Version 1.0

Introduction

- Today we are going to talk about software development, systems, and how to prevent them from getting out of hand

Motivating Questions

- How do you allow teams to release software, often that relies on one another, quickly without coordination?
    - All software is “software as a service”
- First major concept of the night: what is a version?

About Me

- Software and Devops engineer with a special interest in analyzing systems as a whole and how they run, interact, and eventually break
- Have worked through various issues that have cropped up due to improper versioning
- I’ve kinda made this my personal crusade for 2020 and I think it’s going to have massive benefits to our workflow and I think yours as well

Why You Should Care

- I think we inherently exist in an area of software engineering that has unique exposure to systems in the abstract
- This insight allows us to make small process tweaks that can have profound impacts
- I believe versioning is one of these processes
- But really, you should care because you are probably afraid to upgrade versions of software. “Pinning versions” is “best practices”, but I rarely see those versions updating. Software is a living system (as seen above), and in order to achieve

optimal system security and performance, we must tame that and incorporate these changes into our system
- I believe correct adherence to versioning allows us to get over that fear

How Software Is Developed

- Git
    - We will be talking about Git a good amount tonight, but many of the principles still apply in other version control tools. I am interested to hear about additional analogies from the audience
    - Various flows, but real production code often (and should) end up in Master
    - Often time though, we have crazy git processes that lead to many different “versions” of code floating around in space
        - You can use this approach, you just need to have SOME resolution eventually
    - We arrive at our first example/opportunity to analyze a versioning scheme
    - Git is a runner-up for my favorite piece of software ever
        - Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency
    - Git SHAs are a fantastic idea to use internal to Git, but give little information to consumers when used as a tagging mechanism
    - Let’s jump to our first concrete example and look at two docker images tagged with Git SHAs
        - Can anyone tell me which version is newer?
        - What about the relationship to the docker image you are running now?
            - Does it have breaking changes?
        - Ultimately we have no way of knowing and this is why we need a different tool
    - In comes Git Tags
        - This is a lot better, we already see one example of an “alpha” build, this seems to be to be “lower” or “earlier” than 0.2.1, this tells me something and it allows us to be programmatic, but also human readable
- APIs and large systems
    - External vs internal apis
        - What people outside your organization use versus what you and your tools use
    - Public facing APIs are a straightforward case

        - Customers are writing software assuming a level of stability when integrating with your API; you can't rip out their endpoints in a normal release, you have to think of backwards compatibility
            - Second major concept of the night: backwards compatibility
                - Adding new features without breaking old features
                    - Not entirely accurate
                    - We will define a few times throughout this talk
                - For a public API, it means adding new endpoints without modifying existing endpoints EXTERNAL facing interface; same api endpoint, same parameters
        - Most people are probably familiar with interfacing with public APIs and have likely lived through a version upgrade from a v1 to a v2
            - These are referred to as "Major" versions and often (most of the time) contain breaking changes and are NOT backwards compatible
    - Internal Versioniong and Distributed Systems
        - Our next topic is dependency management, but in a way we are already doing that here
        - If you are working with a small team on a single monolithic codebase, versioning internal systems is not as high of a priority, but when you are running dozens to thousands of individual services, you run into issues if you don't have a system
        - How do you A) create a stable interface between components of a system, B) allow those independent components to receive bug fixes, be refactored or receive new features?
            - The answer is using a versioning system that creates a clear "contract" to consumers and confidence that they can make up their own minds on how they want to take updates to your service. This allows the source system to continue to iterate and not become stagnate, without destroying the balance of the system
        - Here I have a few examples of "complex" systems that all have these same problems
            - Factorio, if you remove a component or update it, you don't even know the ripple effects it will have throughout the delicately balanced system
            - Systems are inherently unstable, you have to "simulate" stability by engineering for failure, one major way of doing that is strict adherence to a versioning scheme
            - Take another example, and I believe so much can be learned from the aviation industry and pilot culture, but here is the fuel system in a simple single propeller plane. If we hone in on a single component, the magneto, that itself is a highly complex

            piece of electronics and hardware. That device alone has received a number of updates and iterations over the years, but the interface to the oil pumps
            - "Stable interfaces" can also be seen in the construction of my favorite plane, the Airbus A320
                - It has components built over 4 countries, and I'm sure as hell they have versions for the individual bolts
            - What we can take away from this is that large systems are difficult, and we can't take the path of fear of updating something once it works; there are security updates, performance improvements that can all take place. If you have a safe interface, you can do that without disturbing the beast.
- Dependencies
    - Look at the CNCF landscape, I hope each one of these projects strictly adheres to a versioning scheme. I think in an ideal world, we would all be comfortable accepting all non-breaking changes from all of these packages. We should! Kubernetes is still in a very early state and things are moving quickly. We don't have to be afraid to update though.
    - The unfortunate situation is that two few smaller dependencies successfully adhere to correct versioning procedures.
- 100% of things
    - HELM
        - This is a cool thing because it already gives you a version
            - Makes upgrades very easy
            - Nice way to control releases of Kubernetes manifests which are inherently unversioned
        - You can make both code changes (docker image version updates) and supporting infrastructure and "runtime" changes (helm versions) independently and run multiple versions in the same cluster which is good for A/B testing/canary deployment etc
    - Video Game
        - One example is a video game I'm working on (as I mentioned, I'm a huge video game nerd). We just simply needed a way to associate patch notes with a build and I decided semver was the way to go. It has worked perfectly and in addition to versioning or builds with semver, we version all assets with semver (music, even fonts)

So How Do We Do It?

- Semver
    - Major

        - Breaking changes
    - Minor
        - Backwards-compatible features
    - Patch
        - All backwards-compatible
        - Bugs
        - Refactor
    - Prerelease information
        - Alpha tags
    - Build metadata
        - Can pretty much be arbitrary data, but is not an “official” part of the version
    - Why is it nice?
        - So easy to compare
        - Very flexible system that fits many applications (video game mentioned above, speaker notes, etc)
- **Start with 0.1.0**
- Auto-semver
    - Tool I developed (and still developing), to take some of the “mechanical” process out of versioning. I want you to _think_ about how your software is changing, and let the tool do the minutia for you
    - This is motivated by empowering the developer to ask how things are changing and if it breaks anything?
    - Front loading this question has very profound implications and aids in understanding how things change
    - This tool uses git tags, and I recommend you do the same regardless of if you use this tool or not
- Git Flow
    - We have a git flow that helps us “automate” some of these processes
    - This example is mostly for docker tags, but can easily be extended to helm releases / kubernetes releases (we are doing this) as well as binaries (if you have that need)
    - Note: The “commit” message bit is a limitation of our current CI system, ideally it can be the branch name of the branch being merged into master
- Create a system that works for you
    - Maybe you use a different version control software
    - Maybe you want to manually git tag and version things, that’s fine, I manually version the game example above, just use auto-semver to grab the latest tag and increment the desired value