

# Building the next cloud compute primitive — in Rust

Luca Casonato

# Who am I?

**Luca Casonato**

Software Engineer at Deno Land Inc

Working on Deno and Deno KV

Previously lead of Deno Deploy team

Now mostly special projects



I do standards work in the JavaScript space: WHATWG, W3C, TC39, WinterCG



Building the next cloud compute primitive — in Rust

Luca Casonato [lcas.dev](https://lcas.dev) [@lcasdev](https://twitter.com/lcasdev)

# What is Deno?

Deno is a next-generation JavaScript runtime.

Unlike the Rust ecosystem, the JavaScript ecosystem is terrible.

JS tooling and ecosystem are fragmented and fragile.

No single standard library, no standard formatter, linter, doc generator, testing, etc.

We're fixing this. **Deno is an all in one toolbox for developing web apps.**



Building the next cloud compute primitive — in Rust

Luca Casonato [lucas.dev](https://lucas.dev) [@lucasdev](https://twitter.com/lucasdev)

# Demo



Building the next cloud compute primitive — in Rust

Luca Casonato [lcas.dev](https://lucas.dev) [@lcasdev](https://twitter.com/lcasdev)

# Deno ❤️ Rust

Deno is built in Rust. We maintain >500k LOC of Rust.

About 70-80% of our code is open source and published to crates.io.

We publish >55 crates ourselves: *v8*, *urlpattern*, *fastwebsocket*, *monch*, *acme2*, *rustls-tokio-stream*

And help maintain others: *rust-url*, *wgpu*



Building the next cloud compute primitive — in Rust

Luca Casonato [lucas.dev](https://lucas.dev) [@lucasdev](https://twitter.com/lucasdev)

# Deno is built on the shoulders of giants

We don't build our own JavaScript engine (VM), but use Google's V8 (C++).

We provide safe, idiomatic Rust bindings to V8 as a crate.

All of our services are powered by the excellent Tokio and Hyper crates.



# But why Rust?

Deno was originally written in Go.

Go is a great language for async HTTP servers. But not reliable, strict, customizable, and performant enough.

Needed something strict (reliable), lightweight, performant, and customizable.



## Rust provided us this

Result, Option, and ADT enums enforces exhaustive error handling and recovery.

Explicit memory makes performance characteristics transparent. Borrow checker enforces memory correctness.

Can easily(-ish) integrate with C++ (needed to bind to V8).

No hidden runtime overhead (GC, implicit atomics, etc).

**Incredible** built-in tooling.





## Tangent: built in tooling

Rust users are incredibly lucky to have such an excellent set of built in tooling.

- linter
- formatter
- testing
- documentation generator
- package manager

Most ecosystems do not have this (C++, JS, Swift, C#, ...)



# Back to Deno: what we're building

So, we don't just build a runtime and tooling.

Really, Deno is a cloud company. We host your code on our platform using *isolates*. Global distribution, quick deployment, no config.

Isolates: the next frontier in cloud computing.



Building the next cloud compute primitive — in Rust

Luca Casonato [lucas.dev](https://lucas.dev) [@lucasdev](https://twitter.com/lucasdev)

# Demo



Building the next cloud compute primitive — in Rust

Luca Casonato [lucas.dev](https://lucas.dev) [@lucasdev](https://twitter.com/lucasdev)

# Cloud computing compared

	<b>Bare metal</b>	<b>On-prem VM</b>	<b>Cloud VM</b>	<b>Containers</b>	<b>Isolates</b>
<b>Deployment unit</b>	OS + Runtime + Code	OS + Runtime + Code	OS + Runtime + Code	Runtime + Code	Code
<b>Lead time</b>	days to weeks	hours to weeks	instant	instant	instant
<b>Investment</b>	Hardware	Hardware	-	-	-
<b>Usage billing</b>	-	-	Real time	Real time	CPU time
<b>Scale to zero</b>	✗	✗	✓	✓	✓
<b>Cold start time</b>	-	-	~10s	~1s	~0.1s
<b>Utilization</b>	Low	Low	Low	Higher	Highest



# Cold starts?

Scale to zero: don't run the app if there's no traffic

First request results in start of the app. This time is the cold start.

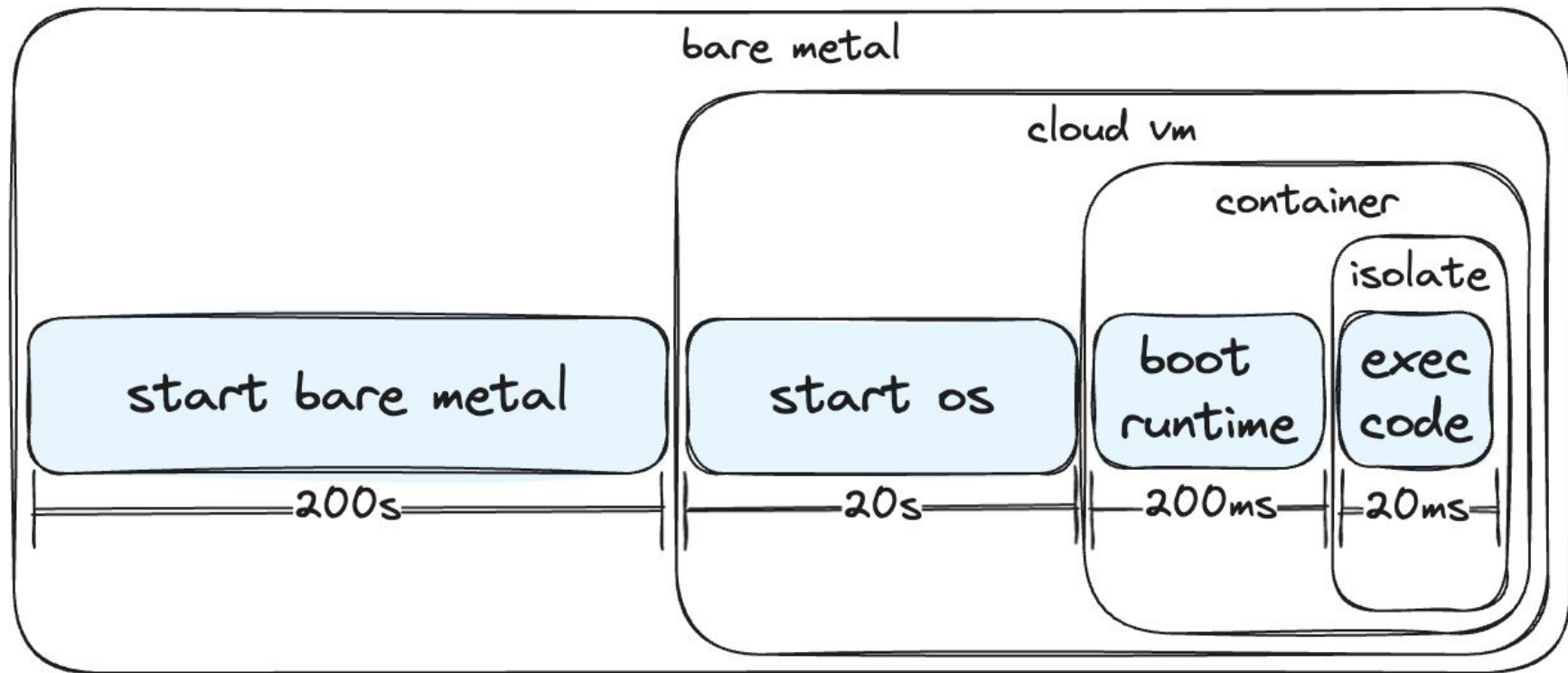
Cold starts result from the work performed that is specific to a tenant.

More work per tenant means higher cold starts.

**=> Reducing differences between tenants equals lower cold starts.**




























# Cold starts across compute primitives



not to scale

# Cold starts compared

 *Shared across tenants*  
 *Specific to a tenant*

	VM	Container	Isolate
Boot machine			
Start kernel			
Initialize networking stack			
Configure kernel virtualization	-		
Start and configure runtime			
Boot user code			
Connect to database			
Serve request			



# Utilization?

How well can you make use of the underlying hardware resources?

Billing by real time:

- Reserve some resources for every tenant
- Users pay for resources they didn't use

Billing by CPU time:

- Everyone uses as many resources as required (within limits)
- Pay only for resources used
- Requires achieving high average utilization





# Achieving high utilization

Requires high packing density.

Packing density: amount of tenants per physical resource.

High density requires efficient tenants.

Sharing resources between tenants requires less memory / CPU per tenant.

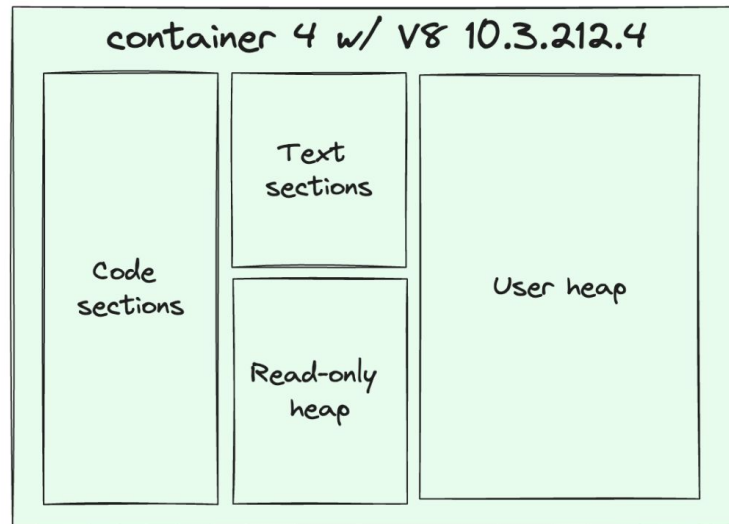
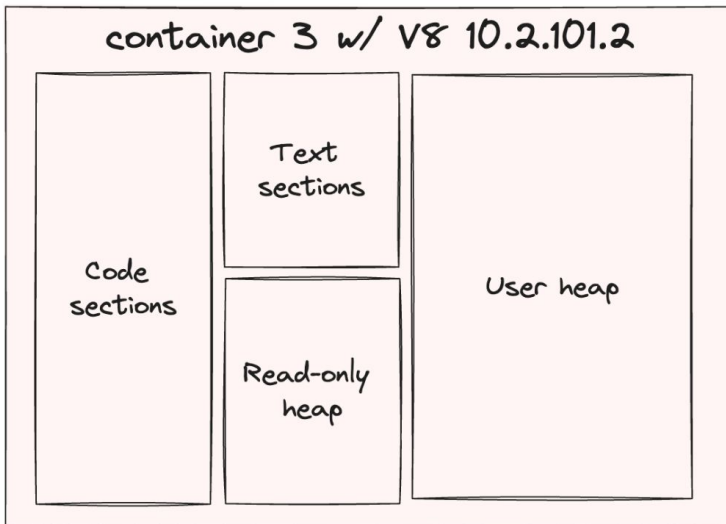
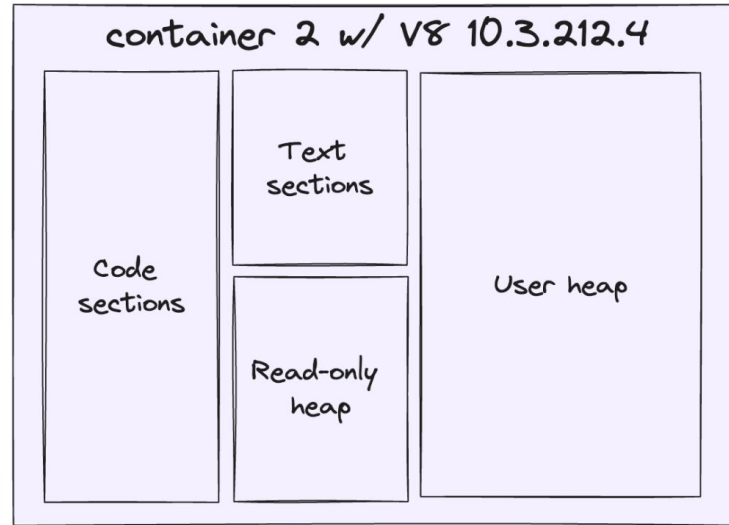
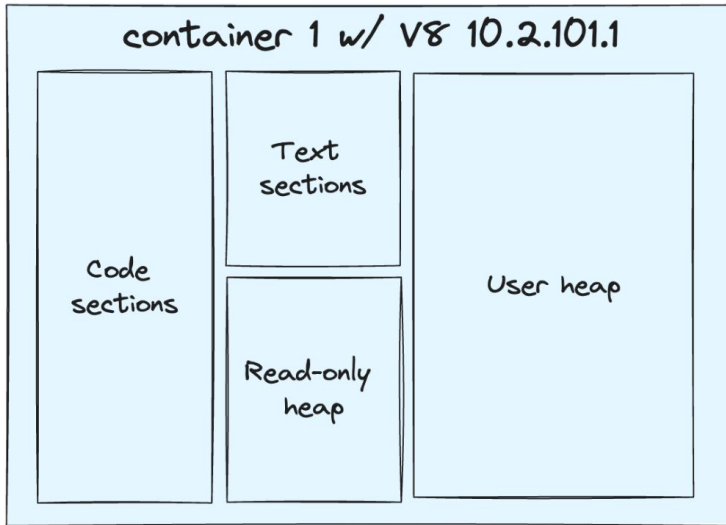
**Low baseline memory / CPU => higher density.**

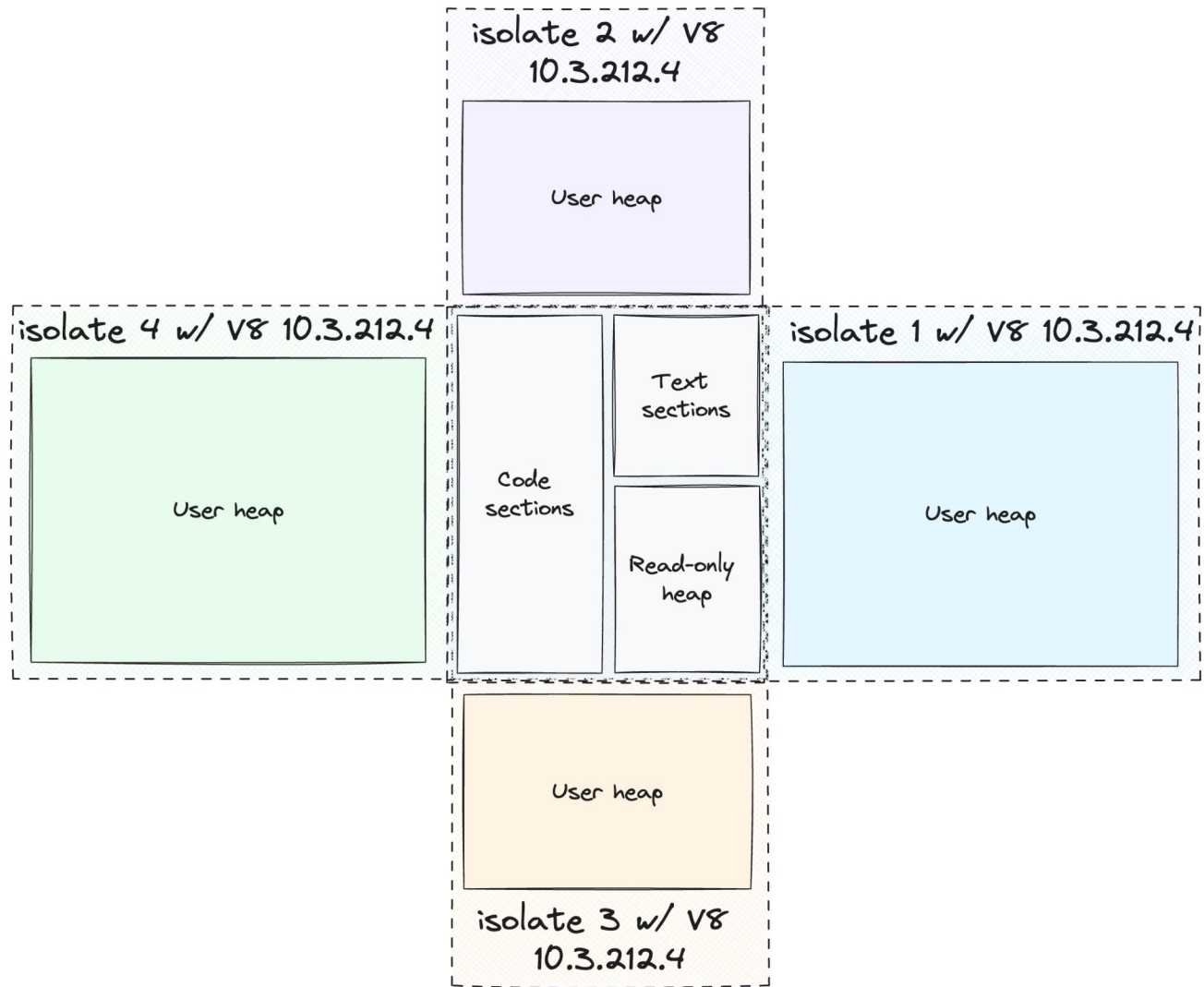


# Comparing baseline

	<b>VM</b>	<b>Container</b>	<b>Isolate</b>
<b>OS</b>	~100 MB	Shared	Shared
<b>Runtime</b>	~10 MB	~10 MB	Shared
<b>User heap</b>	~5 MB	~5 MB	~5 MB
<b>Baseline per tenant</b>	~115 MB	~15 MB	~5MB







# We're not just building a runtime

Also:

- A multi-tenant runtime hypervisor
- Load balancers
- Control plane API servers
- A distributed database



# These have different requirements

Hypervisors require:

- Low overhead
- Very high failure tolerance

HTTP load balancers require:

- High performance
- Predictable prioritization of work under high load

Control plane servers:

- Robust I/O with user (parsing, errors, HTTP)
- Third-party integration and SDKs



# Rust let's us solve all of these\*

**Low overhead**

✅ no GC to fight with about memory

**Very high failure tolerance**

✅ Result, Option, ADT enums, *thiserror*

**High performance**

✅ performance is explicit, not magic

**Predictable prioritization of work**

✅ manual scheduling with `Future::poll`

**Robust I/O with user**

✅ hyper, serde\_json, prost, etc

**Third-party integration and SDKs**

😞 This one is not good



# Rust is not great for API servers (yet)

**AWS**  but not production ready

**GCP**  no ongoing work

**Azure**  in development

**Stripe**  third party

**Twilio** 

...





# This problem gets worse

There are too many flavors a library could be developed for:

- synchronous
- tokio + openssl
- tokio + rustls
- async-std + openssl
- async-std + rustls

Finding a library for the API you need, doesn't mean you can use it.



