

[Y](#) **Hacker News** [new](#) | [threads](#) | [past](#) | [comments](#) | [ask](#) | [show](#) | [jobs](#) | [submit](#)[maliker \(1087\)](#) | [logout](#)

Ask HN: Who operates at scale without containers?

584 points by disintegore 2 days ago | [flag](#) | [hide](#) | [past](#) | [favorite](#) | [429 comments](#)

In other words, who runs operations at a scale where distributed systems are absolutely necessary, without using any sort of container runtime or container orchestration tool?

If so, what does their technology stack look like? Are you aware of any good blog posts?

edit : While I do appreciate all the replies, I'd like to know if there are any organizations out there who operate at web scale without relying on the specific practice of shipping software with heaps of dependencies. Whether that be in a container or in a single-use VM. Thank you in advance and sorry for the confusion.

[add comment](#)

▲ [smilliken](#) 2 days ago | [next \[-\]](#)

My company runs without containers. We process petabytes of data monthly, thousands of CPU cores, hundreds of different types of data pipelines running continuously, etc etc. Definitely a distributed system with lots of applications and databases.

We use Nix for reproducible builds and deployments. Containers only give reproducible deployments, not builds, so they would be a step down. The reason that's important is that it frees us from troubleshooting "works on my machine" issues, or from someone pushing an update somewhere and breaking our build. That's not important to everyone if they have few dependencies that don't change often, but for an internet company, the trend is accelerating towards bigger and more complex dependency graphs.

Kubernetes has mostly focused on stateless applications so far. That's the easy part! The hard part is managing databases. We don't use Kubernetes, but there's little attraction because it would be addressing something that's already effortless for us to manage.

What works for us is to do the simplest thing that works, then iterate. I remember being really intimidated about all the big data technologies coming out a decade ago, thinking they are so complex that they must know what they're doing! But I'd so often dive in to understand the details and be disillusioned about how much complexity there is for relatively little benefit. I was in a sort of paralysis of what we'd do after we outgrew postgresql, and never found a good answer. Here we are years later, with a dozen+ postgresql databases, some measuring up to 30 terabytes each, and it's still the best solution for us.

Perhaps I've read too far into the intent of the question, but maybe you can afford to drop the research project into containers and kubernetes, and do something simple that works for now, and get back to focusing on product?

[reply](#)

▲ [tharne](#) 2 days ago | [parent](#) | [next \[-\]](#)

> What works for us is to do the simplest thing that works, then iterate.

The older I get, the more often I'm reminded that this un-sexy approach is really the best way to go.

When I was younger, I always thought the old guys pushing boring solutions just didn't want to learn new things. Now I'm starting to realize that after several decades of experience, they simply got burned enough times to learn a thing or two had developed much better BS-detectors than 20-something me.

[reply](#)

▲ [tra3](#) 2 days ago | [root](#) | [parent](#) | [next \[-\]](#)

Choose boring technology [0].

For me, the choice is a trade off between the journey and the destination.

Destination is the final objective of your project, business value or whatever.

The journey is tinkering with "stuff".

Depending on the project there's value and enjoyment in both.

[0]: <https://mcfunley.com/choose-boring-technology>

[reply](#)

👤 mateusz 2 days ago | root | parent | next [-]

> Chose boring technology

Doesn't fit in this case, I'd say Nix is still cutting edge.

It may also be simple, but it's not easy (in the Rich Hickey sense).

[reply](#)

👤 spicybright 2 days ago | root | parent | next [-]

Can you elaborate more? From what I've generally heard, Nix has been a good foundation for a lot of companies.

[reply](#)

👤 mateusz 2 days ago | root | parent | next [-]

As far as I know the final effect is great - getting reproducible, statically linked programs.

However, learning Nix / NixOS is quite difficult - it requires a specific set of skills (functional programming, shell, Nix building blocks which you can learn from existing Nix code), the documentation is lacking, error messages are cryptic, debugging support is bad.

[reply](#)

👤 Ericson2314 2 days ago | root | parent | next [-]

Yeah Nix is why "choose boring" is a bit of a misnomer.

The truth is tautologically unhelpful: choose *good* technologies.

Most people have trouble separating snake oil and fads from good things, so old/boring is a safe heuristic.

Nix is not boring, but it is honest. It's hard because it doesn't take half measures. It's not an easy learning curve gaslighting to you to a dead end.

But taste in choosing good technologies is hard to teach, especially when people's minds are warped by a shitty baseline.

[reply](#)

👤 tharne 2 days ago | root | parent | next [-]

> It's not an easy learning curve gaslighting to you to a dead end.

This is a really good way of describing a certain type of technology. I'm stealing this line.

[reply](#)

👤 Ericson2314 2 days ago | root | parent | next [-]

Please do! In this current era of VCs loving developer tool businesses, we are seeing a whole lot of this!

[reply](#)

👤 mountainriver 2 days ago | root | parent | prev | next [-]

Thank you, I've always hated the Boring Technology idiom. The right tool doesn't have to be boring, but it certainly can be

[reply](#)

👤 charcircuit 2 days ago | root | parent | prev | next [-]

Nix isn't a good option for developing rails applications. The nix ecosystem doesn't have good support for ruby meaning you will waste a lot of time and run into a lot of issues getting stuff to work. That's not boring. An example of being boring would be using Nix for something like C where the ecosystem around it is already built up.

I was really pro Nix at the time I took on a rails project and it caused many problems for my team due to Nix just not having good support for my use case. If I could go back in time I would definitely choose something else.

[reply](#)

▲ Ericson2314 2 days ago | root | parent | next [-]

Please check out <https://github.com/NixOS/rfcs/pull/109> !

I absolutely agree language-specific package managers need better support. Nix RFC 109 is a simple-stupid way to get some "important from derivation" in Nixpkgs, so we can collectively dogfood them "lang2nix" tools and finally make them good.

It is really sad to me that many users understandably think "hmm, C is crazy, surely this Nix should be even better with more uniform language ecosystems packages!", only to die on the hell of the lang2nixs all being just *not quite good enough*. This is huge stumbling block for development shops that try to use Nix that I want to see removed!

[reply](#)

▲ dqv 2 days ago | root | parent | prev | next [-]

What was the problem you were having with Ruby (on Rails) and Nix? It wasn't ergonomic for me, but I was able to get things to work.

[reply](#)

▲ charcircuit 2 days ago | root | parent | next [-]

I think one of the main issues was depending on a custom gem. And then something about sandboxing, and outdated things being cached (maybe this was just a rails thing though). Also setting up rails project was convoluted. It was just obvious that there wasn't much time devoted towards that use case of Nix.

The rails support was more of just to handle packaging existing software as opposed to being good for a development environment.

[reply](#)

▲ debaserab2 2 days ago | root | parent | next [-]

What do you mean exactly? At the end of the day, rails is just a collection of ruby gems. If nix has first class support for ruby and Gemfiles in theory there shouldn't be a ton of problems.

I've not used Nix so I legitimately don't know (but I am quite interested in it)

[reply](#)

▲ Ericson2314 1 day ago | root | parent | next [-]

There are no fundamentals hurdles, but because we are currently prevented from properly dogfooding these tools in Nixpkgs, they tend to be unpolished in various ways, they are just used to by a few random projects in relative isolation.

[reply](#)

▲ charcircuit 2 days ago | root | parent | prev | next [-]

I don't remember exactly the problem since it was a couple years ago. There was a lot of issues in regards to failing to run / build rails which regarded doing special nix stuff to fix. It wasn't possible to include a custom gem from a separate directory. I had to disable some sort of sandboxing to prevent Nix from complaining. Nix kept getting in my way and as my team members were not familiar with Nix they couldn't fix it themselves. Sadly, I ruined their opinion of Nix by showing them it with this project.

[reply](#)

▲ ParetoOptimal 2 days ago | root | parent | prev | next [-]

The problem is that programming is about managing complexity.

Typically, technologies I see described as boring have poor features for managing complexity.

Usually with poor reasoning justify meeting some silly definition of "simple" and "boring" that ultimately makes your life harder.

[reply](#)

▲ 0x20cowboy 2 days ago | root | parent | prev | next [-]

Jonathan blow had a great take on this that really spoke to me. I can't do it justice, but paraphrasing as best I can:

Get the simple to understand, basic thing working and push the harder refactor / abstraction until later. Leave the fancy stuff for a future engineer who better understands the problem.

That future engineer is you with more experience with the actual problem.

[reply](#)

▲ xmprt 2 days ago | root | parent | next [-]

The way I see it, I can either prematurely use the complex solution where I don't fully understand the problem or the solution OR I use the simple solution and learn why the complex solution is needed and more importantly what parameters and specific ways I can best leverage the complex solution for this specific problem.

[reply](#)

▲ chousuke 2 days ago | root | parent | next [-]

Simple solutions are also easier to refactor into more complex solutions when the complexity becomes necessary. Going the other way is *much* harder.

In simple systems, you often have the option of simply throwing away entire components and remaking them *because* the system is simple and the dependencies between components are still clear.

[reply](#)

▲ toast0 2 days ago | root | parent | prev | next [-]

Sometimes the simple solution works well enough for as long as you need a solution, too. Often times, it just works, or the problem changes, or the company changes.

[reply](#)

▲ mikepurvis 2 days ago | root | parent | prev | next [-]

This works well if you know you're building it all yourself regardless. I think the calculus does change a bit though if you have frameworks available in your problem domain.

Like, there's no point in scratching away at your own web CMS just to prove to yourself that you really do need Rails or Django. Especially when part of the purpose of a framework is to guide your design toward patterns which will fit with the common, preexisting building blocks that you are likely to end up needing. If you do the design separately, you risk ending up in a place where what you've built isn't really suited to being adapted to other stuff later.

For a humble example of this from myself, I built an installer system for Linux computers— initially very simple, just a USB boot, perform partitioning, unpack a rootfs tarball, and set up the bootloader. Then we

added the ability to do the install remotely using kexec. Then we added an A/B partition scheme so there would always be fallback. It got more and more complicated and fragile, and the market around us matured so that various commercial and OSS solutions existed that hadn't when we first embarked on this effort. But the bigger and less-maintainable the homegrown system became, the more features and capabilities it had which it made it difficult to imagine ditching it in favour of something off the shelf. There isn't really a moral to this story other than it would be even worse if we'd built the in-house thing without doing a thorough evaluation of what was indeed already out there at the time.

[reply](#)

👤 oblio 2 days ago | root | parent | prev | next [-]

> That future engineer is you with more experience with the actual problem.

With the average tenure of a software developer at a company being about 2 years, that future engineer is for sure some other poor soul.

[reply](#)

👤 caffeine 2 days ago | root | parent | prev | next [-]

Being really good at using boring tech is better than being a noob with cool tech.

So as you get older and you get more competent, the relative benefits of boring tech grow.

Whereas if you're a noob at everything, being a noob with cool tech is better.

If the problems are hard enough, and you are competent, you iterate on boring tech for a while to solve the hard problems, and then it becomes cool tech.

[reply](#)

👤 asiachick 2 days ago | root | parent | prev | next [-]

Hmmm, me, I want batteries included. Why I don't I have continuous deployment? Because that would require work doing something I don't have time to do. Why don't I have a staging server? Because more work. Why don't I have backups. Again, more work. Why don't I have metrics so I can see where my bottlenecks are? More work. Why don't I have logging? More work. Why don't I have a way to do updates without taking the servers down? No time, more work, too complicated.

Sure I can write a server in 5 lines of code but adding all that stuff is many many many person years of work and plus I'd have to go through all the learning pains of doing it wrong, crashing things, losing data, before I'd add it correctly.

I want some existing solution that domain experts already created that do things right.

It has nothing to do with "sexy/unsexy". It's unsexy to use UnrealEngine. It's sexy to write your own. But unsexy ships while sexy is spending time recreating stuff that already exists, unsexy is moving on to the actual content and shipping.

