Introducing commit trailers

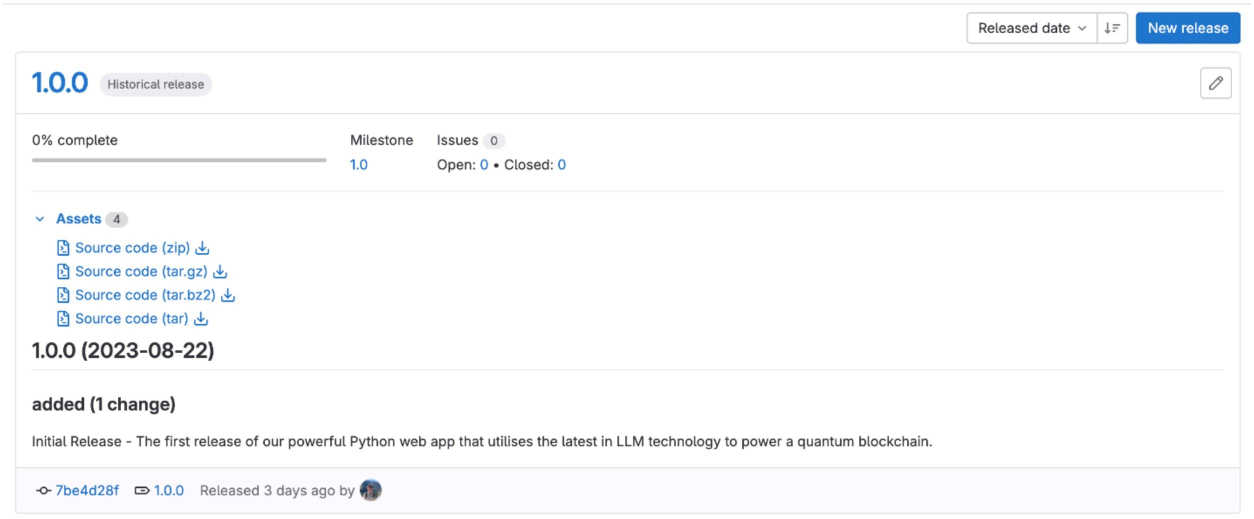
Commit trailers are structured entries in your git commits, created by adding simple <HEADER>:<BODY> format messages to the end of your commit. The git CLI tool can then parse and extract these for use in other systems. An example you might have already used is git commit --sign-off to sign off on a commit. This is implemented by adding a Signed-off-by: <Your Name> trailer to the commit. We can add any arbitrary structured data here, which makes it a great place to store information that could be useful for our changelog.

In fact, if we use a Changelog: <added/changed/removed> trailer in our commits, the GitLab Changelog API will parse these and use them to create a changelog for us automatically!

Let's see this in action by making some changes to a real codebase and performing a release and generating release notes and changelog entries.

Our example project

I'm using a simple repository. Let's pretend Version 1.0.0 of the application was just released and is the current version of the code. I've also created a 1.0.0 release in GitLab, which I did manually because we haven't created our automated release pipeline yet:



1.0.0 (2023-08-22)

added (1 change)

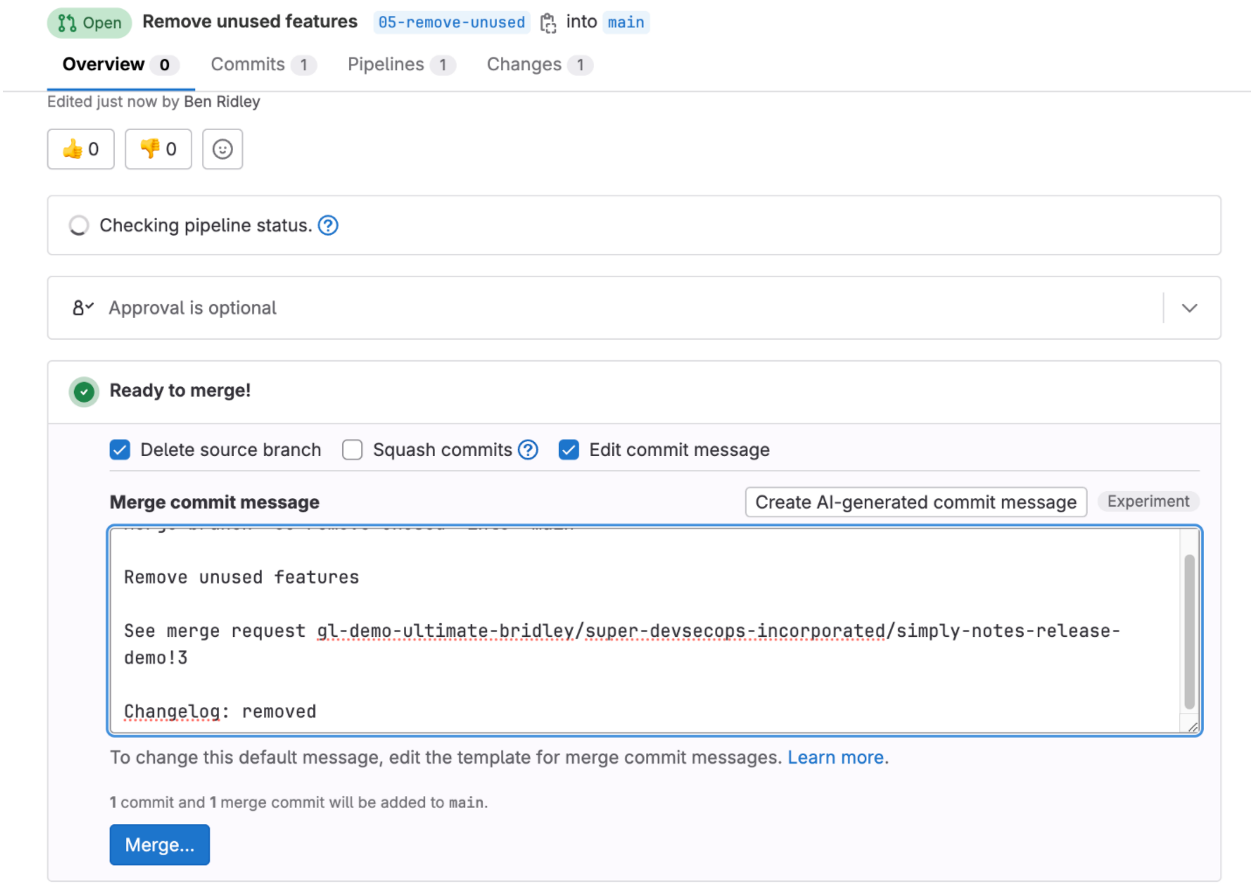
Initial Release - The first release of our powerful Python web app that utilises the latest in LLM technology to power a quantum blockchain.

7be4d28f 1.0.0 Released 3 days ago by

Making our changes

We're in rapid development mode, so we're going to be working on releasing Version 2.0.0 of our application today. As part of our 2.0.0 release, we're going to be adding a new feature to our app: A chatbot! And we're also going to be removing the quantum blockchain feature, because we only needed that for our first venture capital funding round. Also, we're going to be adding an automated release job to our CI/CD pipeline for our 2.0.0 release.

First, let's remove unneeded features. I've created a merge request that contains the necessary removals. Importantly, we need to ensure we have a commit message that includes the Changelog: removed trailer. There are a few ways to do this, such as including it directly in a commit, or performing an interactive rebase and adding it using the CLI. But I think the easiest way in our situation is to leave it until the end and then use the Edit commit message button in GitLab to add the trailer to the merge commit like so:



Remove unused features `05-remove-unused` into `main`

Edited just now by Ben Ridley

Merge commit message

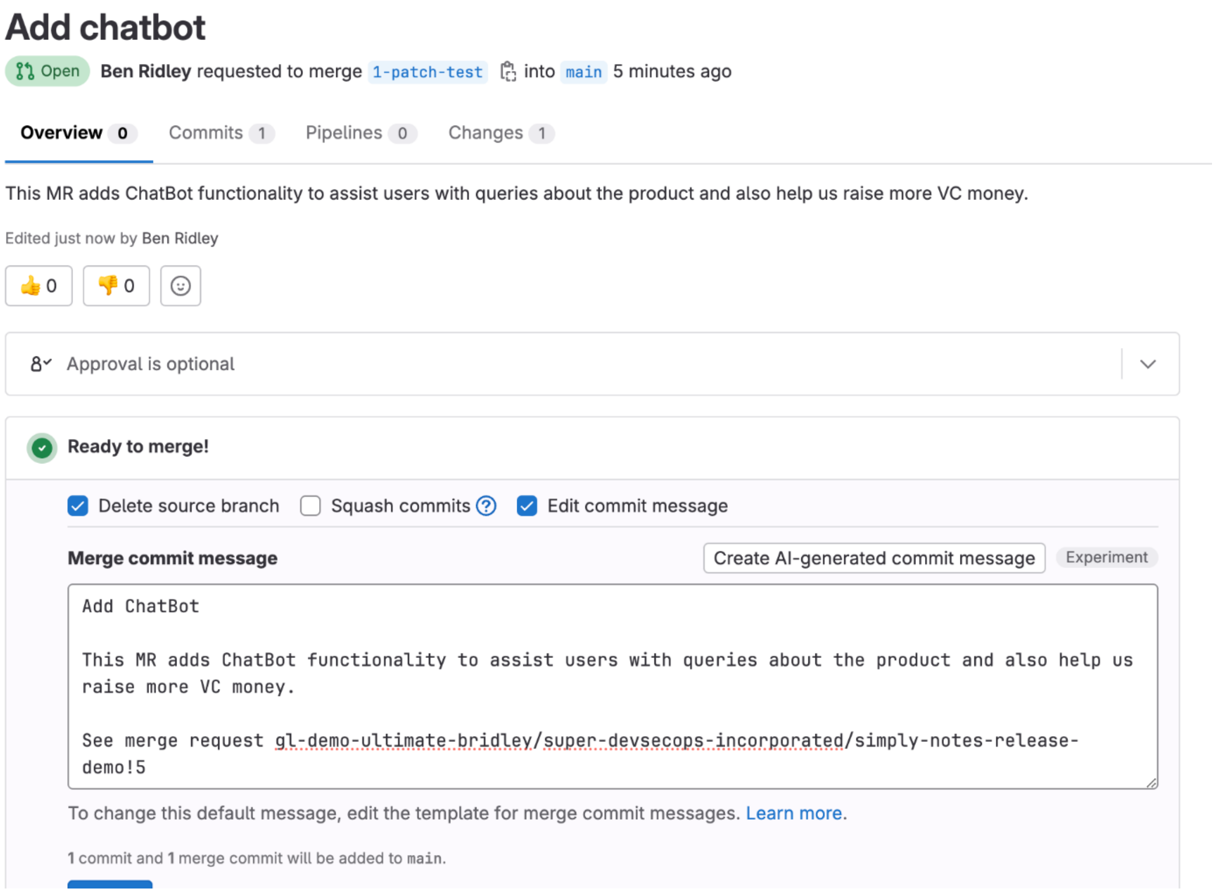
```
Remove unused features

See merge request gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-
demo!3

Changelog: removed
```

To change this default message, edit the template for merge commit messages. Learn more.

1 commit and 1 merge commit will be added to `main`.

If you use this method, you can also change the merge commit title to something more succinct. I've changed the title of my merge commit to 'Remove Unused Features', as this is what will appear in the changelog entry.

Next, let's add some new functionality for the 2.0.0 release. Again, all we need to do is open another merge request that includes our new features and then edit the merge commit to include the Changelog: added trailer and edit the commit title to be more succinct:


Add chatbot

Open Ben Ridley requested to merge `1-patch-test` into `main` 5 minutes ago

This MR adds ChatBot functionality to assist users with queries about the product and also help us raise more VC money.

Edited just now by Ben Ridley

[✓] Delete source branch [ ] Squash commits [✓] Edit commit message

Merge commit message

```
Add ChatBot

This MR adds ChatBot functionality to assist users with queries about the product and also help us
raise more VC money.

See merge request gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-
demo!5
```

To change this default message, edit the template for merge commit messages. Learn more.

1 commit and 1 merge commit will be added to main.

Now we're pretty much ready to release 2.0.0. But we don't want to create our release manually this time. So before our release we're going to add some jobs to our .gitlab-ci.yml file that will perform the release for us automatically, and generate the respective release notes and changelog entries, when we tag our code with a new version like 2.0.0.