# Our experience with this

You end up writing a lot of mini-SDKs for all the APIs yourself.

We internally have:

- Google Cloud Storage
- Google Cloud IAM
- Google Cloud Secrets Manager
- Google Cloud Instance Metadata
- Postmark
- NPM
- ACME (Let's Encrypt)
- GitHub
- *and probably more...*



# Hurdles appear at the extremes

Internal repository that builds 14 binaries from one Cargo workspace, with >25 test binaries.

Various challenges:

- Parallel incremental builds require upwards of 40GB of RAM
- No `cargo test` sharding
- No machine readable reporting (JUnit) in `cargo test`
- Effective CI caching is *very difficult*
- target/ directories upwards of 250 GB



## Example: custom `cargo test` tooling

Build test binaries on one machine

Execute on one or more runner machines

Use `--message-format=json` to convert JSON output of `cargo test` into JUnit



# Conclusion

- Rust is fantastic for building cloud services
- Work needed from vendors to ship good Rust SDKs
- Rust has fantastic built in tooling
- Larger teams: be prepared to be at the forefront of tooling innovation



# We're hiring!

Are you interested in working on the next generation of cloud computing?

- Your work has impact: >300k monthly users
- A lot of open source
- 100% of systems are Rust
- Fully remote

Come talk to me :)



Building the next cloud compute primitive — in Rust

Luca Casonato [lucas.dev](https://lucas.dev) [@lucasdev](https://twitter.com/lucasdev)

# Thanks!

<https://deno.com>



Building the next cloud compute primitive — in Rust

Luca Casonato | [lucas.dev](https://lucas.dev) | [@lucasdev](https://twitter.com/lucasdev)