[reply](#)

👤 fpoling 2 days ago | root | parent | prev | next [-]

If a solution has been working for 20 years, then by Lindy effect there is a good chance it will be relevant in 20 years as well. Basically an old but still actively used software or principle implies that it has being compatible with a lot of new things in past and, as such, is rather universal.

With new software one does not know if it is the one that will survive challenges from even newer software.

[reply](#)

👤 kitd 2 days ago | root | parent | prev | next [-]

The simplest thing for "dev" is not necessarily the simplest thing for "ops" though.


I'm in my late 50s and been developing for 35 of them. Honestly, containers are not rocket science and solve a mountain of ops-type problems *simply*.

If you really are only developing a single instance to be deployed in-house, then you may find you don't need them. But as soon as your instance might possibly get exposed to more than one environment, they're a no-brainer for me.


[reply](#)

👤 api 2 days ago | root | parent | prev | next [-]

[https://www.ariel.com.au/jokes/The Evolution of a Programmer...](https://www.ariel.com.au/jokes/The_Evolution_of_a_Programmer...)

[reply](#) ironmagma 2 days ago | root | parent | prev | next [-]


The thing is, "simple" is not trivial to discern. What looks simple to a beginner may look quite hard to maintain to an experienced dev. "GOTO" being the obvious example.

[reply](#) usaphp 2 days ago | root | parent | prev | next [-]


Or it could be that you just became old and don't want to learn new things anymore))

[reply](#) rattlesnakedave 2 days ago | root | parent | next [-]


This certainly seems more common. Every time I've had to refactor 20 year old cruft it's been because of engineers with the launch-and-iterate mentality 20 years ago, that stopped caring about the "iterate" part once they gained enough job security.

[reply](#) hkt 2 days ago | root | parent | next [-]


Blaming the engineers rather than the businesses seems.. odd. It isn't like the people writing that code had control over what the company set as priorities. If anything, what you experienced seems to be a reflection of corporate short termism.

[reply](#) dylan604 2 days ago | root | parent | next [-]


Quite right. The devs probably had a bunch of "TODO: This can be refactored" through out the code or at least mentally stored, yet the pointy haired bosses decided it was good enough and re-assigned the devs to other tasks.

[reply](#) ironmagma 2 days ago | root | parent | prev | next [-]


That dichotomy falls apart because engineers are part of the business. If the business wants to do something but the engineers won't do it, it won't happen.

[reply](#) basisword 2 days ago | root | parent | prev | next [-]


Or their experience helps them have the wisdom to determine which new things are worth learning and which are a waste of time.

[reply](#) CuriouslyC 2 days ago | root | parent | prev | next [-]


Some old things really are better, or at least not worse and already familiar.

[reply](#) goodpoint 2 days ago | root | parent | prev | next [-]


Such "new things" are just a big bunch of unnecessary complexity.

[reply](#) repomies69 2 days ago | root | parent | next [-]

Sometimes yes. But it is impossible to know for sure. New things still get adopted, some of those new things actually stick and provide value

[reply](#) spicybright 2 days ago | root | parent | next [-]

And at the very least, being old lets you be able to evaluate these new things more accurately.

[reply](#) goodpoint 2 days ago | root | parent | next [-]

It obviously does, but experience is not the only way: you can always read what people wrote and did in the past. It only takes some humility.

[reply](#)

👤 selcuka 2 days ago | root | parent | next [-]

You can, and it will certainly help, but as the old saying goes "unfortunately no one can be told what the Matrix is, you have to see it for yourself". In some contexts no amount of reading will give you the insight that experience does.

[reply](#)

👤 spicybright 1 day ago | root | parent | next [-]

And vice versa, too (granted my own learning style leads me to implementing what I read in little test projects, but I digress)

Overall though, I'm finding the more unique sensory info you give your brain, the better it gets at solving problems in general. Even if that info isn't very related to your goal.

It's a bit like trying not to be a hammer that sees everything as a nail.

[reply](#)

👤 rattlesnakedave 2 days ago | root | parent | prev | next [-]

How?

[reply](#)

👤 ParetoOptimal 2 days ago | root | parent | prev | next [-]

> Such "new things" are just a big bunch of unnecessary complexity.

By definition the "old things" you use would also be "just a big bunch of unnecessary complexity" since they were once new.

[reply](#)

👤 rattlesnakedave 2 days ago | root | parent | prev | next [-]

Or is cope for a loss of neuroplasticity? Has to be evaluated case by case. "Experience" doesn't count for much when evaluating new technical tooling. Landscape shifts far too much.

[reply](#)

👤 tomc1985 2 days ago | root | parent | next [-]

The landscape doesn't need to shift nearly as often as it does. But we are an industry obsessed with hiring magpie developers as cheaply as we can, and those magpie developers demand cool merit badges to put on their resumes, and here we are.

"Everything old is new again" does not BEGIN to do tech justice. Compared to nearly every other profession on the planet, tech knowledge churn is like 10x as fast.

[reply](#)

👤 goodpoint 2 days ago | root | parent | next [-]

> tech knowledge churn is like 10x as fast.

Spot on! Moreover, every time the wheel is unnecessarily reinvented previous lessons are lost.

What is even worse is that people really want to reinvent the wheel and get defensive if you point that out.

Comments like "Or is cope for a loss of neuroplasticity" are a good example.

[reply](#)

👤 tomc1985 2 days ago | root | parent | next [-]

It's outright naked ageism.

I have been studying a different trade, completely unrelated to software engineering or tech, and by far the weirdest thing is reading books or watching seminars from 20 or 30 years ago that *are still relevant*. How much of

technology writing has that honor? Very little.

Like I have been obsessed with tech and computers my whole life, I studied computer stuff when all the other kids were outside playing football or chasing girls or whatever, I self-taught to a very high level of expertise and that knowledge has just been getting systematically ripped away. Which is fine, to an extent, gotta embrace change and yada yada yada. But I no longer have the patience to grind to expertise-level on something over a period of years only to find out the industry has moved on to something completely different to do the exact same thing! All because some starry-eyed shit somewhere in FAANG doesn't like something!

[reply](#)

👤 iamstupidsimple 2 days ago | root | parent | next [-]

The vast majority of computer science lectures from 20 years ago are still relevant. A ton has changed in terms of frameworks but the fundamentals are mostly the same.

[reply](#)

👤 tomc1985 1 day ago | root | parent | next [-]

One other point (since I can't edit): none of the used bookstores I am aware of take old technical books. Even the good ones (because they don't know and just see another tome that will likely sit on a shelf and collect dust)

[reply](#)

👤 tomc1985 1 day ago | root | parent | prev | next [-]

It is, but it isn't. Besides, from my vantage point almost all the work is herding frameworks of some kind. Maybe I need to get out of web tech and into something else.

[reply](#)

👤 goodpoint 1 day ago | root | parent | prev | next [-]

20?! More like 60 or 70, especially if around distributed systems and Dijkstra's work in general. [1]

It's amazing how some people are familiar with every framework of the month and don't understand how a computer works...

[1]

https://en.wikipedia.org/wiki/List_of_important_publications...

[reply](#)

👤 tomc1985 1 day ago | root | parent | next [-]

Maybe I was in the wrong subsector of this industry but there was very little theory work in the jobs I have held. A lot of it was just reusing the same design patterns over and over again, and then scaling that out somehow.

I did not mean my comment to say that seminal works in CS theory are useless, just that most of the books published in this field cover more practical matters and they tend to age very quickly. Like I have an e-bookshelf full of Packt freebies back when they did that, and those books aren't really that useful now unless I want to start on an old version of something.

[reply](#)

👤 tomc1985 2 days ago | root | parent | prev | next [-]

Such ageism.

[reply](#)

👤 jenkstom 2 days ago | root | parent | prev | next [-]

This is the antivax sentiment of the IT world. It's new! It's shiny! Give me some Ivermectin because I know things other people don't! (Yes, I realize you're probably joking. I'm not, particularly.)

[reply](#)

👤 nurettin 2 days ago | root | parent | next [-]

It's like you're just repeating the last inane discussion YouTube got you hooked on in order to get some ad clicks.

[reply](#)

👤 ThePowerOfFuet 2 days ago | root | parent | prev | next [-]

What does ivermectin have to do with anything? Are you suffering from chronic parasitic infection?

[reply](#)

👤 rattlesnakedave 2 days ago | root | parent | prev | next [-]

You have it reversed. "I don't want the new vaccine! Who knows what the long term side effects are!" is equivalent to the technical curmudgeon. The voluntary early trial test group are the bleeding edge tech people. The Ivermectin crowd is the guy that suggests rewriting the backend in Haskell for no good reason.

[reply](#)

👤 disintegore 2 days ago | parent | prev | next [-]

I asked the question without context because I didn't want to fire the thread off in the wrong direction but I suppose in a comment chain it's fine. For what it's worth we do use containers and K8s heavily at my current job.

I know that there are quite a few people opposed to the state of containers and the technologies revolving around them. I don't think the arguments they present are bad. [Attacking the premise] (<https://drewdevault.com/2017/09/08/Complicated.html>) isn't particularly hard. What I don't see a lot of, however, is *alternatives*. "Learn how to do ops" is not exactly pertinent when most documentation on the subject will point you towards containers.

In addition the whole principle, while *clearly* proven, does strike me as a patch on existing platforms and executable formats that weren't designed to solve the sorts of problems that we have today. While efforts could have been made to make software universally more portable it seems we opted for finding the smallest feasible packaging method with already existing technology and have been rolling with it for a decade.

So essentially I'm interesting in knowing how people find ways to reproduce the value these technologies offer, what they instead rely on, which things they leave on the table (eg "we work with petabytes of data just fine but deploying updates is a nightmare"), how much manual effort they put into it, etc. New greenfield projects attempting to replace the fundamentals rather than re-orchestrate them are also very pertinent here.

[reply](#)

👤 javajosh 2 days ago | root | parent | next [-]

What you're asking for is an essay on comparative devops architectures, with a focus on k8s alternatives. I think what you'll find is a lot of ad hoc persistent systems that tend to drift over time in unpredictable ways, and take on the feel of a public lobby if you're being generous, a public restroom if you're not. So what you're asking is really a sample of these ad hoc approaches. What I think you'll find are a few inspired gems, and about 1000 bad implementations of K8s, Docker, Salt, Terraform and all the rest.

The case that always interested me is the service that can really fit on one machine. Machines are truly gigantic in terms of memory and disk. Heck, I worked at BrickLink which ran on two beefy machines that serviced 300k active users (SQL Server is actually quite good, it turns out). (A single beefy server is a really great place to start iterating on application design! PostgreSQL + Spring Boot (+ React) is a pretty sweet stack, for example. By keeping it simple, there are so many tools you just don't need anymore, because their purpose is to recombine things you never chose to split. I can't imagine why you'd need more than 16 cores, 128G of RAM and 10T of storage to prove out an ordinary SaaS. That is a ferocious amount of resources.)

[reply](#)

👤 DeathArrow 2 days ago | root | parent | next [-]

>I can't imagine why you'd need more than 16 cores, 128G of RAM and 10T of storage to prove out an ordinary SaaS. That is a ferocious amount of resources.)

In my case because I was working for a checkout solution for a Cash & Carry chain, with 1000 stores in

25 countries, each with tens of tills, self service points, queue busting tills who also had to serve the web shop. And that solution had to work also when there was no Internet access.

[reply](#)

👤 javajosh 1 day ago | root | parent | next [-]

Your project is significantly more complex than an 'ordinary SaaS'. My own canonical example of a SaaS would be something like WuFoo, or Squarespace. These are characterized as pure webapp component producing technologies, from soup to nuts, with through-the-web tooling for everything from specializing your components, to managing your payment details. The system is "closed" in the sense that it only produces and consumes web components for all operations. There is no real-world device that is critical to operation - so you are free to run services like these on any device, or collection of devices, as long as its users can find 'a useful stateful service' at 'name'. For your project, this is only the first foundational requirement.

[reply](#)

👤 anonymousDan 2 days ago | root | parent | prev | next [-]

Availability?

