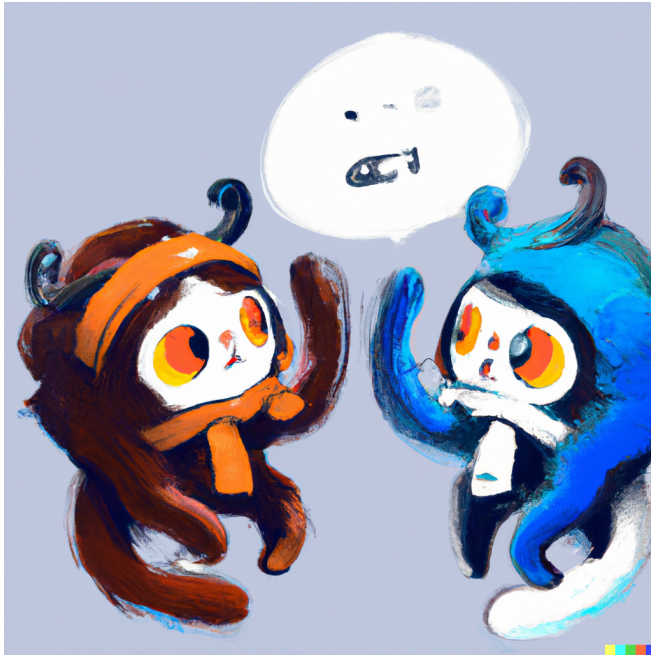# A meta talk about Git strategies

By Morten Olsen

Let me start with a (semi) fictional story; It is Friday, and you and your team have spent the last five weeks working on this excellent new feature. You have written a bunch of unit tests to ensure that you maintain your project's impressive 100

You go home for the weekend satisfied with the great job you did.

On Monday, you open your email to find it flooded with customers screaming that nothing is working! Oh no, you must have made a mistake!!! So you set about debugging and quickly locate the error message in your monitoring, so you checkout the code from Git and start investigating. But the error that happens isn't even possible. So you spend the entire day debugging, again and again, coming to the same conclusion; This is not possible.

Git Rocks!

So finally, you decide to go and read the deployment log line-by-painstakingly-line, and there, on line 13.318, you see it! One of your 12 microservices failed deployment! The deployment used a script with a pipe in it. Unfortunately, the script did not have pipefail configured. The script, therefore, did not generate a non-zero exit code, so the deployment just kept humming along, deploying the remaining 11 with success. This chain of events resulted in a broken infrastructure state and un-happy customers, and you spend the entire Monday debugging and potentially the ENTIRE EXISTANCE coming to an end!

I think most developers would have a story similar to the one above, so why is getting release management right so damn hard? Modern software architecture and the tools that help us are complex machineries, which goes for our deployment tools. Therefore ensuring that every little thing is as planned means that we would have to check hundreds, if not thousands of items, each more to decipher than the last (anyone who has ever tried to solve a broken Xcode builds from an output log will know this).

So is there a better way? Unfortunately, when things break, any of those thousands of items could be the reason, so when stuff does break, the answer is most likely no, but what about just answering the simple question: "Is something broken?". Well, I am glad you asked because I do believe that there is a better way, and it is a way that revolves around Git.

1 Declaring your expected state

So I am going to talk about Kubernetes, yet again - A technology I use less and less but, for some reason, ends up being part of my examples more and more often.

At its core Kubernetes has two conceptually simple tasks; it stores an expected state of the resources that it is supposed to keep track of two; if any of those resources are, in fact, not in the expected state, it tries to right the wrong.

This approach means that when we interact with Kubernetes, we don't ask it to perform a specific task - We never tell it, "create three additional instances of service X," but rather ", There should be five instances of service X".

This approach also means that instead of actions and events, we can use reconciliation - no tracking of what was and what is, just what we expect; the rest is the tool's responsibility.

It also makes it very easy for Kubernetes to track the health of the infrastructure - it knows the expected state. If the actual state differs, it is in some unhealthy state, and if it is unhealthy, it should either fix it or, failing that, raise the alarm for manual intervention.

2 Git as the expected state

So how does this relate to Git? Well, Git is a version control system. As such, it should keep track of the state of the code. That, to me, doesn't just include when and why but also where - to elaborate: Git is already great at telling when something happened and also why (provided that you write good commit messages), but it

should also be able to answer what is the code state in a given context.

So let's say you have a production environment; a good Git strategy, in my opinion, should be able to answer the question, "What is the expected code state on production right now?" And note the word "expected" here; it is crucial because Git is, of course, not able to do deployments or sync environments (in most cases) but what it can do is serve as our expected state that I talked about with Kubernetes.

The target is to be able to compare what we expect, with what is actually there completly independant of all the tooling that sits in between, as we want to remove those as a source of error or complexity.

We want to have something with the simplicity of the Kubernetes approach - we declare an expected state, and the tooling enforces this or alerts us if it can not.

We also need to ensure that we can compare our expected state to the actual state.

To achieve this we are going to focus on Git SHAs, so we will be tracking if a deployed resource is a deployment of our expected SHA.

For a web resource, an excellent way to do this could be through a `/.well-known/deployment-meta.json` while if you are running something like Terraform and AWS, you could tag your resources with this SHA - Try to have as few different methods of exposing this information as possible to keep monitoring simple.

With this piece of information, we are ready to create our monitor. Let's say we have a Git ref called `environments/production`, and its HEAD points to what we expect to be in production, now comparing is simply getting the SHA of the HEAD commit of that ref and comparing it to our `./well-known/deployment-meta.json`. If they match, the environment is in the expected state. If not, it is unhealthy.

Let's extend on this a bit; we can add a scheduled task that checks the monitor. If it is unhealthy, it retriggers a deployment and, if that fails, raises the alarm - So even if a deployment failed and no one noticed it yet, it will get auto-corrected the next time our simple reconciler runs. This can be done simply using something like a GitHub workflow.

You could also go all in and write a crossplane controller and use the actual Kubernetes reconciler to ensure your environments are in a healty state - Go as creazy as you like, just remember to make the tool work for you, not the other way around.

So, now we have a setup where Git tracks the expected state, and we can easily compare the expected state and the actual state. Lastly, we have a reconciliation loop that tries to rectify any discrepancy.

3 Conclusion

So as a developer, the only thing I need to keep track of is that my Git refs are pointing to the right stuff. Everything else is reconciliation that I don't have to worry about - unless it is unreconcilable - and in which case, I will get alerted.

As someone responsible for the infrastructure, the only thing I need to keep track of is that the expected state matches the actual state.

No more multi-tool lookup, complex log dives or timeline reconstruction (until something fails, of course)

I believe that the switch from Git being just the code to being the code state makes a lot of daily tasks more straightforward and more transparent, builds a more resilient infrastructure and is worth considering when deciding how you want to do Git.