the cloud is a steamy pile of shit

lttle.cloud | the manifesto

hello, Steve! please have a seat! it's been a long time since we last spoke. how have you been?

[Steve]: I'm good, tha–

[interrupting with a burst of excitement] yeah yeah, I've been good too. i'm working on this… thing. it's called lttle.cloud. i've been so frustrated with the cloud lately, and i just couldn't take it anymore.

[Steve, looking confused]: what do you mean? why? what's wrong with the cloud?

[ACT I: paying for idle]
yeah. well. there are multiple problems, but for starters, we're literally **drowning in inefficiencies**. let's say you're just done building the mvp for the next unicorn; but, before you can quit your job and go full in on your ai girlfriend saas, there is one more step you need to take. you have to deploy your app, and most of the time, that means getting it in the cloud.

so, you do that. you rent a small ec2 instance on aws, get a domain, and wait for the customers and money to start rolling in. and you wait, and you wait a little longer, but no users are coming. maybe your idea wasn't as good as you thought, maybe the marketing wasn't good, what's certain is that you don't make any money.

but do you know who makes money? aws. if your vm does useful work (aka serve requests for your users), or it sits idle, it doesn't matter–**you're billed the same amount**. and given that you don't have any users yet, you're literally **throwing money in the trashcan**.

[ACT II: scaling is a dangerous road]
but, let's say that suddenly, your ai girlfriend app blows up on the waifu subreddit. if we do some basic math, we get the following: **users** = **traffic** = **useful work**. problem solved, right?

that's true… but there's a bit more math we need to do:
if **users** = **traffic** = **useful work**, that implies that **more users** = **more traffic** = **more useful work**, and that implies that **even more users** = **even more traffic** = … **too much useful work**.

while we were focused on saving money, we forgot about scaling. so, let's scale up. just spin up a few more machines and add a load balancer in front.

now, the app is **blazingly fast™** and the users are happy. money starts rolling in.

but while we were focused on scale, we lost focus on the wallet again. looking at the traffic patterns, you see that for whatever reason, your workload spikes in 5-10 minute **bursts**, during only the night (weird crowd, i know). you are **overprovisioning**, so even if you have a lot of users, you're still **paying for idle** and losing money overall. seems like we just can't catch a break.

but what about autoscaling? we can create an ec2 autoscaling group and set a minimum number of replicas and a maximum number of replicas. the group will scale up/down based on the load.

could work, but the cloud strikes again! your traffic is not predictable—it **changes** on a **scale of milliseconds**, but the autoscaler can react on a **scale of tens of seconds or even minutes**. this means that you could be scaling up to handle a burst, and the burst could be over before you're done scaling up. heck, it takes 30 seconds or more to even start a freaking vm on aws.

[Steve]: ah i see. well, maybe you should've started with serverless.

[ACT III: the lies of serverless]
maybe you're right. maybe. the promise of serverless is just so good. almost too good to be true. pay only for what you use, scale infinitely to handle any amount of load you might encounter.

a model so simple and powerful: **1 request** = **1 instance**; it starts when the request comes, and it gets killed when it's done

but, there's a tiny problem (or a few). **serverless is a lie**.

let's pick aws lambda as an example, but this applies to the other serverless offerings too (gcp functions and whatever azure has).

when you make a request, if no lambda is already available, aws has to spin up an instance with your code before it can handle the request.

this is called a **cold start**, and it can take (depending on your runtime) from **a few hundred milliseconds** to **a few seconds**. cold starts directly translate to a **degraded user experience**, as we just introduced a massive latency in the hot path. even if your code can deliver a response in < 10ms, it doesn't matter—lambda will add at least 200ms on top of it. we have, again, idle time.

and i think you're used to this by now, but who do you think pays for idle time? you, of course.

but wait, it gets even worse. by default, a lambda has a **request concurrency of 1**. yep. you heard me right. your cloud spins up a lambda instance to handle a request, and while it does that, if another request comes, it needs to spawn another lambda.

however, after the request is done, the lambda hangs around for a few minutes (we call this a warm lambda), just in case new requests arrive. say it with me: **idle time, again!** and again, you pay for this!

there is a way to avoid the penalties of cold starts, by having a timer service sending requests regularly to your lambda. aws conveniently offers a feature in cloudwatch that can do just that.

keep in mind this doesn't work if, god forbid, you get hit by concurrent requests, because you are warming only one lambda. you can keep multiple lambdas warm to avoid this, but at this point, you might as well go back to regular vms or containers.

do you even hear the stupidity of this? cold starts are their problem, and yet, **you pay the price**.

so, here's a quick recap: you can deploy your app in the simplest way possible on a vm, but you'll pay for idle, and you won't be able to scale with usage demand. or, you can use autoscaling, but it's too slow and you still pay for idle. or, you can use serverless alternatives, but you'll be hit with complexity, cold starts, and you'll still pay for idle. choose your poison.