[reply](#)

👤 JohnBooty 2 days ago | root | parent | next [-]

Devops is not my area of expertise to put it mildly, but "a single big-ass server" scenarios are often pretty well covered by "a second big-ass server configured as a mirror/hot-spare", right?

Depends on how many nines of uptime you need, of course, and other such things.

(Obviously, there are reams of cases where this *isn't* sufficient)

[reply](#)

👤 senorsmile 2 days ago | root | parent | next [-]

In 2009 I had a pair of (what I thought at the time) very cleverly architected VM servers with complete redundancy between them. Either one could be pulled and there'd maybe be a few seconds of data loss at the worst. One day lightning struck the building and even managed to "jump" the grounded pair of UPS's and fried all hard disks in both nodes.

Thankfully I had both onsite and offsite backups; but it took a couple of weeks to order the new equipment, install it and set everything up again.

Now I "devops" all the things and always have the ability to spin up all the services and restore from backups within minutes. This is the new standard in my opinion.

[reply](#)

👤 exikyut 1 day ago | root | parent | next [-]

Two questions:

1. How would power conditioners and lightning arresters handle these sorts of situations?

2. Was this using VMWare fault tolerant VMs, just out of curiosity? (It continually live-migrates one VM to a hot spare over a 10G link so either can disappear and the VM continues to run.) Or was this a bespoke application architecture implementing effectively the same thing?

[reply](#)

👤 maccolgan 2 days ago | root | parent | prev | next [-]

I think it's always better to have the mirror be located in a separate datacenter and then connect both via a SDN.

[reply](#)

👤 pclmulqdq 2 days ago | root | parent | next [-]

Why not both?

[reply](#)

👤 JohnBooty 2 days ago | root | parent | prev | next [-]

Yeah, there definitely needs to be an offsite spare for sure.

[reply](#)

👤 javajosh 2 days ago | root | parent | prev | next [-]

Why isn't "we're going to risk downtime to speed up our BTM loop for cheap" a good answer?

[reply](#)

👤 NomDePlum 2 days ago | root | parent | next [-]

I've made that very decision.

Partly forced due to internal resource constraints. However, swapping getting a working system out and tested with real users instead of waiting an undefined time to get high availability didn't lose me any sleep.

It's also the case that some systems can withstand a degree of downtime that others can't, or it's not worth paying the cost for the perceived benefit gained.

[reply](#)

👤 SQueeeeeL 2 days ago | root | parent | prev | next [-]

> So essentially I'm interesting in knowing how people find ways to reproduce the value these technologies offer, what they instead rely on, which things they leave on the table

Decades of work typically. That or just standard parallel deployments, most things Docker is good at would take the average developer many unnecessary hours to reproduce.

[reply](#)

👤 pojzon 2 days ago | root | parent | next [-]

Currently there is a multitude of solutions available on the market that make containers and k8s just a complex and hard to maintain choice.

It all boils down to "what bullshit was my management sold on".

Promises of "easy to use, easy to replace, cloud-agnostic, doesn't require domain knowledge" etc etc etc.

It's all bullshit. At some point/scale you need experts in the field that cash in a month more than average dev per year.

[reply](#)

👤 mustache_kimono 2 days ago | root | parent | prev | next [-]

> Attacking the premise isn't particularly hard.

The linked Drew Devault seems to have a glib response for quite a few things. His devs shouldn't build software for users take is just as infuriating. You just want to say "Yeah, I'd rather not, but unfortunately it kinda goes along with writing software. Send 1000 happy users to me to build my software on every platform, because *I'd be glad to let them*. Otherwise, what is your *alternative*?"

[reply](#)

👤 anamax 2 days ago | root | parent | prev | next [-]

> So essentially I'm interesting in knowing how people find ways to reproduce the value these technologies offer, what they instead rely on, which things they leave on the table

It's interesting that you're not interested in what they get by not using containers...

[reply](#)

👤 disintegore 2 days ago | root | parent | next [-]

What? That's the entire point.

[reply](#)

👤 brimble 2 days ago | parent | prev | next [-]

> That's the easy part! The hard part is managing databases.

Ding ding ding ding.

"But what about disk?" (so, relatedly, databases) is the hard part. Balancing performance (network disks suuuuuck) and flexibility ("just copy the disk image to another machine" is fine when it's a few GB—less useful when it's *lots* of

GB and you might need to migrate it *to another city* and also you'd rather not have much downtime) is what's tricky. Spinning up new workers, reliable- and repeatable-enough deployment with rollbacks and such, that's easy, *until* it touches the disk (or the DB schema, that's its own whole messy dance to get right)

[reply](#)

👤 zthrowaway 2 days ago | root | parent | next [-]

People still think stateful things are impossible on k8s but Stateful sets and persistent volumes solves a lot of this. You should be relying on out of the box DB replication to make sure data is available in multiple areas. This is no different on other platforms.

[reply](#)

👤 nijave 2 days ago | root | parent | next [-]

Putting a 50TB+ database that enables eye watering revenue on k8s is a hard sell for a lot of businesses- especially when they have non-containerized solutions that work.

Simple topologies and NoSQL databases (or databases they can handle replication/partitioning/node failures automatically) are pretty easy to stick in StatefulSets.

There's crazy things like this <https://blogs.oracle.com/mysql/post/circular-replication-in-...> that would be fairly difficult to run in a StatefulSet.

In addition, the patch and lifecycle cadence of k8s is pretty quick so rebooting SQL databases with 100ks TPS more than 1-2 times a year tends to be problematic

[reply](#)

👤 koffiezet 1 day ago | root | parent | next [-]

I have a client who's pretty massive database is pretty damn critical, not only for revenue but also legal reasons. It is managed by a separate team. All applications however run on k8s (well, actually openshift), the main reason being organizational scaling of development teams. With the current setup in place, increasing the amount of dev teams we'd need to support would be pretty painless. We do however run certain smaller (<200gb) databases in-cluster where performance is less of a focus, which, because of the storage abstractions currently in place could otherwise become a potential bottleneck.

[reply](#)

👤 evanelias 2 days ago | root | parent | prev | next [-]

> There's crazy things like this <https://blogs.oracle.com/mysql/post/circular-replication-in-...> that would be fairly difficult to run in a StatefulSet.

This isn't a great example to cite, since traditional circular replication in MySQL is a massive anti-pattern... it's incredibly fragile and pretty much has no valid use-case.

That's especially true today when other options like Galera or Group Replication are available. But even 10-15+ years ago, mentioning circular replication was a great way to give a DBA an aneurysm. (Well, I suppose that's arguably a use-case, if you profoundly dislike your company's DBAs...)

[reply](#)

👤 nijave 2 days ago | root | parent | next [-]

Probably a bad example, but the sentiment was: encapsulating years of DBA knowledge and complex enterprise architectures in generic k8s abstractions can be incredibly difficult.

For instance, you put a RDBMS in a StatefulSet. You figure out how to replicate between pods. Now a piece of hardware fails and a replica needs rebuilt--what does that? So you add an operator into the mix. What happens if bin logs have been purged and a replica can't just "rejoin"? Now you have to figure out how to transfer data between pods (PVs) to bootstrap a new replica. Now you probably need some sort of init container solution to keep the pod running and volume mounted while you're copying in data. Now that you have all that, how do you handle master/primary failures and replica promotion? How do you handle backups? How do you handle restores?

Once you've solved all that, how do you performance tune your database? At some point, you're probably going to start looking at kernel tuning which you'll need some additional customizations to enable (probably dedicated node per database pod, then you can either

skip k8s and configure the host directly or wire up allowed sysctls through k8s).

Or, you can skip all that and require human intervention for everything, but now humans need to wade through the k8s abstractions and fix things. With that route, you break basic k8s functionality like restarting pods since it can take down your database topology

[reply](#)

▲ evanelias 1 day ago | root | parent | next [-]

Oh yes to be clear I definitely agree with the overall sentiment. I just wouldn't ever cite circular replication as an example. Basically saying "it's hard to do [inherently flawed terrible thing] on k8s!" just detracts from the argument :)

fwiw my team eventually automated pretty much "everything" for Facebook's database fleet, and it was hundreds of thousands of lines of custom automation code (non-k8s), many years of work for 10+ extremely experienced engineers. In a k8s environment I suspect it would have been even more work.

The open source mysql k8s operators are certainly getting better with time, but there's still a lot of stuff they don't handle.

[reply](#)

▲ DeathArrow 2 days ago | root | parent | prev | next [-]

>Putting a 50TB+ database that enables eye watering revenue on k8s

But you can run the software in k8s while using an external data store for data. With microservices you can also use small databases for each and push the data you want to persist like finished transactions and invoices to external systems.

No reason to have only one giant database to do everything in. At my last place of work we moved from a monolithic software using a giant database to microservices and many small databases, some of them NoSQL.

[reply](#)

▲ nijave 2 days ago | root | parent | next [-]

Parent comment was suggesting you can run databases in StatefulSets on k8s

>No reason to have only one giant database to do everything in

It's the reality for a lot of "legacy" services. Some of these things have 50-100+ apps all connecting to the same DB with incredibly complex schemas, functions, triggers, etc. It's potentially a multi-year project to undo that. When you do that, you're potentially introducing significantly more network hops. If a business transaction spanned multiple pieces of data in the same database before, you've potentially created a saga spanning multiple microservices now.

Even if you do adopt microservices, you'll probably end up consolidating all the data somewhere else for reporting, analytics, data warehousing, etc purposes anyway (so you end up with a giant OLAP DB instead of a giant OLTP one)

[reply](#)

▲ mountainriver 2 days ago | root | parent | prev | next [-]

Yes you can run DBs on kube now, much of people thinking this isn't good comes from years back when it wasn't

[reply](#)

▲ JeremyNT 1 day ago | root | parent | next [-]

Does it buy you much though? The big issue with databases is storage, and you need local storage for optimal performance.

You can ask k8s for a persistent volume on local storage, but at this point you have to treat that pod exactly the same way you'd treat a snowflake database server, because the local storage is what ultimately matters.

You can replicate to other pods with their own persistent volumes, but the replication overhead adds up too both in terms of performance and complexity.

I know there's stuff like vitess and crunchy data, which try to abstract this away, but the amount of layers buried here is very high for benefits that seem kind of nebulous compared to "big ol server" mode.

[reply](#)

▲ senorsmile 2 days ago | root | parent | prev | next [-]

Kelsey Hightower just had a good interview where he disagrees with you:

<https://changelog.com/shipit/44>

Edit: somewhat* disagrees with you. It's a good listen.

[reply](#)

▲ gabrielgrant 2 days ago | root | parent | next [-]

tl;dr?

[reply](#)

▲ bruth 2 days ago | root | parent | next [-]

If you have a low-volume, non-performant, non-critical database, k8s is fine. If you need it to perform and/or need built-in ops (managed backups or replication), use a managed service.

k8s can do stateful, but if a managed service exists for this workload, use it. It is not about can I run it on k8s, it is a should question. Is it worth the cumulative effort required to achieve the same degree of quality.

[reply](#)

▲ hawk_ 2 days ago | root | parent | prev | next [-]

Can you elaborate on "multiple areas"? Do you mean each node inside a db "cluster" should run in a different area? And how does one achieve that?

[reply](#)

▲ fer 2 days ago | root | parent | prev | next [-]

Sometimes I think K8s is largely pushed for Amazon/Google/Microsoft to sell the disk that goes along with it.

[reply](#)

▲ zomglings 2 days ago | root | parent | next [-]

Don't forget network costs (especially for "high availability" clusters spanning multiple regions).

[reply](#)

▲ swiftcoder 2 days ago | root | parent | prev | next [-]

Admittedly it's 5 years since I worked at AWS, but at the time there was basically no internal usage of containers. ECS was pushing customers towards containers, but the the feeling towards containers at the time inside the company could be summed up as "we have a decade worth of operational tooling, why would we throw that all away?"

