Table of Contents

# 1 Smart Commits AI - Comprehensive Documentation

## 1.1 Table of Contents

1. [Introduction](#introduction)
2. [Architecture Overview](#architecture-overview)
3. [Installation Guide](#installation-guide)
4. [Configuration Management](#configuration-management)
5. [Core Functionality](#core-functionality)
6. [API Clients](#api-clients)
7. [Git Integration](#git-integration)
8. [Command Line Interface](#command-line-interface)
9. [Security Features](#security-features)
10. [Advanced Usage](#advanced-usage)
11. [Troubleshooting](#troubleshooting)
12. [API Reference](#api-reference)
13. [Development Guide](#development-guide)
14. [Security Analysis](#security-analysis)
15. [Performance Optimization](#performance-optimization)

## 1.2 1. Introduction

### 1.2.1 1.1 Overview

Smart Commits AI is an enterprise-grade, AI-powered Git commit message generator that automatically creates conventional commit messages using advanced language models. The tool analyzes staged changes in your Git repository and generates professional, standardized commit messages that follow conventional commit specifications.

### 1.2.2 1.2 Key Features

* **🤖 AI-Powered**: Uses state-of-the-art language models (Groq, OpenRouter, Cohere)
* **🔒 Enterprise Security**: Production-ready with comprehensive security features
* **🌍 Universal Language Support**: Works with any programming language
* **⚡ Fast & Efficient**: Optimized for speed and reliability
* **🛡️ Secure by Design**: All security vulnerabilities addressed
* **📋 Conventional Commits**: Follows industry-standard commit conventions
* **🔧 Highly Configurable**: Extensive customization options
* **👥 Team-Ready**: Zero-friction adoption for development teams

### 1.2.3 1.3 Version Information

* **Current Version**: 1.1.0 (Major Security Release)
* **Security Score**: 8.5/10 (Excellent)
* **Production Status**: Enterprise-ready
* **License**: MIT
* **Python Support**: 3.8+

### 1.2.4 1.4 System Requirements

* **Operating Systems**: macOS, Linux, Windows
* **Python**: 3.8 or higher
* **Git**: 2.0 or higher
* **Internet Connection**: Required for AI API calls
* **Disk Space**: ~50MB for installation

## 1.3 2. Architecture Overview

### 1.3.1 2.1 System Architecture

Smart Commits AI follows a modular architecture with clear separation of concerns:

┌─────────────────────────────────────────────────────────────┐  
│ Smart Commits AI │  
├─────────────────────────────────────────────────────────────┤  
│ CLI Interface (cli.py) │  
│ ├── Command parsing and user interaction │  
│ ├── Error handling and user feedback │  
│ └── Security-aware logging and output │  
├─────────────────────────────────────────────────────────────┤  
│ Core Engine (core.py) │  
│ ├── Commit message generation logic │  
│ ├── Git repository analysis │  
│ ├── Input validation and sanitization │  
│ └── Security controls and validation │  
├─────────────────────────────────────────────────────────────┤  
│ Configuration Manager (config.py) │  
│ ├── YAML configuration loading │  
│ ├── Environment variable management │  
│ ├── Security validation and path checking │  
│ └── Default configuration management │  
├─────────────────────────────────────────────────────────────┤  
│ API Clients (api\_clients.py) │  
│ ├── Groq API integration │  
│ ├── OpenRouter API integration │  
│ ├── Cohere API integration │  
│ └── Secure HTTP communication │  
├─────────────────────────────────────────────────────────────┤  
│ Git Integration (git\_hook.py) │  
│ ├── Git hook management │  
│ ├── Repository validation │  
│ ├── Secure file operations │  
│ └── Hook installation and removal │  
└─────────────────────────────────────────────────────────────┘

### 1.3.2 2.2 Data Flow

1. **User Interaction**: User runs git commit or CLI commands
2. **Git Hook Trigger**: prepare-commit-msg hook activates
3. **Repository Analysis**: Core engine analyzes staged changes
4. **Security Validation**: Input sanitization and validation
5. **AI Processing**: API client sends request to chosen provider
6. **Response Processing**: AI response validation and cleaning
7. **Commit Generation**: Final commit message creation
8. **Security Logging**: Secure logging of operations

### 1.3.3 2.3 Security Architecture

Smart Commits AI implements defense-in-depth security:

* **Input Validation**: All inputs sanitized and validated
* **Path Security**: Directory traversal prevention
* **API Security**: Secure HTTP with SSL verification
* **Error Handling**: Information disclosure prevention
* **File Permissions**: Secure file and directory permissions
* **Configuration Security**: Secure YAML loading and validation
* **Logging Security**: Sensitive data masking in logs

## 1.4 3. Installation Guide

### 1.4.1 3.1 Universal Installation

The easiest way to install Smart Commits AI:

curl -fsSL https://raw.githubusercontent.com/Joshi-e8/ai-commit-generator/master/install.sh | bash

This script automatically: - Detects your operating system - Checks Python and pip installation - Installs Smart Commits AI v1.1.0 - Sets up Git hooks - Configures the environment

### 1.4.2 3.2 Manual Installation Methods

#### 1.4.2.1 3.2.1 Using pip

pip install smart-commits-ai==1.1.0  
smart-commits-ai install

#### 1.4.2.2 3.2.2 Using pipx (Recommended for applications)

brew install pipx # macOS  
pipx install smart-commits-ai==1.1.0

#### 1.4.2.3 3.2.3 Using conda

conda install -c conda-forge smart-commits-ai

#### 1.4.2.4 3.2.4 From source

git clone https://github.com/Joshi-e8/ai-commit-generator.git  
cd ai-commit-generator  
pip install -e .

### 1.4.3 3.3 Platform-Specific Instructions

#### 1.4.3.1 3.3.1 macOS

# Using Homebrew (recommended)  
brew install python  
pip3 install smart-commits-ai==1.1.0  
  
# Using MacPorts  
sudo port install python39  
pip3 install smart-commits-ai==1.1.0

#### 1.4.3.2 3.3.2 Linux (Ubuntu/Debian)

sudo apt update  
sudo apt install python3 python3-pip git  
pip3 install smart-commits-ai==1.1.0

#### 1.4.3.3 3.3.3 Linux (CentOS/RHEL)

sudo yum install python3 python3-pip git  
pip3 install smart-commits-ai==1.1.0

#### 1.4.3.4 3.3.4 Windows

# Using Git Bash or WSL  
pip install smart-commits-ai==1.1.0  
  
# Using PowerShell  
python -m pip install smart-commits-ai==1.1.0

### 1.4.4 3.4 Verification

After installation, verify everything works:

# Check version  
smart-commits-ai --version  
  
# Check status  
smart-commits-ai status  
  
# Test configuration  
smart-commits-ai config --validate

## 1.5 4. Configuration Management

### 1.5.1 4.1 Configuration System Overview

Smart Commits AI uses a hierarchical configuration system:

1. **Default Configuration**: Built-in secure defaults
2. **Global Configuration**: System-wide settings
3. **Project Configuration**: Repository-specific settings
4. **Environment Variables**: Runtime overrides
5. **Command Line Arguments**: Immediate overrides

### 1.5.2 4.2 Configuration File Structure

The main configuration file is .commitgen.yml:

# Smart Commits AI Configuration  
api:  
 provider: "groq" # groq, openrouter, cohere  
 models:  
 groq:  
 default: "llama3-70b-8192"  
 alternatives: ["llama3-8b-8192", "mixtral-8x7b-32768"]  
 openrouter:  
 default: "meta-llama/llama-3.1-70b-instruct"  
 alternatives: ["anthropic/claude-3.5-sonnet", "google/gemini-pro-1.5"]  
 cohere:  
 default: "command-r-plus"  
 alternatives: ["command-r", "command-light"]  
  
commit:  
 max\_chars: 72  
 types: ["feat", "fix", "docs", "style", "refactor", "perf", "test", "build", "ci", "chore", "revert"]  
 scopes: ["api", "auth", "ui", "db", "config", "deps", "security", "performance", "i18n", "tests"]  
  
processing:  
 max\_diff\_size: 4000  
 exclude\_patterns:  
 - "\*.key"  
 - "\*.pem"  
 - "\*.p12"  
 - "\*.env\*"  
 - "secrets/\*"  
 - "\*.lock"  
 - "\*.log"  
 - "node\_modules/\*"  
 - ".git/\*"  
 - "dist/\*"  
 - "build/\*"  
 - "\*.min.js"  
 - "\*.min.css"  
 truncate\_files: true  
 max\_file\_lines: 100  
  
security:  
 validate\_inputs: true  
 sanitize\_logs: true  
 max\_log\_size: 10485760 # 10MB  
 timeout: 30  
 verify\_ssl: true  
  
fallback:  
 default\_message: "chore: update files"  
 max\_retries: 3  
 retry\_delay: 1  
  
debug:  
 enabled: false  
 log\_file: ".commitgen.log"  
 save\_requests: false

### 1.5.3 4.3 Environment Variables

Smart Commits AI supports the following environment variables:

#### 1.5.3.1 4.3.1 API Keys

# Groq API Key (free tier available)  
export GROQ\_API\_KEY="gsk\_..."  
  
# OpenRouter API Key  
export OPENROUTER\_API\_KEY="sk-or-..."  
  
# Cohere API Key  
export COHERE\_API\_KEY="..."

#### 1.5.3.2 4.3.2 Configuration Overrides

# Override default provider  
export COMMITGEN\_PROVIDER="openrouter"  
  
# Override model  
export COMMITGEN\_MODEL="anthropic/claude-3.5-sonnet"  
  
# Enable debug mode  
export COMMITGEN\_DEBUG="true"  
  
# Set custom config file  
export COMMITGEN\_CONFIG="/path/to/config.yml"

### 1.5.4 4.4 Configuration Commands

#### 1.5.4.1 4.4.1 View Current Configuration

# Show current configuration  
smart-commits-ai config --show  
  
# Show configuration with validation  
smart-commits-ai config --show --validate  
  
# Show only API configuration  
smart-commits-ai config --show --section api

#### 1.5.4.2 4.4.2 Modify Configuration

# Set provider  
smart-commits-ai config set api.provider groq  
  
# Set model  
smart-commits-ai config set api.models.groq.default llama3-8b-8192  
  
# Set commit message length  
smart-commits-ai config set commit.max\_chars 50  
  
# Add custom commit type  
smart-commits-ai config add commit.types security  
  
# Remove commit type  
smart-commits-ai config remove commit.types security

#### 1.5.4.3 4.4.3 Configuration Validation

# Validate configuration  
smart-commits-ai config --validate  
  
# Validate specific section  
smart-commits-ai config --validate --section security  
  
# Fix common configuration issues  
smart-commits-ai config --fix

## 1.6 5. Core Functionality

### 1.6.1 5.1 Commit Message Generation Engine

The core engine (core.py) is responsible for analyzing Git repositories and generating commit messages.

#### 1.6.1.1 5.1.1 CommitGenerator Class

class CommitGenerator:  
 """Main class for generating AI-powered commit messages."""  
  
 def \_\_init\_\_(self, config: Config):  
 """Initialize with configuration and security validation."""  
 self.config = config  
 self.api\_client = self.\_create\_api\_client()  
 self.\_validate\_security()  
  
 def generate\_commit\_message(self, repo\_path: Optional[str] = None) -> str:  
 """Generate a commit message for staged changes."""  
 # Security validation  
 repo\_path = self.\_validate\_repo\_path(repo\_path)  
  
 # Analyze repository  
 diff\_content = self.\_get\_staged\_diff(repo\_path)  
  
 # Security: Validate and sanitize diff content  
 diff\_content = self.\_sanitize\_diff\_content(diff\_content)  
  
 # Generate message using AI  
 message = self.\_generate\_with\_ai(diff\_content)  
  
 # Security: Clean and validate generated message  
 return self.\_clean\_message(message)

#### 1.6.1.2 5.1.2 Repository Analysis

The engine analyzes Git repositories through several steps:

1. **Repository Validation**: Ensures we’re in a valid Git repository
2. **Staged Changes Detection**: Identifies files with staged changes
3. **Diff Generation**: Creates unified diff of staged changes
4. **Content Filtering**: Removes sensitive files and large binaries
5. **Size Optimization**: Truncates large diffs for API efficiency

def \_get\_staged\_diff(self, repo\_path: Path) -> str:  
 """Get staged changes with security validation."""  
 try:  
 # Security: Use secure subprocess execution  
 result = secure\_subprocess\_run(  
 ["git", "diff", "--cached", "--no-color"],  
 cwd=repo\_path,  
 timeout=30,  
 capture\_output=True,  
 text=True  
 )  
  
 diff\_content = result.stdout  
  
 # Security: Validate diff size  
 if len(diff\_content) > self.config.max\_diff\_size:  
 diff\_content = self.\_truncate\_diff(diff\_content)  
  
 return diff\_content  
  
 except Exception as e:  
 raise GitError(f"Failed to get staged diff: {e}")

#### 1.6.1.3 5.1.3 AI Integration

The engine integrates with multiple AI providers:

def \_generate\_with\_ai(self, diff\_content: str) -> str:  
 """Generate commit message using AI with fallback handling."""  
  
 # Prepare prompt with security considerations  
 prompt = self.\_create\_prompt(diff\_content)  
  
 # Try primary model  
 try:  
 response = self.api\_client.generate\_commit\_message(prompt)  
 return self.\_validate\_ai\_response(response)  
  
 except APIError as e:  
 logger.warning(f"Primary model failed: {e}")  
  
 # Try alternative models  
 for alt\_model in self.config.alternative\_models:  
 try:  
 self.api\_client.model = alt\_model  
 response = self.api\_client.generate\_commit\_message(prompt)  
 return self.\_validate\_ai\_response(response)  
 except APIError:  
 continue  
  
 # Fallback to default message  
 return self.config.fallback\_message

### 1.6.2 5.2 Security Features in Core Engine

#### 1.6.2.1 5.2.1 Input Validation and Sanitization

def \_sanitize\_diff\_content(self, content: str) -> str:  
 """Sanitize diff content for security."""  
 if not isinstance(content, str):  
 raise SecurityError("Invalid diff content type")  
  
 # Remove potential security risks  
 content = re.sub(r'(password|token|key|secret)\s\*[=:]\s\*\S+',  
 r'\1=\*\*\*REDACTED\*\*\*', content, flags=re.IGNORECASE)  
  
 # Validate content length  
 if len(content) > self.config.security.max\_diff\_size:  
 content = content[:self.config.security.max\_diff\_size]  
  
 # Check for suspicious patterns  
 suspicious\_patterns = [  
 r'<script[^>]\*>.\*?</script>', # XSS  
 r'javascript:', # JavaScript injection  
 r'data:.\*base64', # Data URLs  
 r'eval\s\*\(', # Code evaluation  
 ]  
  
 for pattern in suspicious\_patterns:  
 if re.search(pattern, content, re.IGNORECASE | re.DOTALL):  
 logger.warning("Suspicious content detected and sanitized")  
 content = re.sub(pattern, '[SANITIZED]', content, flags=re.IGNORECASE | re.DOTALL)  
  
 return content

#### 1.6.2.2 5.2.2 Repository Path Validation

def \_validate\_repo\_path(self, repo\_path: Optional[str]) -> Path:  
 """Validate and sanitize repository path."""  
 if repo\_path is None:  
 repo\_path = Path.cwd()  
 else:  
 repo\_path = Path(repo\_path)  
  
 # Security: Validate path  
 try:  
 repo\_path = validate\_file\_path(Path.cwd(), repo\_path)  
 except SecurityError as e:  
 raise SecurityError(f"Invalid repository path: {e}")  
  
 # Ensure it's a Git repository  
 if not (repo\_path / ".git").exists():  
 raise GitError(f"Not a Git repository: {repo\_path}")  
  
 return repo\_path

#### 1.6.2.3 5.2.3 Message Cleaning and Validation

def \_clean\_message(self, message: str) -> str:  
 """Clean and validate generated commit message."""  
 if not isinstance(message, str):  
 raise SecurityError("Invalid message type")  
  
 # Remove common AI response artifacts  
 message = re.sub(r'^(Here\'s|Here is|The commit message is):?\s\*', '', message, flags=re.IGNORECASE)  
 message = re.sub(r'^```.\*?\n', '', message) # Remove code block markers  
 message = re.sub(r'\n```.\*?$', '', message)  
  
 # Security: Check for injection attempts  
 dangerous\_patterns = [  
 r'[<>"\']', # HTML/XML injection  
 r'javascript:', # JavaScript injection  
 r'data:', # Data URLs  
 r'[;&|`$()]', # Command injection  
 ]  
  
 for pattern in dangerous\_patterns:  
 if re.search(pattern, message):  
 logger.warning("Potentially dangerous content in AI response")  
 message = re.sub(pattern, '', message)  
  
 # Validate message format  
 message = message.strip()  
 if len(message) > self.config.max\_chars:  
 message = message[:self.config.max\_chars].rsplit(' ', 1)[0]  
  
 # Ensure conventional commit format  
 if not self.\_is\_conventional\_commit(message):  
 message = self.\_make\_conventional(message)  
  
 return message

## 1.7 6. API Clients

### 1.7.1 6.1 API Client Architecture

Smart Commits AI supports multiple AI providers through a unified client interface:

class APIClient(ABC):  
 """Abstract base class for AI API clients."""  
  
 def \_\_init\_\_(self, api\_key: str, model: str):  
 self.api\_key = api\_key  
 self.model = model  
 self.session = self.\_create\_secure\_session()  
  
 def \_create\_secure\_session(self) -> requests.Session:  
 """Create a secure HTTP session."""  
 session = requests.Session()  
 session.verify = True # SSL verification  
 session.timeout = 30 # Request timeout  
  
 # Security headers  
 session.headers.update({  
 'User-Agent': f'smart-commits-ai/{\_\_version\_\_}',  
 'Accept': 'application/json',  
 'Content-Type': 'application/json'  
 })  
  
 return session  
  
 @abstractmethod  
 def generate\_commit\_message(self, diff\_content: str) -> str:  
 """Generate commit message from diff content."""  
 pass

### 1.7.2 6.2 Groq API Client

class GroqClient(APIClient):  
 """Groq API client for fast inference."""  
  
 def \_\_init\_\_(self, api\_key: str, model: str = "llama3-70b-8192"):  
 super().\_\_init\_\_(api\_key, model)  
 self.base\_url = "https://api.groq.com/openai/v1"  
  
 def generate\_commit\_message(self, diff\_content: str) -> str:  
 """Generate commit message using Groq API."""  
  
 # Security: Validate input  
 if not isinstance(diff\_content, str):  
 raise APIError("Invalid diff content type")  
  
 if len(diff\_content) > 8000: # Groq context limit  
 diff\_content = diff\_content[:8000]  
  
 # Prepare request  
 headers = {  
 "Authorization": f"Bearer {self.api\_key}",  
 "Content-Type": "application/json"  
 }  
  
 data = {  
 "model": self.model,  
 "messages": [  
 {  
 "role": "system",  
 "content": self.\_get\_system\_prompt()  
 },  
 {  
 "role": "user",  
 "content": f"Generate a conventional commit message for:\n\n{diff\_content}"  
 }  
 ],  
 "max\_tokens": 100,  
 "temperature": 0.3  
 }  
  
 # Make secure API request  
 try:  
 response = self.\_make\_request(  
 f"{self.base\_url}/chat/completions",  
 headers,  
 data  
 )  
  
 return response["choices"][0]["message"]["content"].strip()  
  
 except Exception as e:  
 raise APIError(f"Groq API error: {e}")

### 1.7.3 6.3 OpenRouter API Client

class OpenRouterClient(APIClient):  
 """OpenRouter API client for multiple model access."""  
  
 def \_\_init\_\_(self, api\_key: str, model: str = "meta-llama/llama-3.1-70b-instruct"):  
 super().\_\_init\_\_(api\_key, model)  
 self.base\_url = "https://openrouter.ai/api/v1"  
  
 def generate\_commit\_message(self, diff\_content: str) -> str:  
 """Generate commit message using OpenRouter API."""  
  
 headers = {  
 "Authorization": f"Bearer {self.api\_key}",  
 "HTTP-Referer": "https://github.com/Joshi-e8/ai-commit-generator",  
 "X-Title": "Smart Commits AI"  
 }  
  
 data = {  
 "model": self.model,  
 "messages": [  
 {  
 "role": "system",  
 "content": self.\_get\_system\_prompt()  
 },  
 {  
 "role": "user",  
 "content": f"Analyze this diff and create a conventional commit message:\n\n{diff\_content}"  
 }  
 ],  
 "max\_tokens": 100,  
 "temperature": 0.2  
 }  
  
 try:  
 response = self.\_make\_request(  
 f"{self.base\_url}/chat/completions",  
 headers,  
 data  
 )  
  
 return response["choices"][0]["message"]["content"].strip()  
  
 except Exception as e:  
 raise APIError(f"OpenRouter API error: {e}")

### 1.7.4 6.4 Cohere API Client

class CohereClient(APIClient):  
 """Cohere API client for enterprise-grade AI."""  
  
 def \_\_init\_\_(self, api\_key: str, model: str = "command-r-plus"):  
 super().\_\_init\_\_(api\_key, model)  
 self.base\_url = "https://api.cohere.ai/v1"  
  
 def generate\_commit\_message(self, diff\_content: str) -> str:  
 """Generate commit message using Cohere API."""  
  
 headers = {  
 "Authorization": f"Bearer {self.api\_key}",  
 "Content-Type": "application/json"  
 }  
  
 # Cohere uses a different API format  
 data = {  
 "model": self.model,  
 "message": f"Create a conventional commit message for this diff:\n\n{diff\_content}",  
 "max\_tokens": 100,  
 "temperature": 0.3,  
 "preamble": self.\_get\_system\_prompt()  
 }  
  
 try:  
 response = self.\_make\_request(  
 f"{self.base\_url}/chat",  
 headers,  
 data  
 )  
  
 return response["text"].strip()  
  
 except Exception as e:  
 raise APIError(f"Cohere API error: {e}")

### 1.7.5 6.5 Secure HTTP Communication

All API clients use secure HTTP communication:

def \_make\_request(self, url: str, headers: Dict[str, str], data: Dict[str, Any]) -> Dict[str, Any]:  
 """Make HTTP request with security controls."""  
 try:  
 logger.debug(f"Making API request to {url}")  
  
 response = self.session.post(  
 url,  
 headers=headers,  
 json=data,  
 timeout=30,  
 verify=True # SSL verification  
 )  
  
 response.raise\_for\_status()  
 return response.json()  
  
 except requests.exceptions.SSLError:  
 raise APIError("SSL verification failed")  
 except requests.exceptions.Timeout:  
 raise APIError("API request timed out")  
 except requests.exceptions.HTTPError as e:  
 if response.status\_code == 401:  
 raise APIError("Invalid API key")  
 elif response.status\_code == 429:  
 raise APIError("Rate limit exceeded")  
 elif response.status\_code >= 500:  
 raise APIError(f"API server error: {response.status\_code}")  
 else:  
 raise APIError(f"API request failed: {e}")  
 except requests.exceptions.RequestException as e:  
 raise APIError(f"Network error: {e}")  
 except ValueError as e:  
 raise APIError(f"Invalid JSON response: {e}")

## 1.8 7. Git Integration

### 1.8.1 7.1 Git Hook Management

Smart Commits AI integrates with Git through the prepare-commit-msg hook:

class GitHookManager:  
 """Manages Git hooks for Smart Commits AI."""  
  
 def \_\_init\_\_(self, repo\_path: Optional[Path] = None):  
 self.repo\_path = self.\_find\_repo\_root(repo\_path)  
 self.hooks\_dir = self.repo\_path / ".git" / "hooks"  
 self.hook\_file = self.hooks\_dir / "prepare-commit-msg"  
  
 def install\_hook(self) -> bool:  
 """Install the prepare-commit-msg hook with security."""  
 try:  
 # Security: Validate repository path  
 self.\_validate\_repo\_security()  
  
 # Create hooks directory if it doesn't exist  
 self.hooks\_dir.mkdir(exist\_ok=True)  
  
 # Generate hook content  
 hook\_content = self.\_generate\_hook\_content()  
  
 # Security: Validate hook content  
 self.\_validate\_hook\_content(hook\_content)  
  
 # Write hook file with secure permissions  
 self.hook\_file.write\_text(hook\_content, encoding="utf-8")  
  
 # Set secure permissions: owner read/write/execute, group read, others none  
 self.hook\_file.chmod(0o750)  
  
 logger.info(f"Git hook installed: {self.hook\_file}")  
 return True  
  
 except Exception as e:  
 raise GitError(f"Failed to install hook: {e}")

### 1.8.2 7.2 Hook Content Generation

def \_generate\_hook\_content(self) -> str:  
 """Generate secure Git hook content."""  
  
 # Get Python executable path  
 python\_path = sys.executable  
  
 # Security: Validate Python path  
 if not Path(python\_path).exists():  
 raise SecurityError("Invalid Python executable path")  
  
 hook\_content = f'''#!/bin/bash  
# Smart Commits AI - prepare-commit-msg hook  
# Generated automatically - do not edit manually  
  
# Security: Set strict error handling  
set -euo pipefail  
  
# Security: Validate environment  
if [ ! -f "{python\_path}" ]; then  
 echo "Error: Python executable not found"  
 exit 1  
fi  
  
# Check if this is an interactive commit (not merge, rebase, etc.)  
if [ "$#" -eq 1 ] || [ "$2" = "message" ]; then  
 # Security: Use absolute path and validate execution  
 if command -v smart-commits-ai >/dev/null 2>&1; then  
 # Generate commit message with timeout  
 timeout 60 smart-commits-ai generate --hook "$1" 2>/dev/null || {{  
 echo "# Smart Commits AI: Generation failed, using manual commit"  
 }}  
 else  
 echo "# Smart Commits AI: Command not found, using manual commit"  
 fi  
fi  
  
# Security: Ensure clean exit  
exit 0  
'''  
  
 return hook\_content

### 1.8.3 7.3 Hook Security Validation

def \_validate\_hook\_content(self, content: str) -> None:  
 """Validate hook content for security."""  
  
 # Check for dangerous patterns  
 dangerous\_patterns = [  
 r'rm\s+-rf', # Dangerous file operations  
 r'sudo\s+', # Privilege escalation  
 r'curl\s+.\*\|\s\*sh', # Remote code execution  
 r'eval\s+', # Code evaluation  
 r'exec\s+', # Code execution  
 r'\$\([^)]\*\)', # Command substitution  
 r'`[^`]\*`', # Command substitution  
 ]  
  
 for pattern in dangerous\_patterns:  
 if re.search(pattern, content):  
 raise SecurityError(f"Dangerous pattern detected in hook: {pattern}")  
  
 # Validate content length  
 if len(content) > 10000: # 10KB limit  
 raise SecurityError("Hook content too large")  
  
 # Ensure proper shebang  
 if not content.startswith('#!/bin/bash'):  
 raise SecurityError("Invalid hook shebang")

### 1.8.4 7.4 Repository Validation

def \_validate\_repo\_security(self) -> None:  
 """Validate repository for security concerns."""  
  
 # Check if we're in a valid Git repository  
 if not (self.repo\_path / ".git").exists():  
 raise SecurityError("Not a valid Git repository")  
  
 # Security: Check repository ownership  
 try:  
 repo\_stat = self.repo\_path.stat()  
 current\_uid = os.getuid() if hasattr(os, 'getuid') else None  
  
 if current\_uid and repo\_stat.st\_uid != current\_uid:  
 logger.warning("Repository owned by different user")  
 except Exception:  
 pass # Skip ownership check on Windows  
  
 # Check for suspicious repository structure  
 suspicious\_paths = [  
 ".git/hooks/pre-receive",  
 ".git/hooks/post-receive",  
 ".git/hooks/update"  
 ]  
  
 for path in suspicious\_paths:  
 if (self.repo\_path / path).exists():  
 logger.warning(f"Suspicious Git hook detected: {path}")  
  
 # Validate hooks directory permissions  
 if self.hooks\_dir.exists():  
 hooks\_stat = self.hooks\_dir.stat()  
 if hooks\_stat.st\_mode & 0o002: # World writable  
 raise SecurityError("Hooks directory is world-writable")

### 1.8.5 7.5 Hook Management Commands

def uninstall\_hook(self) -> bool:  
 """Safely uninstall the Git hook."""  
 try:  
 if self.hook\_file.exists():  
 # Security: Validate before removal  
 content = self.hook\_file.read\_text(encoding="utf-8")  
 if "Smart Commits AI" not in content:  
 raise SecurityError("Hook file not created by Smart Commits AI")  
  
 # Remove hook file  
 self.hook\_file.unlink()  
 logger.info("Git hook uninstalled")  
 return True  
 else:  
 logger.info("Git hook not found")  
 return False  
  
 except Exception as e:  
 raise GitError(f"Failed to uninstall hook: {e}")  
  
def is\_hook\_installed(self) -> bool:  
 """Check if the Git hook is installed."""  
 if not self.hook\_file.exists():  
 return False  
  
 try:  
 content = self.hook\_file.read\_text(encoding="utf-8")  
 return "Smart Commits AI" in content  
 except Exception:  
 return False  
  
def update\_hook(self) -> bool:  
 """Update existing Git hook."""  
 if self.is\_hook\_installed():  
 self.uninstall\_hook()  
 return self.install\_hook()

## 1.9 8. Command Line Interface

### 1.9.1 8.1 CLI Architecture

The CLI (cli.py) provides a user-friendly interface with comprehensive error handling:

@click.group()  
@click.version\_option(version=\_\_version\_\_)  
@click.option('--debug', is\_flag=True, help='Enable debug mode')  
@click.option('--config', type=click.Path(), help='Custom config file path')  
def cli(debug: bool, config: Optional[str]):  
 """Smart Commits AI - AI-powered Git commit message generator."""  
  
 # Security: Configure logging  
 if debug:  
 logging.basicConfig(level=logging.DEBUG)  
 os.environ['COMMITGEN\_DEBUG'] = 'true'  
  
 # Security: Validate config path if provided  
 if config:  
 config\_path = Path(config)  
 try:  
 validate\_file\_path(Path.cwd(), config\_path)  
 os.environ['COMMITGEN\_CONFIG'] = str(config\_path)  
 except SecurityError as e:  
 click.echo(f"Error: Invalid config path: {e}", err=True)  
 sys.exit(1)

### 1.9.2 8.2 Core Commands

#### 1.9.2.1 8.2.1 Generate Command

@cli.command()  
@click.option('--hook', type=click.Path(), help='Hook mode: commit message file')  
@click.option('--repo', type=click.Path(), help='Repository path')  
@click.option('--dry-run', is\_flag=True, help='Show message without committing')  
@handle\_errors  
def generate(hook: Optional[str], repo: Optional[str], dry\_run: bool):  
 """Generate AI-powered commit message."""  
  
 try:  
 # Security: Validate paths  
 repo\_path = None  
 if repo:  
 repo\_path = validate\_file\_path(Path.cwd(), Path(repo))  
  
 # Initialize generator  
 config = Config(repo\_path=repo\_path)  
 generator = CommitGenerator(config)  
  
 # Generate message  
 message = generator.generate\_commit\_message(repo\_path)  
  
 if hook:  
 # Hook mode: write to commit message file  
 hook\_path = validate\_file\_path(Path.cwd(), Path(hook))  
 hook\_path.write\_text(message, encoding="utf-8")  
 console.print(f"✅ Generated message: {mask\_sensitive\_data(message, 50)}")  
 elif dry\_run:  
 # Dry run: display message  
 console.print(f"Generated commit message:\n{message}")  
 else:  
 # Interactive mode: show and confirm  
 console.print(f"Generated message: {message}")  
 if click.confirm("Use this commit message?"):  
 # Execute git commit with the message  
 result = secure\_subprocess\_run(  
 ["git", "commit", "-m", message],  
 cwd=repo\_path or Path.cwd(),  
 timeout=30  
 )  
 if result.returncode == 0:  
 console.print("✅ Commit successful!")  
 else:  
 console.print("❌ Commit failed!")  
  
 except Exception as e:  
 logger.error(f"Generate command failed: {e}")  
 console.print(f"[red]❌ Error: {e}[/red]")  
 sys.exit(1)

#### 1.9.2.2 8.2.2 Install Command

@cli.command()  
@click.option('--repo', type=click.Path(), help='Repository path')  
@click.option('--force', is\_flag=True, help='Force reinstallation')  
@handle\_errors  
def install(repo: Optional[str], force: bool):  
 """Install Git hooks for automatic commit message generation."""  
  
 try:  
 # Security: Validate repository path  
 repo\_path = None  
 if repo:  
 repo\_path = validate\_file\_path(Path.cwd(), Path(repo))  
  
 # Initialize hook manager  
 hook\_manager = GitHookManager(repo\_path)  
  
 # Check existing installation  
 if hook\_manager.is\_hook\_installed() and not force:  
 console.print("✅ Git hooks already installed")  
 console.print("Use --force to reinstall")  
 return  
  
 # Install hooks  
 if hook\_manager.install\_hook():  
 console.print("✅ Git hooks installed successfully!")  
 console.print("\nNext steps:")  
 console.print("1. Set up your API key in .env file")  
 console.print("2. Run 'git commit' to test AI generation")  
 else:  
 console.print("❌ Failed to install Git hooks")  
 sys.exit(1)  
  
 except Exception as e:  
 logger.error(f"Install command failed: {e}")  
 console.print(f"[red]❌ Error: {e}[/red]")  
 sys.exit(1)

#### 1.9.2.3 8.2.3 Status Command

@cli.command()  
@click.option('--verbose', is\_flag=True, help='Show detailed status')  
@handle\_errors  
def status(verbose: bool):  
 """Show Smart Commits AI status and configuration."""  
  
 try:  
 console.print(f"[blue]🤖 Smart Commits AI v{\_\_version\_\_}[/blue]")  
 console.print(f"[blue]🔒 Security Status: Enterprise-Ready[/blue]")  
 console.print()  
  
 # Initialize configuration  
 config = Config()  
  
 # Repository status  
 try:  
 repo\_root = config.repo\_root  
 console.print(f"Repository: [green]✅ {repo\_root}[/green]")  
 except Exception:  
 console.print("Repository: [red]❌ Not in Git repository[/red]")  
 return  
  
 # Hook status  
 hook\_manager = GitHookManager(repo\_root)  
 if hook\_manager.is\_hook\_installed():  
 console.print("Git hooks: [green]✅ Installed[/green]")  
 else:  
 console.print("Git hooks: [yellow]⚠️ Not installed[/yellow]")  
 console.print(" Run 'smart-commits-ai install' to set up")  
  
 # Configuration status  
 console.print(f"Provider: [cyan]{config.provider}[/cyan]")  
 console.print(f"Model: [cyan]{config.model}[/cyan]")  
 console.print(f"Config file: [dim]{config.config\_file}[/dim]")  
 console.print(f"Env file: [dim]{config.env\_file}[/dim]")  
  
 # API key status (securely masked)  
 try:  
 api\_key = config.api\_key  
 masked\_key = mask\_sensitive\_data(api\_key, 4)  
 console.print(f"API key: [green]✅ Configured[/green] ({masked\_key})")  
 except ConfigError:  
 console.print("API key: [red]❌ Not configured[/red]")  
 console.print(f" Set {config.provider.upper()}\_API\_KEY in .env file")  
  
 # Security status  
 console.print("\n[blue]🔒 Security Features:[/blue]")  
 console.print("✅ Input validation enabled")  
 console.print("✅ Path traversal protection")  
 console.print("✅ API key masking")  
 console.print("✅ Secure subprocess execution")  
 console.print("✅ SSL verification enabled")  
 console.print("✅ Error sanitization active")  
  
 if verbose:  
 console.print(f"\n[dim]Configuration details:[/dim]")  
 console.print(f"Max chars: {config.max\_chars}")  
 console.print(f"Max diff size: {config.max\_diff\_size}")  
 console.print(f"Max retries: {config.max\_retries}")  
 console.print(f"Retry delay: {config.retry\_delay}s")  
 console.print(f"Commit types: {', '.join(config.commit\_types)}")  
  
 except Exception as e:  
 logger.error(f"Status command failed: {e}")  
 console.print(f"[red]❌ Error: {e}[/red]")  
 sys.exit(1)

### 1.9.3 8.3 Configuration Commands

#### 1.9.3.1 8.3.1 Config Show Command

@cli.group()  
def config():  
 """Configuration management commands."""  
 pass  
  
@config.command('show')  
@click.option('--section', help='Show specific configuration section')  
@click.option('--validate', is\_flag=True, help='Validate configuration')  
@handle\_errors  
def config\_show(section: Optional[str], validate: bool):  
 """Show current configuration."""  
  
 try:  
 cfg = Config()  
  
 if validate:  
 console.print("[blue]🔍 Validating configuration...[/blue]")  
 try:  
 cfg.validate()  
 console.print("[green]✅ Configuration is valid[/green]")  
 except (ConfigError, SecurityError) as e:  
 console.print(f"[red]❌ Configuration error: {e}[/red]")  
 sys.exit(1)  
  
 # Show configuration  
 if section:  
 # Show specific section  
 config\_data = cfg.\_config.get(section, {})  
 if not config\_data:  
 console.print(f"[red]❌ Section '{section}' not found[/red]")  
 sys.exit(1)  
 else:  
 # Show all configuration (with sensitive data masked)  
 config\_data = cfg.\_config.copy()  
  
 # Security: Mask sensitive values  
 if 'api' in config\_data and 'key' in str(config\_data['api']):  
 # This is handled by the config class internally  
 pass  
  
 # Pretty print configuration  
 console.print(yaml.dump(config\_data, default\_flow\_style=False))  
  
 except Exception as e:  
 logger.error(f"Config show failed: {e}")  
 console.print(f"[red]❌ Error: {e}[/red]")  
 sys.exit(1)