the latest `genai` crate with support for newest models like o1-mini, Gemini 2.0, and Claude 3.5
2. **Real-time Streaming**: Advanced streaming with proper event handling and reasoning chunk support
3. **Zero-Configuration**: Automatic environment variable detection with manual override options
4. **Developer-Friendly**: Comprehensive logging, error handling, and debugging capabilities
5. **Production-Ready**: Thread-safe, async-first design with proper resource management

## Supported Providers

The module supports multiple LLM providers through the genai crate (v0.3.5):

- **OpenAI**: GPT-4, GPT-3.5, GPT-4o, o1-mini, o1-preview, o3-mini, o4-mini models
- **Anthropic**: Claude 3 (Opus, Sonnet, Haiku), Claude 3.5 models
- **Google**: Gemini Pro, Gemini 1.5 Pro/Flash, Gemini 2.0 models
- **Groq**: Llama 3.x models with ultra-fast inference
- **Cohere**: Command R/R+ models
- **XAI**: Grok models (grok-3-beta, grok-3-fast-beta, etc.)
- **DeepSeek**: DeepSeek chat and reasoning models (deepseek-chat, deepseek-reasoner)
- **Ollama**: Local model hosting (requires local setup)

## Architecture

The provider uses the genai crate's `Client` as the underlying interface, which handles:

- Authentication via environment variables
- Provider-specific API endpoints and protocols
- Request/response serialization
- Rate limiting and retry logic

Core Types

GenAIProvider

```
pub struct GenAIProvider {
    client: Client,
}
```

Main LLM provider implementation using the `genai` crate for unified access to multiple AI providers.

Design Philosophy:

The provider is designed around the principle of “configure once, use everywhere”. It automatically handles provider-specific authentication requirements, API endpoints, and response formats while presenting a consistent interface to the application.

Configuration Methods:

1. **Auto-configuration**: Uses environment variables (recommended)
2. **Explicit configuration**: API keys and provider types via constructor
3. **Runtime configuration**: Dynamic provider switching (future enhancement)

Thread Safety: The provider is thread-safe and can be shared across async tasks using `Arc<GenAIProvider>`. The underlying genai client handles concurrent requests efficiently.

Methods:

new()

```
pub fn new() -> Result<Self>
```

Creates a new GenAI provider with automatic configuration.

This constructor creates a provider instance using the genai client’s default configuration, which automatically detects and uses environment variables for authentication. This is the recommended approach for production use.

Environment Variables:

The client will automatically detect and use these environment variables:

- `OPENAI_API_KEY`: For OpenAI models
- `ANTHROPIC_API_KEY`: For Anthropic Claude models
- `GEMINI_API_KEY`: For Google Gemini models
- `GROQ_API_KEY`: For Groq models
- `COHERE_API_KEY`: For Cohere models
- `XAI_API_KEY`: For XAI Grok models
- `DEEPSEEK_API_KEY`: For DeepSeek models
- `OLLAMA_API_BASE`: For Ollama local models (optional, defaults to http://localhost:11434)

Returns:

- `Result<Self>` - A configured provider instance or configuration error

Errors:

This method can fail if:

- The genai client cannot be initialized
- Required system dependencies are missing
- Network configuration prevents client creation

Example:

```
// Set environment variable first
std::env::set_var("OPENAI_API_KEY", "sk-your-key");

// Create provider with auto-configuration
let provider = GenAIProvider::new()?;
```

new_with_config()

```
pub fn new_with_config(provider_type: Option<&str>, api_key: Option<&str>) -> Result<Self>
```

Creates a new GenAI provider with explicit configuration.

This constructor allows explicit specification of provider type and API key, which is useful for CLI applications, testing, or when configuration needs to be provided at runtime rather than through environment variables.

Arguments:

- `provider_type` - Optional provider identifier (e.g., “openai”, “anthropic”)
- `api_key` - Optional API key for authentication

Provider Type Mapping:

- `"openai"` → Sets `OPENAI_API_KEY`
- `"anthropic"` → Sets `ANTHROPIC_API_KEY`
- `"google"` or `"gemini"` → Sets `GEMINI_API_KEY`
- `"groq"` → Sets `GROQ_API_KEY`
- `"cohere"` → Sets `COHERE_API_KEY`
- `"xai"` → Sets `XAI_API_KEY`
- `"deepseek"` → Sets `DEEPSEEK_API_KEY`
- `"ollama"` → Sets `OLLAMA_API_BASE` (optional)

Example:

```
// Create provider with explicit configuration
let provider = GenAIProvider::new_with_config(
    Some("openai"),
    Some("sk-your-api-key")
)?;
```

get_available_models()

```
pub async fn get_available_models(&self, provider: &str) -> Result<Vec<String>>
```

Retrieves all available models for a specific provider.

This method queries the specified provider’s API to get a list of all available models that can be used for chat completion. The list includes both current and legacy models, allowing users to choose the most appropriate model for their needs.

Arguments:

- `provider` - The provider identifier (e.g., “openai”, “anthropic”, “google”)

Provider Support:

Model listing is supported for:

- **OpenAI**: GPT-4, GPT-3.5, GPT-4o, o1 series, o3-mini, o4-mini models
- **Anthropic**: Claude 3/3.5 series (Opus, Sonnet, Haiku)

- **Google**: Gemini Pro, Gemini 1.5/2.0 series
- **Groq**: Llama 3.x series with various sizes
- **Cohere**: Command R/R+ models
- **XAI**: Grok models (grok-3-beta, grok-3-fast-beta, etc.)
- **DeepSeek**: DeepSeek chat and reasoning models
- **Ollama**: Requires local setup and running instance

Returns:

- `Result<Vec<String>>` - List of model identifiers or error

Errors:

This method can fail if:

- The provider name is not recognized by genai
- Network connectivity issues prevent API access
- Authentication credentials are invalid or missing
- The provider's API is temporarily unavailable
- Rate limits are exceeded

Example:

```
let provider = GenAIProvider::new()?;

// Get OpenAI models
let openai_models = provider.get_available_models("openai").await?;
for model in openai_models {
    println!("Available: {}", model);
}

// Get Anthropic models
let claude_models = provider.get_available_models("anthropic").await?;
```

generate_response_simple()

```
pub async fn generate_response_simple(&self, model: &str, prompt: &str) -> Result<String>
```

Generates a simple text response without streaming.

This method provides a straightforward way to get a complete response from an LLM without the complexity of streaming. It's ideal for simple Q&A scenarios, testing, or when the entire response is needed before processing.

Arguments:

- `model` - The model identifier (e.g., "gpt-4o-mini", "claude-3-5-sonnet-20241022")
- `prompt` - The user's message or prompt text

Model Compatibility:

Supports all models available through the genai crate:

- OpenAI: `gpt-4o`, `gpt-4o-mini`, `gpt-3.5-turbo`, `o1-mini`, `o1-preview`
- Anthropic: `claude-3-5-sonnet-20241022`, `claude-3-opus-20240229`, etc.
- Google: `gemini-1.5-pro`, `gemini-1.5-flash`, `gemini-2.0-flash`
- Groq: `llama-3.1-70b-versatile`, `mixtral-8x7b-32768`, etc.

Returns:

- `Result<String>` - The complete response text or error

Example:

```
let provider = GenAIProvider::new_with_config(
    Some("openai"),
    Some("sk-your-key")
)?;

let response = provider.generate_response_simple(
    "gpt-4o-mini",
    "What is the capital of France?"
).await?;

println!("AI: {}", response);
```

generate_response_stream_to_channel()

```
pub async fn generate_response_stream_to_channel(
    &self,
    model: &str,
    prompt: &str,
    tx: mpsc::UnboundedSender<String>
) -> Result<()>
```

Generates a streaming response and sends chunks via mpsc channel.

This is the core streaming method that provides real-time response generation, essential for creating responsive chat interfaces. It properly handles the genai crate's streaming events and manages the async communication with the UI layer.

Streaming Architecture:

The method uses an async stream from the genai crate and processes different types of events:

- **Start**: Indicates the beginning of response generation
- **Chunk**: Contains incremental text content (main response text)
- **ReasoningChunk**: Contains reasoning steps (for models like o1)
- **End**: Indicates completion of response generation

Arguments:

- `model` - The model identifier to use for generation
- `prompt` - The user's input prompt or message
- `tx` - Unbounded mpsc sender for streaming response chunks to the UI

Channel Communication:

The method sends content chunks through the provided channel as they arrive. The receiving end (typically the UI) should listen for messages and handle:

- Regular text chunks for incremental display
- End-of-transmission signal (`EOT_SIGNAL`) indicating completion
- Error messages prefixed with "Error: " for failure cases

Event Processing:

1. **ChatStreamEvent::Start** - Logs stream initiation, no content sent
2. **ChatStreamEvent::Chunk** - Sends content immediately to channel
3. **ChatStreamEvent::ReasoningChunk** - Logs reasoning (future: may send to channel)
4. **ChatStreamEvent::End** - Logs completion, caller should send EOT signal

Error Handling:

Stream errors are handled gracefully:

- Errors are logged with full context
- Error messages are sent through the channel
- The method returns the error for caller handling
- Channel send failures are logged but don't halt processing

Returns:

- `Result<()>` - Success (content sent via channel) or error details

Example:

```rust
use tokio::sync::mpsc;
use perspt::EOT_SIGNAL;

let provider = GenAIProvider::new()?;
let (tx, mut rx) = mpsc::unbounded_channel();

// Start streaming in background task
let provider_clone = provider.clone();
tokio::spawn(async move {
    match provider_clone.generate_response_stream_to_channel(
        "gpt-4o-mini",
        "Tell me about Rust programming",
        tx.clone()
    ).await {
        Ok(()) => {
            let _ = tx.send(EOT_SIGNAL.to_string());
        }
        Err(e) => {
            let _ = tx.send(format!("Error: {}", e));
            let _ = tx.send(EOT_SIGNAL.to_string());
        }
    }
});

// Receive and process chunks
while let Some(chunk) = rx.recv().await {
    if chunk == EOT_SIGNAL {
        break;
    } else if chunk.starts_with("Error: ") {
        eprintln!("Stream error: {}", chunk);
        break;
    } else {
        print!("{}", chunk); // Display incremental content
    }
}
```

generate_response_with_history()

```
pub async fn generate_response_with_history(&self, model: &str, messages: Vec<ChatMessage>
↪) -> Result<String>
```

Generate response with conversation history.

Arguments:

- `model` - The model identifier
- `messages` - Vector of ChatMessage objects representing conversation history

Returns:

- `Result<String>` - Complete response text or error

get_supported_providers()

```
pub fn get_supported_providers() -> Vec<&'static str>
```

Get a list of supported providers.

Returns:

- `Vec<&'static str>` - List of supported provider identifiers

Supported Providers:

```
[
    "openai",
    "anthropic",
    "gemini",
    "groq",
    "cohere",
    "ollama",
    "xai"
]
```

test_model()

```
pub async fn test_model(&self, model: &str) -> Result<bool>
```

Test if a model is available and working.

Arguments:

- `model` - The model identifier to test

Returns:

- `Result<bool>` - True if model is working, false otherwise

validate_model()

```
pub async fn validate_model(&self, model: &str, provider_type: Option<&str>) -> Result
↪<String>
```

Validate and get the best available model for a provider.

Arguments:

- `model` - The model identifier to validate
- `provider_type` - Optional provider type for fallback model selection

Returns:

- `Result<String>` - Validated model identifier or fallback model

Utility Functions

str_to_adapter_kind()

```
fn str_to_adapter_kind(provider: &str) -> Result<AdapterKind>
```

Convert a provider string to genai AdapterKind.

Arguments:

- `provider` - Provider string identifier

Returns:

- `Result<AdapterKind>` - Corresponding AdapterKind enum variant

Provider Mapping:

Input String	AdapterKind
`"openai"`	`AdapterKind::OpenAI`
`"anthropic"`	`AdapterKind::Anthropic`
`"gemini"`, `"google"`	`AdapterKind::Gemini`
`"groq"`	`AdapterKind::Groq`
`"cohere"`	`AdapterKind::Cohere`
`"ollama"`	`AdapterKind::Ollama`
`"xai"`	`AdapterKind::Xai`

Usage Examples

Basic Chat Interaction

```
use perspt::llm_provider::GenAIProvider;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize provider with environment variables
    let provider = GenAIProvider::new()?;

    // Simple question-answer
    let response = provider.generate_response_simple(
        "gpt-4o-mini",
        "Explain async programming in Rust"
    ).await?;

    println!("AI: {}", response);
```

(continues on next page)

(continued from previous page)

```
    Ok(())
}
```

Streaming Chat Interface

```
use perspt::llm_provider::GenAIProvider;
use tokio::sync::mpsc;
use perspt::EOT_SIGNAL;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let provider = GenAIProvider::new()?;
    let (tx, mut rx) = mpsc::unbounded_channel();

    // Start streaming
    tokio::spawn(async move {
        let _ = provider.generate_response_stream_to_channel(
            "claude-3-5-sonnet-20241022",
            "Write a haiku about programming",
            tx
        ).await;
    });

    // Display results in real-time
    while let Some(chunk) = rx.recv().await {
        if chunk == EOT_SIGNAL {
            println!("\n[Stream Complete]");
            break;
        }
        print!("{}", chunk);
    }

    Ok(())
}
```

Error Handling Best Practices

```
use perspt::llm_provider::GenAIProvider;
use anyhow::{Context, Result};

async fn robust_llm_call() -> Result<String> {
    let provider = GenAIProvider::new()
        .context("Failed to initialize LLM provider")?;

    // Test model availability first
    let model = "gpt-4o-mini";
    if !provider.test_model(model).await? {
        return Err(anyhow::anyhow!("Model {} is not available", model));
    }

```

(continues on next page)

(continued from previous page)

```
    // Make the actual request with proper error context
    let response = provider.generate_response_simple(
        model,
        "Hello, world!"
    )
    .await
    .context(format!("Failed to generate response using model {}", model))?;

    Ok(response)
}
```

Provider Selection

```
use perspt::llm_provider::GenAIProvider;

async fn choose_best_provider() -> Result<(), Box<dyn std::error::Error>> {
    let provider = GenAIProvider::new()?;

    // Get all supported providers
    let providers = GenAIProvider::get_supported_providers();

    for provider_name in providers {
        println!("Checking provider: {}", provider_name);

        // Get available models for each provider
        if let Ok(models) = provider.get_available_models(provider_name).await {
            println!("  Available models: {:?}", models);

            // Test the first model
            if !models.is_empty() {
                let works = provider.test_model(&models[0]).await.unwrap_or(false);
                println!("  Model {} works: {}", models[0], works);
            }
        }
    }

    Ok(())
}
```

Implementation Details

GenAI Crate Integration

The module is built on the modern `genai` crate which provides:

Unified Client Interface:

```
use genai::Client;

// Single client handles all providers
let client = Client::default();
let models = client.all_model_names(AdapterKind::OpenAI).await?;
```

Automatic Authentication:

```
// Environment variables are automatically detected:
// OPENAI_API_KEY, ANTHROPIC_API_KEY, GEMINI_API_KEY, etc.
let client = Client::default();
```

Streaming Support:

```
use genai::chat::{ChatRequest, ChatMessage};

let chat_req = ChatRequest::default()
    .append_message(ChatMessage::user("Hello"));

let stream = client.exec_chat_stream("gpt-4o-mini", chat_req, None).await?;
```

Event Processing:

```
use genai::chat::ChatStreamEvent;

while let Some(event) = stream.stream.next().await {
    match event? {
        ChatStreamEvent::Start => println!("Stream started"),
        ChatStreamEvent::Chunk(chunk) => print!("{}", chunk.content),
        ChatStreamEvent::ReasoningChunk(chunk) => println!("Reasoning: {}", chunk.content),
        ChatStreamEvent::End(_) => println!("Stream ended"),
    }
}
```

Error Handling

The module uses `anyhow::Result` for comprehensive error handling:

- **Configuration Errors**: Missing API keys, invalid provider types
- **Network Errors**: Connection timeouts, API rate limits
- **Model Errors**: Invalid model names, unavailable models
- **Stream Errors**: Interrupted streams, malformed responses
- **Authentication Errors**: Invalid API keys, expired tokens

Example Error Handling:

```
use anyhow::{Context, Result};

async fn safe_llm_call() -> Result<String> {
    let provider = GenAIProvider::new()
        .context("Failed to create provider")?;

    let response = provider.generate_response_simple(
        "gpt-4o-mini",
        "Hello"
    )
    .await
    .context("Failed to generate response")?;

    Ok(response)
}
```

Advanced Error Recovery:

```
// Graceful fallback to alternative models
async fn robust_generation(provider: &GenAIProvider, prompt: &str) -> Result<String> {
    let preferred_models = ["gpt-4o", "gpt-4o-mini", "gpt-3.5-turbo"];

    for model in preferred_models {
        match provider.generate_response_simple(model, prompt).await {
            Ok(response) => return Ok(response),
            Err(e) => {
                log::warn!("Model {} failed: {}, trying next", model, e);
                continue;
            }
        }
    }

    Err(anyhow::anyhow!("All models failed"))
}
```

Performance Considerations

Async Streaming:

The streaming implementation is designed for optimal performance:

- Non-blocking async operations
- Immediate chunk forwarding (no batching delays)
- Minimal memory footprint
- Proper backpressure handling

Memory Management:

```
// Unbounded channels for streaming (careful with memory)
let (tx, rx) = mpsc::unbounded_channel();

// Alternative: bounded channels with backpressure
let (tx, rx) = mpsc::channel(1000);
```

Logging and Debugging:

Comprehensive logging is built-in for performance monitoring:

```
// Enable debug logging to track stream performance
RUST_LOG=debug ./perspt

// Logs include:
// - Chunk counts and timing
// - Content length tracking
// - Stream start/end events
// - Error conditions and recovery
```

Testing

Unit Tests:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_str_to_adapter_kind() {
        assert!(str_to_adapter_kind("openai").is_ok());
        assert!(str_to_adapter_kind("invalid").is_err());
    }

    #[tokio::test]
    async fn test_provider_creation() {
        let provider = GenAIProvider::new();
        assert!(provider.is_ok());
    }
}
```

Integration Tests:

```
// Test with real API keys (requires environment setup)
#[tokio::test]
#[ignore] // Only run with --ignored
async fn test_live_openai() -> Result<()> {
    let provider = GenAIProvider::new()?;
    let response = provider.generate_response_simple(
        "gpt-3.5-turbo",
        "Say hello"
    ).await?;
    assert!(!response.is_empty());
    Ok(())
}
```

See Also

- *Configuration Module* - Configuration module for provider setup and authentication
- *Main Module* - Main module for application orchestration and LLM provider integration
- *User Interface Module* - UI module for displaying streaming responses
- GenAI Crate Documentation - Underlying LLM integration library
- Tokio Documentation - Async runtime used for streaming

Related Files:

- `src/llm_provider.rs` - Source implementation
- `src/config.rs` - Configuration and provider setup
- `src/main.rs` - Provider initialization and usage
- `tests/` - Integration and unit tests

Unified interface for AI provider integration using the modern GenAI crate. Handles:

- **Provider Abstraction** - Single interface across OpenAI, Anthropic, Google, Groq, Cohere, and XAI
- **Auto-Configuration** - Environment variable detection and automatic setup
- **Streaming Support** - Real-time response streaming with proper event handling

- **Error Handling** - Comprehensive error categorization and recovery

Key Components:

- `GenAIProvider` struct using the `genai` crate client
- Auto-configuration via environment variables
- Model listing and validation capabilities
- Streaming response generation to channels

ui

User Interface Module

The `ui` module implements the terminal-based user interface for Perspt using the Ratatui TUI framework. It provides a modern, responsive chat experience with real-time streaming responses, enhanced cursor navigation, markdown rendering, and comprehensive state management.

Overview

The UI module is the core interactive component of Perspt, providing a rich terminal-based chat interface. It handles everything from user input and cursor management to real-time streaming display and markdown rendering.

Key Capabilities:

- **Real-time Streaming UI**: Immediate, responsive rendering during LLM response generation with intelligent buffering
- **Enhanced Input System**: Full cursor movement, editing capabilities, and visual feedback with blinking cursor
- **Smart Content Management**: Optimized streaming buffer preventing memory overflow while maintaining responsiveness
- **Rich Markdown Rendering**: Live formatting with syntax highlighting, code blocks, lists, and emphasis
- **Intelligent Error Handling**: Categorized error types with user-friendly messages and recovery suggestions
- **Smooth Animations**: Typing indicators, progress bars, and cursor blinking for better user experience
- **Input Queuing**: Seamless message queuing while AI is responding to maintain conversation flow

Architecture Overview

The UI follows a layered, event-driven architecture designed for responsiveness and maintainability:

```
┌─────────────────────────────────────────────────────────────┐
│                    Perspt UI Architecture                   │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌─────────────┐    ┌──────────────┐    ┌─────────────────┐ │
│  │     App     │────│ ChatMessage  │────│ MessageType     │ │
│  │(Controller) │    │   (Data)     │    │  (Styling)      │ │
│  │             │    │              │    │                 │ │
│  │ + State     │    │ + Content    │    │ + User          │ │
│  │ + Streaming │    │ + Timestamp  │    │ + Assistant     │ │
│  │ + Cursor    │    │ + Markdown   │    │ + Error/System  │ │
│  └─────────────┘    └──────────────┘    └─────────────────┘ │
│           │                                       │         │
│           ▼                                       ▼         │
│  ┌─────────────┐    ┌──────────────┐    ┌─────────────────┐ │
│  │ ErrorState  │    │   AppEvent   │    │Event Processing │ │
│  │ + Categories│    │ + Key Events │    │ + Async Loop    │ │
```

(continues on next page)

(continued from previous page)

```
│  │ + Recovery  │    │ + UI Updates │    │ + Priorities    │ │
│  │ + Messages  │    │ + Timers     │    │ + Non-blocking  │ │
│  └─────────────┘    └──────────────┘    └─────────────────┘ │
│                                                             │
│  ┌─────────────┐    ┌──────────────┐    ┌─────────────────┐ │
│  │ Rendering   │    │  Animation   │    │   Markdown      │ │
│  │ + Layout    │    │ + Spinners   │    │ + Parsing       │ │
│  │ + Cursor    │    │ + Progress   │    │ + Highlighting  │ │
│  │ + Scrolling │    │ + Blinking   │    │ + Code Blocks   │ │
│  └─────────────┘    └──────────────┘    └─────────────────┘ │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Key Design Principles:

1. **Responsiveness**: Immediate feedback for all user actions with optimized rendering
2. **State Consistency**: Centralized state management in the App struct prevents race conditions
3. **Memory Efficiency**: Smart buffer management prevents overflow during long responses
4. **User Experience**: Visual feedback, animations, and clear error messages guide the user

Core Types and Data Structures

MessageType

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum MessageType {
    User,      // Blue styling for user input
    Assistant, // Green styling for AI responses
    Error,     // Red styling for error messages
    System,    // Cyan styling for system notifications
    Warning,   // Yellow styling for warnings
}
```

Determines the visual appearance and behavior of messages in the chat interface. Each type has distinct styling to help users quickly identify message sources.

Message Styling:

Type	Color	Icon	Purpose
`User`	Blue	□	User input messages and questions
`Assistant`	Green	🤖	AI responses and assistance
`Error`	Red	□	Error notifications and failures
`System`	Cyan	□□	System status and welcome messages
`Warning`	Yellow	□□	Warning messages and alerts

Example:

```rust
use perspt::ui::MessageType;

let user_msg = MessageType::User;      // Blue with □ icon
let ai_msg = MessageType::Assistant;   // Green with 🤖 icon
let error_msg = MessageType::Error;    // Red with □ icon
```

ChatMessage

```
#[derive(Debug, Clone)]
pub struct ChatMessage {
    pub message_type: MessageType,
    pub content: Vec<Line<'static>>,
    pub timestamp: String,
}
```

Core data structure for chat messages with rich formatting support and automatic timestamp management.

Fields:

- `message_type` - Determines styling, color, and icon display
- `content` - Pre-formatted content as styled Ratatui lines with full markdown support
- `timestamp` - Creation time in HH:MM format (automatically set by `App::add_message()`)

Features:

- **Rich Markdown Support**: Automatic parsing of markdown with syntax highlighting
- **Responsive Formatting**: Content adapts to terminal width changes
- **Icon Integration**: Automatic icon assignment based on message type
- **Timestamp Management**: Automatic timestamping when added to chat history

Example:

```
use perspt::ui::{ChatMessage, MessageType};
use ratatui::text::Line;

// Simple text message (timestamp will be auto-generated)
let message = ChatMessage {
    message_type: MessageType::User,
    content: vec![Line::from("Hello, AI!")],
    timestamp: String::new(), // Auto-populated by App::add_message()
};

// Rich content with markdown (automatically parsed)
let ai_response = ChatMessage {
    message_type: MessageType::Assistant,
    content: markdown_to_lines("Here's some **bold** text and `code`"),
    timestamp: App::get_timestamp(),
};
```

ErrorState and Error Handling

ErrorState

```
#[derive(Debug, Clone)]
pub struct ErrorState {
    pub message: String,
    pub details: Option<String>,
    pub error_type: ErrorType,
}
```

Comprehensive error information system with automatic categorization and user-friendly messaging.

Fields:

- `message` - Primary user-facing error message (concise and actionable)
- `details` - Optional technical details for debugging and troubleshooting
- `error_type` - Error category for appropriate styling, handling, and recovery suggestions

ErrorType

```rust
#[derive(Debug, Clone)]
pub enum ErrorType {
    Network,        // Connectivity and network issues
    Authentication, // API key and provider auth failures
    RateLimit,      // API rate limiting and quota exceeded
    InvalidModel,   // Unsupported or invalid model requests
    ServerError,    // Provider server errors and outages
    Unknown,        // Unclassified or unexpected errors
}
```

Advanced error categorization system that automatically analyzes error messages and provides appropriate user guidance.

Error Categories with Recovery Guidance:

Type	Description & Auto-Generated Recovery Guidance
`Network`	Connectivity issues, timeouts, DNS failures. *"Check internet connection and try again."*
`Authentication`	Invalid API keys, expired tokens, permission errors. *"Verify API key configuration."*
`RateLimit`	API quota exceeded, too many requests. *"Wait a moment before sending another request."*
`InvalidModel`	Unsupported models, malformed requests. *"Check model availability and request format."*
`ServerError`	Provider outages, internal server errors. *"Service may be temporarily unavailable."*
`Unknown`	Unclassified errors requiring investigation. *"Please report if this persists."*

Automatic Error Categorization Example:

```rust
// The categorize_error() function automatically analyzes error messages
fn categorize_error(error_msg: &str) -> ErrorState {
    let error_lower = error_msg.to_lowercase();

    if error_lower.contains("api key") || error_lower.contains("unauthorized") {
        ErrorState {
            message: "Authentication failed".to_string(),
            details: Some("Please check your API key is valid".to_string()),
            error_type: ErrorType::Authentication,
        }
    } else if error_lower.contains("rate limit") {
        ErrorState {
            message: "Rate limit exceeded".to_string(),
            details: Some("Please wait before sending another request".to_string()),
            error_type: ErrorType::RateLimit,
        }
    }
    // ... other categorizations
}
```

App (Main Controller)

```
pub struct App {
    // Core Application State
    pub chat_history: Vec<ChatMessage>,
    pub input_text: String,
    pub status_message: String,
    pub config: AppConfig,
    pub should_quit: bool,

    // Navigation and Display Management
    scroll_state: ScrollbarState,
    pub scroll_position: usize,
    pub show_help: bool,

    // Input Processing and Queue Management
    pub is_input_disabled: bool,
    pub pending_inputs: VecDeque<String>,
    pub is_llm_busy: bool,
    pub current_error: Option<ErrorState>,

    // Enhanced Cursor and Input Handling
    pub cursor_position: usize,
    pub input_scroll_offset: usize,
    pub cursor_blink_state: bool,
    pub last_cursor_blink: Instant,

    // Real-time Streaming and Animation
    pub typing_indicator: String,
    pub response_progress: f64,
    pub streaming_buffer: String,
    pub last_animation_tick: Instant,

    // Performance and UI Optimization
    pub needs_redraw: bool,
    pub input_width: usize,
    pub terminal_height: usize,
    pub terminal_width: usize,
}
```

Enhanced central application controller managing all aspects of the chat interface, including real-time streaming, cursor navigation, input queuing, and responsive UI updates.

State Organization:

Core Application State: * `chat_history` - Complete conversation with automatic timestamps and rich markdown formatting * `input_text` - Current user input with full text editing support (insert, delete, cursor movement) * `status_message` - Dynamic status with contextual information and error states * `config` - Application configuration and LLM provider settings * `should_quit` - Clean shutdown flag for the event loop

Enhanced Input System: * `cursor_position` - Current cursor position within input text (character-level precision) * `input_scroll_offset` - Horizontal scroll offset for long input lines * `cursor_blink_state` - Visual cursor blinking animation state (500ms intervals) * `input_width` - Available input area width for accurate scroll calculations * `is_input_disabled` - Input protection during streaming to prevent conflicts

Real-time Streaming Management: * `is_llm_busy` - Active response generation state flag * `streaming_buffer` -

Real-time content accumulation from LLM (with 1MB overflow protection) * `response_progress` - Visual progress indicator (0.0 to 1.0 scale) * `typing_indicator` - Animated spinner for visual feedback (10-frame cycle)

Navigation and UI State: * `scroll_position` - Current chat history view position with bounds checking * `scroll_state` - Internal scrollbar state synchronized with position * `show_help` - Help overlay visibility toggle * `needs_redraw` - Performance optimization flag for efficient rendering

Advanced Features: * `pending_inputs` - Message queue for seamless conversation flow while AI responds * `current_error` - Active error state with categorization and recovery suggestions * `last_animation_tick` - Animation timing for smooth 60fps visual effects * `terminal_height/width` - Current terminal dimensions for responsive layout

Performance Optimizations:

- **Intelligent Redraw**: Only updates UI when `needs_redraw` flag is set, reducing CPU usage
- **Smart Buffer Management**: Prevents memory overflow during long responses with 1MB limit
- **Responsive Input**: Immediate character feedback with optimized cursor rendering
- **Efficient Scrolling**: Content-aware scroll calculations with proper bounds checking
- **Animation Timing**: Balanced update intervals for smooth visuals without CPU waste

Developer Notes:

- The App struct uses interior mutability patterns for safe concurrent access
- All timing-related fields use `Instant` for high-precision animation control
- Buffer management includes overflow protection for production stability
- Input handling supports full terminal editing capabilities (Home, End, arrows, etc.)

AppEvent

```
#[derive(Debug)]
pub enum AppEvent {
    Quit,           // Clean application shutdown
    Redraw,         // Immediate UI refresh needed
    Key(KeyEvent),  // User keyboard input
    Tick,           // Periodic timer for animations
}
```

Event system for the responsive async UI loop, supporting immediate user feedback and smooth animations.

Event Types:

- `Quit` - Triggered by Ctrl+C/Ctrl+Q for clean application shutdown
- `Redraw` - Immediate UI refresh for responsive input feedback
- `Key(KeyEvent)` - User keyboard input with full key details and modifiers
- `Tick` - Periodic updates for animations, cursor blinking, and status updates

Event Processing Priority:

The event loop processes events with the following priority order:

1. **Highest**: LLM response chunks (real-time streaming)
2. **High**: Terminal input events (immediate user feedback)
3. **Medium**: UI rendering updates (~60 FPS)
4. **Low**: Background tasks and periodic cleanup

Core Methods

Application Lifecycle

new()

```
pub fn new(config: AppConfig) -> Self
```

Creates a new App instance with enhanced welcome message, optimized state initialization, and responsive UI setup.

Parameters:

- `config` - Application configuration with LLM provider settings

Returns:

- `Self` - Fully initialized App instance with welcome message and default state

Features:

- **Rich Welcome Message**: Multi-line welcome with quick help, shortcuts, and visual styling
- **State Initialization**: All cursors, buffers, and timers properly initialized to safe defaults
- **Performance Setup**: Optimized default values for responsive operation

Implementation Details:

The constructor creates a comprehensive welcome message that includes:

```
// Welcome message with styling and helpful shortcuts
let welcome_msg = ChatMessage {
    message_type: MessageType::System,
    content: vec![
        Line::from("□ Welcome to Perspt - Your AI Chat Terminal"),
        Line::from("□ Quick Help:"),
        Line::from("  • Enter - Send message"),
        Line::from("  • ↑/↓ - Scroll chat history"),
        Line::from("  • Ctrl+C/Ctrl+Q - Exit"),
        Line::from("  • F1 - Toggle help"),
        Line::from("Ready to chat! Type your message below..."),
    ],
    timestamp: Self::get_timestamp(),
};
```

Example:

```
use perspt::ui::App;
use perspt::config::AppConfig;

let config = AppConfig::load().unwrap();
let app = App::new(config);

assert!(!app.should_quit);
assert!(!app.chat_history.is_empty()); // Contains rich welcome message
assert_eq!(app.cursor_position, 0);    // Cursor at start
assert!(!app.is_llm_busy);             // Ready for input
```

get_timestamp()

```
pub fn get_timestamp() -> String
```

Generates a formatted timestamp string for message display.

Returns:

- `String` - Timestamp in HH:MM format for current system time

Usage:

```
let timestamp = App::get_timestamp();
// Returns format like "14:30" for 2:30 PM
```

Message Management

add_message()

```
pub fn add_message(&mut self, mut message: ChatMessage)
```

Adds a message to chat history with automatic timestamping, scroll management, and immediate UI updates.

Parameters:

- `message` - ChatMessage to add (timestamp will be automatically set to current time)

Behavior:

1. **Automatic Timestamping**: Sets current time in HH:MM format
2. **Smart Scrolling**: Automatically scrolls to show new message
3. **Immediate Feedback**: Triggers redraw for instant visibility
4. **State Consistency**: Maintains proper scroll and display state

Example:

```
let message = ChatMessage {
    message_type: MessageType::User,
    content: vec![Line::from("What's the weather like?")],
    timestamp: String::new(), // Will be set automatically
};

app.add_message(message);
// Message immediately visible with current timestamp
```

add_error()

```
pub fn add_error(&mut self, error: ErrorState)
```

Adds an enhanced error message with automatic categorization, recovery suggestions, and visual prominence.

Parameters:

- `error` - ErrorState containing error information and category

Enhanced Behavior:

1. **Dual Display**: Error appears in both chat history and status bar
2. **Rich Formatting**: Error icon (□), styled text, and optional details

3. **Recovery Guidance**: Context-appropriate suggestions based on error type
4. **Visual Prominence**: Red styling and immediate scroll-to-show

Implementation:

```
// Creates rich error display with icon and details
let error_content = vec![
    Line::from(vec![
        Span::styled("□ Error: ", Style::default().fg(Color::Red).bold()),
        Span::styled(error.message.clone(), Style::default().fg(Color::Red)),
    ]),
];

// Adds optional details if available
if let Some(details) = &error.details {
    full_content.push(Line::from(vec![
        Span::styled("   Details: ", Style::default().fg(Color::Yellow)),
        Span::styled(details.clone(), Style::default().fg(Color::Gray)),
    ]));
}
```

Example:

```
let error = ErrorState {
    message: "API key invalid".to_string(),
    details: Some("Check your configuration file".to_string()),
    error_type: ErrorType::Authentication,
};

app.add_error(error);
// Shows: "□ Error: API key invalid"
//        "   Details: Check your configuration file"
```

clear_error()

```
pub fn clear_error(&mut self)
```

Clears the current error state and removes error display from the status bar.

Features:

- **State Reset**: Removes active error from status bar display
- **Clean Recovery**: Allows normal status messages to be shown again
- **Immediate Effect**: Error clearing is instant and triggers UI update

Usage:

Typically called after user acknowledges an error or when starting a new operation that should clear previous error states.

Example:

```
// Display an error
let error = ErrorState {
    message: "Connection failed".to_string(),
    details: None,
```

(continues on next page)

(continued from previous page)

```
    error_type: ErrorType::Network,
};
app.add_error(error);
assert!(app.current_error.is_some());

// Clear the error
app.clear_error();
assert!(app.current_error.is_none());
// Status bar now shows normal status instead of error
```

set_status()

```
pub fn set_status(&mut self, message: String, is_error: bool)
```

Sets the status bar message with optional error logging.

Parameters:

- `message` - The status message to display in the status bar
- `is_error` - Whether this message represents an error (affects logging level)

Features:

- **Immediate Display**: Status message appears instantly in the status bar
- **Error Logging**: Messages marked as errors are logged appropriately
- **Flexible Usage**: Can be used for both informational and error messages

Example:

```
app.set_status("Processing request...".to_string(), false);
// Status shows: "Processing request..."

app.set_status("Connection failed".to_string(), true);
// Status shows: "Connection failed" and logs as error
```

Enhanced Input System

insert_char()

```
pub fn insert_char(&mut self, ch: char)
```

Inserts a character at the current cursor position with immediate visual feedback and smart scrolling.

Parameters:

- `ch` - Character to insert

Features:

- **Cursor-Aware Insertion**: Character inserted exactly at cursor position
- **Auto-Scroll**: Input view scrolls to keep cursor visible for long text
- **Immediate Feedback**: Instant character appearance and cursor movement
- **Blink Reset**: Cursor blink resets for better visibility during typing
- **Input Protection**: Only works when input is enabled (not disabled during streaming)

Example:

```
app.insert_char('H');
app.insert_char('i');
// Input shows "Hi" with cursor at position 2
```

delete_char_before()

```
pub fn delete_char_before(&mut self)
```

Implements backspace functionality with cursor-aware deletion and visual feedback.

Features:

- **Smart Deletion**: Removes character before cursor position
- **Cursor Movement**: Cursor moves back after deletion
- **Visual Update**: Immediate text and cursor position updates
- **Boundary Safety**: Safe operation at beginning of input

delete_char_at()

```
pub fn delete_char_at(&mut self)
```

Implements delete key functionality, removing character at cursor position.

Features:

- **Forward Deletion**: Removes character at current cursor position
- **Cursor Stability**: Cursor position remains stable after deletion
- **Boundary Safety**: Safe operation at end of input

move_cursor_left() / move_cursor_right()

```
pub fn move_cursor_left(&mut self)
pub fn move_cursor_right(&mut self)
```

Navigate cursor within input text with automatic view scrolling for long input.

Features:

- **Boundary Respect**: Cannot move beyond text boundaries
- **Auto-Scroll**: View adjusts to keep cursor visible in long text
- **Visual Feedback**: Immediate cursor position updates

move_cursor_to_start() / move_cursor_to_end()

```
pub fn move_cursor_to_start(&mut self)
pub fn move_cursor_to_end(&mut self)
```

Jump cursor to beginning or end of input with view reset.

Features:

- **Instant Navigation**: Immediate cursor positioning
- **View Reset**: Automatically adjusts scroll to show cursor
- **Home/End Key Support**: Mapped to Home and End keys

update_input_scroll() (Internal)

```
fn update_input_scroll(&mut self)
```

Updates input scroll offset to keep cursor visible in long input text.

Features:

- **Automatic Scrolling**: Keeps cursor visible when input exceeds display width
- **Smooth Navigation**: Provides seamless editing experience for long input
- **Boundary Management**: Ensures proper scroll boundaries and cursor visibility

Algorithm:

```
// Ensures cursor stays visible by adjusting scroll offset
if self.cursor_position < self.input_scroll_offset {
    // Scroll left to show cursor
    self.input_scroll_offset = self.cursor_position;
} else if self.cursor_position >= self.input_scroll_offset + self.input_width {
    // Scroll right to show cursor
    self.input_scroll_offset = self.cursor_position - self.input_width + 1;
}
```

clear_input()

```
pub fn clear_input(&mut self)
```

Clears input text and resets all cursor and scroll state.

Features:

- **Complete Reset**: Clears text, cursor position, and scroll offset
- **Immediate Update**: Triggers UI redraw for instant feedback

get_visible_input()

```
pub fn get_visible_input(&self) -> (&str, usize)
```

Returns the visible portion of input text and the relative cursor position for display.

Returns:

- `(&str, usize)` - Tuple containing (visible_text_slice, relative_cursor_position)

Features:

- **Scroll-Aware**: Returns only the portion of text visible in the input area
- **Cursor Mapping**: Provides cursor position relative to the visible text
- **Width Adaptive**: Automatically adjusts based on available input width

Usage:

Used internally by the rendering system to display input text with proper scrolling for long input lines.

Example:

```
app.input_text = "This is a very long input that exceeds the terminal width".to_string();
app.cursor_position = 10;
```

(continues on next page)

(continued from previous page)

```
app.input_width = 20; // Limited display width

let (visible, cursor_pos) = app.get_visible_input();
// Returns appropriate slice and relative cursor position
```

take_input()

```
pub fn take_input(&mut self) -> Option<String>
```

Extracts input text for sending, with automatic trimming and state reset.

Returns:

- `Option<String>` - Trimmed input text if not empty and input enabled, None otherwise

Behavior:

- Returns trimmed text only if input is enabled and non-empty
- Automatically clears input and resets cursor after extraction
- Prevents input extraction during streaming or when disabled

Example:

```
if let Some(input) = app.take_input() {
    // Send input to LLM
    println!("Sending: {}", input);
    // Input automatically cleared and cursor reset
}
```

Streaming and Real-time Updates

start_streaming()

```
pub fn start_streaming(&mut self)
```

Initiates streaming mode with state protection, immediate feedback, and clean initialization.

Enhanced Features:

- **State Protection**: Ensures clean state before starting new stream by calling finish_streaming() if already busy
- **Immediate Placeholder**: Creates assistant message with “…” placeholder for streaming content
- **Visual Feedback**: Shows animated spinner (□ frame) and progress indicator starting at 0%
- **Input Management**: Disables input during streaming to prevent conflicts and state corruption
- **Clean Initialization**: Clears streaming buffer and resets progress tracking

Implementation Details:

```
// Clean state enforcement
if self.is_llm_busy {
    log::warn!("Starting new stream while already busy - forcing clean state");
    self.finish_streaming();
}

// Set streaming flags
self.is_llm_busy = true;
```

(continues on next page)

(continued from previous page)

```
self.is_input_disabled = true;
self.response_progress = 0.0;
self.streaming_buffer.clear();

// Create placeholder message
let initial_message = ChatMessage {
    message_type: MessageType::Assistant,
    content: vec![Line::from("...")],
    timestamp: Self::get_timestamp(),
};
self.chat_history.push(initial_message);
```

Example:

```
app.start_streaming();
// UI shows: "🚀 Sending request..." with animated spinner
// Input disabled, progress bar appears
// New assistant message with "..." placeholder added
```

update_streaming_content()

```
pub fn update_streaming_content(&mut self, content: &str)
```

Updates streaming content with intelligent rendering optimization, memory management, and real-time UI updates.

Parameters:

- `content` - New content chunk from LLM response

Advanced Features:

- **Buffer Management**: Prevents memory overflow with 1MB limit and intelligent truncation (keeps last 80% of content for context)
- **Smart Rendering**: Content-aware update frequency based on content characteristics and patterns
- **Memory Safety**: Thread-local size tracking prevents buffer overflow and performance degradation
- **Progress Tracking**: Dynamic progress calculation with visual feedback (0-95% during streaming)
- **Real-time Updates**: Always updates message content regardless of UI throttling for data consistency

Optimization Strategy:

```
// Buffer overflow protection
if self.streaming_buffer.len() + content.len() > MAX_STREAMING_BUFFER_SIZE {
    let keep_from = self.streaming_buffer.len() / 5;
    self.streaming_buffer = self.streaming_buffer[keep_from..].to_string();
}

// Immediate UI update triggers:
// - Small content (< 500 chars) - always responsive
// - Line breaks ("\n") - paragraph completion
// - Code blocks ("```") - syntax highlighting triggers
// - Headers ("##", "###") - section breaks
// - Lists ("- ", "* ") - bullet points
// - Sentence endings (". ", "? ", "! ") - natural breaks
```

(continues on next page)

(continued from previous page)

```
// - Text formatting ("**", "*") - emphasis changes
// - Regular intervals (every 200-250 chars) - prevents freezing
```

Performance Features:

- **Thread-local Tracking**: Efficient size-based update throttling using thread-local storage
- **Content-aware Updates**: Higher frequency for structured content (code, lists, headers)
- **Progressive Enhancement**: Gradual progress indicator updates (0.01-0.05 increments)
- **Memory Optimization**: Intelligent buffer management prevents excessive memory usage

Example:

```
app.update_streaming_content("Hello, this is a streaming response...\n");
// Updates buffer, triggers UI redraw due to line break
// Progress increases, typing indicator continues
// Message content updated in real-time
```

finish_streaming()

```
pub fn finish_streaming(&mut self)
```

Completes streaming with final content preservation, state cleanup, and pending input processing.

Critical Features:

- **Content Preservation**: Forces final UI update to transfer all buffered content to the final message
- **Intelligent Cleanup**: Removes placeholder messages if no content was received
- **State Reset**: Properly resets all streaming-related flags and progress indicators
- **Visual Completion**: Updates progress to 100% and shows ready state with success indicator
- **Message Validation**: Ensures assistant messages are properly finalized with timestamps

Implementation Details:

```
// Force final content update regardless of throttling
if !self.streaming_buffer.is_empty() {
    if let Some(last_msg) = self.chat_history.last_mut() {
        if last_msg.message_type == MessageType::Assistant {
            last_msg.content = markdown_to_lines(&self.streaming_buffer);
            last_msg.timestamp = Self::get_timestamp();
        }
    }
} else {
    // Remove placeholder if no content received
    if let Some(last_msg) = self.chat_history.last() {
        let is_placeholder = last_msg.content.len() == 1 &&
            last_msg.content[0].spans[0].content == "...";
        if is_placeholder {
            self.chat_history.pop();
        }
    }
}

// Complete state reset
self.streaming_buffer.clear();
```

(continues on next page)

(continued from previous page)

```
self.is_llm_busy = false;
self.is_input_disabled = false;
self.response_progress = 1.0;
self.typing_indicator.clear();
```

Recovery Features:

- **Placeholder Removal**: Automatically removes empty assistant messages if no content was received
- **Error Handling**: Gracefully handles edge cases like empty chat history or wrong message types
- **Memory Cleanup**: Clears streaming buffer after content transfer to prevent memory leaks
- **UI Synchronization**: Ensures final scroll position and redraw for proper display

Example:

```
app.finish_streaming();
// All buffered content transferred to final message
// Progress shows 100%, status shows "□ Ready"
// Input re-enabled, streaming flags cleared
```

add_streaming_message() (Internal)

```
fn add_streaming_message(&mut self)
```

Creates a new assistant message with "…" placeholder for streaming content.

Features:

- **Placeholder Creation**: Adds initial assistant message with temporary content
- **Visual Feedback**: Provides immediate indication that AI is responding
- **State Preparation**: Sets up message structure for streaming content updates

Implementation:

```
let assistant_message = ChatMessage {
    message_type: MessageType::Assistant,
    content: vec![Line::from("...")],
    timestamp: Self::get_timestamp(),
};
self.chat_history.push(assistant_message);
```

Usage:

Called internally by *start_streaming()* to prepare the chat interface for incoming AI responses.

Navigation and Display

scroll_up() / scroll_down()

```
pub fn scroll_up(&mut self)
pub fn scroll_down(&mut self)
```

Enhanced scrolling with content-aware positioning, proper bounds checking, and automatic state synchronization.

Features:

- **Bounds Checking**: Prevents scrolling beyond valid content range (0 to max_scroll())

- **State Synchronization**: Automatically updates internal scroll_state for scrollbar display
- **Performance Optimized**: Single-position increments for precise navigation
- **Content Awareness**: Respects actual content height and terminal dimensions

Implementation:

```
// scroll_up()
if self.scroll_position > 0 {
    self.scroll_position -= 1;
    self.update_scroll_state();
}

// scroll_down()
if self.scroll_position < self.max_scroll() {
    self.scroll_position += 1;
    self.update_scroll_state();
}
```

Usage Examples:

```
// Navigate chat history
app.scroll_up();    // Move toward older messages
app.scroll_down();  // Move toward newer messages

// Fast scrolling (5 positions at once)
for _ in 0..5 { app.scroll_up(); }    // Page Up behavior
for _ in 0..5 { app.scroll_down(); }  // Page Down behavior
```

scroll_to_bottom()

```
pub fn scroll_to_bottom(&mut self)
```

Instantly scrolls to the most recent messages with optimized positioning calculations.

Features:

- **Instant Navigation**: Jumps directly to the latest messages without animation
- **Automatic Calculation**: Uses max_scroll() to determine proper bottom position
- **State Synchronization**: Updates scroll_state for consistent scrollbar display
- **Content Tracking**: Automatically adjusts for changing chat history length

Auto-called when:

- New messages added to chat history
- Streaming responses complete
- User sends message
- Content updates require visibility

Implementation:

```
pub fn scroll_to_bottom(&mut self) {
    self.scroll_position = self.max_scroll();
    self.update_scroll_state();
}
```

Example:

```
app.scroll_to_bottom();
assert_eq!(app.scroll_position, app.max_scroll());
// User now sees the most recent messages
```

update_scroll_state()

```
pub fn update_scroll_state(&mut self)
```

Synchronizes internal scroll state with UI scrollbar using accurate text wrapping calculations for consistent display and user feedback.

Enhanced Features:

- **Text Wrapping Aware**: Recalculates total rendered lines with proper wrapping logic
- **Scrollbar Synchronization**: Updates ScrollbarState position and content length accurately
- **Terminal Width Adaptive**: Uses current terminal width for wrapping calculations
- **Consistent Line Counting**: Matches the same logic used in max_scroll() for accuracy
- **State Management**: Maintains internal state consistency across navigation operations

Improved Implementation:

```
pub fn update_scroll_state(&mut self) {
    // Calculate terminal width for text wrapping calculations
    let chat_width = self.input_width.saturating_sub(4).max(20);

    // Calculate total rendered lines accounting for text wrapping
    let total_rendered_lines: usize = self.chat_history
        .iter()
        .map(|msg| {
            let mut lines = 1; // Header line

            // Content lines - account for text wrapping
            for line in &msg.content {
                let line_text = line.spans.iter()
                    .map(|span| span.content.as_ref())
                    .collect::<String>();

                if line_text.trim().is_empty() {
                    lines += 1; // Empty lines
                } else {
                    // Character-based text wrapping calculation
                    let display_width = line_text.chars().count();
                    if display_width <= chat_width {
                        lines += 1;
                    } else {
                        let wrapped_lines = (display_width + chat_width - 1) / chat_width;
                        lines += wrapped_lines.max(1);
                    }
                }
            }

            lines += 1; // Separator line after each message
            lines
        })
```

(continues on next page)

(continued from previous page)

```
        .sum();

    self.scroll_state = self.scroll_state
        .content_length(total_rendered_lines.max(1))
        .position(self.scroll_position);
}
```

Key Improvements:

- **Accurate content length**: Sets scrollbar content length to match actual rendered lines
- **Character-aware wrapping**: Uses *.chars().count()* for proper Unicode text measurement
- **Consistent logic**: Uses identical calculation method as max_scroll() for reliability

max_scroll()

```
pub fn max_scroll(&self) -> usize
```

Calculates the maximum valid scroll position with accurate text wrapping calculations and conservative buffering to ensure content visibility.

Enhanced Features:

- **Text Wrapping Aware**: Accurately calculates wrapped lines using character counts instead of byte lengths
- **Terminal Width Adaptive**: Accounts for actual terminal width and border spacing
- **Conservative Buffering**: Reduces max scroll by 1 position to prevent content cutoff at bottom
- **Separator Line Tracking**: Always includes separator lines after each message for consistency
- **Border Accounting**: Properly calculates available chat area excluding UI borders

Improved Algorithm:

```
// Calculate visible height for the chat area
let chat_area_height = self.terminal_height.saturating_sub(11).max(1);
let visible_height = chat_area_height.saturating_sub(2).max(1); // Account for borders

// Calculate terminal width for text wrapping calculations
let chat_width = self.input_width.saturating_sub(4).max(20); // Account for borders and␣
↪padding

// Calculate the actual rendered lines accounting for text wrapping
let total_rendered_lines: usize = self.chat_history
    .iter()
    .map(|msg| {
        let mut lines = 1; // Header line

        // Content lines - account for text wrapping
        for line in &msg.content {
            let line_text = line.spans.iter()
                .map(|span| span.content.as_ref())
                .collect::<String>();

            if line_text.trim().is_empty() {
                lines += 1; // Empty lines
            } else {
                // More accurate text wrapping calculation using char count
```

(continues on next page)

(continued from previous page)

```
                let display_width = line_text.chars().count();
                if display_width <= chat_width {
                    lines += 1;
                } else {
                    let wrapped_lines = (display_width + chat_width - 1) / chat_width;
                    lines += wrapped_lines.max(1);
                }
            }
        }

        lines += 1; // Separator line after each message
        lines
    })
    .sum();

// Conservative scroll calculation to prevent content cutoff
if total_rendered_lines > visible_height {
    let max_scroll = total_rendered_lines.saturating_sub(visible_height);
    max_scroll.saturating_sub(1) // Buffer to ensure last lines are visible
} else {
    0
}
```

Key Improvements:

- **Character-based wrapping**: Uses *.chars().count()* for accurate Unicode text measurement
- **Conservative max scroll**: Reduces by 1 to ensure bottom content is always accessible
- **Consistent separator handling**: Always adds separator lines for uniform spacing
- **Robust border calculation**: Properly accounts for terminal borders and padding

Animation and Visual Feedback

update_typing_indicator()

```
pub fn update_typing_indicator(&mut self)
```

Updates animated typing indicator with smooth character transitions and context-aware animation.

Features:

- **Smooth Animation**: 10-frame Unicode spinner animation cycle with 100ms timing
- **Context Aware**: Only animates when LLM is actively generating responses (is_llm_busy)
- **Performance Optimized**: Efficient time-based frame selection using system time
- **Memory Efficient**: Clears indicator when not in use to prevent unnecessary updates

Animation Frames:

```
let indicators = ["□", "□", "□", "□", "□", "□", "□", "□", "□", "□"];
let current_time = SystemTime::now()
    .duration_since(UNIX_EPOCH)
    .unwrap()
    .as_millis();
let index = (current_time / 100) % indicators.len() as u128;
```

Implementation Details:

- **Time-based**: Uses system time for consistent animation speed across different systems
- **Frame Rate**: 10 FPS (100ms per frame) for smooth visual experience without CPU waste
- **State Management**: Automatically clears when streaming stops to save resources

Example:

```
app.is_llm_busy = true;
app.update_typing_indicator();
assert!(!app.typing_indicator.is_empty()); // Contains current spinner frame

app.is_llm_busy = false;
app.update_typing_indicator();
assert!(app.typing_indicator.is_empty()); // Cleared when not busy
```

tick()

```
pub fn tick(&mut self)
```

Handles periodic updates for animations, cursor blinking, and visual effects with optimized timing.

Timing System:

- **Cursor Blink**: 500ms intervals for natural text cursor blinking
- **Animation Updates**: 50ms intervals for smooth spinner transitions (20 FPS)
- **Performance Balanced**: Optimized timing to prevent CPU waste while maintaining smooth visuals
- **State-based Updates**: Only triggers redraws when visual state actually changes

Implementation:

```
let now = Instant::now();

// Cursor blinking (500ms cycle)
if now.duration_since(self.last_cursor_blink) >= Duration::from_millis(500) {
    self.cursor_blink_state = !self.cursor_blink_state;
    self.last_cursor_blink = now;
    if !self.is_input_disabled {
        self.needs_redraw = true;
    }
}

// Animation updates (50ms cycle)
if now.duration_since(self.last_animation_tick) >= Duration::from_millis(50) {
    if self.is_llm_busy {
        self.update_typing_indicator();
        self.needs_redraw = true;
    }
    self.last_animation_tick = now;
}
```

Features:

- **Conditional Updates**: Only updates cursor when input is enabled
- **Animation Management**: Handles typing indicator updates during LLM processing
- **Efficient Timing**: Uses separate timers for different visual elements
- **Resource Optimization**: Prevents unnecessary redraws when not needed

Example Usage:

```
// In main event loop (60 FPS render cycle):
app.tick(); // Updates all animations and cursor
if app.needs_redraw {
    terminal.draw(|f| draw_enhanced_ui(f, &mut app, &model_name))?;
    app.needs_redraw = false;
}
```

Event System

AppEvent

```
#[derive(Debug)]
pub enum AppEvent {
    Quit,           // Application should terminate
    Redraw,         // UI needs immediate redraw
    Key(KeyEvent),  // Keyboard input event
    Tick,           // Periodic timer event
}
```

Enhanced event system for the responsive async UI loop, supporting immediate user feedback and smooth animations.

Event Types:

- `Quit` - Clean application shutdown requested
- `Redraw` - Immediate UI refresh needed (for responsive input)
- `Key(KeyEvent)` - User keyboard input with full key details
- `Tick` - Periodic updates for animations and cursor blinking

Event Priorities in Main Loop:

1. **Highest**: LLM response processing (real-time streaming)
2. **High**: Terminal input events (immediate user feedback)
3. **Medium**: Rendering updates (~60 FPS for smooth UI)
4. **Low**: Background tasks and cleanup

Advanced UI Functions

Enhanced Event Loop

run_ui()

```
pub async fn run_ui(
    terminal: &mut Terminal<CrosstermBackend<io::Stdout>>,
    config: AppConfig,
    model_name: String,
    api_key: String,
    provider: Arc<GenAIProvider>
) -> Result<()>
```

Runs the enhanced asynchronous UI event loop with prioritized event handling, real-time responsiveness, and optimized performance.

Features:

- **Async Event Processing**: Non-blocking event handling with proper priority-based processing
- **Real-time Streaming**: Immediate LLM response processing with EOT signal prioritization

- **Responsive Input**: Instant feedback for user typing and navigation (immediate redraw)
- **Smooth Rendering**: ~60 FPS updates for fluid animations and visual feedback
- **Resource Optimization**: Balanced CPU usage, battery efficiency, and memory management

Event Processing Architecture:

```rust
// Setup event channels and intervals
let (tx, mut rx) = mpsc::unbounded_channel();
let mut event_stream = EventStream::new();
let mut tick_interval = tokio::time::interval(Duration::from_millis(50));
let mut render_interval = tokio::time::interval(Duration::from_millis(16)); // ~60 FPS

loop {
    tokio::select! {
        // Priority 1: LLM responses (highest priority for real-time streaming)
        llm_message = rx.recv() => {
            // Collect all available messages to prioritize EOT signals
            let mut all_messages = vec![message];
            while let Ok(additional) = rx.try_recv() {
                all_messages.push(additional);
            }

            // Process EOT signals first to prevent state confusion
            // Then process content messages in order
            for msg in all_messages {
                handle_llm_response(&mut app, msg, &provider, &model_name, &tx).await;
            }

            // Force immediate redraw for streaming responses
            terminal.draw(|f| draw_enhanced_ui(f, &mut app, &model_name))?;
        }

        // Priority 2: User input (immediate feedback)
        event_result = event_stream.next() => {
            if let Some(app_event) = handle_terminal_event(
                &mut app, event, &tx, &api_key, &model_name, &provider
            ).await {
                match app_event {
                    AppEvent::Quit => break,
                    AppEvent::Redraw => {
                        // Force immediate redraw for user input responsiveness
                        terminal.draw(|f| draw_enhanced_ui(f, &mut app, &model_name))?;
                    }
                    _ => {}
                }
            }
        }

        // Priority 3: Regular rendering updates (~60 FPS)
        _ = render_interval.tick() => {
            app.tick(); // Handle animations and cursor blinking
            if app.needs_redraw {
                terminal.draw(|f| draw_enhanced_ui(f, &mut app, &model_name))?;
                app.needs_redraw = false;
```

(continues on next page)

(continued from previous page)

```
            }
        }

        // Priority 4: Background tasks and cleanup
        _ = tick_interval.tick() => {
            // Additional background processing if needed
        }
    }
}
```

Advanced Features:

- **EOT Signal Prioritization**: Processes end-of-transmission signals first to prevent state confusion
- **Message Batching**: Collects multiple messages from channel to optimize processing
- **Immediate Feedback**: Forces UI updates for user input to maintain responsiveness
- **Animation Management**: Separate timing for smooth visual effects and cursor blinking
- **Memory Management**: Efficient event processing without memory leaks or buffer overflow

Performance Optimizations:

- **Non-blocking Events**: Uses 10ms polling for terminal events to balance responsiveness and CPU usage
- **Selective Rendering**: Only redraws when needs_redraw flag is set or immediate feedback required
- **Efficient Intervals**: Separate timers for different update frequencies (16ms render, 50ms tick)
- **Resource Cleanup**: Proper terminal cleanup on exit with error handling

Example Usage:

```
let mut terminal = setup_terminal()?;
let config = AppConfig::load()?;
let provider = Arc::new(GenAIProvider::new(api_key.clone()));

// Run the UI loop
run_ui(&mut terminal, config, model_name, api_key, provider).await?;

// Terminal automatically cleaned up on exit
        // Cleanup and maintenance
    }
}
```

Example:

```
let mut terminal = setup_terminal()?;
let config = AppConfig::load()?;
let provider = Arc::new(GenAIProvider::new()?);

run_ui(&mut terminal, config, "gpt-4o-mini".to_string(),
       api_key, provider).await?;
```

Enhanced Rendering

draw_enhanced_ui()

```
fn draw_enhanced_ui(f: &mut Frame, app: &mut App, model_name: &str)
```

Main rendering function with enhanced layout, cursor visualization, and responsive design.

Layout Structure:

```
┌─────────────────────────────────────────────────────────┐
│ □ Perspt │ Model: gpt-4o-mini │ Status: □ Ready        │ 3 lines
├─────────────────────────────────────────────────────────┤
│               Chat History Area                         │
│ □ You • 14:30                                           │ flexible
│ Hello, can you help me with Rust?                       │ (main area)
│                                                         │
│ 🤖 Assistant • 14:30                                     │
│ Of course! I'd be happy to help with Rust.              │
│ ┌─ Code ─┐                                              │
│ │ let x = 42;                                           │
│ └─────────┘                                             │
│                                                         │
├─────────────────────────────────────────────────────────┤
│ > Type your message here...                   ▌         │ 3 lines
│ ┌─ Progress Bar ─┐                                      │ 2 lines
│ │ ████████████████████████                        │     │
│ └─────────────────────────────────────────────────┘     │
│                                                         │
├─────────────────────────────────────────────────────────┤
│ Status: Ready │ Ctrl+C to exit                          │ 3 lines
└─────────────────────────────────────────────────────────┘
```

Features:

- **Adaptive Layout**: Responds to terminal size changes
- **Rich Header**: Model info, status, and visual indicators
- **Enhanced Chat Area**: Icons, timestamps, and markdown rendering
- **Cursor Visualization**: Blinking cursor with position indication
- **Progress Feedback**: Real-time progress bars during AI responses
- **Contextual Status**: Dynamic status information and shortcuts

draw_enhanced_input_area()

```
fn draw_enhanced_input_area(f: &mut Frame, area: Rect, app: &App)
```

Advanced input area rendering with visible cursor, scrolling support, and contextual feedback.

Features:

- **Visible Cursor**: Blinking cursor with character-level positioning
- **Horizontal Scrolling**: Support for long input text with auto-scroll
- **State Indicators**: Visual feedback for input disabled/enabled states
- **Progress Integration**: Shows typing progress and queue status
- **Contextual Hints**: Dynamic hints based on application state

Cursor Rendering:

```
// Cursor visualization with blinking
let cursor_style = if app.cursor_blink_state {
    Style::default().fg(Color::Black).bg(Color::White)  // Visible
} else {
    Style::default().fg(Color::White).bg(Color::DarkGray) // Dimmed
};
```

Markdown Processing

markdown_to_lines()

```
fn markdown_to_lines(markdown: &str) -> Vec<Line<'static>>
```

Advanced markdown parser converting text to richly formatted terminal output with syntax highlighting and visual enhancements.

Supported Elements:

Element	Syntax	Terminal Rendering
Headers	`# Header`	Colored and bold text by level
Code Blocks	`` `rust\ncode\n` ``	Bordered boxes with syntax highlighting
Inline Code	`` `code` ``	Highlighted background color
Bold Text	`**bold**`	Bold terminal styling
Italic Text	`*italic*`	Italic terminal styling
Lists	`- item` or `* item`	Colored bullet points with proper indentation
Block Quotes	`> quote`	Left border with italic text
Line Breaks	Empty lines	Proper spacing preservation

Code Block Rendering:

```
┌─ rust ─┐
│ let greeting = "Hello, World!";      │
│ println!("{}", greeting);            │
└──────────────┘
```

Features:

- **Syntax-Aware**: Different colors for different code languages
- **Performance Optimized**: Efficient parsing for real-time streaming
- **Terminal-Friendly**: Colors and styles optimized for terminal display
- **Robust Parsing**: Handles malformed markdown gracefully

Example:

~~~
let markdown = r#"
# Example Response

Here's some **bold** text and `inline code`.

```rust
fn main() {
    println!("Hello, World!");
}
```

- First item
- Second item with *emphasis*
"#;

let formatted_lines = markdown_to_lines(markdown);
// Returns fully styled lines ready for terminal display
~~~

Error Handling and Recovery

categorize_error()

```
fn categorize_error(error_msg: &str) -> ErrorState
```

Intelligent error analysis and categorization with automatic recovery suggestions.

Analysis Process:

1. **Pattern Matching**: Analyzes error message content for known patterns
2. **Context Extraction**: Extracts relevant technical details
3. **User Translation**: Converts technical errors to user-friendly messages
4. **Recovery Guidance**: Provides specific next steps based on error type

Recognition Patterns:

```
// Network errors
if error_lower.contains("network") || error_lower.contains("connection") {
    // Suggests checking internet connection
}

// Authentication errors
if error_lower.contains("api key") || error_lower.contains("unauthorized") {
    // Suggests checking API key configuration
}

// Rate limiting
if error_lower.contains("rate limit") || error_lower.contains("too many") {
    // Suggests waiting before retry
}
```

Performance and Optimization

Buffer Management:

```
const MAX_STREAMING_BUFFER_SIZE: usize = 1_000_000; // 1MB limit
const UI_UPDATE_INTERVAL: usize = 500;              // Update frequency
const SMALL_BUFFER_THRESHOLD: usize = 500;          // Immediate updates
```

Rendering Optimization:

- **Intelligent Redraw**: Only updates when `needs_redraw` flag is set
- **Streaming Throttling**: Balances responsiveness with performance
- **Animation Timing**: Optimized intervals for smooth visual effects
- **Memory Management**: Prevents buffer overflow during long responses

Event Loop Efficiency:

- **Priority-Based Processing**: Critical events processed first
- **Non-Blocking Operations**: Prevents UI freezing during long operations
- **Resource Management**: Balanced CPU usage and battery life

See Also

- *Basic Usage* - Basic usage guide
- *Extending Perspt* - UI extension guide
- *Configuration Module* - Configuration module
- *LLM Provider Module* - LLM provider integration

Rich terminal-based user interface using Ratatui framework. Handles:

- **Real-time Chat** - Interactive chat interface with live markdown rendering
- **Streaming Display** - Real-time response streaming with configurable buffer management
- **State Management** - Comprehensive application state with cursor position tracking
- **Error Handling** - User-friendly error display with categorized error types
- **Responsive Design** - Adaptive layout with scrollable history and progress indicators

Key Components:

- `App` structure with enhanced state management and cursor tracking
- `ChatMessage` and `MessageType` for styled message representation
- `ErrorState` and `ErrorType` for comprehensive error handling
- Real-time event handling with non-blocking input processing
- Streaming buffer management with configurable update intervals

Module Interactions

Data Flow

```
User Input → UI Module → Main Module → GenAI Provider → LLM API
     ↑              ↓           ↓             ↓             ↓
Terminal ← Real-time ← Event ← Streaming ← API Response ← Provider
        Rendering    Loop     Channel
```

Enhanced Flow Description:

1. **User Input**: User types in terminal, captured by UI module with cursor tracking
2. **Event Processing**: Main module coordinates actions with comprehensive panic handling
3. **Configuration**: Config module provides auto-configured provider settings
4. **LLM Request**: GenAI provider handles API communication with environment-based auth
5. **Streaming Processing**: Real-time response streaming through unbounded channels
6. **UI Update**: UI module renders responses with markdown formatting and progress indicators

Configuration Flow

```
Config File/CLI → Config Module → Provider Validation → UI Setup
     ↓                ↓                    ↓              ↓
Defaults → Processing → Type Inference → Model List → Application Start
```

Configuration Steps:

1. **Loading**: Configuration loaded from file or generated with defaults
2. **Processing**: Provider type inference and validation
3. **Provider Setup**: LLM provider initialized with configuration
4. **Validation**: Provider configuration validated before use
5. **UI Initialization**: Application state initialized with valid configuration

Error Handling Flow

```
Error Source → Module Handler → Error State → UI Display → User Action
     ↓              ↓              ↓            ↓            ↓
Network ─────→ LLM Provider ───→ Main ────→ UI ─────→ Recovery
```

Error Handling:

- **Network Errors**: Handled by LLM provider with retry logic
- **Configuration Errors**: Caught by config module with helpful messages
- **UI Errors**: Managed by UI module with graceful degradation
- **System Errors**: Handled by main module with proper cleanup

Module Dependencies

Dependency Graph

```
main
├── config (configuration management)
├── llm_provider (AI integration)
├── ui (user interface)
└── External Dependencies:
    ├── tokio (async runtime)
    ├── ratatui (TUI framework)
    ├── crossterm (terminal control)
    ├── anyhow (error handling)
    ├── serde (serialization)
    └── genai (LLM provider APIs)
```

Dependency Relationships:

- `main` depends on all other modules
- `ui` uses `config` for application state
- `llm_provider` uses `config` for provider settings
- All modules use common external dependencies

External Integrations

AI Provider APIs (via GenAI crate):

- OpenAI GPT models (GPT-4o, GPT-4o-mini, o1-preview, o1-mini)
- Anthropic Claude models (Claude 3.5 Sonnet, Claude 3 Opus/Sonnet/Haiku)
- Google Gemini models (Gemini 1.5 Pro/Flash, Gemini 2.0 Flash)
- Groq models (Llama 3.x with ultra-fast inference)
- Cohere models (Command R, Command R+)
- XAI models (Grok)

Terminal and System:

- Cross-platform terminal control via `crossterm` (Windows, macOS, Linux)
- Real-time markdown rendering with `ratatui`
- Async I/O with `tokio` runtime
- Environment variable integration for secure authentication

Module Testing

Each module includes comprehensive testing aligned with current implementation:

Unit Tests:

- Configuration loading with intelligent defaults (`test_load_config_defaults`)
- Provider type inference and validation (`test_load_config_from_json_string_infer_provider_type_*`)
- GenAI provider creation and model listing
- UI state management and error handling
- Panic handling and terminal restoration

Integration Tests:

- End-to-end configuration flow with all supported providers
- GenAI provider initialization with environment variables
- Streaming response handling with channel communication
- UI event processing and state updates
- Error propagation and user-friendly display

Current Test Examples:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[tokio::test]
    async fn test_load_config_defaults() {
        let config = load_config(None).await.unwrap();
        assert_eq!(config.provider_type, Some("openai".to_string()));
        assert_eq!(config.default_model, Some("gpt-4o-mini".to_string()));
        assert!(config.providers.contains_key("openai"));
        assert!(config.providers.contains_key("groq"));
    }

    #[tokio::test]
    async fn test_genai_provider_creation() {
        let provider = GenAIProvider::new();
        assert!(provider.is_ok());
    }
}
    use super::*;

    #[test]
    fn test_config_provider_inference() {
        // Test automatic provider type inference
    }

    #[tokio::test]
    async fn test_llm_provider_integration() {
        // Test LLM provider functionality
    }

    #[test]
    fn test_ui_message_formatting() {
        // Test message display and formatting
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Best Practices

When working with Perspt modules:

Configuration:

- Use `load_config(None)` for defaults or `load_config(Some(&path))` for custom files
- Leverage automatic provider type inference with `process_loaded_config()`
- Validate configuration before GenAI provider initialization
- Use environment variables for secure API key management

GenAI Provider:

- Use `GenAIProvider::new()` for auto-configuration via environment variables
- Use `GenAIProvider::new_with_config()` for explicit provider/key setup
- Handle streaming responses with unbounded channels for real-time display
- Implement proper error handling with `anyhow::Result` for detailed context

UI Development:

- Follow established `MessageType` conventions (User, Assistant, Error, etc.)
- Use `ErrorState` and `ErrorType` for categorized error display
- Maintain responsive UI with configurable streaming buffer intervals
- Implement proper cursor position tracking for enhanced user experience

Error Handling:

- Use `anyhow::Result` throughout for comprehensive error context
- Implement panic hooks for terminal state restoration
- Provide user-friendly error messages with recovery suggestions
- Use `ErrorType` categories for appropriate styling and user guidance

Performance:

- Use streaming buffers with size limits (`MAX_STREAMING_BUFFER_SIZE`)
- Update UI responsively with configurable intervals (`UI_UPDATE_INTERVAL`)
- Leverage async/await patterns for non-blocking operations
- Properly manage terminal raw mode state for clean shutdown

See Also

- *Architecture* - Detailed architecture guide
- *Extending Perspt* - Module extension guide
- *Troubleshooting* - Common issues and solutions

3.7.2 Overview

The Perspt API is organized into four main modules, each with a specific responsibility:

🏷️ Configuration (config.rs) Configuration management, file parsing, and environment variable handling.

Configuration Module 🤖 LLM Provider (llm_provider.rs) Unified interface to multiple AI providers with automatic model discovery.

LLM Provider Module 🎨 User Interface (ui.rs) Terminal-based chat interface with real-time rendering and event handling.

User Interface Module 🚀 Main Application (main.rs) Application entry point, CLI parsing, and lifecycle management.

Main Module

3.7.3 Architecture Overview

```
┌─────────────────────────────────────────────────────┐
│                     main.rs                         │
│              (Application Entry)                    │
├─────────────────────────────────────────────────────┤
│  • CLI argument parsing with clap                   │
│  • Application initialization & config loading      │
│  • Comprehensive panic handling & recovery          │
│  • Terminal setup, cleanup & state management       │
│  • Event loop coordination                          │
└─────────────────┬───────────────────────────────────┘
                  │
    ┌─────────────┼─────────────┐
    ▼             ▼             ▼
┌───────────┐ ┌───────────┐ ┌───────────────┐
│ config.rs │ │   ui.rs   │ │ llm_provider  │
│           │ │           │ │     .rs       │
├───────────┤ ├───────────┤ ├───────────────┤
│ • Multi-  │ │ • Ratatui │ │ • GenAI       │
│   provider│ │   TUI     │ │   client      │
│ • Smart   │ │ • Real-   │ │ • Multi-      │
│   defaults│ │   time    │ │   provider    │
│ • Type    │ │   markdown│ │ • Streaming   │
│  inference│ │ • Scroll- │ │ • Auto-config │
│ • JSON    │ │   able    │ │ • Error       │
│   config  │ │   history │ │   handling    │
└───────────┘ └───────────┘ └───────────────┘
```

3.7.4 Module Dependencies

The modules have clear dependency relationships:

main.rs

- Application orchestrator and entry point
- Uses all other modules for complete functionality
- Handles panic recovery and terminal state management
- Coordinates event loop and user interactions

config.rs

- Standalone configuration management
- Supports 8+ LLM providers with intelligent defaults
- JSON-based configuration with environment variable integration
- Provider type inference and validation

llm_provider.rs

- Uses modern *genai* crate for unified provider interface
- Supports OpenAI, Anthropic, Google (Gemini), Groq, Cohere, XAI, DeepSeek, Ollama
- Auto-configuration via environment variables
- Streaming response handling and model discovery

ui.rs

- Rich terminal UI using Ratatui framework
- Real-time markdown rendering and streaming support
- Scrollable chat history with responsive event handling
- Enhanced input management with cursor positioning

3.7.5 Key Structures and Interfaces

GenAIProvider Struct

The modern unified provider implementation using the genai crate:

```rust
pub struct GenAIProvider {
    client: Client,
}

impl GenAIProvider {
    /// Creates provider with auto-configuration
    pub fn new() -> Result<Self>

    /// Creates provider with explicit configuration
    pub fn new_with_config(
        provider_type: Option<&str>,
        api_key: Option<&str>
    ) -> Result<Self>

    /// Generates simple text response
    pub async fn generate_response_simple(
        &self,
        model: &str,
        message: &str
    ) -> Result<String>

    /// Generates streaming response to channel
    pub async fn generate_response_stream_to_channel(
        &self,
        model: &str,
        message: &str,
        sender: mpsc::UnboundedSender<String>
    ) -> Result<()>

    /// Lists available models for current provider
    pub async fn list_models(&self) -> Result<Vec<String>>
}
```

Supported Providers

The GenAI provider supports multiple LLM services:

Provider	Environment Variable	Supported Models
OpenAI	`OPENAI_API_KEY`	GPT-4o, GPT-4o-mini, GPT-4, GPT-3.5, o1-preview, o1-mini
Anthropic	`ANTHROPIC_API_KEY`	Claude 3.5 Sonnet, Claude 3 Opus/Sonnet/Haiku
Google	`GEMINI_API_KEY`	Gemini 1.5 Pro/Flash, Gemini 2.0 Flash
Groq	`GROQ_API_KEY`	Llama 3.x models with ultra-fast inference
Cohere	`COHERE_API_KEY`	Command R, Command R+
XAI	`XAI_API_KEY`	Grok models

3.7.6 Error Handling

Perspt uses comprehensive error handling with proper context and user-friendly messages:

```rust
use anyhow::{Context, Result};

// All functions return Result<T> with proper error context
pub async fn load_config(config_path: Option<&String>) -> Result<AppConfig> {
    // Configuration loading with detailed error context
}

pub async fn generate_response_simple(
    &self,
    model: &str,
    message: &str
) -> Result<String> {
    // Provider communication with error handling
}
```

3.7.7 Configuration System

The configuration system supports multiple sources with intelligent defaults:

1. **JSON Configuration Files** (explicit configuration)
2. **Environment Variables** (for API keys and credentials)
3. **Intelligent Defaults** (comprehensive provider endpoints)
4. **Provider Type Inference** (automatic detection)

```rust
#[derive(Debug, Clone, Deserialize, PartialEq)]
pub struct AppConfig {
    pub providers: HashMap<String, String>,
    pub api_key: Option<String>,
    pub default_model: Option<String>,
    pub default_provider: Option<String>,
    pub provider_type: Option<String>,
}

// Load configuration with smart defaults
pub async fn load_config(config_path: Option<&String>) -> Result<AppConfig>

// Process configuration with provider type inference
pub fn process_loaded_config(mut config: AppConfig) -> AppConfig
```

Provider Type Inference

The configuration system automatically infers provider types from provider names:

Provider Name	Inferred Type	Notes
`openai`	`openai`	Direct mapping
`anthropic`	`anthropic`	Direct mapping
`google`, `gemini`	`google`	Multiple aliases supported
`groq`	`groq`	Fast inference provider
`cohere`	`cohere`	Command models
`xai`	`xai`	Grok models
Unknown	`openai`	Fallback default

3.7.8 Async Architecture

Perspt is built on Tokio's async runtime for high-performance concurrent operations:

Streaming Responses	Real-time display of AI responses as they're generated using async channels
Non-blocking UI	User can continue typing while AI responses stream in real-time
Concurrent Operations	Multiple API calls and UI updates happen simultaneously without blocking
Resource Efficiency	Minimal memory footprint with efficient async/await patterns

3.7.9 Type Safety

Rust's type system ensures correctness throughout the codebase:

Option Types	Explicit handling of optional values prevents null pointer errors
Result Types	All fallible operations return Result for explicit error handling
Strong Typing	Configuration, messages, and provider types are strongly typed
Compile-time Guarantees	Many errors are caught at compile time rather than runtime

3.7.10 Performance Considerations

Memory Management

- **Streaming buffers** with configurable size limits (1MB max)
- **Efficient VecDeque** for chat history with automatic cleanup
- **RAII patterns** for automatic resource cleanup
- **Minimal allocations** in hot paths for better performance

Network Efficiency

- **GenAI client pooling** handles connection reuse automatically
- **Streaming responses** reduce memory usage for long responses
- **Timeout handling** with proper error recovery
- **Environment-based auth** avoids credential storage

UI Performance

- **Real-time rendering** with responsive update intervals (500 chars)
- **Efficient scrolling** with proper state management
- **Markdown rendering** using optimized terminal formatting
- **Non-blocking input** with cursor position management
- **Progress indicators** for better user feedback

Terminal Integration

- **Crossterm compatibility** across platforms (Windows, macOS, Linux)
- **Raw mode management** with proper cleanup on panic
- **Alternate screen** support for clean terminal experience
- **Unicode support** for international characters and emojis

3.7.11 API Stability

Version Compatibility

Perspt follows semantic versioning:

- **Major versions** may include breaking API changes
- **Minor versions** add features while maintaining compatibility
- **Patch versions** fix bugs without changing public APIs

Deprecation Policy

- **Deprecated features** are marked in documentation
- **Migration guides** provided for breaking changes
- **Compatibility period** of at least one major version
- **Clear communication** about upcoming changes

3.7.12 Usage Examples

3.7.13 Usage Examples

Basic Provider Usage

```rust
use perspt::llm_provider::GenAIProvider;
use tokio::sync::mpsc;

#[tokio::main]
async fn main() -> Result<()> {
    // Create provider with auto-configuration
    let provider = GenAIProvider::new()?;

    // Simple text generation
    let response = provider.generate_response_simple(
        "gpt-4o-mini",
        "Hello, how are you?"
    ).await?;

    println!("Response: {}", response);
    Ok(())
}
```

Streaming Response Usage

```
use perspt::llm_provider::GenAIProvider;
use tokio::sync::mpsc;

#[tokio::main]
async fn main() -> Result<()> {
    let provider = GenAIProvider::new()?;
    let (tx, mut rx) = mpsc::unbounded_channel();

    // Start streaming response
    provider.generate_response_stream_to_channel(
        "gpt-4o-mini",
        "Tell me a story",
        tx
    ).await?;

    // Process streaming chunks
    while let Some(chunk) = rx.recv().await {
        print!("{}", chunk);
        std::io::stdout().flush()?;
    }

    Ok(())
}
```

Configuration Loading

```
use perspt::config::{AppConfig, load_config};

#[tokio::main]
async fn main() -> Result<()> {
    // Load with defaults (no config file)
    let config = load_config(None).await?;

    // Load from specific file
    let config = load_config(Some(&"config.json".to_string())).await?;

    println!("Provider: {:?}", config.provider_type);
    println!("Model: {:?}", config.default_model);
    Ok(())
}
```

Custom UI Events

```
use perspt::ui::{App, AppEvent};
use crossterm::event::{self, Event, KeyCode};

fn handle_events(app: &mut App) -> Result<()> {
    if event::poll(Duration::from_millis(100))? {
        if let Event::Key(key) = event::read()? {
            match key.code {
```

(continues on next page)

(continued from previous page)

```
                KeyCode::Enter => {
                    app.handle_event(AppEvent::SendMessage)?;
                }
                KeyCode::Char(c) => {
                    app.handle_event(AppEvent::Input(c))?;
                }
                _ => {}
            }
        }
    }
    Ok(())
}
```

3.7.14 Testing APIs

3.7.15 Testing APIs

Unit Testing

Each module includes comprehensive unit tests:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[tokio::test]
    async fn test_load_config_defaults() {
        let config = load_config(None).await.unwrap();
        assert_eq!(config.provider_type, Some("openai".to_string()));
        assert_eq!(config.default_model, Some("gpt-4o-mini".to_string()));
    }

    #[tokio::test]
    async fn test_provider_creation() {
        let provider = GenAIProvider::new().unwrap();
        // Provider created successfully
    }
}
```

Integration Testing

End-to-end tests validate complete workflows:

```
#[tokio::test]
async fn test_streaming_response() {
    let provider = GenAIProvider::new().unwrap();
    let (tx, mut rx) = mpsc::unbounded_channel();

    provider.generate_response_stream_to_channel(
        "gpt-4o-mini",
        "Hello",
        tx
```

(continues on next page)

(continued from previous page)

```
    ).await.unwrap();

    // Verify streaming works
    let first_chunk = rx.recv().await;
    assert!(first_chunk.is_some());
}
```

3.7.16 Documentation Generation

API documentation is automatically generated from source code:

```
# Generate Rust documentation
cargo doc --open --no-deps --all-features

# Build Sphinx documentation
cd docs/perspt_book && uv run make html
cargo doc --document-private-items

# Generate for specific package
cargo doc --package perspt
```

3.7.17 Best Practices

When using the Perspt API:

1. **Always handle errors** explicitly with Result types
2. **Use async/await** for all I/O operations
3. **Prefer streaming** for better user experience
4. **Validate configuration** before using providers
5. **Test provider connectivity** before starting conversations
6. **Handle network timeouts** gracefully
7. **Use appropriate logging** levels for debugging

See also

- *Developer Guide* - Development guidelines and architecture
- *User Guide* - User-focused documentation
- GitHub Repository - Source code and examples

3.8 Changelog

All notable changes to Perspt will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

3.8.1 [0.4.6] - 2025-06-29

Added

- □□ **Simple CLI Mode (PSP-000003)**: Brand new minimal command-line interface for direct Q&A without TUI overlay
  - `--simple-cli` flag enables Unix-style prompt interface perfect for scripting and automation

  - `--log-file <FILE>` option provides built-in session logging with timestamps
  - Real-time streaming responses in simple text format for immediate feedback
  - Clean exit handling with `Ctrl+D`, `exit` command, or `Ctrl+C` interrupt
  - Works seamlessly with all existing providers, models, and authentication methods
  - Perfect for accessibility needs, scripting workflows, and Unix philosophy adherents
- **Scripting and Automation Support**: Simple CLI mode enables powerful automation scenarios
  - Pipe questions directly: `echo "Question?" | perspt --simple-cli`
  - Batch processing with session logging for documentation and audit trails
  - Environment integration with aliases and shell scripts
  - Robust error handling that doesn't terminate sessions
- **Enhanced Accessibility**: Simple CLI mode provides better screen reader compatibility and simpler interaction model

Technical Details

- New `cli.rs` module implementing async command-line loop with streaming support
- Integration with existing `GenAIProvider` for consistent behavior across interface modes
- Comprehensive error handling with graceful degradation for individual request failures
- Session logging format compatible with standard text processing tools

Changed

- **Enhanced CLI Argument Support**: Added `--simple-cli` and `--log-file` arguments with proper validation
- **Updated Documentation**: Comprehensive updates to README and Perspt book covering simple CLI mode usage
- **Improved Help Text**: Clear descriptions of new simple CLI mode options and use cases

Examples and Use Cases

```
# Basic usage
perspt --simple-cli

# With session logging
perspt --simple-cli --log-file research-session.txt

# Scripting integration
echo "Explain quantum computing" | perspt --simple-cli

# Environment setup
alias ai="perspt --simple-cli"
ai-log() { perspt --simple-cli --log-file "$1"; }
```

3.8.2 [0.4.2] - 2025-06-09

Added

- 🤖 **Zero-Config Automatic Provider Detection**: Perspt now automatically detects and configures available providers based on environment variables
  - Set any supported API key (`OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, etc.) and simply run `perspt`
  - No configuration files or CLI arguments needed for basic usage
  - Intelligent priority-based selection: OpenAI → Anthropic → Gemini → Groq → Cohere → XAI → DeepSeek → Ollama
  - Automatic default model selection for each detected provider
  - Graceful fallback with helpful error messages when no providers are found

- **Enhanced Error Handling**: Clear, actionable error messages when no providers are configured
- **Comprehensive Provider Support**: All major LLM providers now supported for automatic detection
- **Local Model Auto-Detection**: Ollama automatically detected when running locally (no API key required)

Changed

- **Improved User Experience**: Launch Perspt instantly with just an API key - no config required
- **Better Documentation**: Updated getting-started guide and configuration documentation with zero-config examples
- **Streamlined Workflow**: Reduced friction for new users getting started

Technical Details

- Added `detect_available_provider()` function in `config.rs` for environment-based provider detection
- Enhanced `load_config()` to use automatic detection when no explicit configuration is provided
- Comprehensive test coverage for all provider detection scenarios and edge cases

3.8.3 [0.4.1] - 2025-06-03

Added

- Enhanced documentation with Sphinx
- Comprehensive API reference
- Developer guide for contributors

Changed

- Improved error messages for better user experience
- Optimized memory usage for large conversations

Fixed

- Fixed terminal cleanup on panic
- Resolved configuration file parsing edge cases

3.8.4 [0.4.0] - 2025-05-29

Added

- **Multi-provider support**: OpenAI, Anthropic, Google, Groq, Cohere, XAI, DeepSeek, and Ollama
- **Dynamic model discovery**: Automatic detection of available models
- **Input queuing**: Type new messages while AI is responding
- **Markdown rendering**: Rich text formatting in terminal
- **Streaming responses**: Real-time display of AI responses
- **Comprehensive configuration**: JSON files and environment variables
- **Beautiful terminal UI**: Powered by Ratatui with modern design
- **Graceful error handling**: User-friendly error messages and recovery

Technical Highlights

- Built with Rust for maximum performance and safety
- Leverages *genai* crate for unified LLM access
- Async/await architecture with Tokio
- Comprehensive test suite with unit and integration tests
- Memory-safe with zero-copy operations where possible

Supported Providers

- **OpenAI**: GPT-4, GPT-4-turbo, GPT-4o series, GPT-3.5-turbo
- **Anthropic**: Claude 3 models (via genai)
- **Google**: Gemini models (via genai)
- **Groq**: Ultra-fast Llama inference
- **Cohere**: Command R/R+ models
- **XAI**: Grok models
- **DeepSeek**: Advanced reasoning models
- **Ollama**: Local model hosting

Configuration Features

- Multiple configuration file locations
- Environment variable support
- Command-line argument overrides
- Provider-specific settings
- UI customization options

User Interface Features

- Real-time chat interface
- Syntax highlighting for code blocks
- Scrollable message history
- Keyboard shortcuts for productivity
- Status indicators and progress feedback
- Responsive design that adapts to terminal size

3.8.5 [0.3.0] - 2025-05-15

Added

- Multi-provider foundation with genai crate
- Configuration file validation
- Improved error categorization

Changed

- Refactored provider architecture for extensibility
- Enhanced UI responsiveness
- Better handling of long responses

Fixed

- Terminal state cleanup on unexpected exit
- Configuration merging precedence
- Memory leaks in streaming responses

3.8.6 [0.2.0] - 2025-05-01

Added

- Streaming response support
- Basic configuration file support
- Terminal UI with Ratatui
- OpenAI provider implementation

Changed

- Migrated from simple CLI to TUI interface
- Improved async architecture
- Better error handling patterns

Fixed

- Terminal rendering issues
- API request timeout handling
- Configuration loading edge cases

3.8.7 [0.1.0] - 2025-04-15

Added

- Initial release
- Basic OpenAI integration
- Simple command-line interface
- Environment variable configuration
- Basic chat functionality

Features

- Support for GPT-3.5 and GPT-4 models
- API key authentication
- Simple text-based conversations
- Basic error handling

3.8.8 Migration Guides

Upgrading from 0.3.x to 0.4.0

Configuration Changes:

The configuration format has been enhanced. Old configurations will continue to work, but consider updating:

```
// Old format (still supported)
{
  "api_key": "sk-...",
  "model": "gpt-4"
}

// New format (recommended)
{
  "api_key": "sk-...",
  "default_model": "gpt-4o-mini",
  "provider_type": "openai",
  "providers": {
    "openai": "https://api.openai.com/v1"
  }
}
```

Command Line Changes:

Some command-line flags have been updated:

```
# Old
perspt --model gpt-4

# New
perspt --model-name gpt-4
```

API Changes:

If you're using Perspt as a library, some function signatures have changed:

```
// Old
provider.send_request(message, model).await?;

// New
provider.send_chat_request(message, model, &config, &tx).await?;
```

Upgrading from 0.2.x to 0.3.0

New Dependencies:

Update your *Cargo.toml* if building from source:

```
[dependencies]
tokio = { version = "1.0", features = ["full"] }
# ... other dependencies updated
```

Configuration Location:

Configuration files now support multiple locations. Move your config file to:

- *~/.config/perspt/config.json* (Linux)
- *~/Library/Application Support/perspt/config.json* (macOS)
- *%APPDATA%/perspt/config.json* (Windows)

3.8.9 Breaking Changes

Version 0.4.0

- **Provider trait changes**: *LLMProvider* trait now requires *async fn* methods
- **Configuration structure**: Some configuration keys renamed for consistency
- **Error types**: Custom error types replace generic error handling
- **Streaming interface**: Response handling now uses channels instead of callbacks

Version 0.3.0

- **Async runtime**: Switched to full async architecture
- **UI framework**: Migrated from custom rendering to Ratatui
- **Configuration format**: Enhanced JSON schema with validation

Version 0.2.0

- **Interface change**: Moved from CLI to TUI
- **Provider abstraction**: Introduced provider trait system
- **Async support**: Added Tokio async runtime

3.8.10 Deprecation Notices

The following features are deprecated and will be removed in future versions:

Version 0.5.0 (Upcoming)

- **Legacy configuration keys**: Old configuration format support will be removed
- **Synchronous API**: All provider methods must be async
- **Direct model specification**: Use provider + model pattern instead

Version 0.6.0 (Planned)

- **Environment variable precedence**: Will change to match command-line precedence
- **Default provider**: Will change from OpenAI to provider-agnostic selection

3.8.11 Known Issues

Current Version (0.4.0)

- **Windows terminal compatibility**: Some Unicode characters may not display correctly on older Windows terminals
- **Large conversation history**: Memory usage increases with very long conversations (>1000 messages)
- **Network interruption**: Streaming responses may be interrupted during network issues
- **Ollama connectivity**: Local models may require manual service restart after system reboot

Workarounds:

```
# For Windows terminal issues
# Use Windows Terminal or enable UTF-8 support

# For memory issues with large histories
perspt --max-history 500

# For network issues
perspt --timeout 60 --max-retries 5
```

3.8.12 Planned Features

Version 0.5.0 (Next Release)

- **Local model support**: Integration with Ollama and other local LLM servers
- **Plugin system**: Support for custom providers and UI extensions
- **Conversation persistence**: Save and restore chat sessions
- **Multi-conversation support**: Multiple chat tabs in single session
- **Enhanced markdown**: Tables, math equations, and diagrams
- **Voice input**: Speech-to-text support for hands-free operation

Version 0.6.0 (Future)

- **Collaborative features**: Share conversations and collaborate with others
- **IDE integration**: VS Code extension and other editor plugins
- **Mobile companion**: Mobile app for conversation sync
- **Advanced AI features**: Function calling, tool use, and agent capabilities
- **Performance analytics**: Response time tracking and optimization suggestions

Version 1.0.0 (Stable Release)

- **API stability guarantee**: Stable public API with semantic versioning
- **Enterprise features**: SSO, audit logging, and compliance features
- **Advanced customization**: Themes, layouts, and workflow customization
- **Comprehensive integrations**: GitHub, Slack, Discord, and more
- **Professional support**: Documentation, training, and enterprise support

3.8.13 Contributing

We welcome contributions! Please see our *Contributing* for guidelines.

Types of contributions: - Bug reports and feature requests - Code contributions and optimizations - Documentation improvements - Testing and quality assurance - Community support and advocacy

How to contribute:

1. Check existing issues and discussions
2. Fork the repository
3. Create a feature branch
4. Make your changes with tests
5. Submit a pull request

3.8.14 Support

- **GitHub Issues**: Bug Reports
- **Discussions**: Community Chat
- **Documentation**: This guide and API reference
- **Email**: support@perspt.dev (for enterprise inquiries)

3.8.15 License

Perspt is released under the LGPL v3 License. See *License* for details.

3.8.16 Acknowledgments

Special thanks to:

- The Rust community for excellent tooling and libraries
- Ratatui developers for the amazing TUI framework
- genai crate maintainers for unified LLM access
- All contributors and users who help improve Perspt

See also

- *Installation Guide* - How to install or upgrade Perspt
- *Getting Started* - Quick start guide for new users
- *Contributing* - How to contribute to the project

3.9 License

Perspt is released under the GNU Lesser General Public License v3.0 (LGPL v3).

3.9.1 LGPL v3 License

Copyright (c) 2025 Ronak Rathoer, Vikrant Rathore

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

3.9.2 What This Means

The LGPL v3 is a copyleft license that provides strong protection for software freedom while allowing linking with proprietary software. Here's what it means in practical terms:

☐ **What you CAN do:**

- **Use** Perspt for any purpose, including commercial projects
- **Modify** the source code to fit your needs
- **Distribute** copies of Perspt
- **Link** Perspt as a library in proprietary software
- **Combine** Perspt with software under different licenses
- **Create** derivative works based on Perspt

☐ **What you MUST do:**

- **Provide source code** for any modifications to Perspt itself
- **Include** the LGPL v3 license text with distributions
- **Preserve** copyright notices and license information
- **Allow users** to replace the Perspt library with modified versions
- **Make modified source** available under LGPL v3 terms

🚫 **What we DON'T provide:**

- **Warranty** - The software is provided "as is"
- **Liability coverage** - We're not responsible for any damages
- **Support guarantees** - While we strive to help, support is provided on a best-effort basis

3.9.3 Third-Party Licenses

Perspt depends on several open source libraries, each with their own licenses:

Core Dependencies

Crate	License	Description
tokio	MIT	Async runtime for Rust
ratatui	MIT	Terminal user interface library
serde	MIT/Apache-2.0	Serialization framework
clap	MIT/Apache-2.0	Command line argument parser
anyhow	MIT/Apache-2.0	Error handling library
thiserror	MIT/Apache-2.0	Error derive macros

LLM Integration

Crate	License	Description
genai	MIT/Apache-2.0	Unified LLM provider interface
reqwest	MIT/Apache-2.0	HTTP client library

Terminal and UI

Crate	License	Description
crossterm	MIT	Cross-platform terminal library
unicode-width	MIT/Apache-2.0	Unicode character width calculation
textwrap	MIT	Text wrapping and formatting

Development Dependencies

Crate	License	Description
criterion	MIT/Apache-2.0	Benchmarking library
mockall	MIT/Apache-2.0	Mock object library
tempfile	MIT/Apache-2.0	Temporary file management

3.9.4 License Compatibility

The LGPL v3 is compatible with most other open source licenses:

Compatible Licenses: - Apache License 2.0 - BSD licenses (2-clause, 3-clause) - ISC License - MIT License - Public Domain (CC0) - GPL v3+ (can be upgraded to GPL)

Special Considerations: - GPL v2: Not directly compatible due to version differences - Proprietary licenses: Can link with LGPL libraries but must allow library replacement - Copyleft licenses: LGPL provides weaker copyleft than GPL

3.9.5 Commercial Use

Perspt can be freely used in commercial projects:

☐ **Allowed Commercial Uses:**

- **Internal tools** - Use Perspt as part of your development workflow
- **Linked libraries** - Link Perspt as a library in commercial software
- **Service offerings** - Provide Perspt as part of consulting or hosting services
- **Modified library versions** - Create modified versions for internal use
- **Enterprise solutions** - Build enterprise tools that use Perspt

📋 **Requirements for Commercial Use:**

1. **Include LGPL license text** in your distribution
2. **Maintain copyright notices** from the original code
3. **Provide source code** for any modifications to Perspt itself
4. **Allow library replacement** - users must be able to replace the Perspt library
5. **No trademark usage** without permission (see below)

No additional fees, registrations, or permissions are required.

3.9.6 Trademark Policy

While the source code is LGPL v3 licensed, trademarks are handled separately:

"Perspt" Name and Logo: - The name "Perspt" and any associated logos are trademarks - You may use the name in accurately describing the software - Commercial use of the name/logo as your own brand requires permission - Modified versions should use different names to avoid confusion

Acceptable Uses: - "Built with Perspt" - "Based on Perspt" - "Powered by Perspt" - "Fork of Perspt"

Requires Permission: - Using "Perspt" as your product name - Using Perspt logos in your branding - Implying official endorsement

3.9.7 Contributing and License

By contributing to Perspt, you agree that:

1. **Your contributions** will be licensed under the same LGPL v3 License
2. **You have the right** to license your contributions under LGPL v3
3. **You understand** that your contributions may be used commercially
4. **You retain copyright** to your contributions while granting broad usage rights

Contributor License Agreement (CLA)

For substantial contributions, we may request a Contributor License Agreement to:

- Ensure you have the right to contribute the code
- Provide legal protection for the project and users
- Allow for potential future license changes if needed
- Clarify the rights and responsibilities of contributors

3.9.8 License FAQ

Q: Can I use Perspt in my proprietary software? A: Yes, LGPL v3 allows linking with proprietary software. You must provide the library source and allow replacement.

Q: Can I modify Perspt and sell the modified version? A: Yes, but you must provide the source code for your modifications under LGPL v3.

Q: Do I need to open source my modifications? A: Yes, any modifications to Perspt itself must be made available under LGPL v3.

Q: Can I remove the copyright notices? A: No, you must preserve the copyright notices and license information in all copies.

Q: What if I only use parts of the code? A: The LGPL v3 license still applies to any substantial portions you use.

Q: Can I change the license of my derivative work? A: You can license your own code separately, but Perspt parts must remain LGPL v3.

Q: Do I need to attribute Perspt in my application? A: Yes, you must include the LGPL v3 license and copyright notices.

3.9.9 Getting Legal Advice

This page provides general information about the LGPL v3 License and is not legal advice. For specific legal questions:

- **Consult** with a qualified attorney
- **Review** the full license text carefully
- **Consider** your specific use case and jurisdiction
- **Seek** professional legal counsel for commercial decisions

3.9.10 Reporting License Issues

If you believe there's a license violation or have questions about licensing:

- **Email**: legal@perspt.dev
- **GitHub Issues**: License Questions
- **Include** specific details about the concern or question

We take licensing seriously and will investigate all reports promptly.

See also

- *Acknowledgments* - Credits and thanks to contributors
- *Contributing* - How to contribute to the project
- GNU Project - Official LGPL v3 License text

3.10 Acknowledgments

Perspt is built on the shoulders of giants. We extend our gratitude to the many open-source projects, libraries, and communities that made this project possible.

3.10.1 Core Dependencies

AI and LLM Integration

genai

The foundation of Perspt's multi-provider support. This exceptional crate provides unified interfaces to multiple AI providers and automatically stays up-to-date with new models and capabilities.

- **Project**: genai
- **License**: MIT/Apache 2.0
- **Impact**: Enables seamless integration with OpenAI, Anthropic, Google, Groq, Cohere, XAI, DeepSeek, and Ollama providers

serde & serde_json

Rust's premier serialization framework, powering Perspt's configuration management and API communication.

- **Project**: serde
- **License**: MIT/Apache 2.0
- **Impact**: JSON configuration parsing, API request/response handling

User Interface and Terminal

ratatui

The modern, feature-rich TUI framework that powers Perspt's interactive terminal interface.

- **Project**: ratatui
- **License**: MIT
- **Impact**: Rich terminal UI, markdown rendering, scrollable chat interface

crossterm

Cross-platform terminal manipulation library enabling consistent behavior across operating systems.

- **Project**: crossterm
- **License**: MIT
- **Impact**: Keyboard input handling, terminal control, cross-platform compatibility

Async Runtime and Concurrency

tokio

The asynchronous runtime that enables Perspt's responsive, non-blocking architecture.

- **Project**: tokio
- **License**: MIT
- **Impact**: Async/await support, concurrent LLM requests, responsive UI

Error Handling and Utilities

anyhow

Elegant error handling that makes Perspt's error messages helpful and actionable.

- **Project**: anyhow
- **License**: MIT/Apache 2.0
- **Impact**: Comprehensive error context, user-friendly error messages

clap

Command-line argument parsing that makes Perspt easy to use and configure.

- **Project**: clap
- **License**: MIT/Apache 2.0
- **Impact**: CLI interface, help generation, argument validation

3.10.2 Documentation Tools

Sphinx

The documentation generator that created this beautiful book-style documentation.

- **Project**: Sphinx
- **License**: BSD
- **Impact**: Professional documentation, PDF generation, cross-references

Furo Theme

The modern, accessible Sphinx theme that makes this documentation a pleasure to read.

- **Project**: Furo
- **License**: MIT
- **Impact**: Beautiful documentation design, responsive layout, accessibility

3.10.3 Development Tools

Rust Language

The systems programming language that makes Perspt fast, safe, and reliable.

- **Project**: Rust
- **License**: MIT/Apache 2.0
- **Impact**: Memory safety, performance, excellent tooling ecosystem

cargo

Rust's package manager and build system that makes development smooth and dependency management effortless.

- **Project**: Part of Rust toolchain
- **License**: MIT/Apache 2.0
- **Impact**: Dependency management, build automation, testing framework

3.10.4 Community and Inspiration

AI Provider Communities

OpenAI

For creating GPT models and establishing many of the patterns that define modern AI interaction.

Anthropic

For Claude models and their pioneering work in AI safety and helpful, harmless, and honest AI systems.

Google
For Gemini models and their contributions to accessible AI technology.
Groq
For ultra-fast inference infrastructure and democratizing AI speed.
Cohere
For enterprise-grade language models and excellent developer tools.
XAI
For Grok models and advancing conversational AI capabilities.
DeepSeek
For their contributions to the open-source AI ecosystem.
Ollama
For making local AI model hosting accessible and user-friendly.

Open Source Ecosystem

GitHub
For providing the platform that enables collaborative development and open-source sharing.
crates.io
Rust's package registry that makes sharing and discovering Rust libraries effortless.
docs.rs
For automatically generating and hosting documentation for Rust crates.

Terminal and CLI Inspiration

The terminal and CLI interface draws inspiration from many excellent tools:

- **htop** - For showing how terminal UIs can be both beautiful and functional
- **tmux** - For terminal multiplexing concepts and keyboard navigation patterns
- **vim/neovim** - For modal editing concepts and efficient keyboard shortcuts
- **fzf** - For demonstrating responsive, interactive terminal interfaces

Rust Community Projects

Many patterns and approaches in Perspt were learned from studying excellent Rust projects:

- **ripgrep** - For performance optimization and user experience design
- **bat** - For beautiful terminal output and syntax highlighting
- **exa/eza** - For modern CLI design and colored output
- **gitui** - For TUI application architecture and event handling

3.10.5 Testing and Quality Assurance

Users and Beta Testers
The early adopters and users who provided feedback, reported bugs, and suggested improvements.
Security Researchers
For responsible disclosure of security issues and helping make Perspt more secure.
Documentation Reviewers
For helping improve the clarity and completeness of this documentation.

3.10.6 Special Thanks

AI Safety Research Community
For ongoing work to make AI systems more reliable, interpretable, and aligned with human values.
Open Source Contributors
To everyone who contributes to open-source projects, from major features to documentation fixes.

Rust Community
For creating and maintaining an inclusive, helpful community that makes Rust development a joy.

Terminal Enthusiasts
For keeping the art of terminal-based applications alive and pushing the boundaries of what's possible in text-based interfaces.

3.10.7 Contributing Back

Perspt aims to be a good citizen of the open-source ecosystem. We contribute back by:

Open Source Release
Perspt itself is released under the LGPL v3 license, allowing anyone to use, modify, and distribute it.

Documentation Standards
This comprehensive documentation serves as an example of thorough project documentation.

Best Practices Sharing
Through blog posts, talks, and code examples, we share what we've learned building Perspt.

Upstream Contributions
When we find bugs or missing features in dependencies, we contribute fixes and improvements back to those projects.

3.10.8 License Information

Perspt is licensed under the LGPL v3 License. For complete license information, see *License*.

All dependencies are used in accordance with their respective licenses. We are grateful to all the authors and maintainers who choose to share their work under permissive open-source licenses.

3.10.9 Get Involved

Want to contribute to Perspt or the broader ecosystem?

Report Issues
Help improve Perspt by reporting bugs, suggesting features, or improving documentation.

Contribute Code
See our *Contributing* guide for how to contribute code improvements.

Share Knowledge
Write blog posts, create tutorials, or give talks about your experience with Perspt.

Support Dependencies
Consider contributing to the open-source projects that Perspt depends on.

Spread the Word
Help others discover Perspt and the amazing ecosystem of Rust and AI tools.

—

Thank you to everyone who makes open-source software development possible. Your contributions, large and small, make projects like Perspt possible.

Download as PDF

Chapter 4

🔗 Quick Links

- **Repository:** GitHub
- **Crates.io:** perspt
- **Issues:** Bug Reports & Feature Requests
- **Discussions:** Community Chat

Chapter 5

Indices and Tables

- genindex
- modindex
- search