RFD: Gofer and the Barrel of Cod (2024)

Clint J. Edwards
clint.j.edwards@gmail.com

June 22nd, 2024

Contents

1. Preface

In my last position for a company on the cusp of IPO, I noticed something strange that caught me off-guard. There wasn't a single solution for a problem I called "distributed cron". At that time I was part of the Infrastructure Security team and most of that work was creating small bits of code to protect other pieces of code. I soon became increasingly frustrated by this realization. Before I knew it I fell into my old habits again; I couldn't help myself from peering into the void of infrastructure work once more.

As a company we were struggling to figure out a solution, and rightfully so, the problem of distributed cron is hard. It is a problem of secure remote code execution, coupled with the problem of providing that as a service and also one that requires the solution to be fairly amorphous to accommodate all the reasons one might use remote code execution. In terms of problem spaces it's a nightmare. The end goal is somewhat clear, but the steps on how to get there are extremely blurry.

The company itself was in a bit of a crisis with its infrastructure in general (you can avoid that by checking out my startup internal development platform at https://astra.orreri.dev; yes I am shameless) and honestly we never actually implemented anything to solve this problem that worked properly. Instead, each team implemented what they needed to do in their own way and like a movie we've all seen too often, ended up with many poor solutions to the same problems.

Upon leaving that role the thought came to me that there is a hole in our industry; I became deeply confused on how a sector of tech that had so many players all somehow fell short on what I think a workable solution to the distributed cron problem should feel like. So like most software engineers, devoid of any humility on how hard it might be to build such a system, I started coding. Mostly for myself, partly to have something I could point to on my Github for future company interviews[1].

This is the story of that thing.

2. Firstly, What is "Distributed Cron"

I tend to use the words "Distributed Cron" to cast a wide net on a range of tooling I believe to work mostly in the same paradigm. They are remote code execution platforms, essentially running small pieces of code on some schedule or on the account of some event taking place. As such during the rest of this writing I may use "Distributed Cron", "Remote Code Execution", and "Workflow Executor" all interchangeably. The specific nomenclature of "Distributed Cron" comes from Google SRE's writing on the topic (They didn't coin it, but it's where I first encountered it).

The current state of distributed cron is somewhat of a confusing landscape. I think it can be broken down into three camps of things that seem like they should work the exact same way at a high level:

Camp 1: CI/CD Tooling - This is the Github actions, the Gitlab runners, the Travis CIs, and the Circle CIs of the world. They're focused on one thing, providing you with some way for you to run code against a

[1]Spoiler: Very few companies I ever interview with give a shit about anything I've actually coded on my Github. I have an entire rant about how poor our current industry interview practices are. But what's somewhat more maddening is that very few companies even care despite such over whelming disdain for these practices. We're slowly losing some of what made this industry so exciting. It was never about if you could spit back out efficient leetcode algorithms, but all about what you could produce. If you thought different, were creative, and maybe a little bit of a rebel, you were hired, because there was value in the way you attacked problems. We seem to be maturing past that as an industry and creating our own version of hazing to prove you belong here.

I think it's important to remember that some problems cannot be solved with purely computer based solutions.

code repository. They typically do so by strongly tying Gitops[2] to their operating model and are optimized for automating common software dev processes like code linting.

Camp 2: Thing doers - Concourse and Jenkins are notable for this group. These are truly the swiss army knives, the tooling that understands it is remote code execution and allows the flexibility that paradigm requires. They are generally feature filled and malleable, but lack any native tooling specific to any one domain.

Camp 3: Workflow executors - These are tools like Apache Airflow, Temporal, and Luigi, that focus on tooling to help engineers format what they want to do as a series of steps. These steps usually have conditionals and are usually set up in a directed acyclic graph format that closely resembles the structure of what would otherwise be known as a finite state machine. These tools usually contain other tools that are meant to help with the domain focus, for example, data streaming hooks. These tools also usually value high durability as restarting a several terabyte data ingestion job from the start is very costly.

Every camp, at a high level, works more or less the same way; there is a small bit of code that doesn't quite fit for whatever reason with how the main application(s) is run and needs to be run separately. Usually this code is also run from the result of some other event occurring. This trigger can be the passage of time, an api call, a git push, or just about any other "event" that might occur.

The biggest differentiator between the camps are the reliability guarantees you need for the workloads. For example, CI/CD tooling can afford to fail every once in a while in normal usage circumstances, mostly due to its typical workload being idempotent and the cost of occasional failures being somewhat low stakes. On the other hand, "thing doers" and "workflow executors" often end up being the primary way external jobs are run for some applications. This means depending on the application(for example, finance focused applications) these jobs are as or more important than its parent. The other differences between the camps are immaterial as most of the differences boil down to features that can be replicated in any camp somewhat trivially.

3. Okay, different tooling for different jobs, sounds reasonable. What's wrong with it?

Absolutely nothing!

Most of these tools do exactly what they're meant to do for the domain they were created.

The problem lies in the proliferation of this tooling among too many specific use-cases and the user/development experience nightmare that is birthed from the failure of focus. Taking the concept of remote code execution and splitting it among many, different, bespoke tools that at their core do the same thing in vastly different ways, means big issues for the human brain that has to keep some internal index for these infrequently used tools.

Developers must first identify where something belongs among the gamut of tools that all kinda sound similar, but then they must perform the necessary incantation to make their code work within whatever paradigm this tool thinks is the best. Then when inevitably the system breaks, they must attempt to

[2] A very popular operating model for a lot of software development systems is Gitops. It basically just means that operating a particular system involves making changes to the system via configuration stored in a Git repository. Changes to those configurations or actions performed upon that repository(like pushing to it) causes some state to change or action to happen somewhere else. It's very popular and comes with a number of benefits, namely just allowing developers to use tools they are already familiar with and have.

understand how the system works to appropriately debug the breakage. Again, spread this across x number of applications and it quickly becomes a major timesink.

Not to be left out, the support teams (platform and security usually) managing this tooling has to do so for the many different (often badly documented) shapes this tooling appears in. Obviously, this isn't ideal as human beings have enough of a hard time managing just one system. Let's say the average company has one tool from each of the camps mentioned above. That means security evaluations, permissioning management, user management, reliability evaluation, on call rotations, documentation, and much more that needs to be coordinated for a specific tool to run.

We can do better and the crux of the problem lies in common industry thinking about these tools from too high of an abstraction level. Instead of focusing specifically on CI/CD, it should be possible for us to build a system that is a generalized remote code execution platform. Then, in that platform, we should be able to add features(or allow the user to add features) that bridge the gap between these other abstractions in a way that boosts familiarity with a single platform and still delivers the guarantees and behavior we need from all camps.

The logic is that if the key difference between the camps is in the reliability guarantees, then we should be able to create such a system that has sophisticated reliability with the understanding that nobody ever was dissatisfied with having too much reliability.

The drawback of this approach is the (hopefully slight) erosion in user experience. We're gently shifting the learning curve for end users from being slightly within the discovery and implementation phases of building software, to purely being within the implementation phase. No longer will users have to bikeshed about where a short lived job should be run, instead they will consider patterns for making that job operate how they intended. This is good! And the tradeoff is worth it. This results in developers building skills with tooling that mirrors other behavior that they already know and will use again in the future. An additional boon for the alternative approach lies in the prospect of code reuse. A single distributed system can lean a bit heavier on implementation details for the workloads since the centralization of those workloads means the ability to create templates focused on specific use cases. As the distributed cron system in question matures, so does its single community and its cache of robust solutions.

3.1. We tried building a generalized RCE platform, it's called Jenkins and it's a mess.

What can be said about Jenkins that hasn't already been said through gritted teeth and the prolific use of expletives? I really did not want to say anything good about Jenkins for maybe the last decade of my life(I have the capability to be very petty). Mostly because I feel like I've been somehow abused by my time writing it's config configuration language Groovy. The many late nights and frustrated commits (*"Maybe this will work 42"*) have turned me off to some of the charms Jenkins might have had. But evaluating it against the tooling we have today made me confront the uncomfortable idea that Jenkins did actually do somethings correct. And that is as much praise as it's going to get.

Jenkins has a huge market share in this space[3], for a while every silicon-valley-esque SaaS company had a deployment of Jenkins that they begrudgingly maintained. It was just that useful.

[3]A quick Google search reveals that Jenkins seems to hold somewhere around 50% market share for "continuous integration" tooling. A lead it surely gained from being a pretty good product early on and maintains by being an extendable project in its later years.

What Jenkins got right was mostly in it's extremely extensible interface and it's easy to deploy binary. Jenkins solved the collective problem of "I want to do CI/CD mostly, but also sometimes I want to just run this random script at 2pm, oh and sometimes I just want to press a button and run this script". Where Jenkins went off the rails is that it was an early iteration of the "thing do-er" idea and it was a little… too extensible. Interacting with Jenkins was a massive pain and writing pipelines for Jenkins involved actual sacrifices to the dark lord. (*Honestly, if you wanted to write any significantly complex pipelines in Jenkins, Groovy would make sure that you were never more than a couple keystrokes from quitting your job and becoming a goat farmer*). But if you could grit through it, you would get something that worked and most times it worked the way you wanted it to work. It gave developers and ops teams the massive amounts of control they required to solve hard problems. It was both ahead of its time and couldn't keep up with how fast the industry was learning.

In the end though, every company ended up with a sloppy mess of Jenkins plugins that had every bad software supply chain trait known to man. Plugins were often out of date, incompatible with the version of Jenkins you were on, and their security was questionable at best; which was a bit scary since Jenkins often had access to an enormous amount of secrets and source code.

To be completely fair(*another thing I'm not sure Jenkins deserves*), this is not all Jenkins' fault. A lot of the issue with Jenkins' plugin system still exists in the world of open source software today. When you use tooling created by people who are doing it for free, in their spare time, you take what you can get.

The bet is if a Jenkins were created that leaned a little more modern and a little bit more opinionated, maybe it would fit the mold for all the camps previously mentioned and lower the bar on grokking those different abstractions on both the ops side AND the user side. Any reasonable plugin system will have security concerns, but leveraging our modern understanding of software supply chains and defensive runtimes can at least mitigate some of the glaring ones.

4. Some anti-patterns we need to shuck from the post Jenkins world

As part of the industry's graduation from Jenkins we went a bit stir crazy on what could be done. To quote the great Dr. Ian Malcolm "Your scientists were so preoccupied with whether or not they could that they didn't stop to think if they should.". Using the power of hindsight let's go through a few mechanisms I believe the industry has missed the mark on. But lets save the talk about how to fix them for dessert.

4.1. Complex deployments and management for both Operators and Developers

This point is somewhat critical of the current state of software in general. Software is often created with the users as the focal point, leaving the people who have to maintain such software as the blurred, forgotten lines around the edge. Usually this means that the internal deployment of said software is difficult and the management tools for the software is non-existent. For example, back in the early days of Airflow there was no real way to get the code you wanted to run onto the Airflow host machine (I still don't know if they have a solution for this). Instead, you were required to download the snippets of code you needed in some way that you figured out on your own and then point airflow to where you downloaded it. Maybe I didn't quite understand the user experience of Airflow, but I think that says more about the product than it does about me.

Recently, as an industry, we've been getting better at packing and shipping our software, but in reality it's possible we just got bailed out by the creation of containers. Containers provide the infrastructure

and user experience that we should have paid more attention to before they were introduced. On some level developers need to care about what happens to the product as it goes through the software lifecycle in the hands of their customers.

When writing platforms there should be a greater amount of attention paid on the balance of appeal to ops vs the appeal to developers. Just like investing in testing or debugging tools, long term sustainability is achieved by making it easy to not only use the software for the end user, but manage/develop it as an admin/power user. As an industry we're learning this very slowly, but surely. Each new programming language starts with good tooling baked into the ergonomics of the language, proving that we understand that usability is largely important.

As some extra color, ignoring the maintainability of your application both externally and internally is akin to a city building roads they have a tough time replacing. While the initial road might be exactly what city-dwellers want, within a few short years the same roads are seldomly used due to their current state of dilapidation as the hard to replace roads leads to quicker decay.

But the rabbit hole goes deeper than that!

Packaging software has room for improvement, but what about updating our distributed cron jobs?

What about the developers who only care about their little snippet of code that runs every Tuesday night to backup some important database? In most cases this story is equally as broken as the previous one. Most distributed cron systems lean on Gitops for their answer to "how do I deploy this script I want to run". This is not a claim that Gitops is bad, because that claim would be wholly untrue, instead, Gitops has a few shortcomings we need to talk about before we can make a prescription on what should be fixed about using it to release and manage jobs.

To flesh out an example of this let's dive deeper into a job deployment model that exists currently. CircleCI, a fairly common and popular CI/CD platform, has a common deployment model for how jobs get executed on similar platforms. If we get past the mandatory eye roll that stems from every company smashing the word *AI* into every marketing page and read over the documentation, the deployment model for CircleCI looks something like this:

1. You tell CircleCI where your code exists, using one of three code hosting vendors that they integrate with.
2. We create a project in the CircleCI UI and we correspondingly create a config.yml which you put in a specific folder located in the root of your project.

   At this point you can fashion an argument that having it be mandatory that you use a predefined code hosting vendor is already a sticking point, but in practice this rarely is an issue for many companies. Let's continue:
3. Once the configuration file is registered for your pipeline, CircleCI is essentially connected to this repository. This means that you can set up triggers which will automatically detect changes to this repo and then perform the actions you put in the configuration.
4. When a trigger condition is successfully met, CircleCI starts processing your job. It checks out the targeted branch, and then reads your config file once more to understand what it should execute.
5. We've already run into the problem so I won't explain too much more, only to say that it attempts to spin up your container onto one of its 'runners' and executes that container providing you the output.

This dance is a mirror of what most Gitops processes look like. You check your changes in, those changes cause things to happen. The problem is it's a bit rigid. It is a reasonable workflow when your jobs are mostly just updating how you run tests, or running benchmarking, but it becomes inflexible for anything outside of that paradigm.

As your company grows your compliance will need to as well. You'll need to start running more sophisticated analysis of code…maybe a HCL linter (I already told you I was shameless, I dunno why you're still surprised). This linter isn't just assigned to a single repository, we might have many that need the services of such static analysis. On top of that the static analysis tooling might be maintained by a single team. So we're a little under-served by the traditional gitops model. We can use Gitops to build a version of the static analysis tool, but we still need a way to integrate it in other team's CI/CD workflow.

So? We're engineers, we solve problems! And this one can be solved by how we handle all the other tooling of this sort. We slap a version on it, put it into a container, and then we distribute it out to be run automatically by every team by including the snippet into their `config.yml` file of sorts.

But like most things, the world isn't that simple. Once you get to large enough companies, managing internal versions and making sure they're updated for each team that needs it updated becomes somewhat of a painful, complex process. The issues continue to flow once you realize that because you need to run this static analysis tool on every PR that runs, your tooling is now mission critical! Your HCLvet rules have security rules in them! They can't simply be skipped anymore when they are broken.

Well, now we need to develop some tooling to increase our reliability. We need to test our tool properly before releasing it to the masses. Anyone who has worked in software for a while realizes that despite our early efforts, the best way to avoid downtime from releasing something that simply wasn't ready for production…is to release it to production..but very slowly and carefully.

Now your simple static analysis tool needs a bunch of tooling around it that requires your team to build and maintain code that arguably better belongs in the tool that you're using to run the job in the first place. Long term software platforms like Kubernetes have reliability tooling built in already, because we as an industry understood it was critical to the management of the guest software.

What if instead of relying on Gitops as this core tenant of deployment and management it instead becomes just a feature of the short term job executor that you're using. There would be no need to build a bespoke canary system on top of a system that is already several times bastardized into working the way we need it to. Instead the system underneath understands what a canary release is and you can track new releases of your pipelines and making sure they're not breaking developer workflows. This means you gain the peace of mind to release your software safely and easily, because while it isn't a long term running job, it still needs as close to 100% uptime as we can get.

4.2. Marrying code to the system

This one is more of personal preference, but I haven't found great reasons to justify it so I'm keeping it squarely in my anti-pattern list.[4] Systems like Airflow and Temporal require the code you run to know

[4]This opinion is a tough one to defend at times. Any reasonable software engineer will tell you that there is no clear winner between modular designs and deeply intertwined designs. Each have their own host of pros and cons. The true mark of a good engineer is being able to tell when each paradigm would fit best.

Early on in my career I heavily leaned towards modular designs, but I quickly realized that there is distinct beauty and simplicity in designs that are tightly intertwined such that the pieces connected together are considered more holistically.

about the system that they're running on. Usually through some sort of decorator or library that you must use in order to integrate tightly with their system.

Tight integration of code to the execution platform is not a new idea, but in this particular context it doesn't spark joy. Mostly because it breaks the code's ability to be highly modular. When developers write code they're usually trying to solve a business case. Wrapping the business logic up with the execution logic seems like a bit of a boundary violation. The code isn't just the code anymore, now it's the code **plus** Temporal specific code on top of it. If AWS comes up with some other distributed cron system that better fits my business needs, it's no longer as simple as taking your docker container and switching. Now you must translate large parts of your code due to the fact that they were built with the implementation details of running on Temporal. Now you might be saying "well I really don't plan to change my distributed cron system ever so this is not a problem for me" and, well, my response to that is "keep that same energy"[5]. It's never a problem until it is and I tend not to want to take on tech debt if I don't have to. Tech moves fast, can you guarantee this won't be a problem?

Furthermore, the advantages that you are gifted from this tight coupling aren't things that can't be achieved by simply taking a different approach to either the execution or the code structure. The argument from Temporal's side would be that you'd have to do a lot more work to achieve the same effect, and I would agree, but it's not *that* much more work. For example, Temporal's first early *wow* feature was the ability to sleep for an infinite amount of time right in the middle of your code. This is useful for waiting for some other event to happen. A powerful feature no doubt, but waiting for some time in the future is what workflow executors do for fun. This is simply, explicitly, and durably replicated by any workflow executor worth it's salt(Hello event-driven systems) and without the underlying mystery of how Temporal is internally managing this workflow on your behalf.

4.3. Configuration languages as a DSL

For this hardship, I'm going to blame Hashicorp. I have no evidence that they are responsible, but anecdotally the first time I became excited for declarative configuration was when Vagrant was released. It replaced error prone manual setup of VMs with a short configuration file, written in HCL, that instructed the program how to build my VM the right way every time. In one command it would produce the perfect development machine every time. I'm still chasing the high of my first `vagrant up`. Back then I had never experienced software with declarative configurations. I, like many others, were used to digging into my bash history for esoteric settings that produced my desired output and in light of that, searching the internet for how to glue together the settings I needed in my tools. So writing a configuration file, to reproduce my VMs consistently was nothing short of revolutionary in my eyes.

The ability to declare configuration isn't the bad part though, it's the structure in which we do so that is the sticking point. Most configuration languages should strictly be kept for simple configuration. Anything that might even have a hint of being somewhat complex should probably be done by an actual programming language.

(*Check out Oxide Computer's work for a good example of taking something that was once thought of as bespoke parts and is now treated as a single, holistic entity, with great success.*)

[5] Somewhat of a slang term; When someone says "keep that same energy" usually what they're saying is that "you should keep acting or operating in that same fashion" with the implication being that soon you will realize the error of your ways. Said in another way "keep doing what you're doing and you'll find out how bad what you're doing actually is in time". In this context there is a good chance that you actually will never need to move distributed cron providers, but in my opinion even if I never expect to run into this problem being married to it isn't something I want either, provided there are other equitable options.

It's understandable what designers were going for when HCL or GCL were first introduced, but what usually happens is that for any relatively complex product, a configuration language eventually needs a templating language and then a templating language eventually devolves into inheritance nonsense that nobody actually understands the full state of. Which makes it awful for development and learning, two of the supposed goals for using a configuration language in the first place. The problem is that any reasonably complex declaration of the user's intentions will require tooling and expressions that programming languages already excel at.

The introduction of configuration languages only brought more confusion to declarative configuration. Reading this configuration in a language I can already read pretty well has turned into me looking up bespoke built-in functions for a language I only really ever use if I'm touching this specific tooling. Imagine if every problem set within computer science came with its own specific programming language, its own tooling, its own idiosyncrasies, etc.

This is a departure from current wisdom that the engineers who build the systems should also be the engineers who run the system. Configuration languages were touted as a way to simplify configuration, making it easier for those who didn't know how to code or couldn't code as well (*Maybe you ended up on the management track but still wanted to be somewhat hands on*) to make changes easily. A noble goal, but also somewhat naive.

The largest boon configuration languages have for them is that they prevent arbitrary code from running in places where it would be detrimental to do so. In most, you cannot download a library that puts a backdoor into your ssh sessions. Which is good. But we've already signed on for the supply chain maintenance problem by virtue of being a tech company. This advantage isn't worthless, but it also isn't the huge win we first thought.

There is a lot more to say on this topic as the argument between configuration languages and just using a normal programming language is very heated, but it can be summed up by saying, in the world of configuration languages either nobody uses you anymore and you die a hero, or you live long enough to see yourself become a bastardized programming language.

The one thing you were trying to avoid.

4.4. Poor local development loops

This one is somewhat understandable due to the nature of these systems. It's hard for distributed systems to be tested locally. Mostly because they usually require inputs and data from mechanisms that simply don't exist locally and are hard to mock. That may be secrets or external api calls or something else, but most of the distributed cron tooling is lacking in this regard.

Let's take for example the feedback loop of running something on Github actions[6]. Typically, when one wants to make a change, they open the workflow config files and attempt to update them. This is already anxiety inducing because what I have to interact with is a DSL made from YAML and the only way to understand how that DSL works is to read over multiple pages of bespoke documentation(*e.g. No IDE help*

[6]Here specifically I'm talking about tracking down which events map back to which names for those events in Github and then once I understand that then understanding the incantation on how to do something with those events on certain branches within my project. This goes on quite a long way and to be honest my main gripe with it is that it doesn't closely reflect how I interact with Git or Github on a day to day basis. I can't just write somewhere "git clone && cargo lint", I have to understand how to work within the specific paradigm of what this specific DSL allows. It's a major context switch and frankly an unwelcome one that feels unintuitive.

for you). Sometimes to understand if something takes a certain value and how that value's syntax should be structured I'll have to look over three separate pieces of documentation. Not fun when all I want to do is something trivial like add a different linter to my project.[7]

Inevitably, I will get some of the syntax incorrect because again we're using a custom DSL that has no validation, no IDE help, no autocomplete, no nothing. (At the very least Hashicorp's Vagrant had a `verify` command) And so when I commit this change so that it takes effect, I have to do it within a branch to test it. This is mostly okay, feature branches are good practice, but because of the lack of a local feedback loop we've just extended my "*change -> check -> change*" feedback loop and added several unreasonably slow additions. It's no longer quick and tight, now it's relatively slow and error prone. I do this maybe an average of 5-10 times before I get all the bugs properly worked out and my script works how I expect. It's also important to add that debugging why something isn't working is also painful. To do so I have to understand the bounds of the system I'm working in in the first place. I need to understand that when I run `cargo`
`build` that the target directory will be in a specific location so I can get my assets and do something with them.

This is all pretty disjointed. Most of the distributed cron tooling recommends that you use their special way of uploading and retrieving assets. Suddenly, I'm working within tooling that I'm very much not familiar with and want to exit as soon as possible. It's not enjoyable at all.

4.5. Stunted extensibility

Most of this applies to the hosted solutions for distributed cron. Once you are outside the author's original thought process on what you might want to do you're pretty much out of luck. If they don't offer support for it sorry, work around it. Which is actually pretty fair and how most software should work. But specifically for distributed cron, you should be opinionated, but flexible at the same time. At any reasonably sized software company they have a vision for how this distributed cron software works within that vision and I believe the software should be designed in such a way that allows those companies to marry their vision with the software's vision.

The problem is that too much flexibility results in the Jenkins problem mentioned earlier. Instead, the solution to this problem is to have a clear goal for the interfaces you expose outside of the platform. Anything outside that interface shouldn't be supported. Where you draw that particular line exactly is an incredibly hard question, but we have data from past failures that we can draw upon.

4.6. A lack of tooling that mirrors long-running software

We touched upon this in 3.1, but let's hammer it home again. Observability, canarying, blue/green deployments and other techniques are things missing from many of these distributed cron systems. Usually they're most concerned about running your workflow and conveying the state of that workflow to you. The features that mirror traditional, common long-running software tooling stops there. Some of them don't support exporting metrics and many of them because of the Gitops paradigm, require you to get the implementation right as the next step is always 'production'. For certain types of workloads, especially as your company gets bigger these are all tools you will need to replicate to make sure you adhere your workload to the standard that held for production.

[7] I'm aware that Github Actions has ways of running workloads locally. I've never used them and I'm unsure how good they are, mostly because I don't want to download yet another 3rd party tool to test code that should be runnable locally trivially.

5. How do we go about solving some of these problems?

The hubris I felt pushed me to start writing small bits of code in an effort to see just how far I can get on a system that approaches solving some of these problems. The result of that approach is Gofer. Gofer attempts to address some of the issues mentioned above and it has also acted as a stretch goal to have an excuse to eventually build a distributed system.

The goals of Gofer were simple; make something I would enjoy using at my next gig. And if I could fix a lot of the headache I've experienced stepping through other forms of distributed cron systems, that would be okay too.

I started designing and after many iterations I set my sight on the following goals:

5.1. Must be CLI first

The command line is an extremely powerful tool for professionals…but to be honest what I'm really trying to hide is my ineptitude for frontend development. My designs never intersect with my brand of perfectionism and that pains me in ways I'll never recover from. With this bias in mind, I do truly believe most tools should start with the command line if possible. Creating a command line tool means that we can leverage that tool for our eventual increased needs of automation. A command line tool properly designed today, means that not only have we created everything we need to create a graphic frontend, but it also makes it trivial to move into the arms of our automaton overlords.

5.2. Simple Everything

The goal is to tickle the part of the user's brain that screams familiarity. If during the use of Gofer a user is able to intuitively guess how certain things work and should operate, it has succeed. Instead, if the user is constantly context switching to the documentation, we've failed. The documentation shouldn't have to guide you through too much and the API should be fairly straight forward with what intuitively you want to do. (This will never be perfect until natural speech processing gets really good enough, but the goal is still worthy) The goal is to at every junction of recreating features that we are used to in distributed cron, stop and ask the question, but why?

5.3. No more configuration languages

Many of these topics have enough behind it to create its own paper, but the proliferation of configuration languages and their attempt to solve the problems of programming languages in declarative configuration is possibly the most eligible. It's time to give the gun back to programmers. Get rid of configuration languages and hand ultimate power back to those who are willing to learn how to wield it.

Instead of ham-fisting a configuration language back into the shape of a programming language, using an actual programming language allows us to reap the benefits of not just the programming language itself, but the tooling that the developer probably understands how to use already.

The user should be able to write the programming language in the language of their choice and we can make sure we get a consistent product out of those languages by using a common serialization format like Protobuf or JSON.

The developer experience advocate in me is excited about the prospect of being able to easily analyze pipelines due to this and help developers within my theoretical company navigate some best practices.

5.4. Extensible Architecture

With a remote code execution platform it's almost impossible to internally provide everything the user might want. This is due to the platform not being made with specific code or environments in mind. Instead it must be flexible enough to not only support the different types of code one might run on said platform, but to support how one might run said platform.

This flexibility means two things have to be delivered that both revolve around modularity:

- Providing an interface such that triggering the code or piece of the pipeline is easy.
- Provide the code with the mechanisms it needs that aren't necessarily the focus of the remote code execution

platform (Secret management, caching, notifications, etc)

5.5. DAG support

This one is pretty straight-forward and something most distributed cron systems deliver by default. It's often useful for users to string small bits of code together that depend on each other. Now you could do this purely in code (and I think you should when you can), but sometimes you want to run a bespoke tool right after another bespoke tool based on the outcome of the previous tooling. To do this you would require a graph of sorts that once fully processed, resolves in some concrete outcome.

As previously mentioned this is table stakes for most executors.[8]

5.6. Reliability features

This point isn't borne out in actual experience just yet, only frustrations where it has missed. Gofer has only built the foundation of what would be needed for something like this to exist, but it's something missing from tooling I've used in the past, so it's something I'm keen on exploring.

When I worked for that infrastructure security team I built a static analysis tool that helped us write security rules against our massive Terraform repo. This was the easy part. Eventually, I would need to have my linter run against every Terraform pull request that was created and give the user some helpful feedback on how they could fix whatever issues the linter found.

The standard way to do this is just include the linter into the list of tests and checks we are already performing per pull request. And so I did, I created a new check and sheepishly inserted my linter into the workflow for hundreds of engineers. This all went fine until there was something I wanted to change. I realized there really was no good reliability tooling for my specific check. Much like the thinking of yester-year I had one chance to get it right in dev and staging and then I would unleash it upon the masses who were all already annoyed by the fact that it was running in the first place.

This obviously did not go well sometimes, there were times when what I really wanted was a quick way to roll things back such that users waiting on me for a fix didn't have to be told to *"try rebasing now"* in order for their checks to pass. What would have been even nicer, is for the remote code execution tool to be monitoring the failures of the new version of my linter and have a function that could automagically roll it back globally for me.

[8] An interesting part of DAG support is the choice between static and dynamic DAGs. It is an important feature to some users that the DAG that is run is able to be created on the fly by the DAG nodes itself. This means that while a single node in the DAG is running it may actually influence how many and what child nodes are run after it. This is a much used feature of hatchet. Unfortunately(*or fortunately depending on who you ask*), in order to keep things simple Gofer does not implement this.

These are examples of tooling that have become commonplace among long-running code platforms. We've realized that there is no testing like production testing and we've built tools to make sure releases have a lot less risk than a simple fire and forget. I think there is space for this in remote code execution, greater than simple retrying or an alert when a task fails. I think we can be smarter about pipeline reliability and versioning.

5.7. Easy local testing

Having a quick development loop is paramount for getting anything done in software at all. We're all human and we all make mistakes and the quicker those mistakes are brought to light the quicker we can ship with confidence.

The current state of distributed cron tooling does not make this easy though and honestly it's not something I have fully figured out yet. It's a tough problem. The configuration AND code that we write for remote code execution platforms are subject to the details of how that platform enables you to run your code. You can do as much local testing as you want, but if there is a security group blocking your network requests on your distributed cron platform, there isn't much you can do about that.

At the very least though we can form patterns for how we can get as close as possible. I believe one of those patterns is making the container the centerpiece of code execution. It allows things to be mostly predictable and testable locally. You do your development of your pipeline locally testing within your docker container and then when it's ready you just tell Gofer to run it. This doesn't solve all the problems and you can already do this on other platforms. Gofer's secret magic is that you can run it locally and test out your workloads from localhost. This, accompanied by the reliability tooling mentioned above, should bring the amount of fingers and toes we cross when shipping to production down a notch.

6. Let's start writing!

A new challenge afoot I began coding and iterating on what would eventually be named as Gofer.

6.1. First let's go over the high level design of "Gofer"

6.1.1. #Goals

If it isn't apparent at this point I really value simplicity in software. My favorite piece of software are things that took something I once thought of as difficult and instead made them feel trivial. My goal with Gofer was to take into account all the problems mentioned previously and come up with a way to solve those problems without making a piece of software that was functionally harder to use.

That goal seems easy in the abstract, but like anything worthwhile was tough to adhere to in practice. Fixing some of the issues I found with distributed cron systems in the past meant that I had to take bravely different approaches than other tooling had. One example of this is that Gofer doesn't have any agents. There isn't a piece of code that acts as a minion/runner/whatever helping orchestrate your container running on a single machine. The goal here was to make something that fit in with what Ops teams have already.

So, to that end, Gofer doesn't force you to learn how to properly scale and configure yet another tool, instead Gofer just runs the containers on a platform you're probably already familiar with. While this is a nice goal for ease of use, that also means that Gofer has significantly less control and insight over how your container runs. For Gofer to do fancy things like collect telemetry on your run, it's at the mercy of what your container orchestrator makes available. This means that some of the features you

might get for free on a platform with agents, need to somehow be implemented on a platform with significantly less control. Tough and definitely has some drawbacks, but the hope is that while Gofer might not tie everything in a bow for you, the pros outweigh the cons. There are no foreign agents for you to manage, instead your containers run in places you're probably already familiar with. This is HUGE for troubleshooting and debugging.

The other thing that was hard from a Gofer design perspective, was the goal to make this something that fit the mold of as many environments as possible while being open source. I wanted something that people felt good kicking the tires of and that wasn't going to be thrown overboard the minute the underfunded startup that wrote it ran out of runway. If you're designing with a more focused goal in mind, say for a specific company's implementation of infrastructure, it's somewhat easier to create compelling features. With increased ambiguity, comes features that attempt to solve generalized cases. Leaving them to be less compelling than their counterparts. This is all to say, a tool made for a ton of use cases is usually pretty mediocre at the majority of them.

The ever so slightly comforting thing is that I'm not building rockets for the first time. The pattern in which Gofer works is something already solved by others and the interfaces to the tools that Gofer connects with are mostly welcoming to this exact type of tooling.

6.1.2. Architecture Overview

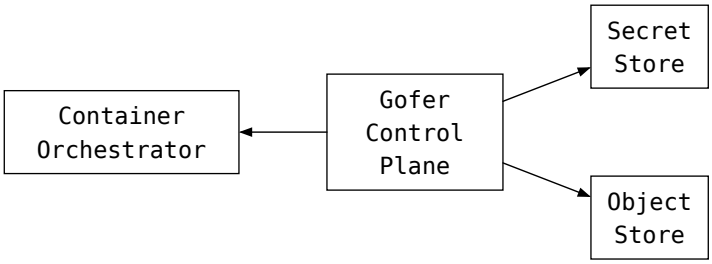

Fig. 1: Gofer's architecture and modular connections

The basics of Gofer are simple:

- Gofer runs as a single static binary which acts as the Gofer control plane. This control plane provides a HTTP REST API.
- It's meant to be deployed internally. Gofer's server binary and CLI binary are both the same. So it's possible to start Gofer in "dev mode" and just try out what you want to do locally. The user can edit their local configuration to talk to a local Gofer server or their internally deployed one. The strategy of a single static binary that does everything gives the end users a tidy package in which all tooling needed for every aspect of Gofer is conveniently located. There is no need to manage a docker-compose stack and then additionally interface with a CLI. Everything is in one place.
- The separate HTTP endpoint mentioned above is purely such that a company can open up access to only that service endpoint. The endpoint is run alongside the main service, but run on a different port. This endpoint is primarily used for external events like webhooks. This means that security teams don't have to worry about figuring out how to prevent people from accessing the entire API and can narrowly grant access to a single, specific port.
- The diagram above shows that Gofer's control plane is focused on integrating within an environment that already has known solutions for various aspects. This is a key feature of Gofer. Use the tooling you and your team are already familiar with. And if you don't have all or any of these services that Gofer needs to run, Gofer provides a rudimentary, but effective default which requires no input or setup from you and can take care of the needs of mid-sized companies easily.

The next diagram here shows how Gofer manages what are called "pipeline extensions".

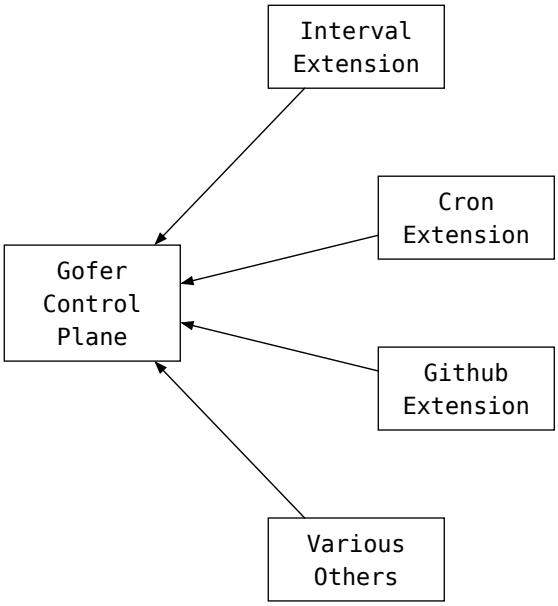

Fig. 2: Extensions

- Gofer needs a way to provide utility services to a user's pipeline. The most common example is providing ways for the user to automatically start their pipeline or once their pipeline is finished post the results somewhere meaningful. Gofer solves this through spinning up long running containers using your preferred container orchestrator.
- This works in the same paradigm as other containers, you tell Gofer where it is and what version to go get and it launches it. Once Gofer has spun up all the "installed" extensions. Gofer can communicate with this extension over an interface established beforehand and implemented by REST.
- The user explicitly tells Gofer to "subscribe" their pipeline to specific extensions and those extensions can take further actions based on different input.
- A quick example: It's possible for a pipeline that checks if a particular website is up, to use the "interval" extension for this. This extension simply calls the Gofer API and starts a new run of a particular pipeline when a previously defined time interval has passed.

6.2. Some of the hard parts

Next let's discuss some of the thorny parts of creating Gofer and distributed cron systems

6.3. What to do about Modularity?

6.3.1. First try or two: Hashicorp's Plugin system

My first approach to modularity revolved around Hashicorp's plugin system. The goal was to choose specific things Gofer would NOT do natively and provide a good interface that developers could write code against to expand its functionality.

The problem with modularity is that it conflicts with the previously mentioned value of simplicity. Writing a single program that does a number of things is simple, the code works with the other code and all is well. Writing something that has to be modular is difficult, it essentially requires predicting the future. Interfaces need to provide the user with enough flexibility for them to achieve their goals, but it's impossible to predict everyone's use case. Furthermore, large interfaces are a pain for developers to grok, develop, and maintain. Ideally the interface is set and you nailed it so perfectly that no further changes

need to be made. A rare, potentially impossible event for any used software. Instead, you try to keep the interface small, general, and focused on the task at hand.

Initially I chose Hashicorp's plugin system due to familiarity. I've considered writing extra tooling for Vault and Consul and found it liberating that all I had to do was write some go code against an interface and bam! It would work. The initial version of Gofer touted this plugin system for a few key areas:

- An object store so that it was easy for users to store various assets.
- A secret store so that users could easily inject secrets into their pipelines.
- And a system I called "triggers" which were various event based plugins that would enable users to run their pipeline based on some external event like the passage of time.
- And lastly, the scheduler, which was the container orchestrator Gofer interfaced with to actually run the container.

From the onset the plugin system had a bit of a learning curve. How Hashicorp's go-plugin system worked was that it created a new process to house the plugin code and then communicated with that code through a RPC interface. I found myself struggling to make it easy for developers who weren't familiar with the system already. While that is a solvable problem I found the other problems not quite as solvable. To "install" a plugin the main binary would need to download the repo, compile it with Go, and then attempt to start it up. Initially I found the security implications of this a bit worrying, but over time it was the issue of the operator's experience that really did me in. I found orchestrating this process a bit cumbersome and hard to explain why this had to be done with people who are unfamiliar with Hashicorp's system. This did not coincide with the vision I had for plugins which are written in any language and soon I abandoned go's plugin architecture for something a little more general.

6.3.2. Third try: Everything is a container!

After souring on the implementation of the previous plugin system it was time to take a step back and weigh other options. There were some steps in-between, but the next major idea was the prospect of everything being a container. This mantra sounded good in my head, almost a nod everything is a file in Linux. Plugins being containers have some real-world, concrete advantages:

- It mirrors the advantages of the RPC implementation go-plugin has.
- You can write the plugin in any language that can implement the API (which is in REST).
- Gofer is already familiar with how to run and manage containers, making the plugin system just launch containers keeps things standardized.
- There isn't any need to attach the Go compiler along with Gofer anymore just to install plugins easily.

This seemingly solves the issue with go-plugin and providing a consistent operating model to how Gofer works (It just runs containers!). Next, I went back to the drawing board and made sure I was taking the opportunity of a refactor to consider all parts of the trigger system. I realized that triggers were only one piece of the puzzle for what is essentially a plugin system for pipelines. A pipeline might want to do more than just be notified when to run, it might want to report its run somewhere like telemetry or simply in chat. Or maybe a pipeline would like a repository automatically loaded into Gofer's object store before it runs. This caused me to rename "triggers" to "*extensions*".

When the Gofer control plane starts up it launches these "extensions" and can subscribe pipelines to those extensions with information the pipeline provides. This new design means that extensions are fairly flexible in what they can do and since they can communicate directly with the Gofer API they have a large amount of data they can work with to perform various actions.

6.3.3. Bonus: The joys of the plugin system

The not so openly stated goal of the plugin system was simply to not be as bad as Jenkins. To this end it was important to evaluate what parts of the system needed to be extendable, how easily they needed to be extendible, and what that interface would look like. It also needed to be somewhat secure. If you were going to run chunks of code from places you're operating on faith haven't included a trojan somewhere, it at least better have a modicum of protections around it. This was the advantage of the container model.

Based on your risk profile you could in theory make your container orchestrator something that uses a more strict workload runtime(something like Firecracker). This means that your pipeline extensions could run in a completely isolated environment and since plugins simply communicate with Gofer through its API, there isn't much villainous remote code execution that could go on under your nose.

This accounts for some of the issues with Jenkins plugins, but just because you run sketchy code in a Firecracker VM doesn't mean all your problems are solved. The nature of Gofer's relationship with these plugins helps a bit in that regard, but whether or not there are protections in place doesn't mean that people want to use buggy, exploit-filled code within their environments.

This is a problem I'm not sure can be solved by Gofer, at least not to completion. The plugin system can be thought of as a very close relative to a programming library. Bits of code, from other developers (whom you often have no idea about), that you want to combine with your bit of code to do something useful. When viewed through this lens, it would suggest that the solutions we use in that paradigm would also apply to this one. Above we've addressed some of the strategies around "okay, when this does run/compile how do we mitigate vulnerabilities from that", but this doesn't address the question of "how do we have some system of trust before we get to that point?". The only way to seemingly address this is to follow the same patterns as our programming language friends; we create a community!

The Vision:

- Gofer maintains a certain subset of extensions as first-class citizens. These are high quality, trusted, and additional to what Gofer already provides. It can be thought of as the standard library of extensions.
- Gofer attempts to start a community around other extensions that are available, where people can register their extensions. This might be a separate site where people can willingly register their extensions and Gofer will keep metrics on said extensions.
- If you register your extension, Gofer can record where that extension lives and run some security static analysis on it beforehand. Since the extension must conform to Gofer's interface, it should limit the amount of crazy things extension authors can do.
- This hopefully will create a subset of second-class extensions, which based on common software heuristics (For example the repo author or number of downloads) people can feel somewhat confident that their extension choice from this pool is pretty safe.