Building an automated release pipeline

For our pipeline to work, we need to create a project access token that will allow us to call GitLab's API to generate changelog entries. Create a project access token with the API scope, and then store the token as a CI/CD variable called CI_API_TOKEN. We'll reference this variable to authenticate to the API.

NOTE:- The below code includes a simple job to increment the tag by patch and then update the .version file with the latest tag. This job was written because that is how tags are being created in ci commons. If you are creating tags manually, please remove this job.

Next, we're going to add three new jobs to our gitlab-ci.yml file:

```
default:
  tags:
    - default

stages:
    - release

workflow:            # Execute the pipeline only when run pipeline button is cli
cked or tag is pushed
   rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
    - if: $CI_COMMIT_TAG

bump_tag:
  image: alpine:latest
  stage: release
  rules:
    - if: '$CI_COMMIT_TAG == null' # Trying to prevent execution when commit tag
is present
  before_script:
    - apk add --no-cache git
    - apk add curl jq
    # putting credentials
    - git config --global user.email "${GITLAB_EMAIL}"
    - git config --global user.name "${GITLAB_USER}"
  script:
    # Checking if file path is getting read
    # - echo $STACK_VERSION_FILE
    - export LATEST_TAG=$(cat .version)
    - echo "$LATEST_TAG"

    # Incrementing by patch
    - NEW_TAG=$(echo $LATEST_TAG | awk -F. '{printf "%d.%d.%d", $1, $2, $3 + 1}')
    - echo "New tag will be $NEW_TAG"

    # POST req to gitlab api endpoint to create a new tag
    - 'curl -H "PRIVATE-TOKEN: $GITLAB_ACCESS_TOKEN" -X POST --
data "tag_name=$NEW_TAG&ref=$CI_COMMIT_SHA" "$CI_API_V4_URL/projects/$CI_PROJECT_
ID/repository/tags"'
    # Updating the .version file
    - echo "$NEW_TAG" > .version
    - git add .
    - git commit -m "automated version bump"
    - git push -
o ci.skip https://root:$GITLAB_ACCESS_TOKEN@$CI_SERVER_HOST/$CI_PROJECT_PATH.git
HEAD:main

prepare_notes:
  stage: release
  image: alpine:latest
  rules:
    - if: '$CI_COMMIT_TAG' # gets triggered at each tag push

  script:
    - apk add curl jq

    # create release notes by using changelog api
    - 'curl -H "PRIVATE-
TOKEN: $GITLAB_ACCESS_TOKEN" "$CI_API_V4_URL/projects/$CI_PROJECT_ID/repository/c
hangelog?version=$CI_COMMIT_TAG&trailer=Type&config_file=changelog_config.yml" |
jq -r .notes > release_notes.md'
  artifacts:
    paths:
      - release_notes.md

generate_descriptive_notes:
  stage: release
  image: python:3.9-alpine

  rules:
    - if: '$CI_COMMIT_TAG' # gets triggered at each tag push
  needs:
    - job: prepare_notes
      artifacts: true
  before_script:
    - apk add curl jq
    - pip install google-generativeai
  script:
    - python gemini.py
  artifacts:
    paths:
      - release_notes.md

release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  needs:
    - job: generate_descriptive_notes
```

```
      artifacts: true
  rules:
    - if: '$CI_COMMIT_TAG'
  script:
    - echo "Creating release"
    - echo "$CI_COMMIT_TAG"
  release:
    name: '$CI_COMMIT_TAG'
    description: release_notes.md
    tag_name: '$CI_COMMIT_TAG'
    ref: '$CI_COMMIT_SHA'
```

In the above configuration, the prepare_job uses curl and jq to call the GitLab Changelog API endpoint and then passes this to our generate_descriptive_notes job. This job uses a python script which calls the endpoint any text generation model of your choice. For the purpose of the demo we have used Gemini 1.5 flash. This is what the python script looks like:

```python
import google.generativeai as genai

genai.configure(api_key="<Your API KEY>")

generation_config = {
  "temperature": 1,
  "top_p": 0.95,
  "top_k": 64,
  "max_output_tokens": 8192,
  "response_mime_type": "text/plain",
}

model = genai.GenerativeModel(
  model_name="gemini-1.5-flash",
  generation_config=generation_config,
)

chat_session = model.start_chat(
  history=[
  ]
)

f = open("release_notes.md", "r+")
data = f.read()
```

```
response = chat_session.send_message("""You’re a helpful assistant specialized in
 generating descriptive release notes. Analyze the list of changes between 2 rele
ases which I am going to provide you using GitLab's Changelog API.  DO NOT COMMUN
ICATE WITH THE USER. Focus only on generating descriptive notes.

1. Provide a brief overview of the release highlighting major changes.
2. List each change with a short, clear description.
3. Categorize changes into different headings. Headings are followed by '###' in
the below changes to be provided
4. Maintin a professional yet approachable tone. Do not make any conclusion about
 how a commit will influence a project if you don't know.
5. Use clear, non-technical language where possible.
6. Use bullet points for individual changes and subheadings for different categor
ies.
7. Always write the date in Month Day Year format. These are the list of changes:
\n """+ data)

f.write("\n"+response.text)

f.close()
```

The analysis generated by the ai model is appended to the changelog file and passed onto the release_job to actually create the release. To break it down further:

- We use the project access token created earlier to call the GitLab Changelog API, which performs the generation of the release notes and we store this as an artifact.
- We're using the $CI_COMMIT_TAG variable as the version. For this to work, we need to be using semantic versioning for our tags (something like 2.0.0 for example), so you'll notice I've also restricted the release job using a rules section that checks for a semantic version tag.
- Semantic versioning is required for the GitLab Changelog API to work. It uses this format to find the most recent release to compare to our current release.
- We use the official release-cli image from GitLab. The release-cli is required to use the release keyword in a job.
- We use the release keyword to create a release in GitLab. This is a special job keyword reserved for creating a release and populating the required fields.
- We can pass a file as an argument to the description of the release. In our case, it's the file we generated in the prepare_job and generate_descriptive_notes job, which was passed to this job as an artifact.

