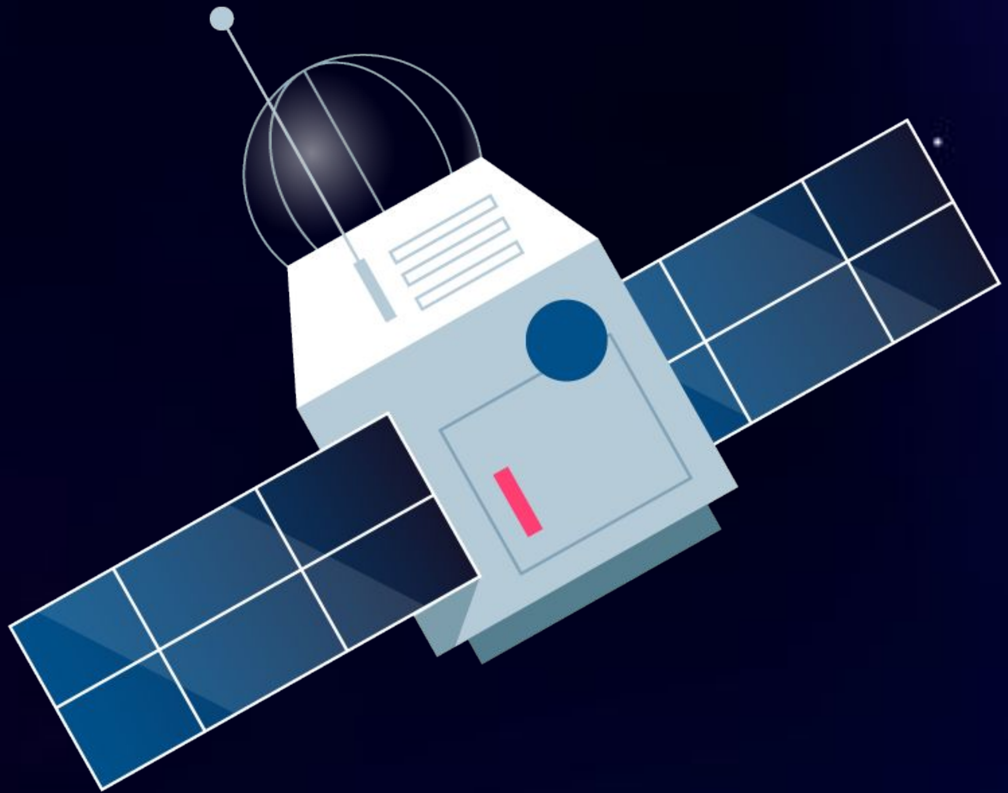
A Pragmatic Tour of Docker Filesystems

Jacob Howard

Docker Captain
Founder @ Mutagen

🐦 @xenoscopic



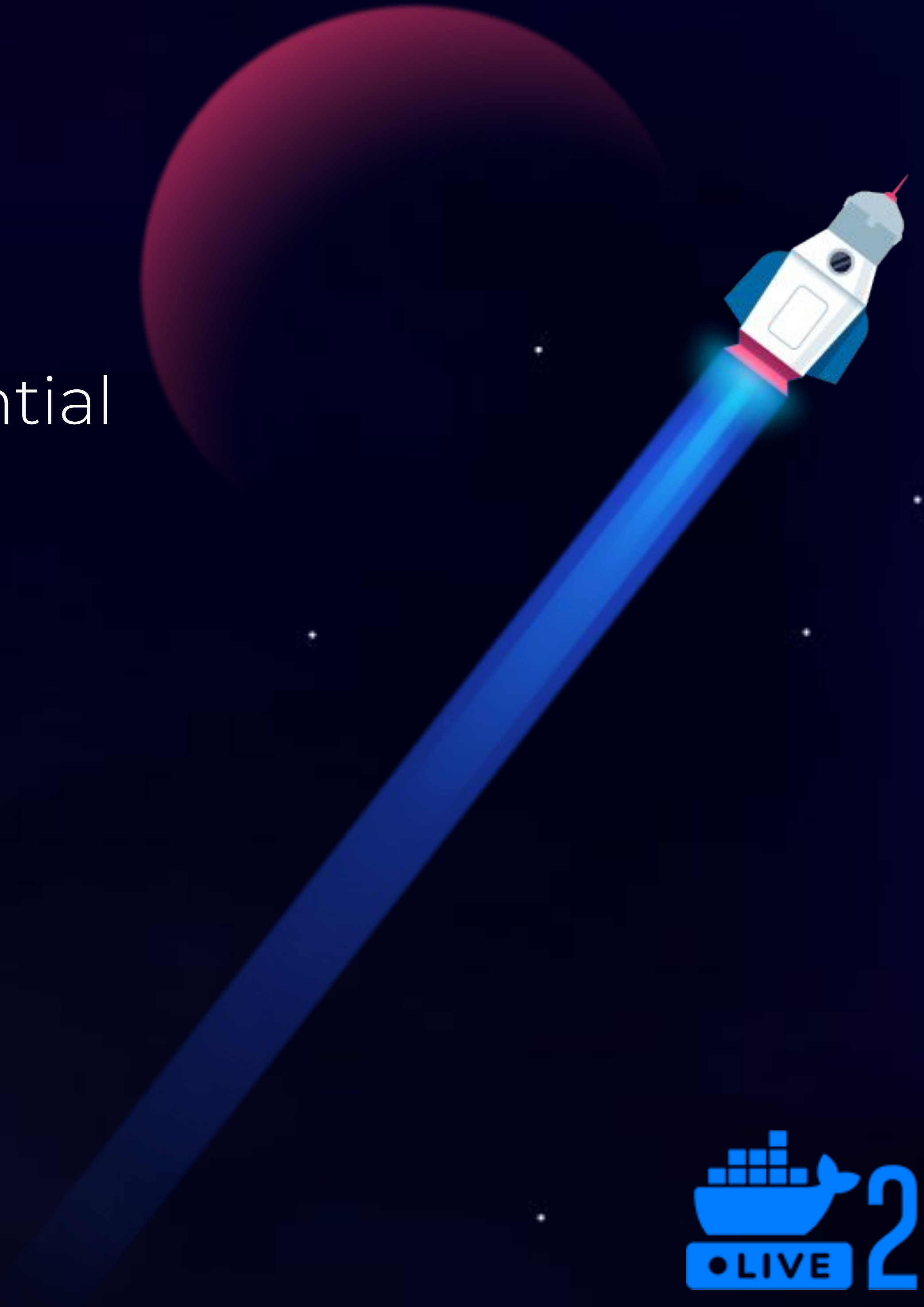
The Short Version: Container Filesystems Aren’t Magic…

1. **They’re Not Beyond Understanding**

The filesystem landscape is complex, but that complexity is essential and it can be leveraged for better performance.

2. **They’re Not Infinitely Performant**

Different filesystems have different purposes, behaviors, features, and performance characteristics.