This will leave a large amount of extensions that aren't in these first two pools, but those pools will also cover the 80% case. For the 20% cases, only using extensions that are open source should give security teams a way to double check what is entering their environment. If you don't trust the author of a "third-class" extension, you could even fork it and then maintain it yourself.

Without going too deeply into rehashing the advantages of containers and a standardized API, the container model is also nice due to how easy it is to create more plugins. You write code against a defined REST API in whatever language you want -> you wrap it in a docker container -> you tell Gofer about that docker container -> That's it!

Gofer provides an SDK that helps with this process and makes it so you only have to care about the business logic of your extension. Furthermore, the extension system means that once your pipeline is subscribed to an extension, that extension could presumably do a great many things to make your life easier. Extensions can query the Gofer API for information, so that means that it's possible to make an extension that does something based on the overall state of your pipeline over many runs. Anything is possible. And "*the sky's the limit (with guardrails)*" is what any successful distributed cron system needs.

6.4. Declarative Triggers

Partially related to the previous modularity section, triggers were more of an interesting feature than I first gave them credit for.

One of the key parts of Gofer is its ability to use events to trigger specific pipelines on the user's behalf. Without this mechanism, a remote code execution platform really isn't much use. So it goes without saying that a lot of time was spent around how to allow users to execute their pipelines in various ways that were all easy to manage.

Initially I came up with the "Trigger" system. Here is how it worked:

1. A user would declaratively define the trigger they wanted in their pipeline manifest, in code.
2. They would then submit that manifest to Gofer, in which case Gofer would attempt to "subscribe" that trigger to that pipeline.
3. This subscription process was simply Gofer communicating (via API) to the container to tell it "Hey, whenever you have an event with x settings send us back a request to run y pipeline".
4. When the trigger had a relevant event it simply sent a request back to the Gofer master to say "hey you should run this pipeline for this reason".

The initial idea to define the trigger within the manifest was due to wanting to get as many benefits out of the paradigm of Gitops. Anything your pipeline wanted to do differently was always defined within that file, which was very beneficial for the standard Gitops reasons.

The idea quickly broke down when faced with the challenge of how I was going to meld this declarative system (whose backend process was over the network and anything but declarative) with my ideas around reliability.

One execution case seemed overwhelmingly hard to solve though:

1. A user configures their pipeline with a trigger that says "run this pipeline every 5 mins".
2. The user hits submit and Gofer subscribes the pipeline to this trigger.
3. Some time passes and the user realizes that they want to try a different trigger or maybe they want to change their current trigger.
4. Sadness occurs.

The sadness lies in thinking through the process of a canary deployment under these circumstances. What is the correct process here? Do we unsubscribe you immediately from your old subscriptions and try to immediately subscribe you to the new ones? That seems not in the fashion of a canary deploy at all. But if we don't do a cut and dry unsub -> sub model then your pipeline will actually be subscribed to multiple triggers at the same time and those triggers have no knowledge of your previous subscriptions. So you might have changed 5 mins to 10 mins, but if we were truly canarying in a true sense of the process you'd have multiple runs going off all at different times because multiple triggers ran. Definitely not what a normal user would expect.

The problem lies in the framing of how each system thinks about what a pipeline is. Gofer itself thinks about pipelines in versions, this enables it to track changes and provide the reliability features. Triggers need to think about pipelines as singletons, this is important because it's also how users think about pipelines. They are a single workflow or unit of code which is expected to be run as a cohesive unit with certain guarantees. Breaking this paradigm wouldn't make much sense when paired with the principle of least surprise.

This realization of incompatibility caused me to rethink how triggers should be defined. My first thought was preservation; how could I make this system still work? Can I update the trigger interface such that they have a better grasp on how things should work when a pipeline is midway through a canary process? This didn't seem likely, the logic for how a trigger should intelligently understand which pipeline to run and when seemed non-deterministic. Then I thought about just keeping the abrupt switch of triggers intact. Even during a canary, when you uploaded a new version which altered the trigger settings we would unsubscribe you from all of them, only to resubscribe you with the correct settings. This didn't make sense in a world with canaries either, if we immediately removed your old triggers when you made a configuration change, then we wouldn't be able to appropriately canary your pipeline. If the successful run of an older version was in any way tied to those triggers then the old version would fail immediately, not only would this cause a canary that provides less stability, but there isn't any guarantee that we could rollback and restore these settings.

6.4.1. Simplicity is hard and consistency is important.

The only appropriate move here seemed to be to return power to the user so they are able to choose their destiny. Since the pipeline manifest represented objects and settings that were subject to the reliability features of Gofer I figured that to remove confusion I would remove their inclusion within the manifest file entirely.

Instead the user defines and submits their pipeline manifest and then once that is verified by Gofer they can assign triggers(or rather by this point in time "extensions") to their pipeline through the command line. This detracts away from the ultimate user experience that would be the ability to simply define extension settings at the top of your pipeline manifest and have them automagically work, but it gives the user the ability to choose how extensions should be handled for a new version of their pipeline.

6.5. Caching

Since remote code execution platforms can run just about anything it is appropriate to reason that some of this code may benefit from a bit of statefulness. The issue is the mere mention of statefulness goes against some of the early ideas for how container orchestrators should work. The killer feature of throwing containers around is that, well, you can throw containers around. Which is made much more difficult if those containers have some state you constantly have to account for. Once containers start having state then they transition away from their intended goal of cattle and become pets again.

Not good.

The hard part is that a distributed cron platform without some level of caching is kind of wasteful and slow. The reason being is that these platforms often just run the same small bits of code over and over with different inputs. As such, a low hanging example would be the utility of caching when you create a pipeline that just compiles your code for distribution. Anyone who has written anything in Rust will realize that without some type of build caching you'll waste a lot of CPU cycles downloading the same libraries just so you can compile them again just like you did the first time around. Better, would be to

save the incremental builds Rust gives you for free somewhere, so the next run takes 15 seconds instead of 15 minutes.

6.5.1. So okay, caching is a difficult but table stakes feature, what do we do about it?

This is one of those pervasive features that Gofer's goals and values made difficult to achieve. Gofer accounts for the fact that the end user will run this on almost any container orchestrator and due to this freedom it's somewhat difficult to give a worry free, natural mechanism for caching.

6.5.2. The first solution attempted to use FUSE

At first I thought about using the solution closest to what most other providers offer. Docker/ OCI supports the ability to attach volumes and most CI/CD systems will have a certain amount of workers that your container runs on. If you control the worker layer like this then you can simply mount a consistent volume the container declares before the container starts. If your worker pool doesn't shift around often you might even keep the mounts alive when the container has run on a worker at least once. It's also possible for this model that you might implement a copy-on-write mechanism due to your ability to control the layer between storage writes from the container to the actual device.

I've thought about this a lot, since it gives the best user experience. Through the glory of FUSE you declare a mount you want and then Gofer figures out how to give you that mount point connected to your container from wherever you want it. Within your container you just treat that mount point as if it was just another directory and it acts as such.

