Changelog Generator: A Tool for Automated Changelog Generation from Git History

Sunskar Dhanker
University of Illinois Urbana-Champaign
Champaign, Illinois, USA
dhanker2@illinois.edu

ABSTRACT

This paper presents the Changelog Generator, a Python-based tool designed to automate the creation of changelogs from git commit history. The tool supports local repositories and GitHub URLs, offering features like commit filtering, interactive selection, and intelligent commit scoring. It utilizes the Anthropic Claude API to generate human-readable changelog summaries. The implementation includes a command-line interface with various customization options and a scoring system that evaluates commit importance based on message content and file changes.

ACM Reference Format:
Sunskar Dhanker. 2025. Changelog Generator: A Tool for Automated Changelog Generation from Git History. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Repository: https://github.com/sunnyskar/changelog-generator

1 INTRODUCTION

Maintaining accurate and meaningful changelogs is crucial for software projects, yet it remains a time-consuming task often neglected. Changelogs are essential documentation for developers, project managers, and end users, providing a clear record of new features, bug fixes, and other significant changes across project releases. Well-maintained changelogs improve transparency, facilitate collaboration, and help teams track progress, plan releases, and communicate updates to stakeholders.

Despite their importance, changelogs are frequently overlooked or updated inconsistently. Manual changelog creation is prone to human error, omissions, and inconsistencies, especially in fast-paced development environments with frequent commits and multiple contributors. As projects scale, the volume and complexity of commit histories make it increasingly difficult to extract relevant information and summarize changes effectively.

The Changelog Generator addresses these challenges by automating the creation of changelogs from git commit history. By leveraging structured commit data and modern AI capabilities, the tool ensures that changelogs are comprehensive, accurate, and consistently formatted. Automation not only saves developer time but also improves the quality and reliability of project documentation.

The tool is designed to be both user-friendly for developers and powerful enough to handle complex repository histories. Its command-line interface and structured output make it particularly suitable for integration with Large Language Models (LLMs) and AI coding assistants. LLMs can leverage the Changelog Generator to:

- Access up-to-date information about code changes through the command-line interface
- Generate human-readable summaries of recent changes using the Claude API integration
- Filter and analyze commit history based on specific criteria
- Provide developers with accurate information about recent updates and modifications

This integration enables LLMs to maintain current knowledge about codebases and provide more accurate assistance to developers. The tool's ability to process both local repositories and GitHub URLs makes it versatile for various development scenarios.

In summary, the Changelog Generator bridges the gap between raw commit history and user-friendly changelogs, empowering both human developers and AI systems to better understand and communicate the evolution of software projects.

2 SYSTEM OVERVIEW

The Changelog Generator is implemented in Python and provides the following core features:

- Support for both local git repositories and GitHub URLs
- Commit filtering by date range and patterns
- Interactive commit selection
- Commit scoring system based on message content and file changes
- Integration with Anthropic Claude API for changelog generation
- Command-line interface with extensive customization options

At a high level, the system is designed with modularity and extensibility in mind. The architecture separates repository access, commit processing, scoring, and changelog generation into distinct components, making it easy to extend or adapt the tool for additional use cases or integrations in the future.

2.1 Architecture and Workflow

The typical workflow of the Changelog Generator consists of the following stages:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(1) **Repository Access:** The tool determines whether the target is a local directory or a GitHub URL. It then fetches the commit history using the appropriate backend (GitPython for local repositories, GitHub API for remote).
(2) **Commit Filtering:** Users can specify filters such as date ranges, commit message patterns, tags, or custom exclusion rules to narrow down the set of commits to be included in the changelog.
(3) **Commit Scoring and Categorization:** Each commit is analyzed and scored based on its message content, the number and type of files changed, and other heuristics. Commits are optionally grouped into user-defined categories for improved readability.
(4) **Interactive Review (Optional):** In interactive mode, users can review, select, or deselect commits to further refine the changelog contents.
(5) **Changelog Generation:** The selected commits, along with their metadata and scores, are passed to the Anthropic Claude API. The API generates a well-structured, human-readable changelog in markdown format, with changes grouped and summarized under appropriate headings.
(6) **Output and Integration:** The resulting changelog can be previewed, saved to a file, or integrated into release workflows, documentation pipelines, or LLM-based coding assistants.