[reply](#)

▲ rootusrootus 2 days ago | parent | prev | next [-]

> But I'd so often dive in to understand the details and be disillusioned about how much complexity there is for relatively little benefit.

I feel similarly about many cloud services in e.g. AWS. I get where it's sometimes handy, but the management overhead can get pretty insane. I used to hear people say "just do X in AWS and ta-da you're done!" only to find out that in many cases it isn't actually a net improvement until you are doing a *lot* of the same thing. And then it costs \$\$\$\$\$\$.

[reply](#)

▲ JohnBooty 2 days ago | root | parent | next [-]

Yeah, I feel this way a lot.

In recent years I've done work for companies where it feels like the end result of a lot of *very* expensive AWS infra is... more expensive, and not really superior to things we had ten years ago when hosting on-prem.

There's a level of scale for which distributed AWS/Azure is necessary, but I'm not sure how many shops need that.

The devops guy at a previous gig thought I was some kind of naive simp for insisting we didn't need a *cluster* of Amazon's version of Redis. We were storing like, a megabyte of data in there. That's not a typo.

It's not that he was an idiot. Smart guy actually. Not sure if he was just padding his resume with experience ("sure, I've admin'd Redis clusters!") or if he was locked into some kind of default thinking that wouldn't even allow him to consider some kind of minimal setup.

[reply](#)

voidfunc 2 days ago | root | parent | next [-]

The default setup trap thing resonates. Also a lot of DevOps folks I have met are former sysadmins with unflexing mindsets about the way things should be done. It all just culminates as cargo cult "best practices" that bloat your infra.

[reply](#)

throwbigdata 2 days ago | root | parent | prev | next [-]

You are correct

[reply](#)

jmt_ 2 days ago | root | parent | prev | next [-]

Totally agree. Something I find myself thinking/saying more and more is "don't underestimate what you can accomplish with a \$5 VPS". Maybe a little hyperbolic, but computers are really fast these days.

Elastic resources are pretty magical when you need them, but the majority of software projects just don't end up requiring as many resources as devs/management often seem to assume. Granted, migrating to something like AWS, if you do end up needing it, is a pain, but so is adding unnecessary layers of AWS complexity for your CRUD app/API backend.

Also wonder how many devs came up on AWS and prefer it for familiarity and not having to know and worry too much about aspects of deployment outside of the code itself (i.e not having to know too much about managing postgres, nginx, etc).

[reply](#)

barbazoo 2 days ago | parent | prev | next [-]

> Containers only give reproducible deployments, not builds

Could someone elaborate on this please? Doesn't it depend entirely on your stack how reproducible your build is? Say I have a Python app with its OS level packages installed into a (base) image and its Python dependencies specified in a Pipfile, doesn't that make it pretty reproducible?

Is the weak spot here any OS dependencies being installed as part of the continuous build?

[reply](#)

smilliken 2 days ago | root | parent | next [-]

You already got a few good answers, but I'll echo them: you can do reproducible builds in containers, and nothing's stopping you from using nix inside containers. But you're at the mercy of all the different package managers that people will end up using (apt, npm, pip, make, curl, etc). So your system is only as good as the worst one.

I inherited a dozen or so docker containers a while back that I tried to maintain. Literally none of them would build— they all required going down the rabbit hole of troubleshooting some build error of some transitive dependency. So most of them never got updated and the problem got worse over time until they were abandoned.

The reason Nix is different is because it was a radically ambitious idea to penetrate deep into how all software is built, and fix issues however many layers down the stack it needed to. They set out to boil the ocean, and somehow succeeded. Containers give up, and paper over the problems by adding another layer of complexity on top. Nix is also complex, but it solves a much larger problem, and it goes much deeper to address root causes of issues.

[reply](#)

jka 2 days ago | root | parent | next [-]

Do your developers run the same Nix packages that you deploy to production?

(that sounds like a energy-level-transition in developer productivity and debugging capability, if so)

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Yeah, the environment is bit-for-bit identical in dev and prod. Any difference is an opportunity for bugs.

OK, there's one concession, there's an env var that indicates if it's a dev and prod environment. We try to use it sparingly. Useful for stuff like not reporting exceptions that originate in a dev environment.

Basically, there's a default.nix file in the repo, and you run nix-shell and it builds and launches you into the environment. We don't depend on anything outside of the environment. There's also a dev.nix and a prod.nix, with that single env var different. There's nothing you can't run and test natively, including databases.

Oh, it also works on MacOS, but that's a different environment because some dependencies don't make sense on MacOS, so some stuff is missing.

[reply](#)

👤 jka 2 days ago | root | parent | next [-]

That sounds extremely impressive, thank you. I hope to find some time to try out Nix soon (appending it to the list of technologies to learn...).

[reply](#)

👤 kazinator 2 days ago | root | parent | prev | next [-]

Does that mean you turn off security-related randomizations in everything, like address space randomization and hash table randomization?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

No, we have address space randomization and hash table randomization since those happen at runtime. /dev/random works as you'd expect.

The immutability is just at build time. So chrome and firefox aren't able to seed a unique ID in the binaries like you might be accustomed to. Funny story, we had a python dependency that would try to update itself when you imported it. I noticed because it would raise an exception when it was on a read only mount.

[reply](#)

👤 Gwypaas 2 days ago | root | parent | prev | next [-]

How do you manage quick iteration loops?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

We use python. If we were writing in a compiled language, we'd use the same compiler toolchain as everyone else, but with the versions of all of our dependencies exactly the same from nix. We have some c extensions and compile Typescript and deploy those build artifacts. In the case of javascript, our node modules is built by nix, and our own code is built by webpack --watch in development.

[reply](#)

👤 nunez 2 days ago | root | parent | prev | next [-]

I don't know Nix and can't comment on that, but in my experience, when I've inherited containers that couldn't build, this was usually due to its image orphaned from their parent Dockerfiles (i.e. someone wrote a Dockerfile, pushed an image from said Dockerfile, but never committed the Dockerfile anywhere, so now the image is orphaned and unreproducible) or due to the container being mutated after being brought up with `docker exec` or similar.

Assuming that the container's Dockerfile is persisted somewhere in source control, the base image used by that Dockerfile is tagged with a version whose upstream hasn't changed, and that the container isn't

modified from the image that Dockerfile produced, you get extremely reproducible builds with extremely explicit dependencies therein.

That said, I definitely see the faults in all of this (the base image version is mutable, and the Dockerfile schema doesn't allow you to verify that an image is what you'd expect with a checksum or something like that, containers can be mutated after startup, containers running as root is still a huge problem, etc), but this is definitely a step up from running apps in VMs. Now that I'm typing this out, I'm surprised that buildpacks or Chef's Habitat didn't take off; they solve a lot of these problems while providing similar reproducibility and isolation guarantees.

[reply](#)

👤 tonyarkles 2 days ago | root | parent | next [-]

So as a quick example from my past experiences, using an Ubuntu base image is fraught. If you don't pin to a specific version (e.g. pinning to ubuntu:20.04 instead of ubuntu:focal-20220316), then you're already playing with a build that isn't reproducible (since the image you get from the ubuntu:20.04 tag is going to change). If you *do* pin, you have a different problem: your apt database, 6 months from now, will be out of date and lots of the packages in it will no longer exist as that specific version. The solution is "easy": run an "apt update" early on in your Dockerfile... except that goes out onto the Internet and again becomes non-deterministic.

To make it much more reproducible, you need to do it in two stages: first, pinning to a specific upstream version, installing all of the packages you want, and then tagging the output image. Then you pin to to that tag and use that to prep your app. That's... probably... going to be pretty repeatable. Only downside is that if there is, say, a security update released by Ubuntu that's relevant, you've now got to rebuild both your custom base and your app, and hope that everything still works.

[reply](#)

👤 nunez 2 days ago | root | parent | next [-]

Yup, that can, indeed, be a problem. Relying on apt (for example) is generally a bad idea, hence why you'd want to vendor everything that your app needs if a specific version of, say, libcurl is something that your app requires.

This along with the supply chain issues you mentioned is why some maintainers are moving towards using distroless base images instead, though these can be challenging to debug when things go wrong due to them being extremely minimal, down to not having shells.

[reply](#)

👤 hinkley 2 days ago | root | parent | prev | next [-]

This is something that even the Docker core team is not entirely clear on (either they understand it and can't explain it, or they don't understand it).

The RUN command is Turing complete, therefore cannot be idempotent. Nearly every usable docker image and base image include the RUN command, often explicitly to do things that are not repeatable, like "fetch the latest packages from the repository".


This is all before you get to application code which may be doing the same non-repeatable actions with gems or crates or maven or node modules, or calling a service and storing the result in your build. Getting repeatable docker images usually is a matter of first getting your build system to be repeatable, and then expanding it to include the docker images. And often times standing up a private repository so you have something like a reliable snapshot of the public repository, one you can go back to if someone plays games with the published versions of things.

[reply](#)

👤 meatmanek 2 days ago | root | parent | next [-]

pedantic nit-pick that doesn't detract from your main point: Turing completeness doesn't imply that it can never be idempotent. In fact we'd expect that a particular Turing machine given a particular tape should reliably produce the same output, and similarly any Turing-complete program given the exact same input should produce the exact same output. Turing machines are single-threaded, non-networked machines with no concept of time.

The main reason the RUN statement isn't idempotent, in my opinion, is that it can reach out to the network and e.g. fetch the latest packages from a repository, like you mention. (Other things like the clock, race conditions in multi-threaded programs, etc. can also cause RUN statements to do different things on multiple runs, but I'd argue those are relatively uncommon and unimportant.)

[reply](#) dilyevsky 2 days ago | root | parent | next [-]


Yup, the problem is not that RUN is Turing Complete the problem is that it's non-hermetic.

[reply](#) hinkley 17 hours ago | root | parent | next [-]

▼ If it's Turing Complete you can't prove that it's hermetic. You can't even prove if it'll ever finish running (halting problem).


Maybe someday someone will invent a sort of EBPF for containers, but usually the first batch of RUN commands and the last are calling package managers, and in between you're doing things like creating users and setting permissions using common unix shell commands, which have the same problems.

With the possible exception of the unix package managers, we have no hope of those ever being rewritten inside of a proof system, because the package manager is often a form of dogfooding for the language community. The Node package manager is going to be written in Node and use common networking and archive libraries. Same for Ruby, Rust, you name it.

[reply](#) treis 2 days ago | root | parent | prev | next [-]


>Getting repeatable docker images

But there's no real need for repeatable build docker images. You copy the image and run it where ever you need to. The entire point of Docker is to not have to repeat the build.

[reply](#) hinkley 2 days ago | root | parent | next [-]

If you could explain that to the Docker team you would be performing a great humanitarian service.


They for many years close issues and reject PRs based on the assertion that Dockerfile output needs to be repeatable (specifically, that the requested feature or PR makes it not repeatable). It's not, it can't be without a time machine, and if it was I believe you'd find that ace icing traction would have been more difficult, even impossible.

[reply](#) treis 2 days ago | root | parent | next [-]

Do you have an example?

[reply](#) spion 2 days ago | root | parent | prev | next [-]

Until there is a critical vulnerability in one of the components present on that image (system packages or application packages).


[reply](#) kronin 2 days ago | root | parent | next [-]

This is why I haven't adopted the practice of "Build your artifact along with the docker image that packages it".

Instead, build your artifact and publish it to an artifact repository, just like we used to.

then wrap that artifact in a Docker image.

Vulnerability found in the docker image? No problem. Build a new image with the same artifact.

[reply](#) barbazoo 19 hours ago | root | parent | next [-]

▼ I'm curious why you're being downvoted for this. Whoever disagrees, please share some context.

What you said about "wrapping" I interpret as: based on an image for instance with OS level dependencies you create another image with application level artifacts, e.g. a python application. When your app changes, you don't build the base image again, you only build the app image. This makes sense to me.

[reply](#)

👤 treis 1 day ago | root | parent | prev | next [-]

At that point you're not repeating a build. You're building a new image.