The problem with this is Gofer isn't in control of its workers. Gofer just trusts that whichever container orchestrator you give it can run containers and that's about it. Gofer doesn't require you to install a worker process on the downstream system and as such it's harder to orchestrate something like the above without passing that difficulty on to the Ops teams supporting this. I thought about possibly allowing users to use a specific container that already has the FUSE implementation within it and that would allow containers to auto-mount things. Then Gofer provides an object store which can serve as the storage layer to be mounted automagically. The problem with this is it's a bit too restrictive. Part of product design is not surprising your users with weird implementations of ideas that are consistent everywhere else. The prospect of maintaining a special container that is the only way to receive this functionality didn't seem like good design. Additionally, it turns out that network storage solutions are just really hard.

6.5.3. Then I decided none of that will work

Like the solution to modularity I decided to put a little more work on the user side to avoid headache on the ops side of things. The final solution was leaving the container layer alone and instead implementing a caching mechanism a little higher up the stack. On every run of a pipeline the user can elect to have Gofer inject an environment variable called `GOFER_API_KEY` into each run(An opt-in feature for each pipeline). This key gives a 'pipeline run' access to the Gofer API.

This enables users to handle caching on their own using the Gofer CLI and Gofer's object store. The user is meant to use the Gofer CLI, with the included key, to orchestrate their container into handling the restoring of files they may need before their run starts.

This is categorically a worse feature from a user experience standpoint than simply declaring a mount and using normal POSIX operations. You can feel this when creating your individual pipeline steps; with this system you might need to repeat the action of pulling a cache many times over an entire pipeline as Gofer cannot simply just run all tasks on one worker and keep the mount point intact. The hope is that

it makes up for those shortcomings in raw simplicity and flexibility. The modularity of Gofer means that your object store (if it doesn't already) can have a cache in front of it and implement the same bandwidth/time saving mechanisms as any other platform.

Suffice to say, it's a work in progress feature. Right now, on current workloads, it seems to be working, but I could see any significantly evolved company realizing that this model won't work for them. The sheer amount of bandwidth that would be used in transferring the same files to a container over and over again would be a deal breaker. It also breaks Gofer's goal of allowing you to test locally easily. Under this model users would have to account for this by making sure their test container has specific mount points emulating the Gofer object store calls. It's something I'm not fully happy with and I'm keeping my eye out for other solutions.

6.6. Language Agnosticism

6.6.1. Stepping away from Gitops

A well beaten horse at this point, it was important that Gofer move away from the common mechanism of declarative pipelines written in custom configuration languages. To avoid rehashing the above arguments for why configuration languages are bad I'll simply say here that, from inception, Gofer's plan was to have users write pipelines in an actual programming language. The vision was for the user to spend more time in areas of familiarity rather than learning the ins and outs of yet another custom YAML DSL.

This smuggles in another departure from other tooling that Gofer makes a point of. Gofer isn't Gitops based like most other tooling in this space. This is an important distinction since the model of Gitops brings lots of advantages for developers, namely there is very little "drift" in what is checked into a branch and what is running currently. Despite this, as mentioned in the point on reliability features, I felt that Gitops wasn't quite flexible enough to achieve the control one would want for storing their distributed cron code within their repository and also making sure that code is deployed exactly when it's ready. The reliability features Gofer attempts to achieve needs the ability for the deployer to make changes and then test and release those changes in a manner they deem fit. This flexibility of release style means that implementing Gitops would be somewhat difficult here, as Gofer will often need to manage multiple versions of the pipeline at the same time. Performing these releases while conforming to proper Gitops is technically possible, but would be somewhat complicated and hamper some of the real time choices that Gofer's users benefit from.

6.7. Implementation

To this end, the design for allowing users to write in any language is having some standard that all languages can "compile" to. For this we need some common serialization format. My first thought just like everyone else was to provide an SDK in each language supported, that would enable users to simply import a library and start building pipelines immediately. The SDK would provide functions which all resolved to the serialization format of JSON. The serialization step for each pipeline would be handled by the user's local Gofer CLI and then that CLI would send the Gofer control plane the pipeline represented by this `JSON`.

There are some really cool pros here, the best of which is that once you have a common serialization language anyone can generate tooling to write/store their configs how they like. There is probably one person that really likes everything in YAML and that means with a minimal amount of work, that person can translate Gofer config mechanisms into YAML and manage their Gofer pipelines with YAML alone. Yay flexibility! (Boo `YAML`)

The drawbacks of a language agnostic design is in its maintainability. While Gofer the control plane doesn't quite care what language you implement your pipeline in, it takes what was previously a single point of maintenance (a single configuration language) and turns it into something that now has several points. Gofer can always declare that it will only support something like the top ten most popular languages, but that is still ten different languages that Gofer has to make sure appropriately works with every feature of pipelines.

Most companies tackle this by utilizing language generation based on golden files of what each SDK should provide. The idea being that I can create a single file that appropriately describes what I need in each language and then some other tool can automatically generate those functions in each language. This is something I need to do a bit more research on, but for the time being Gofer simply severely limits which languages you can use currently. Maintaining the supported Rust and Go SDKs isn't a terrible amount of work just yet, but even these two libraries have significant differences with how they are implemented and changing things within the two is extremely error prone.

Nevertheless, this language agnostic system works beautifully. Not attempting to rewrite language tooling like testing or formatting is a huge time save on Gofer maintenance and prospective users seem more excited about maintaining pipelines which are in their programming language of choice.

7. The Myth of the Catfish and the Cod

There is a good (but unsubstantiated and probably very untrue) story that is told sometimes about fishermen and cod.

As the story goes:

> *Long ago, fishermen used barrels to ship freshly caught cod to consumers.*
>
> *But when recipients would crack open the barrels, the fish were often mushy and tasteless.*
>
> *Confused, a few smart fishermen came to the realization that packing the cod into the barrel caused them to slow their movement on the long journey to their destination. Being in this state of dormancy for long periods of time caused the fish to lose muscle definition and in turn their flavor and firmness.*
>
> *But what to do?*
>
> *The fishermen still wanted to supply far away places with their fish, but nobody would pay for the quality of cod they received by the end of the barrel's journey.*
>
> *In a stroke of genius, these clever fishermen came up with a novel solution.*
>
> *They would throw a few catfish into the barrel along with the cod!*
>
> *The catfish would never be able to eat the entire barrel of cod, but the fish, now preoccupied with survival, would maintain their firmness and tastiness over the long voyage.*

This story is often told to remind us that sometimes we need someone to be a catfish to us, making sure we're continuously improving, maintaining our edge and musculature instead of growing complacent.

To extend the metaphor, Gofer is a catfish for distributed cron systems. It tests the bounds of what has become familiar to us over time. These familiarities, like the small rock in your shoe, isn't exactly what we want, but are bearable enough that we can keep striding towards our real destination. Distributed

cron systems are very rarely anyone's true destination, but they are a very important part of the vehicle you use to get to that destination. And because of that fact they deserve to be attended to with more rigor than we have in the past.

The future of Gofer is firmly planted in making sure the mettle of these familiarities are tested often and complacency doesn't creep into the tooling we depend on. Hopefully, providing the chase our short term job executors need to get to their destination full of flavor.

Thanks for reading!