2.2 Design Philosophy and Extensibility

The Changelog Generator is built to be both robust and flexible:

- **Modularity:** Each major function (repository access, filtering, scoring, generation) is encapsulated, allowing for easy maintenance and future enhancements.
- **Customizability:** The command-line interface exposes a wide range of options, enabling users to tailor the tool to their workflow and project conventions.
- **Extensibility:** The system is designed to support additional repository platforms, alternative AI summarization services, or custom scoring heuristics with minimal changes.
- **Integration-Ready:** Structured outputs and API-driven design make it easy to plug the tool into CI/CD pipelines, documentation systems, or AI-powered developer tools.

By combining traditional software engineering techniques with modern AI capabilities, the Changelog Generator provides a comprehensive, automated solution for changelog management that scales with project complexity and team size.

3 IMPLEMENTATION DETAILS

The system follows a pipeline architecture with four main stages:

(1) **Data Acquisition**: Fetch raw commit data from local or remote repositories
(2) **Processing Pipeline**: Filter, score, and categorize commits
(3) **AI Integration**: Generate human-readable summaries via Claude API
(4) **Output Generation**: Produce final changelog in specified format

3.1 Core Components

3.1.1 Repository Handler. Handles repository access through two parallel implementations:

- **Local Repositories**: Uses GitPython's `Repo` object with:

```
repo = git.Repo(path)
commits = list(repo.iter_commits())
```

- **GitHub Repositories**: Implements GitHub API v3 client with:
  - OAuth authentication using personal access tokens
  - Pagination handling for large commit histories
  - Rate limit monitoring and retry logic

3.1.2 Commit Scoring System. Implements a weighted scoring algorithm:

$$\text{Score} = \sum(\text{KeywordWeights}) + \log(\text{FilesChanged}) + \frac{\text{Insertions}}{10} + \text{DirectoryBonus}$$

Where:

- **KeywordWeights**: Predefined weights for commit message keywords (e.g., "fix"=3, "feat"=5)
- **DirectoryBonus**: Additional points for changes in critical directories (e.g., `src/`=2, `tests/`=1)
- **FileTypeWeights**: Bonus points for specific file types (e.g., `.py`=1, `.md`=0.5)

3.1.3 Changelog Generation. The Claude API integration features:

- A structured prompt template with:
  - Project-specific instructions
  - Example-based formatting guidance
  - Context-aware summarization rules
- Response validation with:
  - Markdown syntax checking

CONTENT SAFETY FILTERS

  - Fallback to template-based generation on API failure

3.2 Error Handling

The system implements robust error handling through:

- **Repository Access**:
  - Graceful recovery from corrupted git objects
  - Network error retries with exponential backoff
  - Authentication failure detection
- **API Integration**:
  - Request timeout management (30s default)
  - Rate limit tracking (requests/minute)
  - Response validation against schema
- **User Input Validation**:
  - Date format verification
  - Path sanitization
  - Injection attack prevention

3.3 Performance Optimizations

Key performance features include:

- **Caching**:

  - 1-hour TTL for GitHub API responses
  - Local commit metadata cache
  - Batch processing of AI requests
- **Parallel Processing**:
  - Concurrent scoring of commits
  - Async API requests
  - Background I/O operations
- **Memory Management**:
  - Streaming of large commit histories
  - Selective field loading for git objects
  - Paginated API response handling

3.4 Testing Strategy

The implementation includes:

- Unit tests for core scoring algorithms
- Integration tests with mock repositories
- Golden master testing for AI output
- Fuzz testing for edge case handling
- Performance benchmarking suite

3.5 Security Considerations

- Secure credential handling via environment variables
- Input sanitization for GitHub URL parameters
- Read-only repository access
- HTTPS-only API communication
- Audit logging for sensitive operations

4 USAGE

4.1 Installation

```
pip install -r requirements.txt
export ANTHROPIC_API_KEY="your-api-key-here"
```

4.2 Command Line Interface

The tool provides a rich command-line interface with the following options:

- Repository path (local or GitHub URL)
- Number of commits to include
- Date range filtering
- Pattern-based commit exclusion
- Custom categories for changelog organization
- Tag-based commit filtering
- Interactive mode for commit selection
- Preview mode for commit review

4.3 Example Usage

```
# Generate changelog for last 10 commits
python changelog_generator.py /path/to/repo 10

# Filter by date range
python changelog_generator.py /path/to/repo 10
--from-date 2024-01-01 --to-date 2024-02-01

# Use custom categories and exclude patterns
python changelog_generator.py /path/to/repo 10
-c "New Features" -c "Bug Fixes" -e "chore:" -e "docs:"
# Interactive mode
python changelog_generator.py /path/to/repo 10 --interactive
```

5 IMPLEMENTATION EXAMPLES

5.1 Commit Scoring

```python
def score_commit(commit: Dict, repo: Optional[git.Repo] =
    None) -> Tuple[int, str]:
    score = 0
    message = commit['message'].lower()

    # Score based on keywords
    for keyword, points in IMPORTANT_KEYWORDS.items():
        if keyword in message:
            score += points

    # Score based on file changes
    if repo and 'hash' in commit:
        stats = repo.commit(commit['hash']).stats.total
        if stats['files'] > 5:
            score += 2
        elif stats['files'] > 2:
            score += 1

    return score, explanation
```

Listing 1: Commit Scoring Implementation

5.2 Changelog Generation

```python
def generate_changelog(commits: List[Dict], categories:
    Optional[List[str]] = None) -> str:
    prompt = f"""Based on the following git commit
    history, generate a user-friendly changelog in
    markdown format.
    Requirements:
    1. Use proper markdown formatting
    2. Categorize changes under headers
    3. Group related commits together
    4. Focus on user-relevant changes
    """

    response = requests.post(API_URL, headers=headers,
    json={
        "model": "claude-3-sonnet-20240229",
        "messages": [{"role": "user", "content": prompt}]
    })

    return response.json()["content"][0]["text"]
```

Listing 2: Changelog Generation with Claude API

6 CONCLUSION

The Changelog Generator provides a practical and automated solution for generating changelogs directly from git commit history, addressing a persistent challenge in software engineering. The tool streamlines a process that is often tedious and error-prone when performed manually by combining robust repository analysis, intelligent commit scoring, and integration with advanced AI summarization models.

Beyond its immediate utility, the Changelog Generator exemplifies the growing synergy between traditional software tooling and AI-driven workflows. Its design enables seamless integration into

modern development pipelines, continuous integration/continuous deployment (CI/CD) systems, and AI-powered coding assistants. This saves developer time and ensures that changelogs remain accurate, comprehensive, and accessible to all project stakeholders.

The tool's modular architecture and extensible interface lay the groundwork for future enhancements. Potential directions for future work include supporting additional version control platforms (such as Mercurial or Subversion), incorporating more advanced natural language processing techniques for commit analysis, and enabling direct integration with project management and release automation tools. Further, as AI models evolve, generated changelogs' quality and contextual relevance will only improve, opening new possibilities for automated documentation and developer support.

In summary, the Changelog Generator bridges the gap between raw commit data and user-friendly documentation, empowering human and AI collaborators to understand, maintain, and communicate the evolution of software projects. By automating changelog creation, the tool enhances developer productivity and contributes to higher standards of transparency and collaboration in software development.

7 FUTURE WORK: INTEGRATION WITH RETRIEVAL-AUGMENTED GENERATION (RAG) FRAMEWORKS

Integrating the Changelog Generator with Retrieval-Augmented Generation (RAG) frameworks is a promising avenue for enhancing it. This approach would enable advanced, context-aware querying and automated documentation workflows by leveraging large language models (LLMs) and efficient retrieval systems.

7.1 Connecting to a RAG Framework

By connecting the Changelog Generator to a RAG pipeline, LLMs can make the generated changelogs, commit histories, and related documentation searchable and retrievable in real time. This integration would allow developers and AI assistants to answer natural language questions about project history, code changes, and rationales, precisely referencing the underlying data. For example, a user could query, "What security fixes were introduced in March 2025?" The RAG system would retrieve and summarize the relevant changelog entries.

7.2 Chunking Strategies for Changelog Data

Effective RAG systems require breaking down (chunking) large documents or datasets into smaller, manageable, and semantically meaningful pieces called "chunks." For changelog and commit history data, chunking can be performed in several ways:

- **Fixed-size chunking:** Split changelogs or commit logs into uniform blocks, such as every 100 lines or 1,000 tokens.
- **Semantic chunking:** Divide data based on logical units, such as individual commits, release notes, or categorized changelog sections (e.g., "Bug Fixes," "New Features").
- **Paragraph or sentence-based chunking:** Especially useful for long-form summaries or detailed commit messages, ensuring each chunk remains contextually coherent.
- **Overlapping chunks:** Maintain context across chunk boundaries by allowing some overlap (e.g., the last sentence of one chunk appears at the start of the next), which helps preserve meaning and continuity.

Each chunk should be enriched with metadata (e.g., commit hash, author, date, affected files, tags) to facilitate precise retrieval and filtering.

7.3 Indexing for Efficient Retrieval

After chunking, each segment is converted into a vector embedding, a numerical representation capturing its semantic content, using an embedding model. These embeddings are then stored in a vector database, which supports fast similarity search and retrieval. Key indexing strategies include:

- **Approximate Nearest Neighbors (ANN):** Balances retrieval speed and accuracy for large datasets.
- **Hierarchical Navigable Small World (HNSW):** Efficient for real-time applications, providing quick and accurate retrieval.
- **Inverted File Index (IVF) and Product Quantization (PQ):** Useful for massive datasets, optimizing memory usage and search times.

The choice of indexing strategy depends on the scale of the changelog data and the application's latency/accuracy requirements.

7.4 LLM Tool Calling Integration

Modern LLMs can be configured to "tool call" external systems, enabling them to interact programmatically with tools like the Changelog Generator [5]. In this setup, the LLM can:

- Receive a user query (e.g., "List all new features added since version 2.0").
- Issue a tool call to the Changelog Generator, specifying parameters such as date range, categories, or search terms.
- Retrieve the relevant changelog entries or summaries, either directly from the generator or via the RAG retrieval system.
- Synthesize a user-facing response that references or summarizes the retrieved data.

This workflow allows the LLM to act as an intelligent assistant, dynamically leveraging the most current and relevant project information without manual intervention [6].

7.5 Benefits and Implementation Roadmap

Integrating the Changelog Generator with a RAG framework and enabling LLM tool calls would unlock several benefits:

- **Natural language querying** of changelog and commit history, enabling both developers and AI assistants to extract precise, context-rich answers.
- **Automated, up-to-date documentation** that evolves with the codebase, reducing manual effort and improving transparency.
- **Enhanced troubleshooting and code review** by enabling targeted retrieval of historical changes, rationales, and their impacts.

- **Scalability** for large projects, as chunking and indexing strategies ensure efficient retrieval even from massive repositories.

A possible implementation roadmap includes:

(1) Develop a preprocessing pipeline to chunk and embed changelog and commit data.
(2) Store embeddings and metadata in a scalable vector database.
(3) Integrate with a RAG framework (such as LangChain or LlamaIndex) to enable retrieval and generation workflows.
(4) Expose an API or command-line interface for querying and interactive exploration, enabling LLMs to tool call the system as needed.

Following this roadmap, the Changelog Generator can become a foundational component in intelligent, retrieval-augmented software development environments, supporting more dynamic and context-aware developer workflows.

8 ACKNOWLEDGMENTS

Thanks to the open-source community for the libraries that made this project possible.

REFERENCES

[1] GitPython Documentation, https://gitpython.readthedocs.io/
[2] Click Documentation, https://click.palletsprojects.com/
[3] Questionary Documentation, https://questionary.readthedocs.io/
[4] Anthropic API, https://docs.anthropic.com/claude/reference/complete_post
[5] Vapi, "Custom LLM Tool Calling Integration," https://docs.vapi.ai/customization/tool-calling-integration
[6] Slaptijack, "Auto-Generating Changelogs with Git Hooks and OpenAI," https://slaptijack.com/programming/auto-generating-changelogs-with-git-hooks-and-openai.html