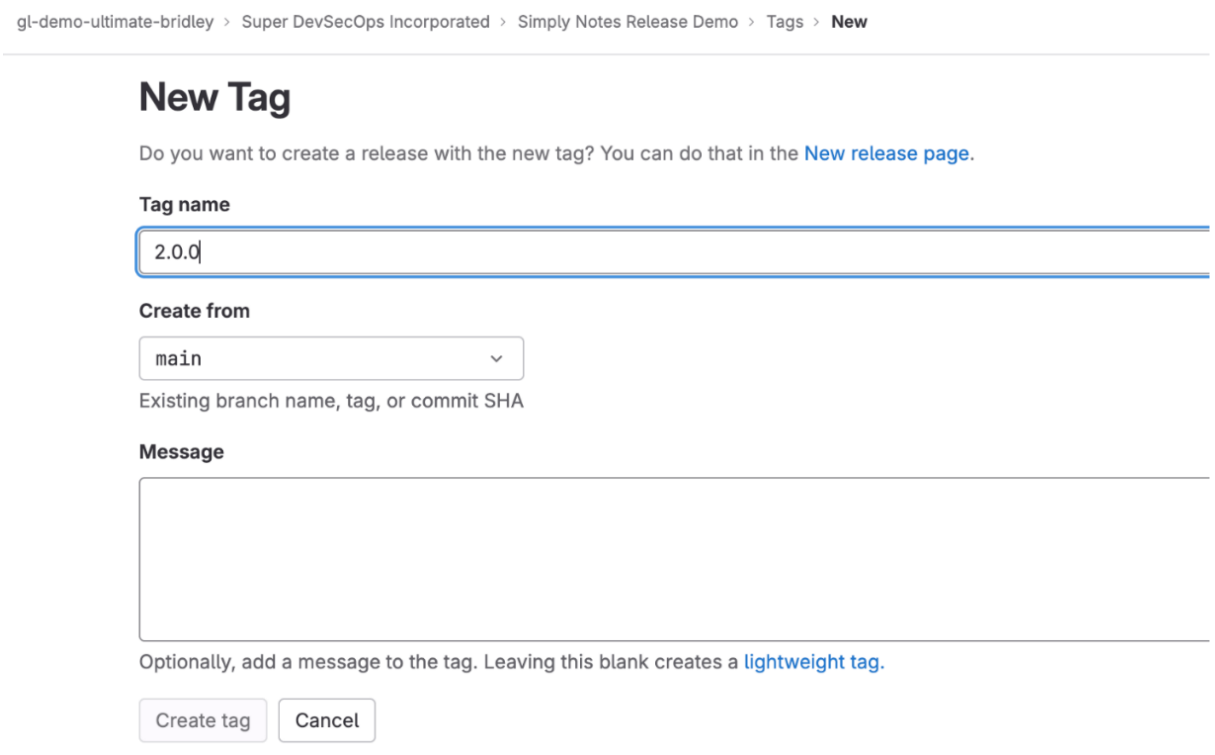
Performing an automated release

Tag creation through pipelines:

With this setup, all we need to do is execute the pipeline.

Manual Tag creation:

To perform a release is push a tag to our repository that follows our versioning scheme. You can simply push a tag using the CLI, this example uses GitLab's UI to create a tag on the main branch. Create a tag by selecting Code -> Tags -> New Tag on the sidebar:


gl-demo-ultimate-bridley › Super DevSecOps Incorporated › Simply Notes Release Demo › Tags › New

New Tag

Do you want to create a release with the new tag? You can do that in the New release page.

Tag name

2.0.0

Create from

main

Existing branch name, tag, or commit SHA

Message

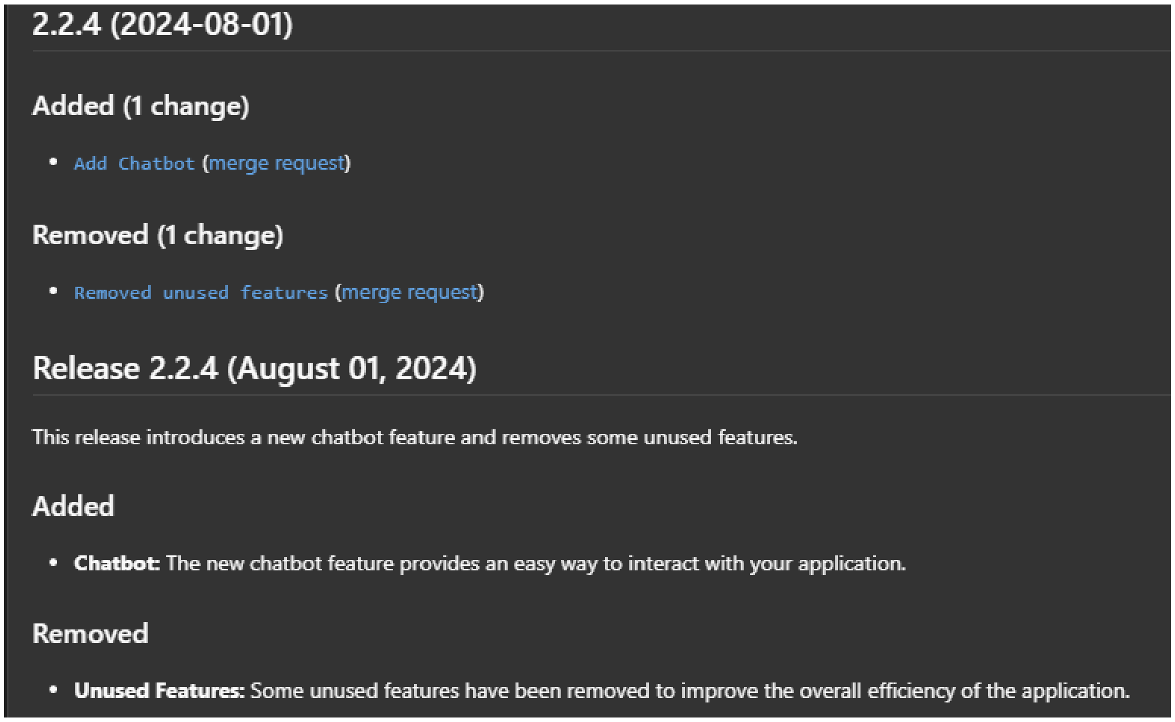
Optionally, add a message to the tag. Leaving this blank creates a lightweight tag.

- On creation, our pipelines will start to execute. The GitLab Changelog API will automatically generate release notes for us as markdown, which contains all the changes between this release and the previous release.

Here's the resulting markdown generated in our example:

```
## 2.0.0 (2023-08-25)

### added (1 change)

- [Add ChatBot](gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-demo@0c3601a45af617c5481322bfce4d71db1f911b02) ([merge request](gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-demo!4))

### removed (1 change)

- [Remove Unused Features](gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-demo@463d453c5ae0f4fc611ea969e5442e3298bf0d8a) ([merge request](gl-demo-ultimate-bridley/super-devsecops-incorporated/simply-notes-release-demo!3))
```

- As you can see, GitLab has extracted the entries for our release notes automatically using our git commit trailers. In addition, it's helpfully provided links back to the merge request so readers can see more details and discussion around the changes.
- And now, our final release:



2.2.4 (2024-08-01)

Added (1 change)

- `Add Chatbot` (merge request)

Removed (1 change)

- `Removed unused features` (merge request)

Release 2.2.4 (August 01, 2024)

This release introduces a new chatbot feature and removes some unused features.

Added

- **Chatbot:** The new chatbot feature provides an easy way to interact with your application.

Removed

- **Unused Features:** Some unused features have been removed to improve the overall efficiency of the application.

Changelogs

Changelogs are generated based on commit titles and Git trailers. To be included in a changelog, a commit must contain a specific Git trailer. Changelogs are generated from commit titles, and categorized by Git trailer type. You can enrich changelog entries with additional data, such as a link to the merge request or details about the commit author. Changelog formats can be customized with a template.

Each section in the default changelog has a title containing the version number and release date, like this:

```
## 1.0.0 (2021-01-05)


### Features (4 changes)


- [Feature 1](gitlab-org/gitlab@123abc) by @alice ([merge request](gitlab-
org/gitlab!123))

- [Feature 2](gitlab-org/gitlab@456abc) ([merge request](gitlab-org/gitlab!456))

- [Feature 3](gitlab-org/gitlab@234abc) by @steve

- [Feature 4](gitlab-org/gitlab@456)
```

The date format for sections can be customized, but the rest of the title cannot. When adding new sections, GitLab parses these titles to determine where to place the new information in the file. GitLab sorts sections according to their versions, not their dates.

Each section contains changes sorted by category (like **Features**), and the format of these sections can be changed. The section names derive from the values of the Git trailer used to include or exclude commits.

Commits for changelogs can be retrieved when operating on a mirror. GitLab itself uses this feature, because patch releases can include changes from both public projects and private security mirrors.

Add a trailer to a Git commit

You can add trailers manually when you write a commit message. To include a commit using the default trailer of `Changelog` and categorize it as a feature, add the string `Changelog: feature` to your commit message, like this:

```
<Commit message subject>

<Commit message description>

Changelog: feature
```

Create a changelog

Changelogs are generated from the command line, using either the API or the GitLab CLI. The changelog output is formatted in Markdown, and you can customize it.

From the API

To use the API to generate changelogs with a `curl` command, see Add changelog data to a changelog file in the API documentation.

From the GitLab CLI

Introduced in `glab` version 1.30.0.

Prerequisites:

- You have installed and configured the GitLab CLI, version 1.30.0 or later.
- Your repository's tag naming schema matches the expected tag naming format.
- Commits include changelog trailers.

To generate the changelog:

1. Update your local copy of your repository with `git fetch`.
2. To generate a changelog for the current version (as determined by `git describe --tags`) with the default options:
    - Run the command `glab changelog generate`.
    - To save the output to a file, run the command `glab changelog generate > <filename>.md`.

3. To generate a changelog with customized options, run the command `glab changelog generate` and append your desired options. Some options include:
    - `--config-file [string]`: The path to the changelog configuration file in your project's Git repository. This file must exist in your project's Git repository. Defaults to `.gitlab/changelog_config.yml`.
    - Commit range:
        - `--from [string]`: The start of the range of commits (as a SHA) to use for generating the changelog. This commit itself isn't included in the changelog.
        - `--to [string]`: The end of the range of commits (as a SHA) to use for generating the changelog. This commit *is* included in the list. Defaults to the `HEAD` of the default project branch.
    - `--date [string]`: The date and time of the release, in ISO 8601 (`2016-03-11T03:45:40Z`) format. Defaults to the current time.
    - `--trailer [string]`: The Git trailer to use for including commits. Defaults to `Changelog`.
    - `--version [string]`: The version to generate the changelog for.

To learn more about the parameters available in GitLab CLI, run `glab changelog generate --help`. See the API documentation for definitions and usage.

Customize the changelog output

To customize the changelog output, edit the changelog configuration file, and commit these changes to your project's Git repository. The default location for this configuration is `.gitlab/changelog_config.yml`. The file supports these variables:

- `date_format`: The date format, in `strftime` format, used in the title of the newly added changelog data.
- `template`: A custom template to use when generating the changelog data.
- `include_groups`: A list of group full paths containing users whose contributions should be credited regardless of project membership. The user generating the changelog must have access to each group for credit to be given.
- `categories`: A hash that maps raw category names to the names to use in the changelog. To alter the names displayed in the changelog, add these lines to your configuration file and edit them to meet your needs. This example renders the category titles as `### Features`, `### Bug fixes`, and `### Performance improvements`:

```
- ---

- categories:

-   feature: Features

-   bug: Bug fixes

-   performance: Performance improvements
```

Custom templates

History

Category sections are generated using a template. The default template:

```
{% if categories %}

{% each categories %}

### {{ title }} ({% if single_change %}1 change{% else %}{{ count }} changes{% end
%})

{% each entries %}

- [{{ title }}]({{ commit.web_url }})\

{% if author.credit %} by {{ author.reference }}{% end %}\

{% if merge_request %} ([merge request]({{ merge_request.web_url }})){% end %}

{% end %}

{% end %}

{% else %}

No changes.

{% end %}
```

The `{% ... %}` tags are for statements, and `{{ ... }}` is used for printing data. Statements must be terminated using a `{% end %}` tag. Both the `if` and `each` statements require a single argument.

For example, for a variable called `valid`, you can display “yes” when this value is true, and display “nope” otherwise by doing the following:

```
{% if valid %}

yes

{% else %}

nope

{% end %}
```

The use of `else` is optional. A value is considered true when it’s a non-empty value or boolean `true`. Empty arrays and hashes are considered false.

Looping is done using `each`, and variables inside a loop are scoped to it. Referring to the current value in a loop is done using the variable tag `{{ it }}`. Other variables read their value from the current loop value. Take this template for example:

```
{% each users %}

{{name}}

{% end %}
```

Assuming `users` is an array of objects, each with a `name` field, this would then print the name of every user.

Using variable tags, you can access nested objects. For example, `{{ users.0.name }}` prints the name of the first user in the `users` variable.

If a line ends in a backslash, the next newline is ignored. This allows you to wrap code across multiple lines, without introducing unnecessary newlines in the Markdown output.

Tags that use `{%` and `%}` (known as expression tags) consume the newline that directly follows them, if any. This means that this:

```
---

{% if foo %}

bar

```

```
{% end %}

---
```

Compiles into this:

```
---

bar

---
```

Instead of this:

```
---

bar

---
```

You can specify a custom template in your configuration, like this:

```
---

template: |

  {% if categories %}

  {% each categories %}

  ### {{ title }}

  {% each entries %}

  - [{{ title }}]({{ commit.web_url }})\

  {% if author.credit %} by {{ author.reference }}{% end %}

  {% end %}

  {% end %}

  {% else %}

  No changes.
  {% end %}
```

When specifying the template you should use `template: |` and not `template: >`, as the latter doesn't preserve newlines in the template.

Template data

History

At the top level, the following variable is available:

- `categories`: an array of objects, one for every changelog category.

In a category, the following variables are available:

- `count`: the number of entries in this category.
- `entries`: the entries that belong to this category.
- `single_change`: a boolean that indicates if there is only one change (`true`), or multiple changes (`false`).
- `title`: the title of the category (after it has been remapped).

In an entry, the following variables are available (here `foo.bar` means that `bar` is a sub-field of `foo`):

- `author.contributor`: a boolean set to `true` when the author is not a project member, otherwise `false`.
- `author.credit`: a boolean set to `true` when `author.contributor` is `true` or when `include_groups` is configured, and the author is a member of one of the groups.
- `author.reference`: a reference to the commit author (for example, `@alice`).
- `commit.reference`: a reference to the commit, for example, `gitlab-org/gitlab@0a4cdd86ab31748ba6dac0f69a8653f206e5cfc7`.
- `commit.web_url`: a URL to the commit, for example, `https://gitlab.com/gitlab-org/gitlab/-/commit/0a4cdd86ab31748ba6dac0f69a8653f206e5cfc7`.
- `commit.trailers`: an object containing all the Git trailers that were present in the commit body.

  These trailers can be referenced using `commit.trailers.<name>`. For example, assuming the following commit:

  ```
  Add some impressive new feature


  Changelog: added
  ```

```
Issue: https://gitlab.com/gitlab-org/gitlab/-/issues/1234

Status: important
```

The `Changelog`, `Issue` and `Status` trailers can be accessed in the template as follows:

```
{% each entries %}

{% if commit.trailers.Issue %} ([link to issue]({{ commit.trailers.Issue
}})){% end %}

{% if commit.trailers.Status %}Status: {{ commit.trailers.Status }}{% end %}

{% end %}
```

- `merge_request.reference`: a reference to the merge request that first introduced the change (for example, `gitlab-org/gitlab!50063`).
- `merge_request.web_url`: a URL to the merge request that first introduced the change (for example, `https://gitlab.com/gitlab-org/gitlab/-/merge_requests/50063`).
- `title`: the title of the changelog entry (this is the commit title).

The `author` and `merge_request` objects might not be present if the data couldn't be determined. For example, when a commit is created without a corresponding merge request, no merge request is displayed.

Customize the tag format when extracting versions

GitLab uses a regular expression (using the re2 engine and syntax) to extract a semantic version from tag names. The default regular expression is:

```
^v?(?P<major>0|[1-9]\d*)\.(?P<minor>0|[1-9]\d*)\.(?P<patch>0|[1-9]\d*)(?:-
(?P<pre>(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-
zA-Z-]*))*))?(?:\+(?P<meta>[0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$
```

This regular expression is based on the official semantic versioning regular expression, and also includes support for tag names that start with the letter `v`.

If your project uses a different format for tags, you can specify a different regular expression. The regular expression used *must* produce the following capture groups. If any of these capture groups are missing, the tag is ignored:

- `major`
- `minor`

- `patch`

The following capture groups are optional:

- `pre`: If set, the tag is ignored. Ignoring `pre` tags ensures release candidate tags and other pre-release tags are not considered when determining the range of commits to generate a changelog for.
- `meta`: Optional. Specifies build metadata.

Using this information, GitLab builds a map of Git tags and their release versions. It then determines what the latest tag is, based on the version extracted from each tag.

To specify a custom regular expression, use the `tag_regex` setting in your changelog configuration YAML file. For example, this pattern matches tag names such as `version-1.2.3` but not `version-1.2`.

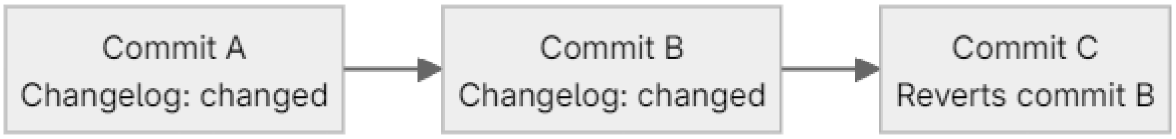
```yaml
---

tag_regex: '^version-(?P<major>\d+)\.(?P<minor>\d+)\.(?P<patch>\d+)$'
```

To test if your regular expression is working, you can use websites such as regex101. If the regular expression syntax is invalid, an error is produced when generating a changelog.

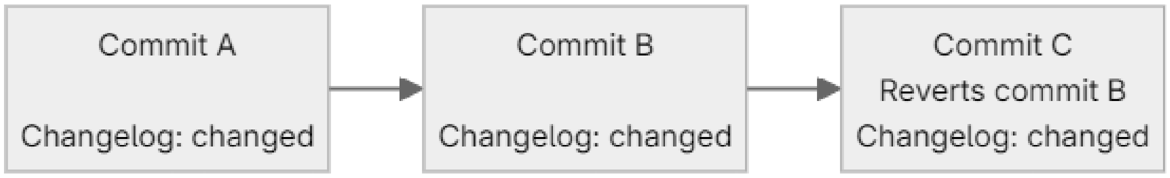
Reverted commit handling

To be treated as a revert commit, the commit message must contain the string `This reverts commit <SHA>`, where `SHA` is the SHA of the commit to be reverted.

When generating a changelog for a range, GitLab ignores commits both added and reverted in that range. In this example, commit C reverts commit B. Because commit C has no other trailer, only commit A is added to the changelog:



However, if the revert commit (commit C) *also* contains a changelog trailer, both commits A and C are included in the changelog:



Commit B is skipped.

Changelog entries

This section contains instructions for when and how to generate a changelog entry file, as well as information and history about our changelog process.

Overview

Each list item, or **entry**, in our `CHANGELOG.md` file is generated from the subject line of a Git commit. Commits are included when they contain the `Changelog` Git trailer. When generating the changelog, author and merge request details are added automatically.

The `Changelog` trailer accepts the following values:

- `added`: New feature
- `fixed`: Bug fix
- `changed`: Feature change
- `deprecated`: New deprecation
- `removed`: Feature removal
- `security`: Security fix
- `performance`: Performance improvement
- `other`: Other

An example of a Git commit to include in the changelog is the following:

```
Update git vendor to gitlab

Now that we are using gitaly to compile git, the git version isn't known

from the manifest, instead we are getting the gitaly version. Update our

vendor field to be `gitlab` to avoid cve matching old versions.

Changelog: changed
```

If your merge request has multiple commits, make sure to add the `Changelog` entry to the first commit. This ensures that the correct entry is generated when commits are squashed.

Overriding the associated merge request

GitLab automatically links the merge request to the commit when generating the changelog. If you want to override the merge request to link to, you can specify an alternative merge request using the `MR` trailer:

```
Update git vendor to gitlab

Now that we are using gitaly to compile git, the git version isn't known

from the manifest, instead we are getting the gitaly version. Update our

vendor field to be `gitlab` to avoid cve matching old versions.

Changelog: changed

MR: https://gitlab.com/foo/bar/-/merge_requests/123
```

The value must be the full URL of the merge request.

GitLab Enterprise changes

If a change is exclusively for GitLab Enterprise Edition, **you must add** the trailer `EE: true`:

```
Update git vendor to gitlab

Now that we are using gitaly to compile git, the git version isn't known

from the manifest, instead we are getting the gitaly version. Update our

vendor field to be `gitlab` to avoid cve matching old versions.

Changelog: changed

MR: https://gitlab.com/foo/bar/-/merge_requests/123

EE: true
```

Do not add the trailer for changes that apply to both EE and CE.

What warrants a changelog entry?

- Any change that introduces a database migration, whether it’s regular, post, or data migration, **must** have a changelog entry, even if it is behind a disabled feature flag.
- Security fixes **must** have a changelog entry, with `Changelog` trailer set to `security`.
- Any user-facing change **must** have a changelog entry. Example: “GitLab now uses system fonts for all text.”
- Any client-facing change to our REST and GraphQL APIs **must** have a changelog entry. See the complete list what comprises a GraphQL breaking change.
- Any change that introduces an advanced search migration **must** have a changelog entry.
- A fix for a regression introduced and then fixed in the same release (such as fixing a bug introduced during a monthly release candidate) **should not** have a changelog entry.
- Any developer-facing change (such as refactoring, technical debt remediation, or test suite changes) **should not** have a changelog entry. Example: “Reduce database records created during Cycle Analytics model spec.”
- *Any* contribution from a community member, no matter how small, **may** have a changelog entry regardless of these guidelines if the contributor wants one.
- Any experiment changes **should not** have a changelog entry.
- An MR that includes only documentation changes **should not** have a changelog entry.

For more information, see how to handle changelog entries with feature flags.

Writing good changelog entries

A good changelog entry should be descriptive and concise. It should explain the change to a reader who has *zero context* about the change. If you have trouble making it both concise and descriptive, err on the side of descriptive.

- **Bad:** Go to a project order.
- **Good:** Show a user’s starred projects at the top of the “Go to project” dropdown list.

The first example provides no context of where the change was made, or why, or how it benefits the user.

- **Bad:** Copy (some text) to clipboard.

- **Good:** Update the “Copy to clipboard” tooltip to indicate what’s being copied.

Again, the first example is too vague and provides no context.

- **Bad:** Fixes and Improves CSS and HTML problems in mini pipeline graph and builds dropdown list.
- **Good:** Fix tooltips and hover states in mini pipeline graph and builds dropdown list.

The first example is too focused on implementation details. The user doesn’t care that we changed CSS and HTML, they care about the *end result* of those changes.

- **Bad:** Strip out `nils` in the Array of Commit objects returned from `find_commits_by_message_with_elastic`
- **Good:** Fix 500 errors caused by Elasticsearch results referencing garbage-collected commits

The first example focuses on *how* we fixed something, not on *what* it fixes. The rewritten version clearly describes the *end benefit* to the user (fewer 500 errors), and *when* (searching commits with Elasticsearch).

Use your best judgement and try to put yourself in the mindset of someone reading the compiled changelog. Does this entry add value? Does it offer context about *where* and *why* the change was made?

How to generate a changelog entry

Git trailers are added when committing your changes. This can be done using your text editor of choice. Adding the trailer to an existing commit requires either amending to the commit (if it’s the most recent one), or an interactive rebase using `git rebase -i`.

To update the last commit, run the following:

```
git commit --amend
```

You can then add the `Changelog` trailer to the commit message. If you had already pushed prior commits to your remote branch, you have to force push the new commit:

```
git push -f origin your-branch-name
```

To edit older (or multiple commits), use `git rebase -i HEAD~N` where `N` is the last N number of commits to rebase. Let’s say you have 3 commits on your branch: A, B, and C. If you want to update commit B, you need to run:

```
git rebase -i HEAD~2
```

This starts an interactive rebase session for the last two commits. When started, Git presents you with a text editor with contents along the lines of the following:

```
pick B Subject of commit B

pick C Subject of commit C
```

To update commit B, change the word `pick` to `reword`, then save and quit the editor. Once closed, Git presents you with a new text editor instance to edit the commit message of commit B. Add the trailer, then save and quit the editor. If all went well, commit B is now updated.

Since you changed commits that already exist in your remote branch, you must use the `--force-with-lease` flag when pushing to your remote branch:

```
git push origin your-branch-name --force-with-lease
```

Add changelog data to a changelog file

History

Generate changelog data based on commits in a repository.

Given a semantic version and a range of commits, GitLab generates a changelog for all commits that use a particular Git trailer. GitLab adds a new Markdown-formatted section to a changelog file in the Git repository of the project. The output format can be customized.

For user-facing documentation, see Changelogs.

```
POST /projects/:id/repository/changelog
```

Supported attributes

Changelogs support these attributes:

Attribute	Type	Required	Description
`version`	string	yes	The version to generate the changelog for. The format must follow semantic versioning.
`branch`	string	no	The branch to commit the changelog changes to. Defaults to the project's default branch.

`config_file`	string	no	Path to the changelog configuration file in the project’s Git repository. Defaults to `.gitlab/changelog_config.yml`.
`date`	datetime	no	The date and time of the release. Defaults to the current time.
`file`	string	no	The file to commit the changes to. Defaults to `CHANGELOG.md`.
`from`	string	no	The SHA of the commit that marks the beginning of the range of commits to include in the changelog. This commit isn’t included in the changelog.
`message`	string	no	The commit message to use when committing the changes. Defaults to `Add changelog for version X`, where `X` is the value of the `version` argument.
`to`	string	no	The SHA of the commit that marks the end of the range of commits to include in the changelog. This commit *is* included in the changelog. Defaults to the branch specified in the `branch` attribute. Limited to 15000 commits unless the feature flag `changelog_commits_limitation` is disabled.
`trailer`	string	no	The Git trailer to use for including commits. Defaults to `Changelog`. Case-sensitive: `Example` does not match `example` or `eXaMpLE`.

Requirements for `from` attribute

If the `from` attribute is unspecified, GitLab uses the Git tag of the last stable version that came before the version specified in the `version` attribute. For GitLab to extract version numbers from tag names, Git tag names must follow a specific format. By default, GitLab considers tags using these formats:

- `vX.Y.Z`
- `X.Y.Z`

Where `X.Y.Z` is a version that follows semantic versioning. For example, consider a project with the following tags:

- `v1.0.0-pre1`
- `v1.0.0`

- `v1.1.0`
- `v2.0.0`

If the `version` attribute is `2.1.0`, GitLab uses tag `v2.0.0`. And when the version is `1.1.1`, or `1.2.0`, GitLab uses tag `v1.1.0`. The tag `v1.0.0-pre1` is never used, because pre-release tags are ignored.

The `version` attribute can start with `v`. For example: `v1.0.0`. The response is the same as for `version` value `1.0.0`. Introduced in GitLab 17.0.

If `from` is unspecified and no tag to use is found, the API produces an error. To solve such an error, you must explicitly specify a value for the `from` attribute.

Migrating from a manually-managed changelog file to Git trailers

When you migrate from an existing manually-managed changelog file to one that uses Git trailers, make sure that the changelog file matches the expected format. Otherwise, new changelog entries added by the API might be inserted in an unexpected position. For example, if the version values in the manually-managed changelog file are specified as `vX.Y.Z` instead of `X.Y.Z`, then new changelog entries added using Git trailers are appended to the end of the changelog file. Issue 444183 proposes customizing the version header format in changelog files. However, until that issue has been completed, the expected version header format in changelog files is `X.Y.Z`.

Examples

These examples use cURL to perform HTTP requests. The example commands use these values:

- **Project ID**: 42
- **Location**: hosted on GitLab.com
- **Example API token**: `token`

This command generates a changelog for version `1.0.0`.

The commit range:

- Starts with the tag of the last release.
- Ends with the last commit on the target branch. The default target branch is the project's default branch.

If the last tag is `v0.9.0` and the default branch is `main`, the range of commits included in this example is `v0.9.0..main`:

```
curl --request POST --header "PRIVATE-TOKEN: token" \

  --data "version=1.0.0" \

  --url "https://gitlab.com/api/v4/projects/42/repository/changelog"
```

To generate the data on a different branch, specify the `branch` parameter. This command generates data from the `foo` branch:

```
curl --request POST --header "PRIVATE-TOKEN: token" \

  --data "version=1.0.0&branch=foo" \

  --url "https://gitlab.com/api/v4/projects/42/repository/changelog"
```

To use a different trailer, use the `trailer` parameter:

```
curl --request POST --header "PRIVATE-TOKEN: token" \

  --data "version=1.0.0&trailer=Type" \

  --url "https://gitlab.com/api/v4/projects/42/repository/changelog"
```

To store the results in a different file, use the `file` parameter:

```
curl --request POST --header "PRIVATE-TOKEN: token" \

  --data "version=1.0.0&file=NEWS" \

  --url "https://gitlab.com/api/v4/projects/42/repository/changelog"
```

Generate changelog data

Generate changelog data based on commits in a repository, without committing them to a changelog file.

Works exactly like `POST /projects/:id/repository/changelog`, except the changelog data isn't committed to any changelog file.

```
GET /projects/:id/repository/changelog
```

Supported attributes:

Attribute	Type	Required	Description
`version`	string	yes	The version to generate the changelog for. The format must follow semantic versioning.
`config_file`	string	no	The path of changelog configuration file in the project's Git repository. Defaults to `.gitlab/changelog_config.yml`.
`date`	datetime	no	The date and time of the release. Uses ISO 8601 format. Example: `2016-03-11T03:45:40Z`. Defaults to the current time.
`from`	string	no	The start of the range of commits (as a SHA) to use for generating the changelog. This commit itself isn't included in the list.
`to`	string	no	The end of the range of commits (as a SHA) to use for the changelog. This commit *is* included in the list. Defaults to the HEAD of the default project branch.
`trailer`	string	no	The Git trailer to use for including commits. Defaults to `Changelog`.

```
curl --header "PRIVATE-TOKEN: token" \

  --url "https://gitlab.com/api/v4/projects/42/repository/changelog?version=1.0.0"
```

Example response, with line breaks added for readability:

```
{

  "notes": "## 1.0.0 (2021-11-17)\n\n### feature (2 changes)\n\n-

    [Title 2](namespace13/project13@ad608eb642124f5b3944ac0ac772fecaf570a6bf)

    ([merge request](namespace13/project13!2))\n-

    [Title 1](namespace13/project13@3c6b80ff7034fa0d585314e1571cc780596ce3c8)

    ([merge request](namespace13/project13!1))\n"

}
```