[Steve]: man, that's sad.

yeah. our pain is making it rain money for jeff bezos. we're drowning in inefficiency and they don't do anything about it.

why? because it works and because it's profitable. there's no incentive for them to improve their services. why would they? there's no alternative. it's not like everyone can build their own cloud.

[ACT IV: a walled garden]

and even if you make it work on one cloud, you will be so **tied to that cloud's services**, that switching will be… at least not easy.

you start with a lambda, and of course you'll also need an api gateway. most likely your app will need file storage, so let's plug an s3 bucket into the mix. you might also want to use their service x but to do that you need to connect it to your lambda with their event bus product. or, you might need to put your lambdas (multiple at this point) in a vpc to access private resources in your infrastructure, like a database. but if you want to connect to other public services from your lambdas, you need to add an internet gateway to your vpc. what? your app is slow? maybe it's time to move that database in aws too, and while you're at it, why not add a managed redis instance to cache some of the slow operations? need indexing or search? no problem, use their managed elastic stack alternative. what? your company launched a satellite and you need radio communications with it? no problem. aws ground station to the rescue. your satellite operations plugged directly in your cloud stack.

and if you want to move from on-premise to aws but you have petabytes of data, there's no problem. you can rent a freaking truck filled with ssds from aws, load it up at your on-prem location, send it back to aws and they will bulk import that on your shiny cloud stack. this is not a joke. i'm not making this up. it's called aws snowball, look it up.

there is **no escaping the cloud once you're that deep into it**. it's almost like they are **doing this intentionally**…

they want to build this **walled garden**, but they invested so much time in the wall, and so little in the garden that instead we get a **tightly fenced, huge and steamy pile of shit**.

you can't escape the cloud… i can't escape the cloud. they got us good.

[ACT V: the reveal]
so yeah, i'm building a cloud to solve these problems.

[Steve, searching for his words]: wait, what?? didn't you say it's not possible?

i didn't say you can't do it. i only said it would be hard, and not worth it for everybody to build themselves one.

imagine a **serverless-like platform**, with a few twists:

- it uses **hardware-level isolation** for workloads, this is a must for multi-tenancy. no containers, not good enough.
- it has **no cold starts**. your workload always **starts in less than 10ms from when a request arrives**. also, it can **handle concurrent requests**, **autoscaling**, and it has a **timeout of less than 10 seconds** before killing the instance. you truly **only pay for what you use**.
- you can deploy your app or other services (like postgres and redis) with **0 modifications**. just ship it as a docker image.
- designed to provide very **high density** of guests on the same host (well into the thousands).

now, what if i told you that you can already **try this out here**. it's just a demo for now, but the runtime is **fully functional**. you can read more about the crazy tech that makes it work **here**.

[Steve]: ok, ok! let's say that this runtime mostly solves the problem with the inefficient cloud, but what about the walled garden? won't you end up just creating another hard-to-escape cloud offering?

no. and here's why. everything is built in public, **open-sourced** with a very permissive license. the runtime prototype is already available **here**.

you can **build your own cloud if you want**. start your own single node, just for you. deploy it on your development server and cut down the costs of having multiple versions of multiple projects running at the same time. or, just use it in production.

also, lttle will always be compatible exclusively with **OCI images**, including existing ones. we won't create our own managed services variants. if you want a database, just deploy a postgres image with a persistent volume yourself.

if that sounds hard, don't worry, there will be a way to deploy service templates (ex. from github) and configure them. instead of figuring out how to do postgres with read-only replicas, you will be able to configure and deploy an existing template made by someone who knows postgres better.

steve, still with me? you've been silent for some time. tell me, what do you think?

[Steve]: