The Twelve-Factor Container

Can good design be containerised?

David Carboni Follow
Aug 7, 2018 · 7 min read

> *Containers hit the tech headlines around 2013 and are now more the rule than the exception. I had the privilege of working with a forward-thinking team in a well known government organisation around that time and by 2015 was running Docker in production.*

This project delivered a seismic shift in the ability of the organisation to not only deliver, but deliver at pace. Let's be clear, containers were only part of that success. The team was the first to be able to work from an agile mindset, we prioritised a high-performing culture and, crucially, the sponsorship and support of transformational leadership made the project possible.

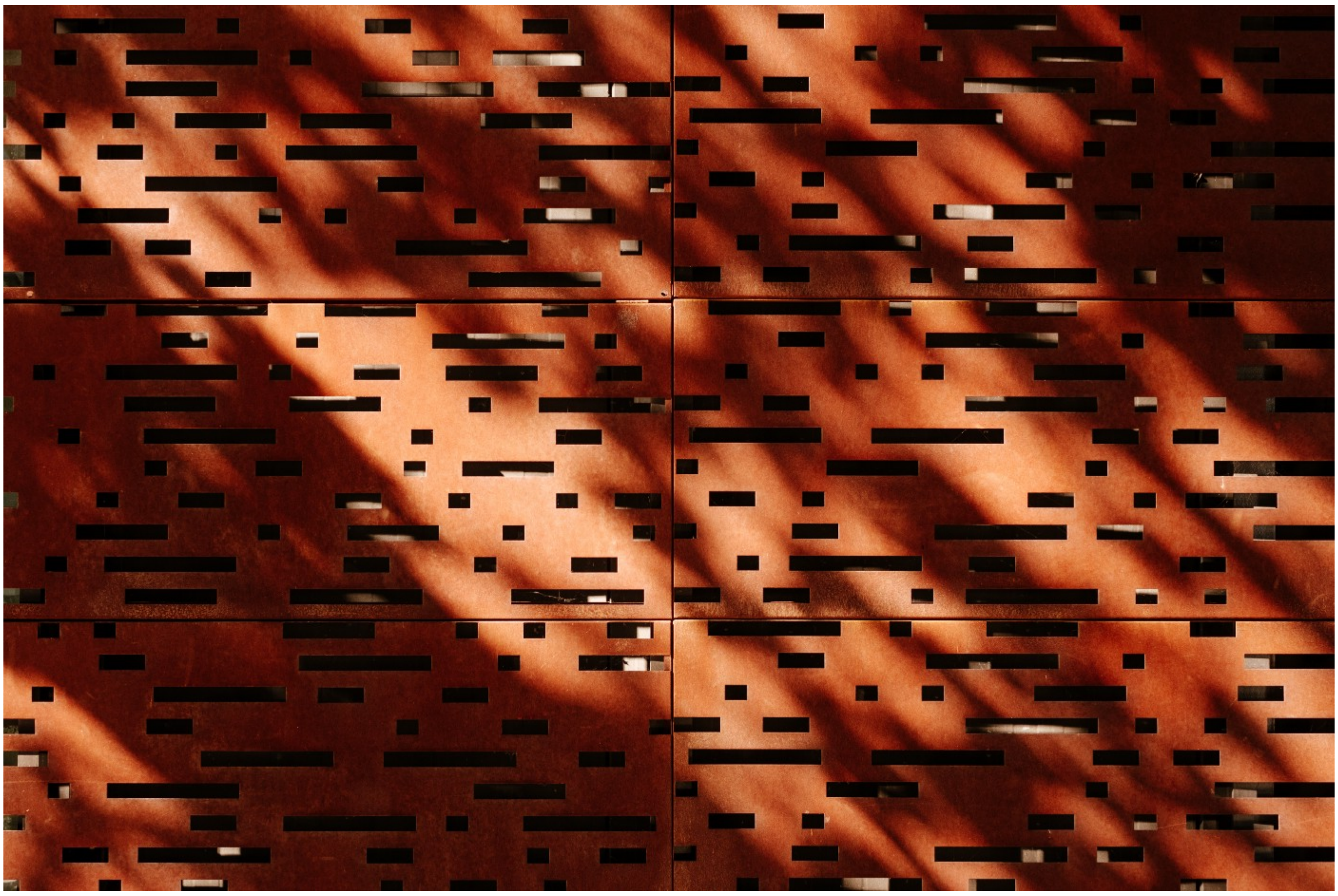

Photo by Samuel Zeller on Unsplash

Within this generative structure of strong culture and leadership, on a practical level, the system used containers and microservices, but that wasn't what made it work.

