Pretty CODESYS Git Diffs

Roy Ratcliffe[a]

[a]https://github.com/royratcliffe

Published in *Roy's Code Chronicles*, 2026

Git is great for tracking changes in text files, but CODESYS project objects are stored as single-line JSON, which makes even small edits look like one huge blob on a single line. That makes history harder to read and review outside CODESYS' own tools. This article shows a simple workaround: a Python pre-commit hook that prettifies staged `.object` files before they are committed. It only touches staged files, skips unstaged work, and avoids partially staged content so the repository stays safe and predictable. The result is a Git history that behaves like normal text-based diffs: you can see what changed, review it in a familiar way, and keep a clearer trail of how the project evolved. It is a small fix, but it makes a big difference if you want to work with CODESYS projects in a Git workflow without losing readability.

CODESYS | Git | Python

Introduction

Lately, I've been climbing the CODESYS learning curve.

CODESYS version 3.5 lets you store projects, including library projects, in a Git repository.

It encodes projects in a proprietary binary-coded format by default, just one big binary blob for the entire project, which is not Git-friendly. Git prefers to see lines of text that can become subject to difference comparison as changes arise.

If you attach a Git repository to a project, CODESYS caches a project in the repository in the form of JSON object files.

One Long Line. One small niggle does exist, however. The project objects[1] are JSON, but **not** pretty JSON. Each committed file is a JSON dictionary object on one long line—not a single unescaped newline anywhere. Call it ugly JSON. This is not a problem per se for CODESYS. Its built-in comparison tool unpacks and renders differences easily enough. Git's difference granularity cannot expand into the content of the line, however. From one version to the next, Git sees one line before and one line after. Not very helpful outside the built-in tools!

Manually formatting is possible. VS Code offers a "Format Document" command for JSON content, provided that you identify the "object" extension as JSON. Still, not ideal.

Pre-Commit Hook

The problem? Prettify staged objects automatically. Git lets you run pre-commit hooks. This would be a straightforward solution. JSON is a space-agnostic format. Provided that CODESYS does not validate spacing when loading from a pretty repository[2], then pretty JSON is as good as ugly, compressed JSON.

Add the following python script to a `.githooks` folder within your repository clone. Download it here. It prettifies all *staged* object files, skipping un-staged files.

```
#!/usr/bin/env python3
"""Pretty-print staged .object JSON files (CODESYS project objects) before commit."""

import json
import subprocess
import sys


def staged_object_files():
    """Return a list of staged .object files."""
    # ACM stands for added-copied-modified; ignore deleted files which
    # no longer exist on disk.
    out = subprocess.run(
        ["git", "diff", "--cached", "--name-only", "--diff-filter=ACM"],
        capture_output=True,
        text=True,
        check=True,
    ).stdout
    return [f for f in out.splitlines() if f.endswith(".object")]


def unstaged_files():
    """Return the set of files with unstaged working-tree changes."""
    out = subprocess.run(
```

[1] Files with `.object` extension but JSON content

[2] it does not

```
        ["git", "diff", "--name-only"],
        capture_output=True,
        text=True,
        check=True,
    ).stdout
    return set(out.splitlines())


def prettify(path):
    """Prettify a single .object JSON file."""
    print(f"pre-commit: prettifying {path}")
    # Read the file content first. Encoding "utf-8-sig" strips a BOM if
    # present, so re-saved files stay as plain UTF-8.
    with open(path, "r", encoding="utf-8-sig") as f:
        content = f.read()
    # Attempt to parse the content as JSON.
    try:
        data = json.loads(content)
    except json.JSONDecodeError:
        # .object files aren't guaranteed to be JSON; leave non-JSON
        # ones untouched.
        print(f"pre-commit: skipping {path} (not valid JSON)", file=sys.stderr)
        return False
    # Compare the pretty-printed content with the original content. If
    # they differ, overwrite the file with the pretty-printed version.
    # The default dict insertion order with sort_keys=False (default)
    # preserves the original key order; indent=2 only controls
    # formatting, while ensure_ascii=False keeps non-ASCII text
    # readable.
    pretty = json.dumps(data, indent=2, ensure_ascii=False) + "\n"
    if pretty == content:
        print(f"pre-commit: {path} is already pretty")
        return False
    with open(path, "w", encoding="utf-8", newline="\n") as f:
        f.write(pretty)
    return True


def main():
    """Prettify all staged .object JSON files."""
    files = staged_object_files()
    # Reading and writing the working tree would pull in unstaged edits
    # on partially staged files, silently committing more than the user
    # staged; bail out instead.
    partially_staged = sorted(set(files) & unstaged_files())
    if partially_staged:
        print(
            "pre-commit: aborting, these staged .object files also have unstaged "
            "changes (stage fully or stash unstaged changes first):",
            file=sys.stderr,
        )
        for path in partially_staged:
            print(f"  {path}", file=sys.stderr)
        sys.exit(1)
    changed = [path for path in files if prettify(path)]
    if changed:
        # Re-stage the reformatted files so the commit captures the pretty version.
        subprocess.run(["git", "add", *changed], check=True)
        print(f"pre-commit: prettified {len(changed)} .object file(s)")


if __name__ == "__main__":
    main()
```

Note:

1. The script does **not** prettify unstaged object files: they represent work in progress.

2. Neither does it prettify partially-staged objects, as that would be a tricky operation. CODESYS Git tools do not currently support partial commits, but once pretty, partial hunk commits become possible using other tools.

Run it manually using, assuming the script is located in the `.githooks` directory:

```
python ./.githooks/prettify_json.py
```

Installing the hook. The script needs a shell launcher and a Git configuration setting to set up your CODESYS Git repository.

```
#!/bin/sh
# Prettify staged .object JSON files (CODESYS project objects) before commit.
# Not run automatically: enable with `git config core.hooksPath .githooks`.
# Must use LF line endings: git-bash's sh fails to parse CRLF scripts.
hook_dir="$(dirname "$0")"
# Try python3/python/py in order since availability varies by platform.
PYTHON="$(command -v python3 || command -v python || command -v py)"
if [ -z "$PYTHON" ]; then
    echo "pre-commit: python not found, skipping .object prettification" >&2
    exit 0
fi
"$PYTHON" "$hook_dir/prettify_json.py"
```

Run the following command line from your repository to set up the hook. It makes an assumption about where the Python script and its associated shell script appear: in `.githooks` in my case. Adjust accordingly. The hooks need to live outside the `.git` folder in order to commit the hook.

```
git config core.hooksPath .githooks
```

Conclusions

Perhaps one day CODESYS will automatically output pretty JSON.

Until then, a pre-commit hook will improve the Git experience. It exposes the contents and changes to the Git commit pipeline in a clear, pretty, line-by-line fashion. This has advantages for storage, as well as visualisation. Git stores changes as patches of difference hunks. When lines do not change, they will not appear in the patch set. Committing a change on a single line replaces the old line with the new line. Git does **not** parse out the sub-line character changes. Pretty JSON unfolds the object contents as lines, subject to text-style patching with lines added, lines removed and context lines.

Small changes thereby become clear in the repository outside the CODESYS tools. You can readily see changes in other tools also, e.g. VS Code. It also gives the developer a peek under the bonnet. Changes in the GUI reflect in the objects.