[reply](#)

👤 spion 1 day ago | root | parent | next [-]

From that point on, the build may become "fully unrepeatable" because previous steps may have reused cached stuff that can no longer be reproduced.

You could break the build simply by cleaning some things from the cache, without changing the Dockerfile, even though that Dockerfile "succeeded to build"

[reply](#)

👤 convolvatron 1 day ago | root | parent | prev | next [-]

what happens when you need to create a new one with a small incremental change?

[reply](#)

👤 jffry 2 days ago | root | parent | prev | next [-]

> Containers only give reproducible deployments, not builds

I think that means containers alone are insufficient for creating reproducible builds, not that containers make reproducible builds impossible.

[reply](#)

👤 lazide 2 days ago | root | parent | next [-]

I believe that is correct. It's also hard to make container builds really reproducible (even with other parts of the build being so). Some sibling comments have talked about why.

Containers are to ops like C is to programming languages. Man is it useful, but boy are there footguns everywhere.

Usually still better than the ops equivalent of assembly, which is what came before though - all the custom manual processes.

Kubernetes is maybe like early C++. A bit weird, still has footguns, and is it really helping? Mostly it is in most cases, but we're also discovering many more new failure modes and having to learn some new complex things.

As it matures, new classes of problems will be way easier to solve with less work, and most of the early footguns will be found and fixed.

There will be new variants that hide the details and de-footgun things too, and that will have various trade offs.

No idea what will be the equivalent of Rust or Java yet.

[reply](#)

👤 thinkharderdev 2 days ago | root | parent | next [-]

I love this analogy

[reply](#)

👤 thinkingkong 2 days ago | root | parent | prev | next [-]

The dependencies can change out from underneath you transparently unless you pin everything all the way down the stack. Upstream docker images for example are an easy to understand vector of change. The deb packages can all change minor versions between runs, the npm packages (for example) can change their contents without making a version bump. Theres tons of implicit trust with all these tools, build wise.

[reply](#)

👤 pojzon 2 days ago | root | parent | next [-]

Tbh I hope ppl do pin stuff to specific versions and have their own repositories of packages because you dont want to have external dependencies during builds that can fail.

DevOps 101 more or less..

[reply](#)

👤 Osiris 2 days ago | root | parent | prev | next [-]

npm packages can change without a version change?

Can you explain this?

npm doesn't allow you to delete any published versions (you can only deprecate them). You aren't allowed to publish a version that's already been published.

Even when there have been malicious packages published the solution has been to publish newer versions of the package with the old code. There's no way to delete the malicious package (maybe npm internally can do it?).

[reply](#)

👤 thinkingkong 2 days ago | root | parent | next [-]

Sorry, without a minor version change. You can easily publish a patch version and most people don't pin that part of their dependency.

[reply](#)

👤 LunaSea 2 days ago | root | parent | next [-]

They don't need to pin it directly.

They only need to "npm ci" (based on package-lock.json) instead of "npm install" (based on package.json) within the Docker container to get a fully reproducible build.

[reply](#)

👤 monocasa 2 days ago | root | parent | prev | next [-]

I think that's what they're saying, that containers are orthogonal to reproducible builds.

[reply](#)

👤 q_eng_anon 2 days ago | root | parent | prev | next [-]

<https://github.com/GoogleContainerTools/distroless>

This is the container community's response to this ambiguity I think.

[reply](#)

👤 claytonjy 2 days ago | parent | prev | next [-]

is Nix a hurdle when onboarding engineers? do only a few people need to know the language? I've wanted to use Nix and home-manager for personal stuff but the learning curve seems big.

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

In my case, adopting Nix was a response to having a poor onboarding process for new engineers. It was always fully automated, but it wasn't reliable before nix. So somebody would join the team, and it was embarrassing because the first day would be them troubleshooting a complex build process to get it to work on their machine. Not a great first impression.

So I adopted Nix "in anger" and now new machines always build successfully on the first try.

For me it was easy to set up. 80% done on the first day. It helped that I understood the conceptual model and benefits first. There's a lot of directions you can take it, I'd recommend getting something really simple working then iterating. Don't try to build the Sistine Chapel with all the different things integrated on day one.

It's hard to overstate how well that decision has worked out for me. Computers used to stress me out a lot more because of dependencies and broken builds. Now I have a much healthier relationship with computers.

[reply](#)

👤 john-shaffer 2 days ago | root | parent | next [-]

Have you had problems with Nix on macOS? Nix works great for me, but I can't use it much to deal with

installs or synchronize dependencies because the devs on macOS can't get Nix running.

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

A few people I've worked with have used our nix build successfully on MacOS, but I stick to Linux. They've told me it works fine after making some dependencies conditional. I would've expected it to be death by a million papercuts, so I'm delighted it works and a VM isn't needed.

[reply](#)

👤 mrslave 2 days ago | root | parent | prev | next [-]

We have devs on MacOS with Nix working, but we do encounter inconsistencies. Production & other devs are on Linux and they are having a much better time.

[reply](#)

👤 silviogutierrez 2 days ago | root | parent | prev | next [-]

It's a hurdle to be able to write Nix stuff, but to consume it is pretty easy.

Shameless plug here: <https://www.reactivated.io/documentation/why-nix/>

[reply](#)

👤 olifante 2 days ago | root | parent | next [-]

I just tried Nix yesterday for the first time in order to play with reactivated. Was surprised by how simple it was to get it running. Made me think that perhaps containers are not as indispensable as I thought.

And reactivated is awesome, by the way. I just worry about how brittle it will be as both Django and React evolve.

[reply](#)

👤 silviogutierrez 2 days ago | root | parent | next [-]

Thanks! Hopefully not brittle at all. Specially since both React and Django are very stable projects, the latter even more than the former.

[reply](#)

👤 abecedarius 2 days ago | root | parent | prev | next [-]

Just my experience with Nix, two or three years back: I went to the Nix website for the install instructions, saw something like "curl | sh", said "what? no way" and used the alternative non-recommended out-of-date instructions, which turned into a full-day rabbit hole until it would build. A year later I went through this again for another machine, and this time just gave up.

So I would say curl|sh is probably the way to go, which I guess means I'd only want to install it into a container. But I do really like the idea of Nix.

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

I grabbed a copy of that shell script, reviewed it, and committed it to git. It's extra nice to save a specific copy of it so you get the same version of the nix cli tools across the team. It works with different versions just fine, but still better to standardize.

[reply](#)

👤 deathanatos 2 days ago | parent | prev | next [-]

> Kubernetes has mostly focused on stateless applications so far. That's the easy part! The hard part is managing databases.

Kubernetes can absolutely host stateful applications. A StatefulSet is the app-level construct pretty much intended for exactly that use case. My company runs a distributed database on top of Kubernetes.

(We have another app that uses StatefulSets to maintain a cache across restarts, so that it can come up quickly, as it otherwise needs to sync a lot of data. But, it is technically stateless: we could just sync the entire dataset, which is only ~20 GiB, each time it starts, but that is wasteful, and it makes startup, and thus deployments of new code, quite slow. It would also push the limits of what an emptyDir can accommodate in our clusters' setup.)

(The biggest issue we've had, actually, with that, is that Azure supports NVMe, and Azure has AKS, but Azure — for God only knows what reason — apparently doesn't support combining the two. We'd love to do that, & have managed

k8s & nice disks, but Azure forces us to choose. This is one of my chief gripes about Azure's services in general: they do not compose, and trying to compose them is just fraught with problems. But, that's an *Azure* issue, not a Kubernetes one.)

[reply](#)

👤 pojzon 2 days ago | root | parent | next [-]

For anyone looking for confirmations look for Zalando's Postgres Operator. They host 100++ db clusters using that also for production workloads.

[reply](#)

👤 fnordpiglet 1 day ago | parent | prev | next [-]

I think containers and k8s start to make sense when you need to impose logical isolation on your flat topology networks, often for security but also for preventing everything becoming overly complex in their dependencies. But this IMO isn't often a concern until your enterprise is very large and complex with some very crufty dependencies that you realize eventually you can never practically demise because you can't even fully determine what all depends on it at any given time.

That said, I would prefer managing that stuff through SDNs rather than trying to make user space overly complex software replication of commodity capabilities. Or, just use a cloud provider to start managing the isolation for the parts that benefit for it. What really baffles me is the use of k8s in aws. Why?

BTW - congrats on effective use of Nix. I use it extensively at home for my home services. It's got some weirdness to it for sure but it's a heck of a lot better than almost anything I've ever seen to date. I can't wait to see what evolves out of it.

[reply](#)

👤 higeorge13 2 days ago | parent | prev | next [-]

It sounds interesting. You mentioned big data; what is your tech stack other than postgres?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Python, PostgreSQL, Nix, Linux. Javascript and Typescript for frontend. Everything else is effectively libraries. After 300k lines of Python, with the occasional C extension, we haven't found any limit.

[reply](#)

👤 david38 2 days ago | parent | prev | next [-]

Ehhh, I run a high performance database system in Kubernetes and it works great. It's distributed, uses EBS volumes. That's about as opposite of stateless as it gets.

[reply](#)

👤 tetha 2 days ago | root | parent | next [-]

Do note that you have offloaded a good chunk of state management into EBS volumes via CSI. Attaching the CSI volumes is one thing, running the disks hosting the volumes is another thing.

[reply](#)

👤 n42 2 days ago | parent | prev | next [-]

I'm leading my team through a migration to Nix environments right now, and everything you've said has rung true to my own experiences so far. I'd love to hear more about the hard lessons you've learned, if you have any thoughts there

[reply](#)

👤 mamcx 2 days ago | parent | prev | next [-]

Any hints in how use nix with DBs?

I have a semi-docker setup where I stay with the base OS + PostgreSQL and the rest is docker. But that means that upgrade the OS/DB is not that simple.

I tried before dockerize PG, but still the OS need management anyway and was harder to use diagnostics/psql/backups with docker...

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Yeah, we built a tool for managing postgresql databases with nix. It's called schematic:
<https://gitlab.com/deltaex/schematic>

We've been using it in prod for a couple years. There's a couple dozen production deployments outside the company as well. It's open source, MIT licensed. It doesn't have documentation yet, so it's currently only for people that don't mind reading the source or have talked to us directly about it.

It uses Nix because postgresql has C extensions, which can depend on anything in the software universe. Schema depends on extensions, so it's not technically possible to separate schema migrations from Nix without duct tape and glue. So schematic is a sort of "distribution" of PostgreSQL that has a package manager (for extensions, schema, content, etc), and manages revisions.

If this is interesting to others here, I can do a "Show HN" post after getting the docs in order.

[reply](#)

▲ mamcx 2 days ago | root | parent | next [-]

Yeah, managing the DBs is the major complications on doing this. A little discrepancy in binaries are not that always problematic, but a fail on the DB is catastrophic :)

[reply](#)

▲ ComradePhil 2 days ago | parent | prev | next [-]

If you really believed what you pretend to, you would be programming in .NET on Windows computers and running your apps on Windows... or even Java.

Or take that problem to another level and use golang.

Instead, you use python and try to fix the problems with another immature solution: Nix, which has uniform builds... but not really in certain cases... and it is the same on every Linux... except not really so you kind of have to use NixOS.

[reply](#)

▲ ixxie 2 days ago | parent | prev | next [-]

I would love to hear more about your architecture and deployment... do your services run as NixOS modules? Are you using NixOps or Morph or something else? How does your world look like without K8s and Containers?

[reply](#)

▲ smilliken 2 days ago | root | parent | next [-]

For the first 6 years of using Nix, it was depoyed on Ubuntu. We recently migrated to NixOS. NixOS is fantastic for other reasons that are similar but separate from Nix as a package manager/build system.

It's easier to incrementally switch to Nix first then NixOS later.

We don't use systemd services for our code. We only use NixOS as an operating system, not for application layer. Our code works just the same on any linux distribution, or even macos minus linux-only stuff. That way we don't need to insist that everyone uses the same OS for development, and no one is forced into VMs. Everything can be run and tested locally.