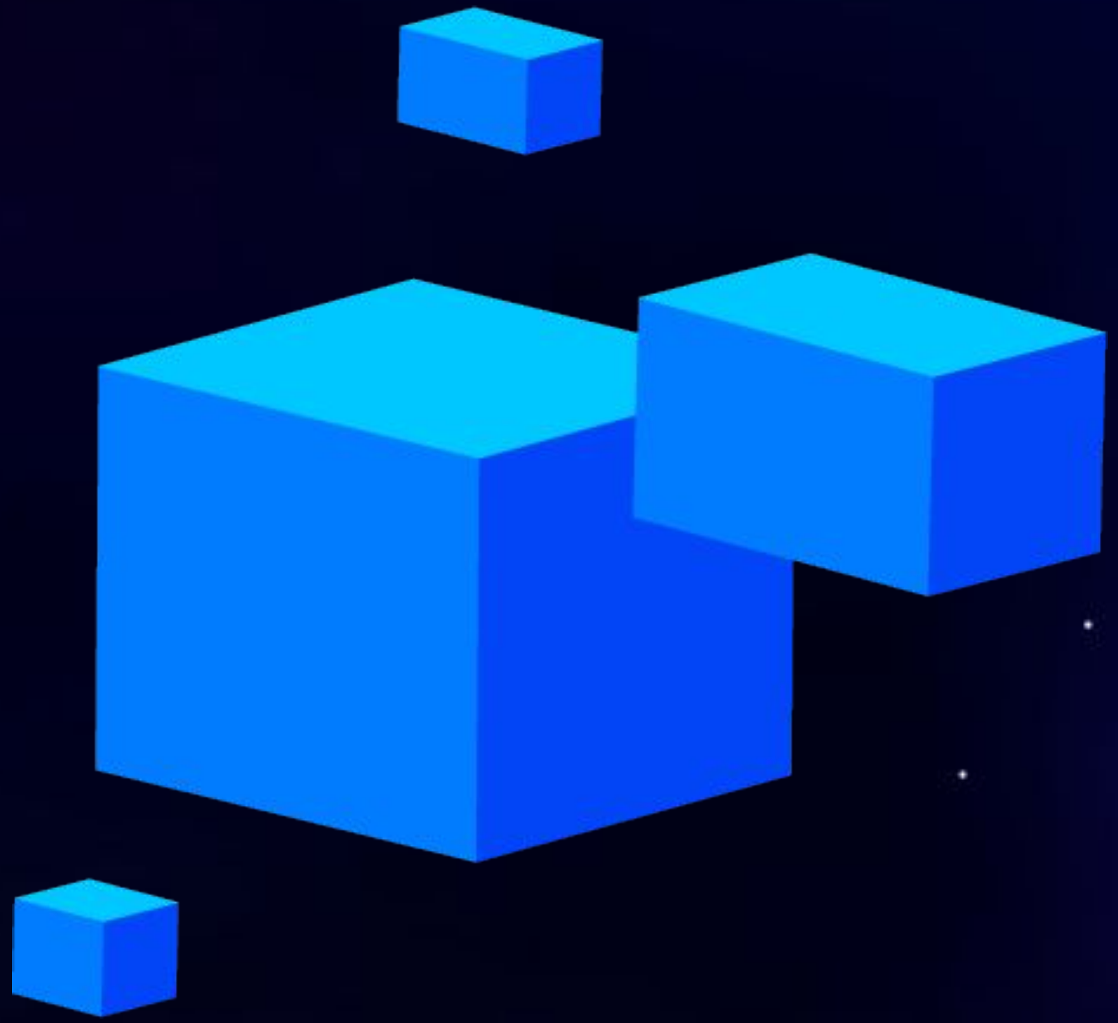


Core Concepts: Containers and Related Filesystems

What Are Containers?

Convenient isolation and portability

- Containers combine Linux kernel mechanisms like **namespaces** and **cgroups** into a *unified abstraction*
- Namespaces allow processes to have different views of OS resources
- Mount namespaces regulate the filesystems that containers can see
- Multiple filesystems are used to support the container abstraction

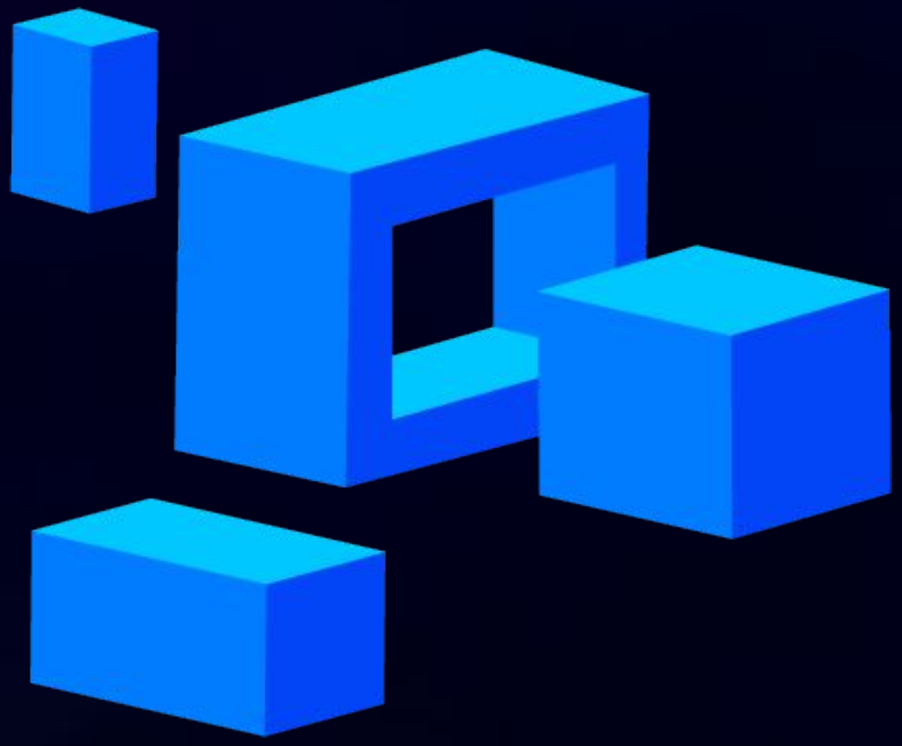


Which Filesystems Enter the Picture?

There are essentially five categories:

- Images
- Container root filesystems
- Bind mounts
- Volumes
- Temporary filesystems

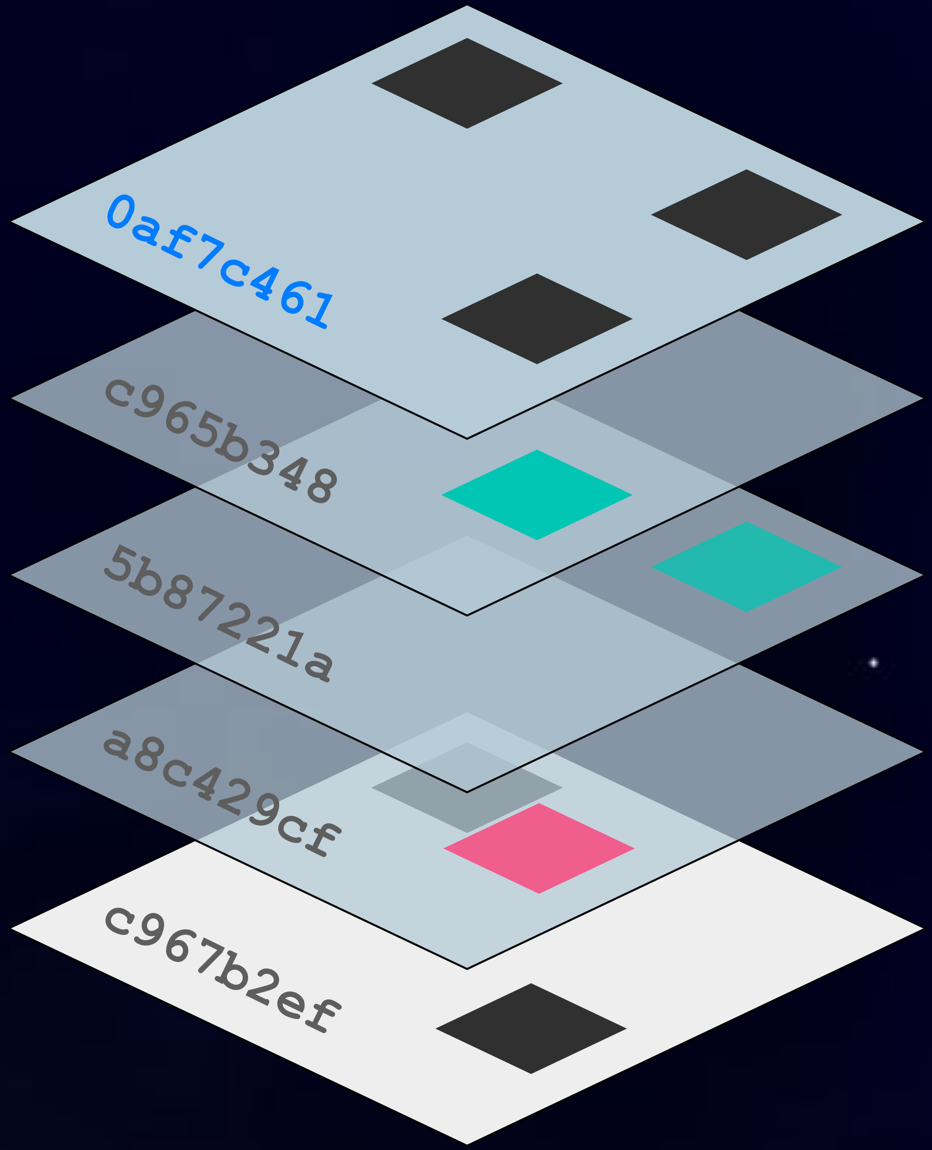
You can ***and should*** leverage each of these.



Images

Static distributable “filesystem” roots

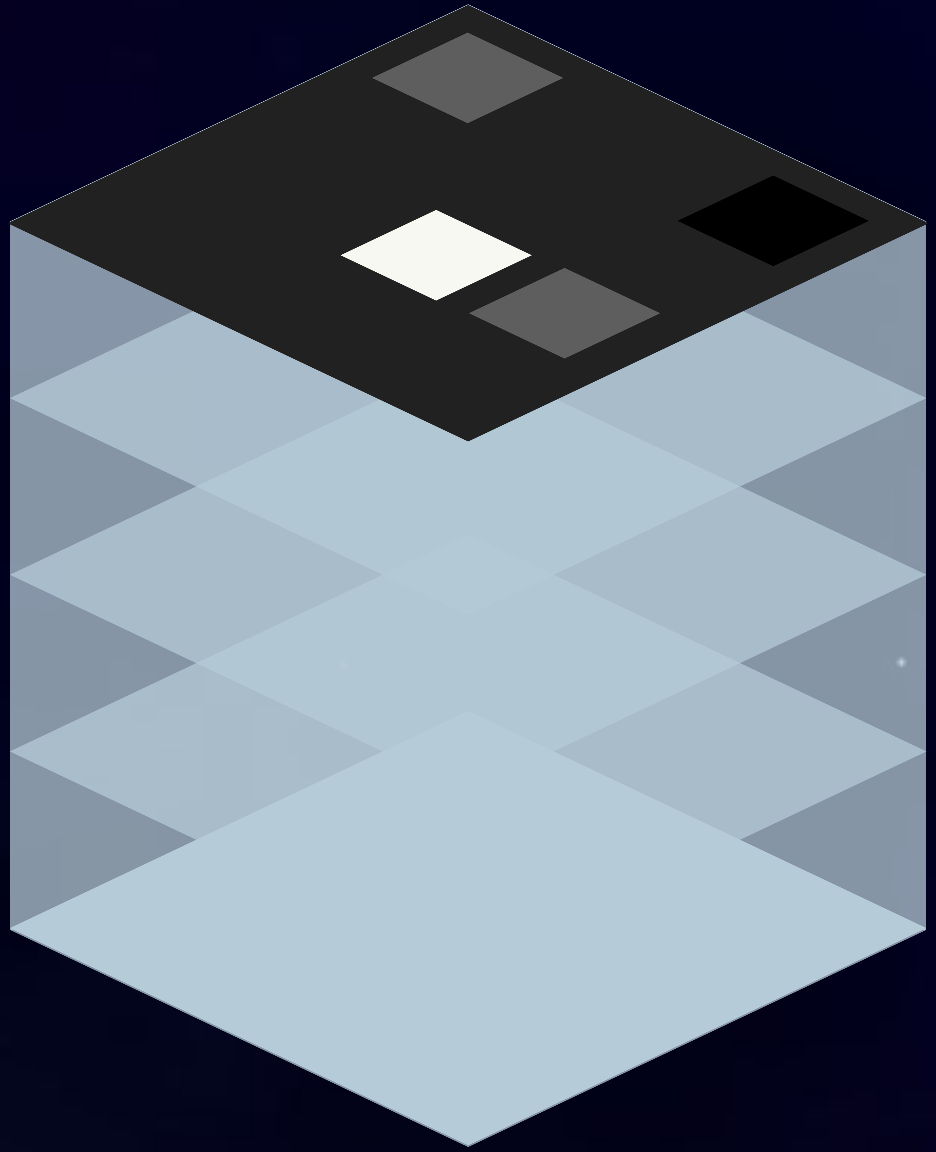
- Snapshots of a filesystem with metadata
- Built and distributed in a layered fashion
- Derived from a base image
- Stored and distributed as tarballs
- Standardized by OCI
- Excellent storage for tools (and some dependencies)



Container Root Filesystems

Images reified for use by containers

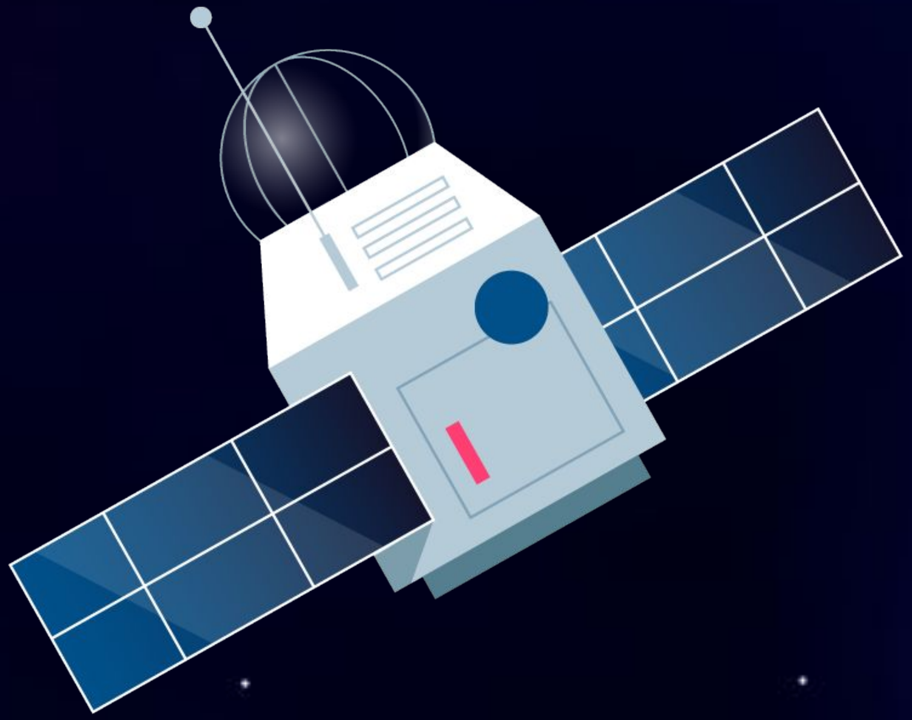
- Layers of an image converted to a mountable filesystem
- **OverlayFS** is the primary mechanism
  - Other storage drivers exist
- Can also track changes to generate new image layers from temporary containers
- Mutable but not persistent
- Reasonable performance for simple tasks



Bind Mounts

Host files made available to containers

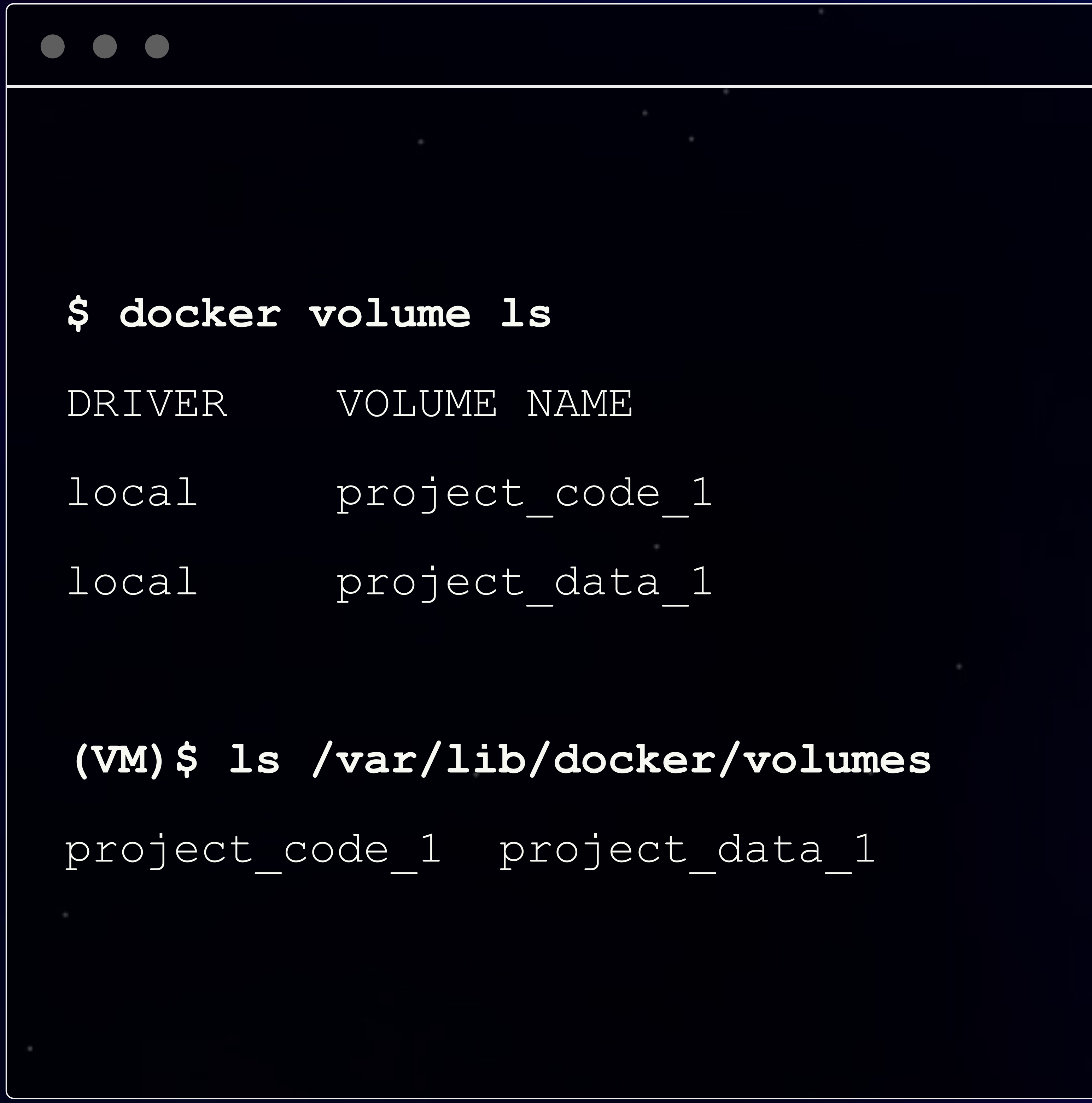
- Existing filesystem paths made accessible in different locations (even. across namespaces)
- Not something specific to containers
- No performance penalty ***natively***
- Implemented using **virtual filesystems** in Docker Desktop
  - **gRPC-FUSE** on macOS (previously osxfs)
  - 9P on Windows (but **native** via WSL2)
- Excellent for code you need to **edit**



Volumes

Persistent, performant, mutable storage

- Bind mounts with arbitrary storage
- Just folders in Docker Desktop
    - But **inside** the virtual machine!
- Plugins exist for alternative storage
- Can be attached to multiple containers simultaneously
- Excellent performance characteristics
- Great for storing data and/or code



```
$ docker volume ls
DRIVER    VOLUME NAME
local     project_code_1
local     project_data_1

(VM)$ ls /var/lib/docker/volumes
project_code_1  project_data_1
```

Temporary Filesystems

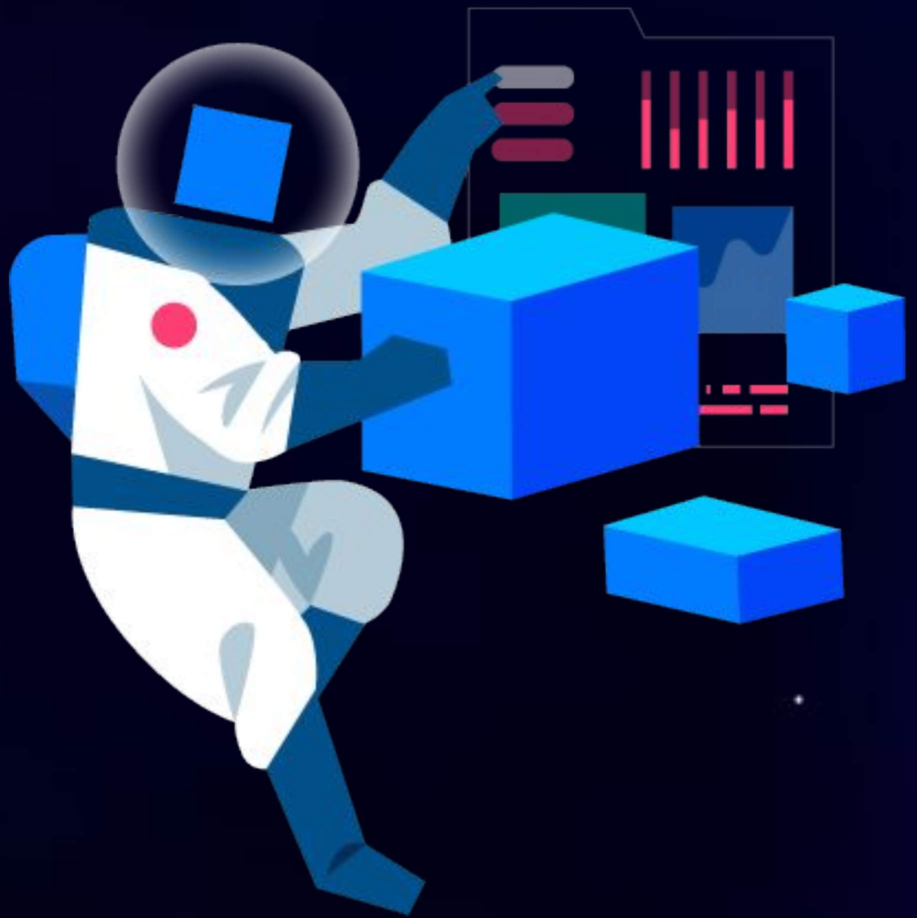
Ephemeral in-memory storage

- Standard Linux **tmpfs** filesystems
- Good performance
- **No persistence**
  - Not a good option for code

Performance Considerations for Containerized Development

Developer Tools Are Different (and Demanding)

- Filesystem access very different than casual computing or production use cases
- Assets loaded dynamically and repeatedly
- Typically O(n_{file}) behavior in terms of **`getdents`**, **`stat`**, **`open`**, **`read`**, and **`close`** system calls
- These don't behave as well on virtual filesystems
- Modern dependency management can easily bring in 10-100K files (or more)
- Also brutal in terms of CPU (e.g. compiling) and memory usage (e.g. linking)



How Should We Approach Filesystem Performance?

If things are fast enough, just leave them

There's no point in prematurely optimizing.

Figure out what slow programs are actually doing

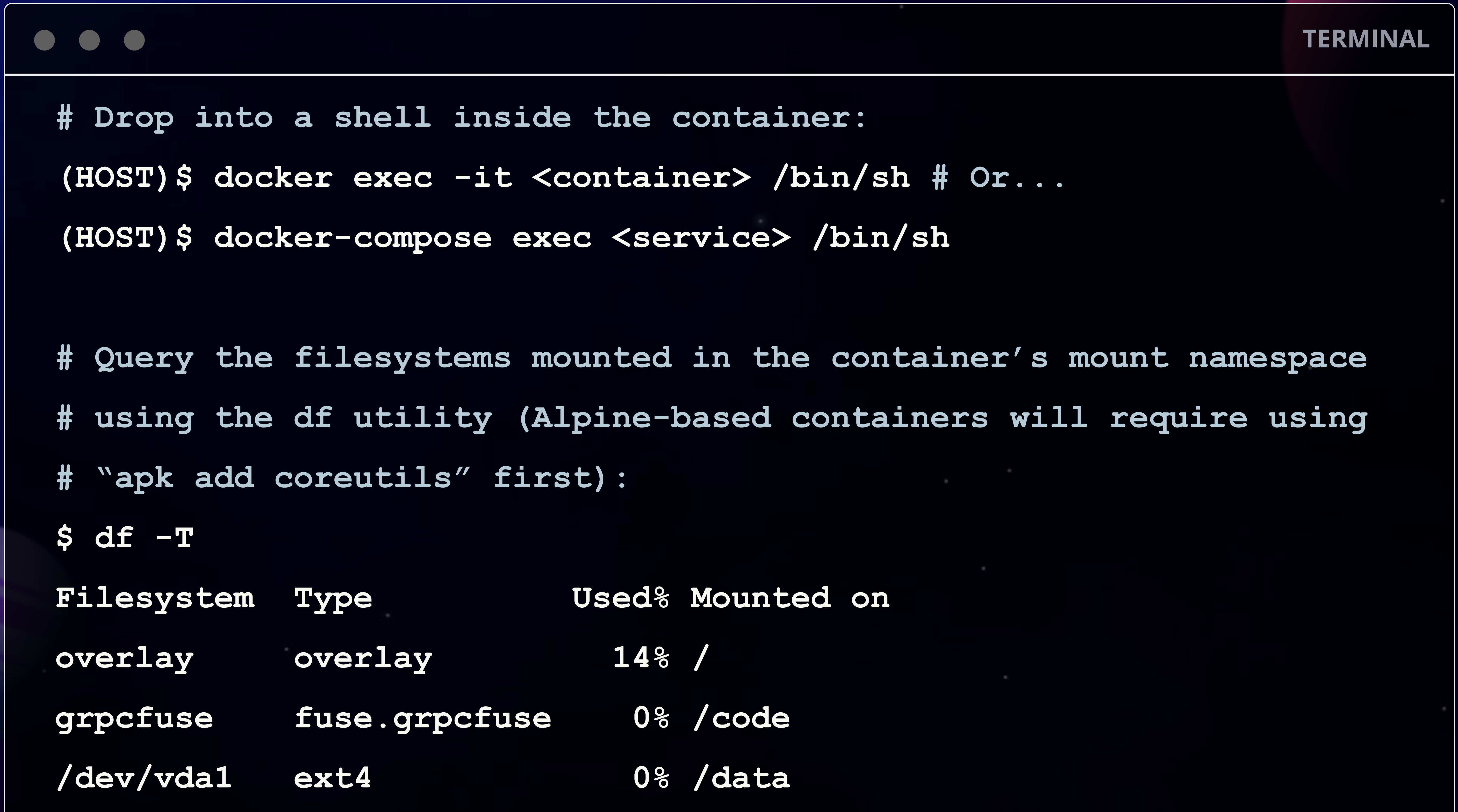
Macrobenchmarks aren't very informative. Microbenchmarks of the wrong thing are irrelevant. Understand *your* tools' system calls.

Perform comparative benchmarks of relevant operations on relevant systems

Use the actual software (or a representative simulation) and hardware to compare filesystems and understand potential gains.

Step 0

Understand Which Filesystems Are Being Used



TERMINAL

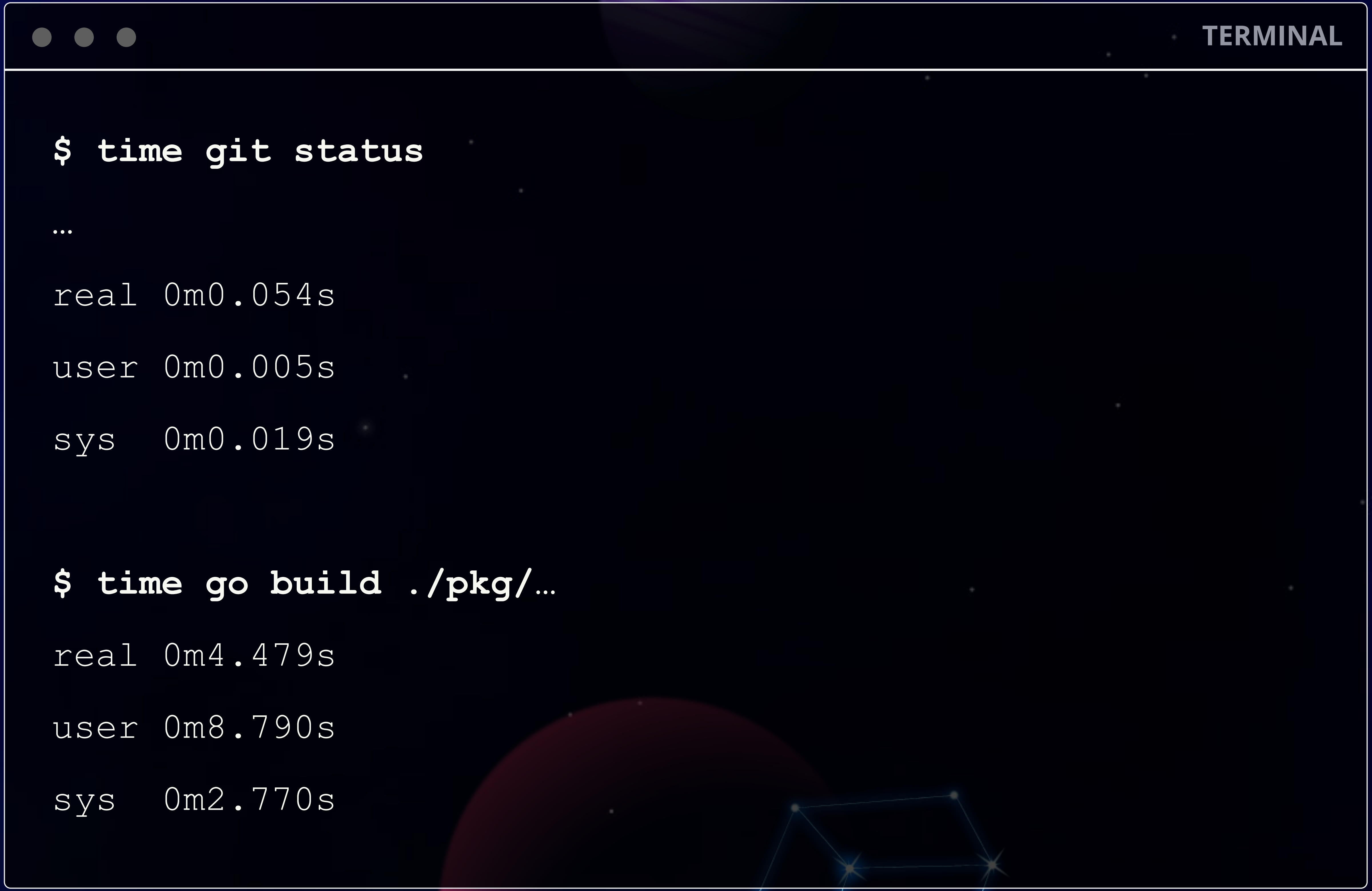
```
# Drop into a shell inside the container:
(HOST)$ docker exec -it <container> /bin/sh # Or...
(HOST)$ docker-compose exec <service> /bin/sh

# Query the filesystems mounted in the container's mount namespace
# using the df utility (Alpine-based containers will require using
# "apk add coreutils" first):
$ df -T
Filesystem  Type           Used% Mounted on
overlay     overlay          14% /
grpcfuse    fuse.grpcfuse     0% /code
/dev/vda1   ext4              0% /data
```

Step 1

Time Operations

Start by using high-level timing to identify problematic operations.



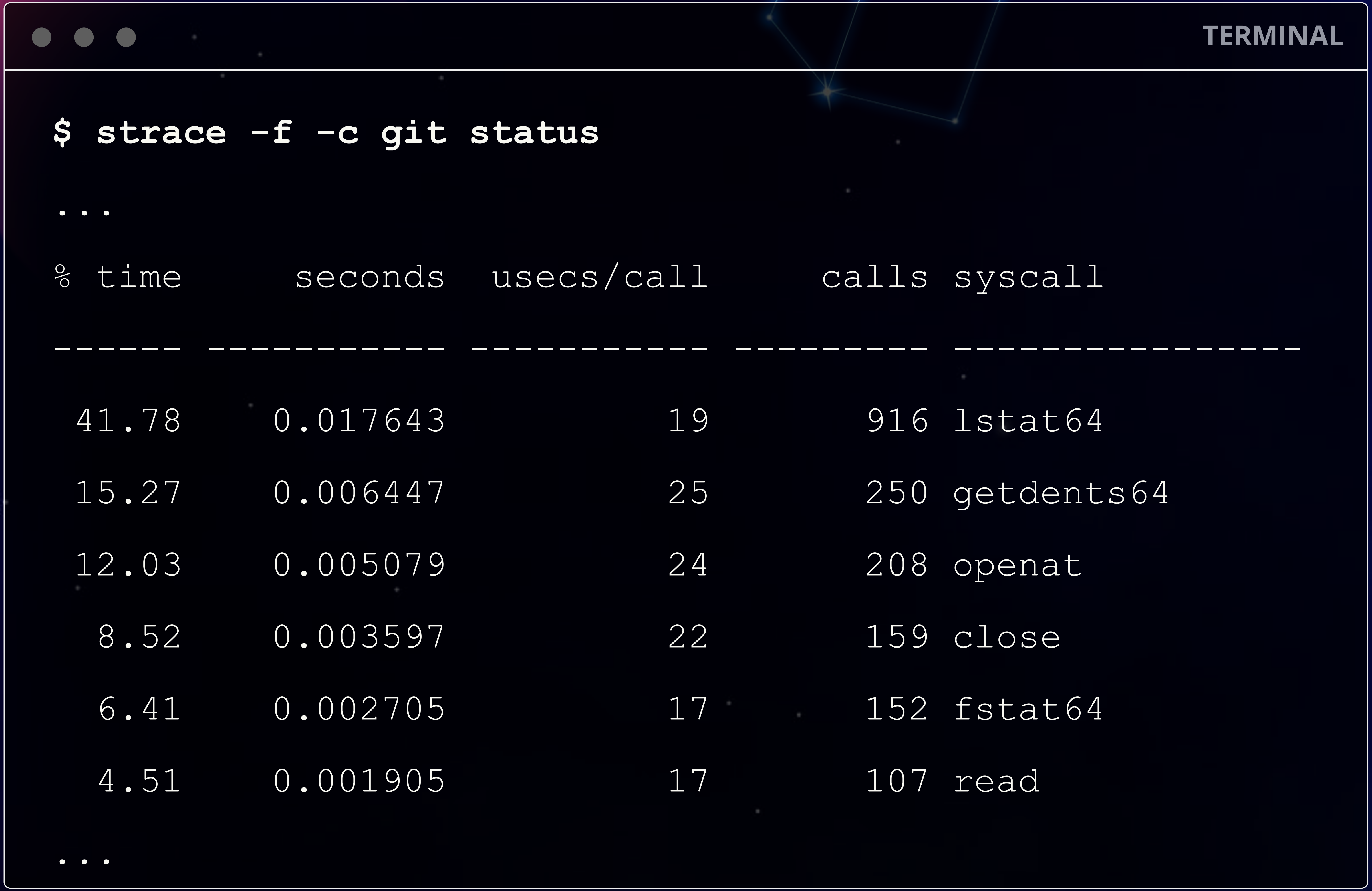
```
$ time git status
…
real  0m0.054s
user  0m0.005s
sys   0m0.019s

$ time go build ./pkg/…
real  0m4.479s
user  0m8.790s
sys   0m2.770s
```

Step 2

Trace Operations

Once you know which operations are slow, use tools like **`strace`** to understand what they're actually doing.



TERMINAL

```
$ strace -f -c git status
...
% time     seconds  usecs/call     calls syscall
------ ----------- ----------- --------- ----------------
 41.78    0.017643          19       916 lstat64
 15.27    0.006447          25       250 getdents64
 12.03    0.005079          24       208 openat
  8.52    0.003597          22       159 close
  6.41    0.002705          17       152 fstat64
  4.51    0.001905          17       107 read
...
```

Step 3

Trace in Detail

If programs are spending a suspicious amount of time doing something, dive into low-level traces to understand exactly what's going on.

```
$ strace -f git status 2>&1 | grep ‘lstat64’
lstat64("cmd/completion.go", {st_mode=S_IFREG|0644,
st_size=451, ...}) = 0
...

$ strace -f go build ./pkg/... 2>&1 | grep ‘jacob’
...
openat(AT_FDCWD, "/home/jacob/mutagen/pkg/configuration",
O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
...
```

The Next Steps...
(Depending on your situation)

Help the computer **do less work**

Identify suspicious or unnecessary work being done by tools and scripts. Removing this work is easier than optimizing.

Try alternatives and understand gains

Perform the same operations on different filesystems and understand the potential gains and tradeoffs.

Dig deeper with other tools

Use more advanced strace features or tools like eBPF to delve even deeper into slow system calls.

🔗 **https://aosabook.org/en/posa/ninja.html**

Recommendations, Strategies, and Rules of Thumb

Understand How Container Filesystems Compare

Performance isn’t the entire picture...

	Images	Root Filesystems	Virtual Filesystems	Volumes	Temporary Filesystems
Host-Editable	Yes	No	Yes	Varied	No
Mutable	No	Yes	Yes	Yes	Yes
Persistent	Yes	No	Yes	Yes	No
Performance	High	High	Low-Medium	High	High
Ease of Use	Varied	Trivial	Trivial	Varied	Complex

Use the Simplest Solution That's *Fast Enough*

Start with bind mounts, defer complex solutions

Complexity is easy to add but hard to remove. Using it sparingly helps to identify bottlenecks.

Prefer built-in and/or idiomatic solutions

Before reaching for a third-party solution, understand exactly why you need it.

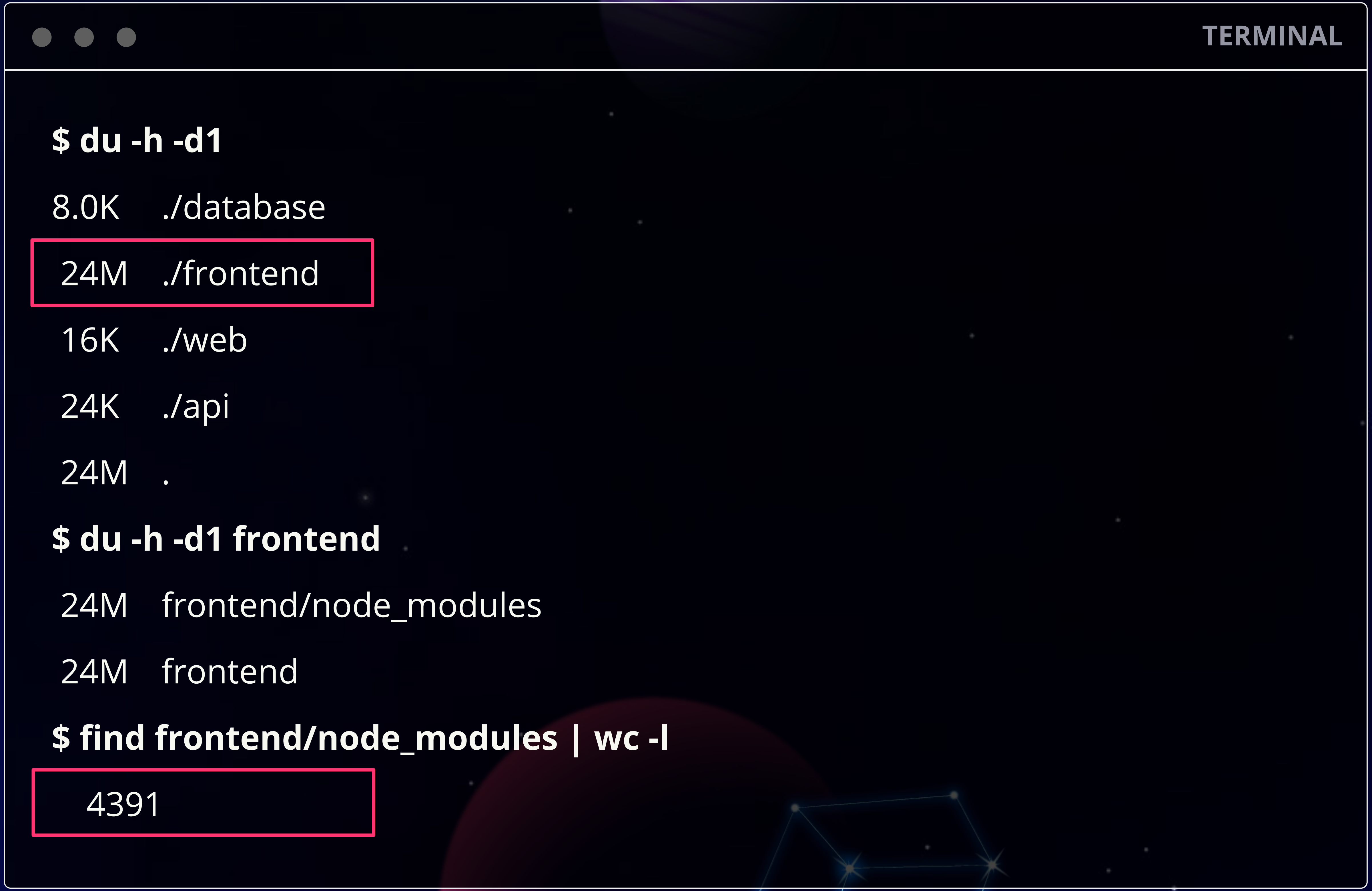
Help the computer work faster by doing less

We're spoiled by performance, but computers aren't magic.

Audit Your Code Size

It’s very easy to bring in **hundreds of MBs** of dependencies, even with a simple project.

Auditing your code and dependencies can help the computer do less work.



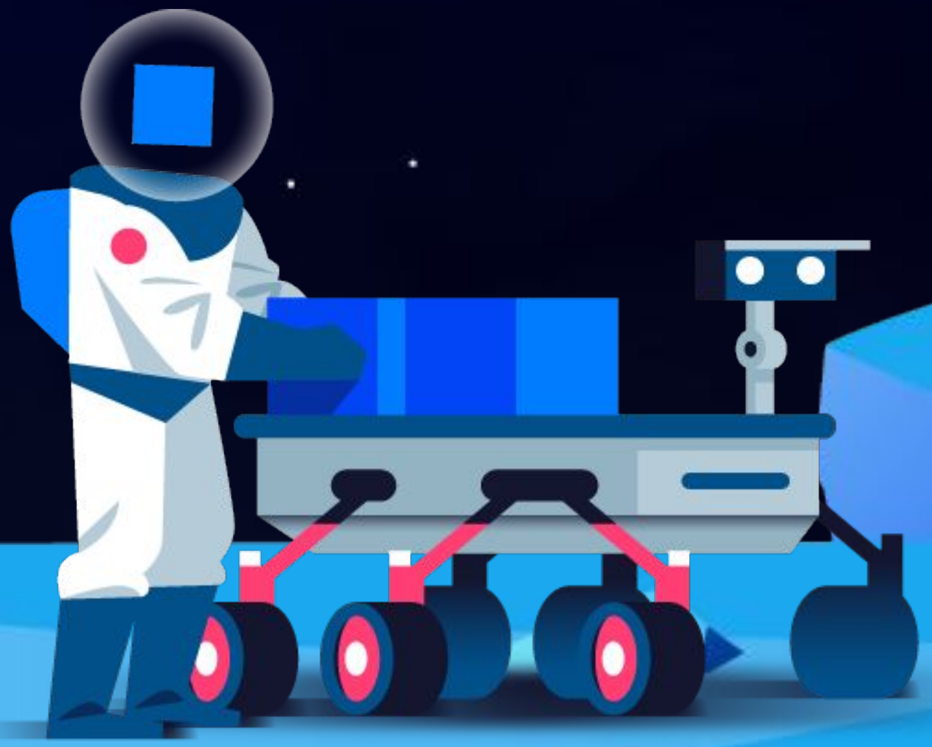
```
$ du -h -d1
8.0K    ./database
24M     ./frontend
16K     ./web
24K     ./api
24M     .
$ du -h -d1 frontend
24M     frontend/node_modules
24M     frontend
$ find frontend/node_modules | wc -l
4391
```

Avoid Crossing the Host/VM Boundary

- Crossing the host/VM boundary turns system calls into **RPC calls**
- Understand what can be static, cached, or dynamically generated inside the VM
- Bind mount only what you need to ***edit***, not dependencies and standard libraries
- Work inside the VM if possible

Don’t Bind Mount Certain Files Across the Host/VM Boundary…

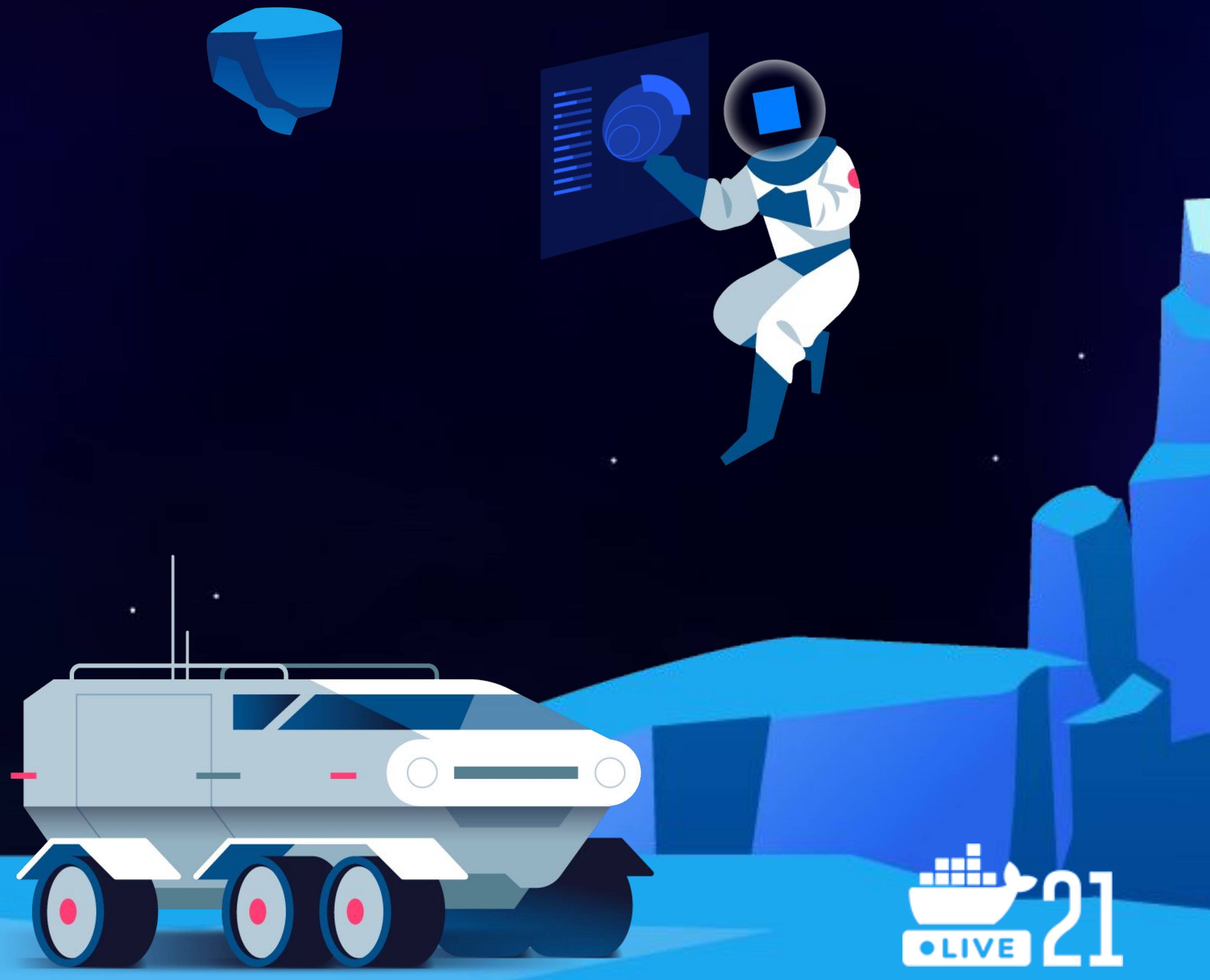
- `mmap`’d files (e.g. databases)
- inode-sensitive files (e.g. `.git` directories)
- Machine-specific code (e.g. `node_modules`)
- Huge updating files (e.g. logs)
- Standard libraries
- Dependencies



Use Filesystem Watching if Possible

Many tools and frameworks can use filesystem watching to **optimize rebuilds** and avoid rescanning an entire codebase. Use these features if you can!

(But note that not all filesystems support event notifications, and they can be spotty...)



Consider Synchronization as a Nuclear Option

- If you need to edit a large codebase, synchronize it into a volume
- Same rules apply: **keep it minimal**
- Keep the volume external from your Compose project to amortize sync costs
- Experiment with different tools
    - Mutagen, docker-sync, VS Code, etc.

Stay Up-To-Date and Collaborate

Watch the Docker Desktop Release Notes, join discussions, and **share ideas!**

Summary

- Different filesystems have different features, behaviors, and performance
- **Understand all of the options** and leverage what you need
- Check what your tools are **actually doing**
- The best option is to **do less work**
- Be kind and share ideas

Thanks for listening!

Send me your questions, ideas, and feedback

🐦 @xenoscopic