Incident Retrospectives as Code

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

I live in Belmont, in the San Francisco Bay Area Peninsula. I wish to acknowledge it as the ancestral homeland of the Ramaytush Ohlone people.

0.1 Background on Retrospectives

Incidents: An Introduction

Align on terms

Let's make sure we are all talking about the same things. Here are the terms I will be using, and my definitions for them.

The industry is young enough that you might be using different terms and different definitions. This is fine! But I want to make sure we are all on the same page.

0.1.1 Incident

Incident

Something bad...

...we didn't want...

...and we would like to prevent.

An incident has to have three things:

- It has to be bad: someone was harmed.
- It has to be *unplanned*: we did not plan to do this harm. This is not trivial. Sometimes trade-offs must be made, and someone needs to be harmed. But that is *not* an incident.
- It has to be something that we are willing to work to prevent. We have to *care* about the harm. We might care about it because remediating is annoying. We might care about it because it led to lost revenue. But if we don't care, it's not an incident.

0.1.2 Retrospective

Retrospective

(AKA "post-mortem")

Analysis...

...and an improvement plan.

A retrospective is sometimes called "post-mortem". Since nobody actually died (I hope!) and since talking about death can be triggering to some people, I prefer the more neutral term "retrospective".

An *incident* is something bad we want to prevent from recurring. The retrospective assumes both parts:

- It analyses the chain of event that led to the harm.

- It offers concrete recommendations to avoid it in the future.

0.1.3 Retrospective Review

Retrospective Review

Part of "signing off" on a retrospective

Feedback

Feedback addressed

Sync/Async

Both the analysis and recommendation parts of a retrospective can be nuanced. Unwarranted assumptions might have been made, or relevant information missed.

A retrospective *review* is the part, or parts, where the team offers feedback on the retrospective.

The feedback might be synchronous, in a meeting, or asynchronous, as written comments. It is often a combination of both.

The feedback is addressed, partially, by modifying the retrospective. At some point, usually, retrospectives are "signed off": the team reaches a consensus about the retrospective being "good enough".

0.1.4 Why do Retrospectives?

Why Retrospectives?

"Make new mistakes"...

...for sufficiently large values of "new".

Retrospectives are a lot of work. Why spend all the time and effort?

The goal of a retrospective is to make sure any existing issues are dealt with. The definition of "existing" is designed to be far reaching.

The goal of a *good* retrospective is to find all the issues that led to the incident. The more we identify and deal with, the better we can mitigate and avoid future incidents.

0.1.5 Why do Retrospective Reviews?

Why Retrospective Reviews?

Verify analysis...

...verify recommendations...

...teach.

Properly analyzing incidents is complicated. There are many things that are easy to miss and many ways to misidentify issues.

Getting feedback from the team allows making the retrospectives more valuable. It also serves as a good opportunity to share knowledge.

0.2 Source Control for Retrospectives

Source Control for Retrospectives

Step 0 of "as code"

Code is kept in source control in every reasonably mature team. The minimum for treating retrospectives "as code" is to do the same.

0.2.1 Why not wiki?

Why not Wiki?

Edit with usual editor!

Local backup

There are alternatives. Some teams keep the retrospectives in a wiki.

Wikis aren't bad! But there are advantages to *not* editing via the web.

A modern editor is an engineer's friend. They spend most day in it – be it VSCode, Sublime, neovim, or anything else.

Keeping retrospectives in source control, as text files, allows taking advantage of that. Each team member can use their favorite editor, and the files are kept locally even if the editor crashes.

0.2.2 Why not shared docs?

Why not Shared Docs?

E.g., GDoc, Dropbox Paper, MS Sharepoint...

Non-proprietary format

Common tooling

There are many shared doc platforms. Google Docs, Dropbox Paper, MS Sharepoint are all popular.

They are all good for a lot of use cases! They are not, however, optimized for retrospectives.

In some of them, even embedding a code-like environment is complicated. For a thing where mentioning specific Kubernetes Deployments a DNS name, or a specific UNIX command is common, this can be useful.

0.2.3 Format

Format

Your favorite "lightweight markup" language

Default: Markdown

Other alternatives: ReStructured Text, Asciidoc

Choose and commit

We do need some sort of markup for the retrospectives. Luckily, there are a plethora of formats that are designed for relatively light-weight markup.