[reply](#)

▲ smilliken 2 days ago | root | parent | next [-]

> Are you using NixOps or Morph or something else? How does your world look like without K8s and Containers?

Not using NixOps or Morph. I recall considering a few others as well. In each case, I wasn't able to understand what they were doing that I couldn't do with a simple script. Instead, there's a python program that does an rsync, installs some configuration files, and runs the nix build. It deploys to any linux OS, but it does some stuff differently for NixOS, and the differences account for about 100 lines while increasing scope (managing filesystems, kernel versions, kernel modules, os users, etc). A deploy to all hosts takes about a minute because it runs in parallel. Deploys are zero downtime, including restarting a bunch of web apps.

The NixOS configuration itself is another ~200 lines, plus a bit per host for constants like hostname and ip addresses. It's really neat being able to define an entire operating system install in a single configuration file that gets type checked.

[reply](#)

👤 osener 2 days ago | root | parent | next [-]

For others interested in deploying like this, this approach sounds exactly like what krops does. Krops is similar to Morph, except it does the derivative building on the remote host.

It is very simple and works great. Deploying from macOS to NixOS is possible as well.

<https://github.com/krebs/krops>

<https://tech.ingolf-wagner.de/nixos/krops/>

[reply](#)

👤 geoduck14 2 days ago | parent | prev | next [-]

Well, I was about to ask if you were looking for a job - but then I saw you were the founder and CTO. So... are you hiring?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Yes, we're hiring! :)

Reach out to dev@mixrank.com. Hiring globally for pretty much all roles, including junior roles.

[reply](#)

👤 sideway 2 days ago | parent | prev | next [-]

Out of curiosity, what industry is your company in?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Details in bio.

[reply](#)

👤 rapsey 2 days ago | parent | prev | next [-]

I've seen comments on here from people with large Nix deployments with like hundreds of thousands of Nix script code and saying it is a complete nightmare to manage.

[reply](#)

👤 Thaxll 2 days ago | parent | prev | next [-]

The thing is, you spent probably a lot of time on things that were granted elsewhere and while the rest of the world is improving on those tech you keep your home grown solution that is harder and harder to maintain. Plus the knowledge that is not transferable.

> Containers only give reproducible deployments, not builds, so they would be a step down.

This is not true, if you use a docker image A with specific version to build then the result is reproducible.

That's what most people should be doing, you pin versions in your build image that's it.

> Kubernetes has mostly focused on stateless applications so far. That's the easy part

Kubernetes can run database, and stateless application is not a solved problem, I mean how does your services get restarted on your system, is it something that you had to re-create as well?

[reply](#)

👤 smilliken 2 days ago | root | parent | next [-]

Dan Luu wrote an essay on this topic recently: <https://danluu.com/nothing-works/>

He ponders why it is that big websites inevitably have kernel developers. Way out of their domain of expertise, right? If you adopt a technology, you're responsible for it.

When Kubernetes inevitably has an issue that is a blocker for us, I don't have confidence in my ability to fix it. When an internal python or shell program has an issue that is a blocker for us, I change it.

PostgreSQL is used by probably millions of people, but we've had to patch it on multiple occasions and run our fork in production. Nobody wants to do that, but sometimes you have to.

The point is, you can't just say "oh, we use kubernetes so we don't have to think about it". No. You added it to your stack, and now you're responsible for it. Pick technologies that you're able to support if they're abandoned,

unresponsive to your feature requests and bug reports, or not interested in your use case. Pick open source, obviously.

This is another reason I like Nix. It's a one-line change to add a patch file to a build to fix an issue. So I can contribute a fix upstream to some project, and then I don't have to wait for their release process, I can use the patch immediately and let go of it whenever the upstream eventually integrates it. It lowers the cost of being responsible for third party software that we depend on.

[reply](#)

👤 Thaxll 2 days ago | root | parent | next [-]

> When Kubernetes inevitably has an issue that is a blocker for us, I don't have confidence in my ability to fix it. When an internal python or shell program has an issue that is a blocker for us, I change it.

I doubt that this happen in reality because Kubernetes cover uses cases for pretty much everyone, now I would doubt the knowledge of a regular dev to try to mimic a solution that k8s already does.

[reply](#)

👤 cj 2 days ago | root | parent | prev | next [-]

> the rest of the world is improving on those tech

Does that matter if the current stack works just fine?

Imagine someone who built a calculator app with jQuery javascript 10 years ago, and all it does is add and subtract numbers. You could spend time porting it to Ember, and then migrating to Angular, and then porting it to React, and then porting it to React with SSR and hooks.

If the calculator app worked with 15 year old code, you can either leave it be, or try to keep up with latest tech trends and continuously refactor and port the code.

I think there are a lot of "calculator app" type components of many systems that, at a certain point, can be considered "done, complete" without ever needing a rebuild or rewrite. Even if they were built on technologies that might now be considered antiquated.

[reply](#)

👤 hinkley 2 days ago | root | parent | next [-]

The people who are invested in their home-grown solutions are usually pretty bad at doing tech support for them. They enjoy writing code, and it doesn't take long before you've created enough surface area that you couldn't possibly keep up with requests even if you wanted to.

Which they don't, because nothing has convinced me of the crappiness of some of my code more thoroughly than watching other people try and fail to use it. They are a mirror and people don't always like what they see in it. That guy who keeps talking trash about how stupid everyone else is for not understanding his beautiful code is deflecting. If they're right then my code does not have value/isn't smart which means *I* don't have value/am not smart, so clearly the problem is that I'm surrounded by assholes.

I don't think we would be using as much 3rd party software without StackOverflow, and almost nobody creates a StackOverflow for internal tools. Nobody writes books for internal tools. Nobody writes competitors for internal tools which either prove or disprove the utility of the original tool. All of those options are how some people learn, and in some cases how they debug. You're cutting off your own nose when you reinvent something and don't invest the time to do all of these other things. If I lose someone important or get a big contract I didn't think I could win, I can't go out and hire anyone with 3 years experience in your internal tool, trading money for time. I have to hire noobs and sweat it out while we find out if they will ever learn the internal tools or not.

The older I get the more I see NIH people as working a protection racket, because it amplifies the 'value' of tenure at the company. The externalities created really only affect new team members, delaying the date when you can no longer dismiss their ideas out of hand. You have in effect created an oligarchy, whereas most of the rest of us prefer a representative republic (actual democracy is too many meetings).

[reply](#)

👤 p10_user 2 days ago | root | parent | next [-]

Nice post. I wish I was reading this 6 years ago, but at least I quickly learned to be humble about the quality of my code.

[reply](#)

👤 hinkley 2 days ago | root | parent | next [-]

I'm not quite sure how to interpret what you've said exactly.

Some of what I said is a performance piece. If I insist that I am human and thus make mistakes, then it makes space both for others to make mistakes and for us to have a frank conversation of how, in our more lucid moments, we can do things that fight/counteract our basic nature.

It's morally equivalent to putting a plastic cover over the Big Red Button so that pushing it becomes a conscious *and* considered act. Deciding to push the button is step 1, moving the cover is step 2, and step 1.5 or 2.5 is realizing this is in fact not what you want to do after all, instead of a momentary bit of inattention, bad judgement, or the Imp of the Perverse.. It's not that I won't make mistakes, but that (hopefully) I make them less often. But making them less often, I'm entrusted with *bigger* potential for mistakes, and so my mistakes tend to cause more harm. In a Precautionary Principle sense, we end up roughly within the same order of magnitude. When I make 5x fewer mistakes I'm allowed to be responsible for problems that are twice as risky.

If you want candor about mistakes you have to start by giving people permission to make them (or at least, to make new ones).

[reply](#)

👤 p10_user 1 day ago | root | parent | next [-]

Everything you are talking about seems centered around cultivating humility. You are absolutely right. It's ultimately the ability to allow yourself to make mistakes. Being wrong is no fun. Best to get it over with, accept it and work forward, rather than to bring yourself into some delusion.

[reply](#)

👤 Thaxll 2 days ago | root | parent | prev | next [-]

It works fine until the core people that know that stack leave your compagny and no one wants to touch it. Then the new people comes in a pich a solution that is widely used.

[reply](#)

👤 Thaxll 2 days ago | root | parent | prev | next [-]

At some point you just don't want to re-invent the wheel, reminds me a lot of:
https://en.wikipedia.org/wiki/Not_invented_here

[reply](#)

👤 zomglings 2 days ago | root | parent | prev | next [-]

The OP is talking about reproducible *builds*. A docker image is a build *artifact*.

If I gave you a Dockerfile for a node.js service that installed its dependencies using npm, and the service used non-trivial dependencies, and I gave you this file 10 years after it was first written, chances are you would not be able to build a new image using that Dockerfile.

This would be a problem for you if you had to make some small change to that node.js service (without changing any dependencies).

[reply](#)

👤 Art9681 2 days ago | root | parent | next [-]

There are other container build tools such as Kaniko that have solved the reproducible builds issue right? If we're operating at scale, it probably means we are using some flavor of Kubernetes and the Docker runtime is no longer relevant. Would Redhat OpenShift's source-to-image (S2I) build process solve the reproducible builds requirement? This space moves quickly and the assumptions we had last week may no longer be relevant.

For example Crunchy Postgres offers an enterprise supported Kubernetes Operator that leverages stateful sets. Yet I am reading in these comments that it is an unsolved issue.

[reply](#)

👤 Thaxll 2 days ago | root | parent | prev | next [-]

Docker is also used to build things not just run them, multistage build with dependency vendoring, you

have a 100% reproducible build.

It's very easy to do in Go.

You pin the version used to build your app (the runtime version) In your git repo you have dependencies That's it.

[reply](#)

👤 jandrewrogers 2 days ago | root | parent | prev | next [-]

Sure, Kubernetes can run a database, but not efficiently. Companies with intensive data infrastructure frequently operate at scale without containers, either VMs or bare metal. The larger the data volume, the less likely they are to use containers because efficiency is more important. It is also simpler to manage this kind of thing outside containers, frankly, since you are running a single process per server.

People have been building this infrastructure since long before containers, thousands of servers in single operational clusters. Highly automated data infrastructure without containers is pretty simple and straightforward if this is your business.

[reply](#)

👤 remram 2 days ago | root | parent | next [-]

Running on Kubernetes doesn't mean all data has to be on network disks.

[reply](#)

👤 anonymousDan 2 days ago | root | parent | prev | next [-]

Can you point to any resources on the perf implications of docker for databases?

[reply](#)

👤 Art9681 2 days ago | root | parent | next [-]

OP is likely making assumptions on obsolete knowledge. Tech moves really fast which means the knowledge we gained last week might no longer be relevant. Here is an article showing it is perfectly acceptable to deploy Postgres in Kubernetes:

<https://www.redhat.com/en/resources/crunchy-data-and-openshi...>

The great majority of Kubernetes experts I have met use a managed service and have not been exposed to the internals or the control plane back end. An abstracted version of Kubernetes that works well for most use cases but makes it more difficult to solve some of the problems listed in this thread.

Most folks dont really know Kubernetes. They know how to deploy X application using X abstraction (Helm, Operators, CD tooling) to some worker node in a public cloud using public sources and thats basically it. It's no wonder they cannot solve some of the common problems because they didnt really deploy the apps. They filled out a form and pushed a button.

[reply](#)

👤 jandrewrogers 1 day ago | root | parent | prev | next [-]

High-performance database engines take complete control of their resources, I/O, and scheduling, completely bypassing the OS kernel. Linux is explicitly designed to allow this. This architecture enables integer factor improvements in throughput on the same hardware, which is why you would design your software this way. Older, slower database engine designs (e.g. Postgres) don't take explicit control of resources in this way and will work reasonably well in containers.