> What really made the difference, day-to-day for pace, progress and results, was a discipline of simplicity in design.

Simplicity not just in technology design, but also in how we expressed agile, designing our work environment and iterating our processes to minimise friction and fit the dynamic of the team. From user experience to team culture to architecture, design made it work.

The twelve factor app

The twelve factor app is an influential collection of practical design principles which grew out of the experiences of the team at Heroku. If you work with technology and haven't come across them, I'd encourage you get to know them: https://12factor.net

> These principles speak softly and carry big sticks.

I see in them the kind of hard-won ease that flows from experience and mastery. Along with the Government Design Principles, 12factor is one of my go-to resources, providing a demanding framework in which I can draw out great design.

I've found myself returning to the twelve factors with the work of my current team and it's got me thinking: I don't suffer out-of-context design lightly, whether from myself or from others. Technical architects expounding me-too aspirations of Google or Netflix scale platforms don't end well with me.

So I'm taking my own test. The twelve factors make sense for single-deployable monolithic web applications, but do they still make sense in containers?

Unboxing the twelve factors

Let's summarise and contextualise. To quote the introduction:

> *Our motivation is to raise awareness of some systemic problems we've seen in modern application development … and to offer a set of broad conceptual solutions to those problems*

I’ve learnt over the years that when it comes to delivery, it’s important to be specific before generic. Generic is really, really, really hard. Which is why aspirations of reuse are more likely to harm than help a project.

> I’ve also learnt that more often than not the right answer is not binary.

Running through those specific designs and implementations there are, nonetheless, generic patterns. Good principles make terrible laws, but held lightly, they provide helpful constraints that might guide you towards a range of good answers.

Specific and generic

I’m arriving in a place. From here my view is that specific detail is as useless as generic platitude when either ignores or belittles the other. In teams, the unit of delivery is the collective and in design the unit of delivery is “yes, and” — depolarising without losing creative tension. A multidisciplinary team where each person knows they are individually necessary but not sufficient for delivery is when a team works.

> So I like broad concepts. A lot.

My favourite Government Design Principle is “Do less”. To me it speaks of the twin traits of humility and confidence. When found together, there’s no need to hide behind a smokescreen of vagueness and clutter. Specific, direct, polite. Clarity is a gift of character. At the same time, “do less” yields no value on its own, until the principled rubber hits the specific road.

Show me an architecture with one too many boxes in it and I’ll say “How could we do less here? What can we subtract to make this better?” If complexity can’t be justified, it’s complicatedness. That’s a red flag for delivery, for maintenance and for surviving the rigours of production. It’s unacceptable risk and it needs to go.

> So I like specifics. A lot.

And I see that these aren’t a contradiction. So lets look at the twelve factors in the specific context of containers.

The twelve

Let’s take a look at each of the principles and how they sit in a container world:

1. **One codebase, tracked in revision control**: to quote, “There is always a one-to-one correlation between the codebase and the app”. In this case an app would correlate to a container, probably a microservice.

2. **Explicitly declare and isolate dependencies**: this principle is well supported in a world of containers. “A twelve-factor app never relies on implicit existence of system-wide packages”, which is very much what container isolation gives you. A container really can contain only the files needed to do its work. With a bit of effort and determination the number of files can be counted on the fingers of one hand.

3. **Store config in the environment**: if there’s a sweet spot for how containers constrain your design to improve yours results, this could well be it. Increasingly it feels uncomfortable to do self-harming things. If you’ve come across the term “immutable artifact”, the idea that software should be built once and the resulting “artifact” deployed unchanged in each environment to avoid unpredictable variations, this is it. *The container is the unit of immutable deployment and the config values provided by an environment are the wiring that plugs it in to the rest of the system.*

4. **Treat backing services as attached resources**: when it comes to databases, queues and file storage, containers nudge you further away from storing data and state in the app. In a world where containers can appear and disappear by the bucketload from moment to moment, it becomes really important to externalise data storage. It then becomes ever clearer that delegating the operation of data storage components to your cloud provider makes sense— they’re an externality that’s going to distract from your core value.

5. **Strictly separate build and run stages**: this one’s a no-brainer with containers. You build an image, push it to a registry and then pull it to a target machine to run as a container. There are ways to muddy this, but you’ll have to work ever harder to make life difficult for yourself in this respect.

6. **Execute the app as one or more stateless processes**: here again, containers take 12-factor a step further. They really are only designed to be a single process, isolated inside a dedicated filesystem. You absolutely can insist on finding ways to get around this, but I think you’ll find pretty quickly that your code smells riper than a well-aged blue cheese. Accept the constraint, it’s good for your design.

7. **Export services via port binding**: this one may not be obvious at first, so here’s a quote: “The contract with the execution environment is binding to a port to serve requests”. This typically holds true for containerised microservices — all the gubbins needed to respond to requests is included in the container image and the only way to

connect to the service is via a port on the container. One particularly neat thing about containers is that each one has a separate address space, so there's potentially even less effort needed to figure out which port to bind to.

8. **Scale out via the process model**: the 12-factor approach to concurrency and scaling is "horizontal" or "scale out". Containers are very much designed with this idea in mind. With a stateless design, you should normally be able to run multiple copies of it to provide additional capacity. This principle is a default assumption in Docker Compose, Docker Swarm and Kubernetes.

9. **Maximize robustness with fast startup and graceful shutdown**: the principle of disposability is a corollary to the ability to scale out. If you can start an additional dozen copies of a microservice, you'll also want to design for the ability to take instances out of service, whether for reduced capacity or rolling updates. I'll touch on startup time here as it's a serious point of friction that I've seen ignored in architecture conversations, particularly where there's a preference for using the JVM, (Java Virtual Machine) especially with the Spring framework. Slow starts (by which I mean anything over one second) destroy both development cycle time and operational agility whilst loading up your cloud hosting bill. In short, expect to pay a lot of money for not a lot of progress. Don't shoot the messenger.

10. **Keep development, staging, and production as similar as possible**: this is a big one for containers. Assuming you're building an image once, in a predictable way, and deploying it to your environments, this is going to help you ensure you're running exactly the same build in each environment. There's more to this principle, but I think it's good to highlight that here again containers are providing a nudge towards 12-factor design.

11. **Treat logs as event streams**: this is one of my favourite container wins. Log rotation was great fun, but ultimately added no real value to applications. I don't think I've seen anyone argue with the practice of writing logs to *stdout* and allowing them to be routed to the container runtime or off to log storage and analysis — ELK, Prometheus, InfluxDB, Stackdriver — the list is long and varied (and your mileage likewise).

12. **Run admin/management tasks as one-off processes**: if you've ever considered that containers have a lot of similarities to functions (e.g. Lambda, GCP functions) then this principle makes perfect sense. If you've come across the intricate solutions that have been touted over the years for database migrations, you may agree that none of them are particularly elegant they generally hang around in your production code like an awkward gooseberry after startup. Putting a one-off task into a one-shot

container that starts, does its work and then exits, is a great way to keep things clean.

The design nudge

For me it’s clear that containers are a natural extension of 12-factor thinking. Ultimately, the ability to do anything at any time isn’t helpful, so convergence at a design level, whilst maintaining an open mind on specifics, helps us bring useful lessons into individual situations.

> *Design constraints give you nudges and apply pressure to help narrow your options and make you think harder about what good looks like.*

Docker Twelve Factor Containers Design In Tech