Markdown is a good default being supported in quite a few platforms. There are sometimes reasons to use other formats, like ReStructuredText or ASCIIdoc.

0.2.4 Template

Template

Common ToC

Clarify what belongs

Note any guidelines

Note mandatory/optional

NOT a replacement for process docs

Templates are useful regardless of format. They are even more essential for retrospectives as code, where a template serves as a reminder for the expected formatting.

Being text, sometimes there's more we can do. For example, writing a script to pre-populate some fields or help out sketch the timeline.

0.2.5 Organization

Organization

Directory structure (flat/hierarchical)

Directory structure (images)

Naming

Beyond a single retrospectives, keeping them in a code repository means deciding on a structure. How will files be named? Where will they be kept?

Do we want one directory for the assets? Or assets directory per retrospective?

0.3 Collaboration for Retrospectives

Collaboration for Retrospectives

Step 1 of "as code"

...because you don't push straight to main in your other repositories

Retrospectives are written collaboratively in the same sense that code is written collaboratively. The model here is not, usually, "pair programming".

The more common model of collaborative code writing falls into:

- Person task is assigned to writes a patch.
- Patch is reviewed for achievement of goal and for complying with repository guidelines.
- Edits are made.
- Repeat until approved.

This model is not that different from how retrospectives work!

0.3.1 Some More Definitions

Definitions

(Sorry)

Source Control Management (SCM): Manage source code history and diffs

Examples: git, Mercurial

Source Collaboration System: Manage source code collaboration

Examples: GitHub, GitLab, Review Board

Draft Patch: A suggested code change that can be commented on and approved

Examples:

"Pull Request" (GitHub)

"Merge Request" (GitLab, BitBucket)

"Review Request" (Review Board)

In order to talk about collaboration, I need to make sure we all agree on some more definitions. Sorry!

A *source control management* system is what you interact with locally to manage source code. Git, Mercurial, and more are all options.

A *source collaboration system* is a server platform which manages source code *collaboration*. It can include an SCM system or not, but this is not relevant to its role here. Examples include GitHub, GitLab, and Review Board.

The essential *unit* of a collaboration system is the "draft patch". Though the collaboration is similar, each system names it differently:

- GitHub calls it "Pull Request" for historical reasons, mostly having to do with how the Linux kernel is developed.
- GitLab and BitBucket call it "Merge Request", because after it is approved, the developer pushes the "Merge" button to add this patch to the main source code branch.
- Review Board calls it "Review Request" because that is the focus of the system. After it is approved, merging is done by a "bot" inside ReviewBoard.

0.3.2 Draft Patch

Draft Patch

"Asking for feedback"

The usual collaboration process starts with a "draft patch". This is the first point where others in the team are asked for feedback.

In retrospectives, this is the point where the original author is satisfied, or reasonably satisfied, that the retrospective is "done". It is possible to ask for feedback earlier, of course, but it needs to be clear if this is "early feedback" or "I see no further issues".

0.3.3 Draft Patch Feedback

Draft Patch Feedback

Line-by-line comments
Overall comments
Ask for changes
Feedback includes:

- A comment on a specific line
- Comments on overall structure or choices
- Specific requets for changes to be made

This is no different in a retrospective. A team member might indicate a specific item in the timeline is wrong or missing details, or they might ask how the recomendations pertain to the issues identified.

As in all comments on a draft patch, it is important to be clear whether this is a "blocker" or whether this is a non-mandatory suggested improvement.

0.3.4 Adding Commits

Addressing Draft Patch Feedback

Push new commits
Reply to comments

Feedback that never changes the patch is no feedback. A draft patch must be allowed to have more commits added, addressing various aspects of the feedback.

Some comments will be wrong or unclear. In those cases, the original answer needs to answer them.

This is the same in retrospectives. The original author is responsible for addressing comments, either by making changes or by engaging in a discussion.

0.3.5 Getting Approval

Getting Approval

Who approves?
"Discussion has resolved on everything"

A collaboration platform needs to specify who is allowed to "approve". The semantics of approval is "there are no more unresolved discussions".

In retrospectives, ideally, this is a "mechanical check". If all discussions are marked somehow as "resolved", then the retrospective can be merged.

0.3.6 Merging

Merging

Merge patch
Close ticket

After approval, a draft patch is merged and the corresponding ticket is closed. This should be the case in retrospectives as well.

This is a good point for declaring the retrospective "complete". Any further changes would need a new draft patch, and a fresh approval.

0.4 Continuous Integration for Retrospectives

Continuous Integration for Retrospectives

...just like for your other code repositories, right?

You have CI for your other repos, right? Right? Respect your retrospectives to have the same!

0.4.1 Wait, What?

CI for Retrospectives: Wait What?

Never send a human to do a machine's job

Wait, what? Retrospectives are mostly prose, not *really* code.

It turns out that at least some things can be done by computers. If it *can* be automated, it *should* be automated.

0.4.2 Linting

CI for Retrospectives: Lint

Automatic checking of guidelines

Linting is one thing which can be useful. Does the timeline move forward? Does it include big gaps? Do all recommendations include a ticket link?

0.4.3 Rendering

CI for Retrospectives: Rendering

Easier to read

Easier to search

Another useful thing to do is to do rendering. This makes it easier for reviewers to *read* the retrospective, when they see the fully-formatted version.

0.4.4 Example Lint

Example Lint

```
def get_timestamps(input_text):
    ...

def check_order(timestamps):
    timestamps = iter(timestamps)
    last = next(timestamps)
    for current in timestamps:
        if current <= last:
```

```
            raise ValueError(
                "got decreasing",
                last,
                current,
            )
        last = current
```

Here is a simple example lint: it grabs the timestamps section, and checks that the timeline is in order. This is tedious, and better left to a computer to verify.

Note that the code here is optimized to fit on a slide. A real linter would also include better error messages if a line is malformatted, and would also find *all* issues, not just the first one.

Example Lint Failure

```
try:
    check_order(get_timestamps(input))
except Exception as exc:
    for arg in exc.args:
        print(arg)

got decreasing
2022-01-02 19:12:33
2022-01-02 19:12:31
```

The output in the CI would look like this error message. The person writing the patch can fix that before any humans have to read the incorrect timeline.

0.5 Analyzing Retrospectives Data

Analyzing Retrospectives' Data

Input, not decisions

Easier

Once the retrospectives are all in a consistent, analyzable, linted, format, we can write code to analyze them. This is not intended to make decisions for us: it is useful as input to decisions so that the decisions can be data-driven.

0.5.1 Retrospective Process

Retrospective Process

Theory: Incident, Research, Draft, Review, Approve, Implement

Real life: Infinite variations

The retrospective process is simple, *in theory*. In practice, the details matter and there are infinite variations.

0.5.2 Iterating on Retrospective Process

Iterating Retrospective Process

Is our variation good?

Analyzing retrospectives is useful for answering the question "is our variation good"?

0.5.3 Example: Length of Recommendations

Number of Recommendations

```
def find_section(input_text, name):
    ...

def get_recommendations(input_text):
    bullets = find_section(
        input_text,
        "Recommendations",
    )
    recommendations = bullets.findall("list_item")
    return len(recommendations)

get_recommendations(input_text)
```

2

For example, we can count how many recommendations are made. This means that we can see how it changes over time.

0.5.4 Recommendations Done

Recommendation Done

Difficult to fit on slide

Similar ideas + API to ticket system

Similar code, but more complicated, could check which recommendations are finished. This is useful to see how we are addressing recommendations.

0.5.5 Share Analysis

Share Analysis

One off: Distribute Notebook

Repeated: Dashboards

It is important to share any analysis. For one-off, distribute something like Jupyter Notebooks. For a repeated analysis, create a dashboard.

0.6 Getting Started

Getting Started

This is great.

What's Next?

This is pretty cool, you're probably thinking. But this seems like it would be a big pain to implement, right?

Not really!

0.6.1 Light Markup Language

Light Markup Language

Choose

Markdown

(ReST, Asciidoc)

The first step is to choose a light markup language. Choose Markdown, unless you have a reason not to.

0.6.2 Template

Template

Validate that language works

You probably already have a template if you are doing retrospectives. If not, adapt something from the internet or from another team.

Write it in the language you chose. This helps validate that it is appropriate.

0.6.3 Organization

Organization

Create directories

Create the directory structure you want. Commit it to your source control system.

0.6.4 Repo

Repo

Create repo

(Not trivial): Set access controls

Create the repository in your collaboration system. Set appropriate access control, especially for approvals of merge requests.

0.6.5 Start Today!

Start!

That's all there is!

That is it! All of you are just five minutes away from treating your retrospectives as code. When are you going to do it?