The original Linux container design was never intended with this type of software in mind, and tacitly reduces the control a process has over its resources. Invariants required for performance are violated, and therefore performance suffers relative to bare metal (or conditionally VMs). The v2 container implementation, which isn't widely available yet, recognizes these problems and attempts to remedy some of these poor behaviors of the v1 container implementation.

If your database does not assume strict control of its underlying hardware resources then this does not affect you. However, this kind of strict resource control is idiomatic in all the highest performing databases architectures, to great effect, so it is the most performant databases that are most adversely affected by containerization.

FWIW, virtual machines used to have similar terrible performance for high-performance databases for similar reasons. Now there are mechanisms for the database to effectively bypass the VM

similar to the kernel, and the loss of performance is minimal. Container environments typically don't offer similar bypass mechanisms.

[reply](#)

▲ Inxg33k1 2 days ago | root | parent | prev | next [-]

I also don't agree with the concept, and I've had problems about with docket using a specific OS version but pulling in some minor software version which broke, so I think even if being slightly true, that statement represent something that might be solved by specifying minor versions for the packages you depend on, which is a level of effort compatible with the one you need on nix to specify packages hashes, so you can have both with docker, a strict and non strict approach, while I guess nix only supports strict

[reply](#)

toast0 2 days ago | prev | next [13 more]

▲ wanderr 2 days ago | prev | next [-]

Grooveshark didn't use any of that. We were very careful about avoiding dependencies where possible and keeping our backend code clean and performant. We supported about 45M MAU at our biggest, with only a handful of physical servers. I'm not aware of any blog posts we made detailing any of this, though. And if you're not familiar with the saga, Grooveshark went under for legal, not technical reasons. The backend API was powered by nginx, PHP, MySQL, memcache, with a realtime messaging server built in Go. We used Redis and Mongodb for some niche things, had serious issues with both which is understandable because they were both immature at the time, but Mongodb's data loss problems were bad enough that I would still not use them today.

That said, I'm using Docker for my current side project. Even if it never runs at scale, I just don't want to have to muck around with system administration, not to mention how nice it is to have dev and prod be identical.

[reply](#)

▲ brimble 2 days ago | parent | next [-]

> That said, I'm using Docker for my current side project. Even if it never runs at scale, I just don't want to have to muck around with system administration, not to mention how nice it is to have dev and prod be identical.

This is why I use docker, at work and for my own stuff. No longer having to give a shit whether the hosting server is LTS or latest-release is *wonderful*. I barely even have to care which distro it is. Much faster and easier than doing something similar with scripted-configuration VMs, plus the hit to performance is much lower.

[reply](#)

▲ turkeywelder 2 days ago | parent | prev | next [-]

I miss Grooveshark so much - Licensing issues aside it was one of the best UIs for music ever. I'd love to hear more stories about the backend

[reply](#)

▲ gibspaulding 2 days ago | root | parent | next [-]

My feelings as well! The way it put your queue front and center and gave you so much control over how things were added worked really well for me. Spotify leaves a lot to be desired in its UI.

I still remember opening the site one day and reading the weird apology letter. I think that was the first time I saw something I really cared about just disappear off of the internet.

[reply](#)

▲ Aachen 2 days ago | parent | prev | next [-]

Man I miss Grooveshark still today. Spotify is okay but still a step down. Needing billion-dollar licensing schemes to even get started makes this such a hard market to actually get into and provide a competitively superior experience.

[reply](#)

▲ mhitza 2 days ago | parent | prev | next [-]

What a great service. I'd be curious if you could go into details how the radio feature worked back then, because I found myself receiving worse suggestions when I used similar features in Spotify/Google Play Music.

[reply](#)

▲ wanderr 1 day ago | root | parent | next [-]

Oh man, I should write a blog post about that, as I built that feature myself. It was meant to be a stopgap until

we could get some real machine learning in there, but nothing else we tried did as well. First, for efficiency all recommendations were artist to artist, not song to song. That works well for a lot of genres but is pretty bad for others. We started with a free DB of artist similarities, I don't remember where we got that from, maybe musicbrainz? We built a shitty internal interface for adding and removing links between artists and adjusting the weights of those links and then made it available to all employees to mess with. As you might imagine just about everyone there was passionate about music so it didn't take long to crowdsource a huge catalog of quality recommendations and then for really obscure stuff we would fall back to the open db. So the actual algorithm would look at your seeds - artists you put in the queue before turning on radio or artists with songs that you liked while radio was on, pull the top n linked artists for each of your seed artists, and do some weighted shuffling. It would also make sure to space out artists so you don't hear the same one too often etc. Then for genre radio we just secretly selected a bunch of artists we felt were representative of the genre and used those as the seeds. Oh yeah and if you disliked a song we'd prevent that artist from playing for the rest of your session. We also would look at anomalies like popular artists with not many recommendations, or artists that, when used as seeds, lead to shorter listening sessions (implying that the recommendations need to be cleaned up). Most attempts to replace this with something smarter ran into 2 problems: 1. Popular stuff is popular, so it looks like a good recommendation for anything, and 2. ML is hard and takes a lot of time, which we never had enough of

[reply](#)

▲ fougerejo 2 hours ago | root | parent | next [-]



I want to salute you for the Grooveshark recommendation engine. To this day, that's THE feature that I used a LOT on Grooveshark (hours & hours), and that I'm frustrated about in Spotify. You did an amazing job on this one.

[reply](#)

▲ efreak 1 day ago | root | parent | prev | next [-]

I'm not aware of an MB similar artist database. I'm guessing you used music map[0], it's the only free database I know of for similar artists that doesn't require scraping.

[0]: <https://www.music-map.com/>

[reply](#)

▲ mhitza 5 hours ago | root | parent | next [-]



Around that time freebase was still a thing, and dbpedia/wikipedia could also be used to sample relationships.

[reply](#)

▲ wanderr 21 hours ago | root | parent | prev | next [-]



Ah, I think you're right that it wasn't MB because I remember having to match artists by name rather than mbid. Music map doesn't sound familiar but my boss negotiated the access, I think I was just ingesting the data from a csv dump so it could have been anywhere.

[reply](#)

▲ hungryforcodes 2 days ago | parent | prev | next [-]

Groove shark was the best.

[reply](#)

▲ hexfish 2 days ago | parent | prev | next [-]

Thanks for giving some insight into this. Grooveshark was absolutely great!

[reply](#)

▲ jimbob45 2 days ago | parent | prev | next [-]

I miss Grooveshark to this day. Thanks for building such an excellent product!

[reply](#)

▲ pineconewarrior 2 days ago | parent | prev | next [-]

I loved Grooveshark! thanks for your work

[reply](#)

▲ maxk42 2 days ago | prev | next [-]

Back in 2010 I built and operated MySpace' analytics system on 14 EC2 instances. Handled 30 billion writes per day. Later I was involved in ESPN's streaming service which handled several million concurrent connections with VMs but no containers. More recently I ran an Alexa top 2k website (45 million visitors per month) off of a single container-free EC2 instance. Then I spent two years working for a streaming company that used k8s + containers and would fall over if it had more than about 60 concurrent connections per EC2 instance. K8s + docker is much heavier than advertised.

[reply](#)

👤 danielrhodes 2 days ago | parent | next [-]

Docker is far heavier - the overhead is the flexibility and process isolation you get. I imagine that's really useful for certain types of workloads (e.g. an ETL pipeline), but is crazy inefficient for something single purpose like a web app.

[reply](#)

👤 ryanjkirk 2 days ago | root | parent | next [-]

Docker is heavier (and more dangerous) because of dockerd, the management and api daemon that runs as root. Actual process isolation is handled by cgroup controls which are already built into the kernel and have been for years. You can apply them to any process, not just docker ones.

However, Docker is essentially dead; the future is CRI-O or something similar which has no daemon and runs as an unprivileged user. And you still get the flexibility and process isolation, but with more security.

[reply](#)

👤 freebuju 2 days ago | root | parent | next [-]

All the so-called "docker killers" are essentially unfinished products. They don't compare 1:1 to docker in feature set and even if they run as rootless, they still are vulnerable to namespace exploits in the Linux kernel. Though docker runs as root, it's still well protected out-of-the-box for the average user and is a very mature technology.

[reply](#)

👤 ryanjkirk 2 days ago | root | parent | next [-]

Are you from 2018? Everyone running OpenShift is using CRI-O and that footprint is not small. We made the switch in our EKS and vanilla k8s clusters in 2021. Docker has now even made their API OCI-compliant in order to not be left behind. And the point is that most people don't want a docker feature-for-feature running in prod. The attack surface is simply too large. I don't need an API server running as root on all my container hosts.

Use docker on your laptop, sure. Its time in prod is over.

[reply](#)

👤 Art9681 2 days ago | root | parent | next [-]

Agreed. Tons of obsolete assumptions in this thread. We have been using Podman / OpenShift in production and never ran into a use case where Docker was needed.

[reply](#)

👤 p_l 2 days ago | root | parent | prev | next [-]

One of the biggest benefits of k8s for me, back in 2016 when I first used it in prod, was that it threw away all the extra features of Docker and implemented them directly by itself - better. Writing was already in the wall that docker will face stern competition that doesn't have all of its accidental complexity (rktnetes and hypernetes were a thing already)

[reply](#)

👤 richardwhiuk 2 days ago | root | parent | next [-]

Kubernetes used to (tediously) pass everything through to Docker, but since 1.20, that's resolved, and it now uses containerd.

[reply](#)

👤 p_l 2 days ago | root | parent | next [-]

Not everything - for a bunch of things, the actual setup increasingly happened *outside* docker then docker was just informed how to access it, bypassing all the higher level logic in Docker.

1.20 is when docker mode got deprecated, IIRC, but many of us were already happily

running in containerd for some time.

[reply](#)

👤 qwertywert_ 2 days ago | root | parent | prev | next [-]

I thought its moving to CRI-o as well instead of containerd, or is k8s just not using docker, and containerd still supported in the future?

[reply](#)

👤 p_l 2 days ago | root | parent | next [-]

CRI-O is the target and containerd is the most common runtime implementing it at the moment.

[reply](#)

👤 qwertywert_ 1 day ago | root | parent | next [-]

Are you sure? This isn't my subject area but CRI-O looks like an alternative to containerd and implements the OCI compliant runtime like containerd does. And then there is a 3rd which is docker engine which is the one being dropped.

[reply](#)

👤 p_l 1 day ago | root | parent | next [-]

Sorry, I mixed up CRI and CRI-O. The roadmap is to remove dockershim (the interface to docker-compatible container runtime) and use only CRI - of which containerd and CRI-O are two compatible implementations.

[reply](#)

👤 richardwhiuk 2 days ago | root | parent | prev | next [-]

Kubernetes has removed docker, so I think that's basically it from a large scale perspective.

[reply](#)

👤 freedomben 1 day ago | root | parent | next [-]

Yep. Mantis has announced intent to continue maintaining the docker shim (which allows k8s to talk to docker programmatically, but I can't imagine many people switching the default to docker unless they are manually installing k8s on their nodes, which used to be common but no longer is.

[reply](#)

👤 stickyricky 2 days ago | parent | prev | next [-]

> Handled 30 billion writes per day.

Writes to what?

[reply](#)

👤 tptacek 2 days ago | prev | next [-]


Ironically, here at Fly.io, we run containers (in single-use VMs) for our customers, but none of our own infrastructure is containerized --- though some of our customer-facing stuff, like the API server, is.

We have a big fleet of machines, mostly in two roles (smaller traffic-routing "edge" hosts that don't run customer VMs, and chonky "worker" hosts that do). All these hosts run `fly-proxy`, a Rust CDN-style proxy server we wrote, and `attache`, a Consul-to-sqlite mirroring server we built in Go. The workers also run our orchestration code, all in Go, and Firecracker (which is Rust). Workers and WireGuard gateways run a Go DNS server we wrote that syncs with Consul. All these machines are linked together in a WireGuard mesh managed in part by Consul.

The servers all link to our logging and metrics stack with Vector and Telegraf; our core metrics stack is another role of chonky machines running VictoriaMetrics.

