1 Smart Commits AI - Comprehensive Documentation
- 1.1 Table of Contents
- 1.2 1. Introduction
  - 1.2.1 1.1 Overview
  - 1.2.2 1.2 Key Features
  - 1.2.3 1.3 Version Information
  - 1.2.4 1.4 System Requirements
- 1.3 2. Architecture Overview
  - 1.3.1 2.1 System Architecture
  - 1.3.2 2.2 Data Flow
  - 1.3.3 2.3 Security Architecture
- 1.4 3. Installation Guide
  - 1.4.1 3.1 Universal Installation
  - 1.4.2 3.2 Manual Installation Methods
  - 1.4.3 3.3 Platform-Specific Instructions
  - 1.4.4 3.4 Verification
- 1.5 4. Configuration Management
  - 1.5.1 4.1 Configuration System Overview
  - 1.5.2 4.2 Configuration File Structure
  - 1.5.3 4.3 Environment Variables
  - 1.5.4 4.4 Configuration Commands
- 1.6 5. Core Functionality
  - 1.6.1 5.1 Commit Message Generation Engine
  - 1.6.2 5.2 Security Features in Core Engine
- 1.7 6. API Clients
  - 1.7.1 6.1 API Client Architecture
  - 1.7.2 6.2 Groq API Client
  - 1.7.3 6.3 OpenRouter API Client
  - 1.7.4 6.4 Cohere API Client
  - 1.7.5 6.5 Secure HTTP Communication
- 1.8 7. Git Integration
  - 1.8.1 7.1 Git Hook Management
  - 1.8.2 7.2 Hook Content Generation
  - 1.8.3 7.3 Hook Security Validation
  - 1.8.4 7.4 Repository Validation
  - 1.8.5 7.5 Hook Management Commands
- 1.9 8. Command Line Interface
  - 1.9.1 8.1 CLI Architecture
  - 1.9.2 8.2 Core Commands
  - 1.9.3 8.3 Configuration Commands

1 Smart Commits AI - Comprehensive Documentation

1.1 Table of Contents

1. Introduction
2. Architecture Overview
3. Installation Guide
4. Configuration Management
5. Core Functionality
6. API Clients
7. Git Integration
8. Command Line Interface
9. Security Features
10. Advanced Usage
11. Troubleshooting
12. API Reference
13. Development Guide
14. Security Analysis
15. 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:
```

```yaml
    - "*.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

```bash
# 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

```python
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

```python
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:

```python
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

```python
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

```python
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:

```python
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

```python
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)
```