We build our code with a Buildkite-based CI system and deploy with a mixture of per-project `ctl` scripts and `fcm`, our in-house Ansible-like. Built software generally gets staged on S3 and pulled by those tools.

Happy to answer any questions you have. I think we fit the bill of what you're asking about, even though if you read the label on our offering you'd get the opposite impression.

[reply](#) po_ta_toes 2 days ago | parent | next [-]

Hi there,

Very interesting read!

I work for a large news org, The team I'm in primarily uses elixir, which I know the people at fly.io love too!

Why did you decide not to containerise your own infrastructure?

We use some 'chonky' ec2s but are thinking about using containers.

Given that the BEAM has quite a large footprint, do you think it still a good candidate for containers, or would that introduces too much overhead?

[reply](#) tptacek 2 days ago | root | parent | next [-]

We would containerize everything if we could! But our infrastructure components usually live outside the security boundaries our container interface sets up. The proxy has to be able to talk to every app, not just apps in its own organization, and it needs direct access to other infrastructure components. The orchestration code needs to be able to launch Firecracker VMs --- it can't itself be a Firecracker VM.


We can design around all these constraints! We just haven't gotten around to it yet. I think there's a general consensus that hiding things inside Firecrackers is a good design, and we'll do it where we can.

Elixir runs *great* inside Firecrackers. The idea behind Firecracker is that it introduces a minimal load, by dint of being ruthlessly simple. When you run an Elixir app on an EC2 instance, you're running hypervised as well (and probably by a hypervisor that's less efficient --- not a knock, Firecracker is Amazon code too).

The thing that makes this doable at Fly.io is that we run our own hardware, so we're not trying to nest VMs inside VMs. Of course, that's what EC2 is doing too. So whatever your intuitions are about things that run well on vanilla EC2, they should carry over to us.

[reply](#) freedomben 1 day ago | root | parent | next [-]


What are you using for the host OS on your bare metal?

[reply](#) tptacek 1 day ago | root | parent | next [-]

Linux. :)

[reply](#) freedomben 1 day ago | root | parent | next [-]

heh, could you share if it's more of an rpm flavor or more of a deb flavor? :-) (or some third option. no false dichotomies intended. actually it would be badass if you're running Arch :-D)

[reply](#) tptacek 1 day ago | root | parent | next [-]

At this moment I will go as far as to say that it is not a badass flavor of Linux.


:)

[reply](#) cpach 2 days ago | parent | prev | next [-]

Intriguing!

This makes me curious: How does one learn to design and build systems like this...?

Also: How do you folks at Fly decide what parts to use "as is" and what parts to build from scratch? Do you have any specific process for making those choices?

[reply](#) tptacek 2 days ago | root | parent | next [-]

We had to build the orchestration stuff (it was originally a Nomad driver, but has outgrown that) because the

tooling to run OCI containers as Firecracker VMs didn't exist in a deployable form when we started doing this stuff.

Most of the big CDNs seem to start with an existing traffic server like Nginx, Varnish, or ATS. One way to look at what we did with our "CDN" layer is that rather than building on top of something like Nginx, we built on top of Tokio and Hyper and its whole ecosystem. We have more control this way, and our routing needs are fussy.

By comparison, we use VictoriaMetrics and Elasticsearch (I don't know about "as-is" --- lots of tooling! --- but we don't muck with the cores of these packages), because our needs are straightforwardly addressed by what's already there.

Lots of companies doing stuff similar to what we're doing have elaborate SDN and "service mesh" layers that they built. We get away with the Linux kernel networking stack and a couple hundred lines of eBPF.

We definitely don't have a specific process for this stuff; it's much more an intuition, and is more about our constraints as a startup than about a coherent worldview.

[reply](#)

👤 [illfit](#) 2 days ago | [root](#) | [parent](#) | [prev](#) | [next](#) [-]

Google uses "Non-Abstract Large System Design (NALSD)" <https://sre.google/workbook/non-abstract-design/> for this style of design.

The emphasis on a concrete design with concrete numbers can help identify the main scaling and reliability limitations, and put a cost on these. "Design X costs \$A/year for Y scheduled fly.io tasks".

To build such a design relies on knowing fundamentals such as the performance characteristics of CPU/disk/network. "How many disks would it take to serve 50k QPS at 20ms, each time performing 1k of random disk I/O."

Knowing this helps identify where in your stack you want flexibility, and why you'd want it.

"We log 100MB/s spread across 200 machines, which can be done with vanilla mature Elastic search"

"We need to be able to perform container routing at 0.5ms overhead, and updates need to be atomic. eBPF can do this but existing solutions are immature. Since this is also our core competency, let's do this ourselves."

[reply](#)

👤 [freedomben](#) 1 day ago | [root](#) | [parent](#) | [prev](#) | [next](#) [-]

I can't speak for fly.io, but as someone who architects a lot of systems, you learn by doing, failing, and doing again. If you're smart your failures will all be proofs-of-concept rather than real-world, but some amount of real-world failure is inevitable.

You must learn how to learn from mistakes, because the more typical human reaction to failure is to get emotional and try to rationalize the failure away (including blaming others). You need to become ruthlessly analytical. Good analytical intuition can only come from experience/failure so you can't rush it.

Start with simple/straight forward problems to solve. For example, design an architecture for a headless, stateless API server. Start with a single instance exposed directly to the internet, and then put a load balancer in front of it and scale it up to at least 3 instances (horizontal scaling).

Now add a relational database (like postgres or mysql). Now make the server stateful (only for practice, avoid stateful services this like the plague in real life).

Now add a websocket to the server (which will need to go through the load balancer).

Now add a UI. Start with a few server-rendered pages, and then add a SPA. Try serving the SPA from the server app, and also try separating it into two distinct, independently deployable apps.

Now add some UDP to the app (adding WebRTC is an easy way to do this without having to write a lot of code).

Now add asynchronous jobs (sometimes called delayed jobs) just using the existing database.

Now add a queue system (like redis) for the jobs to communicate through and scale up the number of job instances.

Now add logging and metrics collection. I recommend EFK stack for logging and Prometheus/Grafana for metrics. At this point if you haven't used containers or kubernetes yet, it's probably a good time to repeat the whole process in k8s.

Now start adding microservices! Start with simple ones and get increasingly complex. Add a service that is never accessed directly by users, and put mTLS in front of it. Make sure that your services are aggregating logs

to your logging solution and metrics are being collected.

Now add some services that use gRPC, protobufs, etc. You already have some non-HTTP in the form of UDP, but try adding some non-HTTP TCP based services. An SMTP server is a good challenge. You'll have to start getting creative!

To be well rounded, try using various products/approaches and see how they solve problems you have. It's good to do things "the hard way" (such as setup manually on VMs) to learn how they work, but IRL you won't want to do everything that way. Kubernetes is exploding for a reason. Make sure you learn some industry tools/solutions as well. I prefer open source to vendors as the vendors often obscure all the learning (which can be good for a real company, but isn't good for your goal of learning).

As you get better and more experienced, many of these scenarios can be done entirely as thought experiments, although if you are doing it for real, you should try to build a PoC (proof of concept) for *anything* that hasn't been proven before.

Lastly, bounce ideas off of people with experience! Be prepared to have your ideas challenged, and be prepared to critically think about others ideas and challenge them! In the real world you will rarely design it all yourself.

Pretty soon, you will be able to design and build complex systems. Also the Google SRE book(s) can be really helpful.

Holy cow this went a lot longer than I intended. I think I'll turn this into a blog post.

[reply](#)

▲ q3k 2 days ago | [prev](#) | [next](#) [-]

Depends what you mean by 'container runtime' or 'container orchestration tool'...

For example, Google's Borg absolutely uses Linux namespacing for its workloads, and these workloads get scheduled automatically on arbitrary nodes, but this doesn't feel at all like Docker/OCI containers (ie., no whole-filesystem image, no private IP address to bind to, no UID 0, no control over passwd...). Instead, it feels much closer to just getting your binary/package installed and started on a traditional Linux server.

[reply](#)

▲ menage 2 days ago | [parent](#) | [next](#) [-]

> no whole-filesystem image

At least in the past, almost all jobs ran in their own private filesystem - it was stitched together in userspace via bind mounts rather than having the kernel do it with an overlayfs extracted from layer tar files (since overlayfs didn't exist back then), but the result was fairly similar.

Most jobs didn't actually request any customization so they ended up with a filesystem that looked a lot like the node's filesystem but with most of it mounted read-only. But e.g. for a while anything running Java needed to include in their job definition an overlay that updated glibc to an appropriate version since the stock Google redhat image was really old.

[reply](#)

▲ isseu 2 days ago | [parent](#) | [prev](#) | [next](#) [-]

Yeah was gonna post this. Not sure if is updated enough but it's worth reading the Borg paper <https://research.google/pubs/pub43438/>

[reply](#)

▲ sitkack 2 days ago | [parent](#) | [prev](#) | [next](#) [-]

Install deps into a dynamically provisioned container, it will feel similar.

[reply](#)

▲ alex_duf 2 days ago | [prev](#) | [next](#) [-]

Hey former Guardian employee here.

The Guardian has hundreds of servers running, pretty much all EC2 instances. EC2 images are baked and derived from official images, similarly to the way you bake a docker image.

We built tools before docker became the de facto standard, so we could easily keep the EC2 images up to date. We integrated pretty well with AWS so that the basic constructs of autoscaling and load balancer were well understood by everyone.

The stack is mostly JVM based so the benefits of running docker locally weren't really significant. We've evaluated moving to

a docker solution a few times and always reached the conclusion that the cost of doing so wouldn't be worth the benefits.

Now for a company that starts today I don't think I'd recommend that, it just so happen that The Guardian invested early on the right tooling so that's pretty much an exception.

[reply](#)

▲ sparsely 2 days ago | parent | next [-]

> We integrated pretty well with AWS so that the basic constructs of autoscaling and load balancer were well understood by everyone.

This is an underappreciated point I think sometimes. Once you have a team which is familiar with your current, working setup, the benefits of moving away have to be pretty huge for it to be worthwhile.

[reply](#)

▲ weego 2 days ago | parent | prev | next [-]

Also ex employee. Riff Raff is absolutely still an excellent mod for build and deploy. At the time I was there it the initial stack build via handwritten cloudformation script that was the friction and pain point.

[reply](#)

▲ alex_duf 1 day ago | root | parent | next [-]

Honestly I miss riff-raff :D

For anyone who doesn't know about it: <https://github.com/guardian/riff-raff>

[reply](#)

▲ bscanlan 2 days ago | parent | prev | next [-]

Intercom is pretty similar. We use EC2 hosts and no containers (other than for development/test environments and some niche third-party software that is distributed as Docker containers). Autoscaling groups are our unit of scalability, pretty much one per workload, and we treat the EC2 hosts as immutable cattle. We do a scheduled AMI build every week and replace every host. We use an internally developed software tool to deploy buildpacks to hosts - buildpacks are pre-Docker technology from Heroku that solves most of the problems containers do.

I wouldn't necessarily recommend building this from scratch today, it was largely put in place around 8 years ago, and there are few compelling reasons for us to switch.

[reply](#)

▲ speleding 2 days ago | parent | prev | next [-]

> Now for a company that starts today I don't think I'd recommend that

I think for a company starting out that AMIs (Amazon Machine Images), which from your description is probably what you're using, is actually a much better way to go than docker containers, because you get a large part of the orchestration for free with the AWS EC2 auto-scaling and health detection without most of docker complexity.

(I would suggest using something like Terraform to set it up in a reproducible way though)

[reply](#)

▲ rr808 2 days ago | parent | prev | next [-]

> Now for a company that starts today I don't think I'd recommend that

Any reason why? It sounds pretty good.

[reply](#)

178 more comments...

Applications are open for YC Summer 2022

[Guidelines](#) | [FAQ](#) | [Lists](#) | [API](#) | [Security](#) | [Legal](#) | [Apply to YC](#) | [Contact](#)

Search: