Challenges of building an API service

Learnings from svix and other APIs

Tom Hacohen
Devcon 2022

@TomHacohen
www.svix.com

Who am I?

- Founder and CEO of Svix
- Open source dev and maintainer
- Previously led teams at Samsung and the Israeli intelligence corps

What is Svix?

- Enterprise ready webhooks service
- We help companies send webhooks
- Customers such as: **Brex** and **LTSE**
- Backed by **Y Combinator**, **Aleph**, and amazing founders and CTOs.

Y Combinator

ALEPH

What is this talk about?

- What to know before starting an API service
- Learnings from Svix and other services
  - Your requirements may differ
  - Apply reasonable judgement
- Forgot to mention anything? Let me know!

Keep It Simple Silly

- Complexity is your biggest enemy - avoid if possible
- Prevents you from moving fast
- Many points of failure
- Hard to reason about

There is good debt and bad debt

Good debt

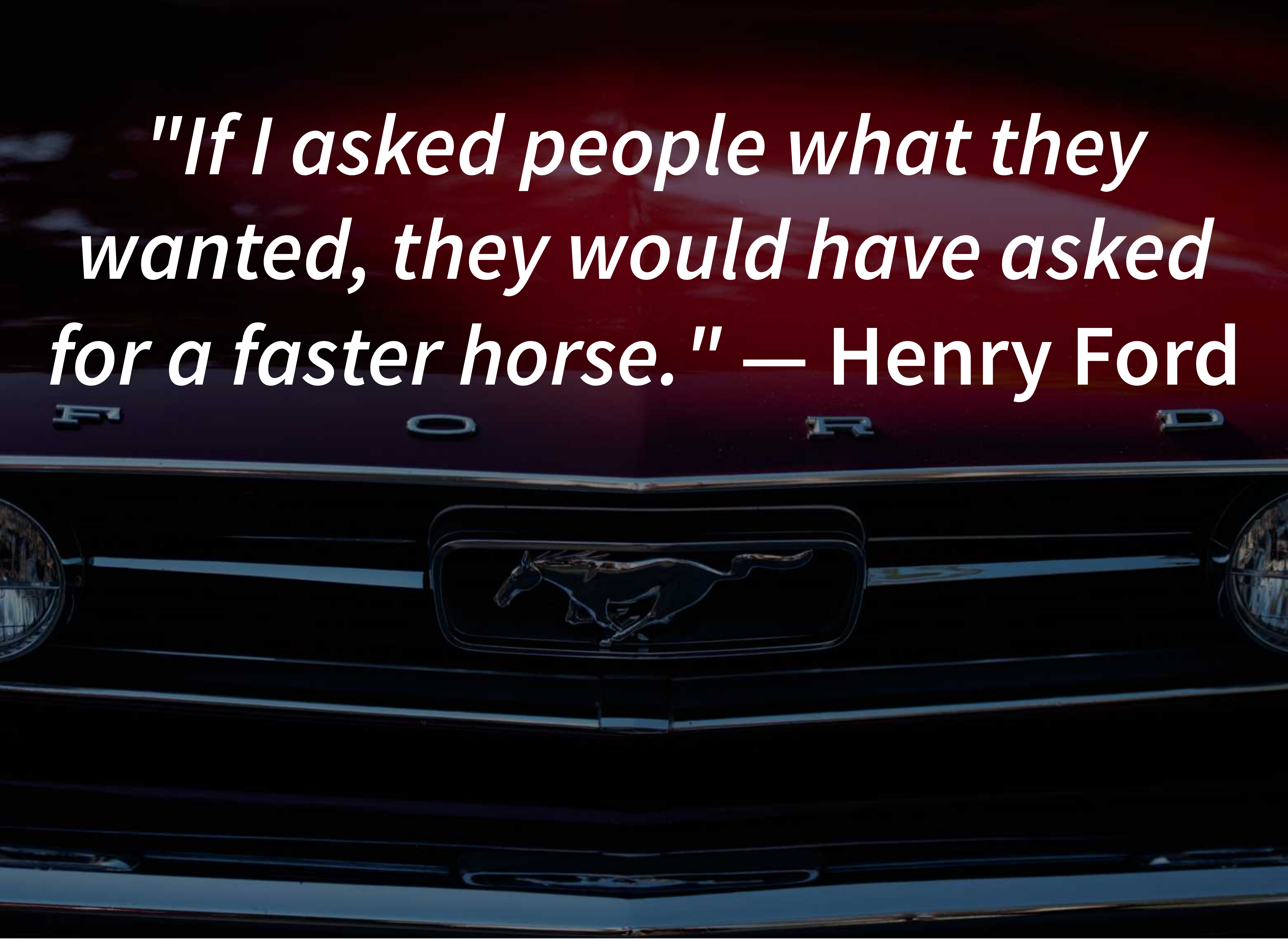
Bad debt

Understand your tech stack

- Different technology comes with different trade-offs
- Do you care about consistency? Availability?
- Know the tools you use and their limitations

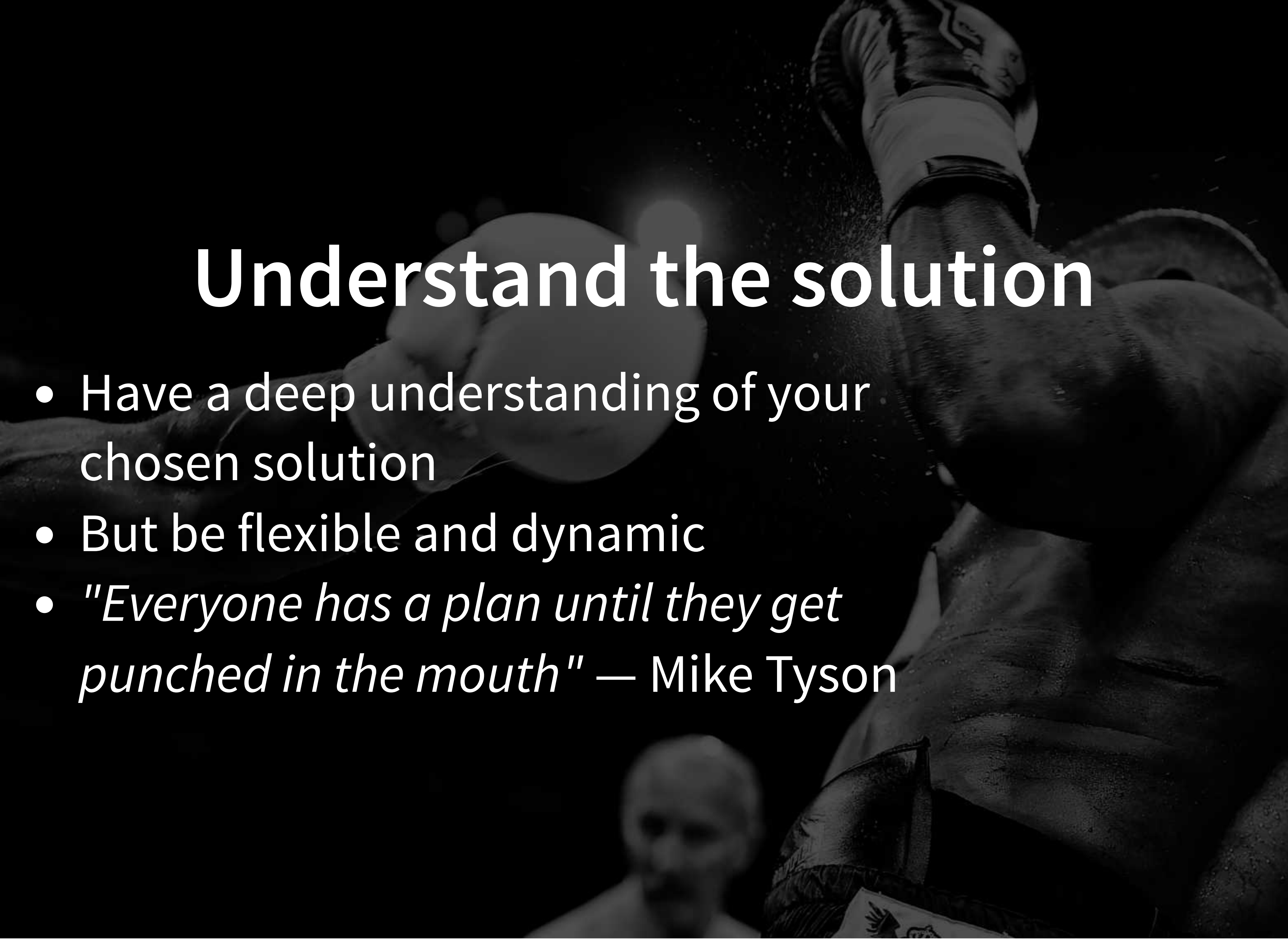
Know your customers

- What do they want?
- What do they care about?
- What would they hate?
- Have the curiosity of a child.



"If I asked people what they wanted, they would have asked for a faster horse." — Henry Ford

It's not what you do,
it's what you enable



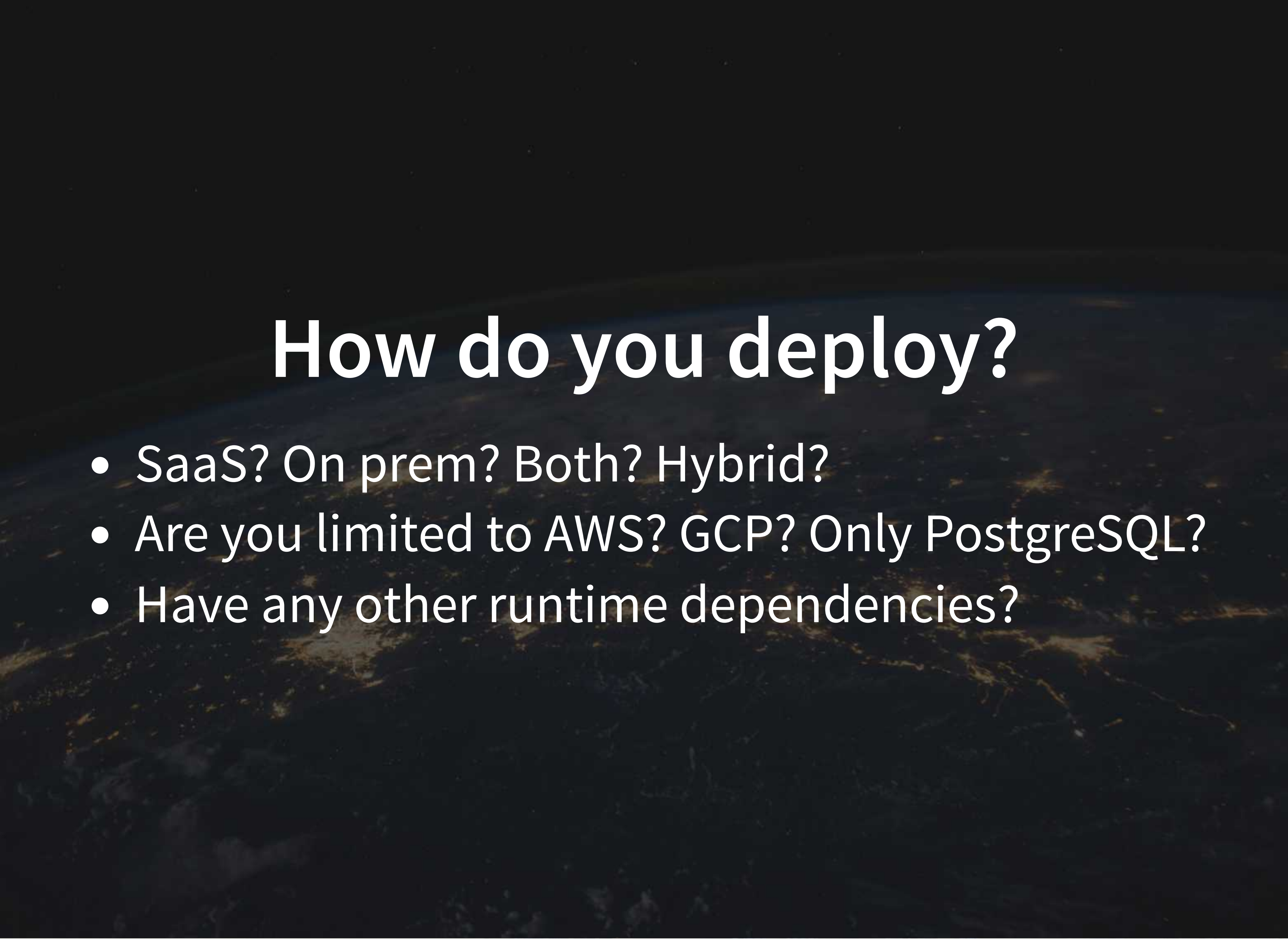
Understand the solution

- Have a deep understanding of your chosen solution
- But be flexible and dynamic
- *"Everyone has a plan until they get punched in the mouth"* — Mike Tyson

Set the right foundations

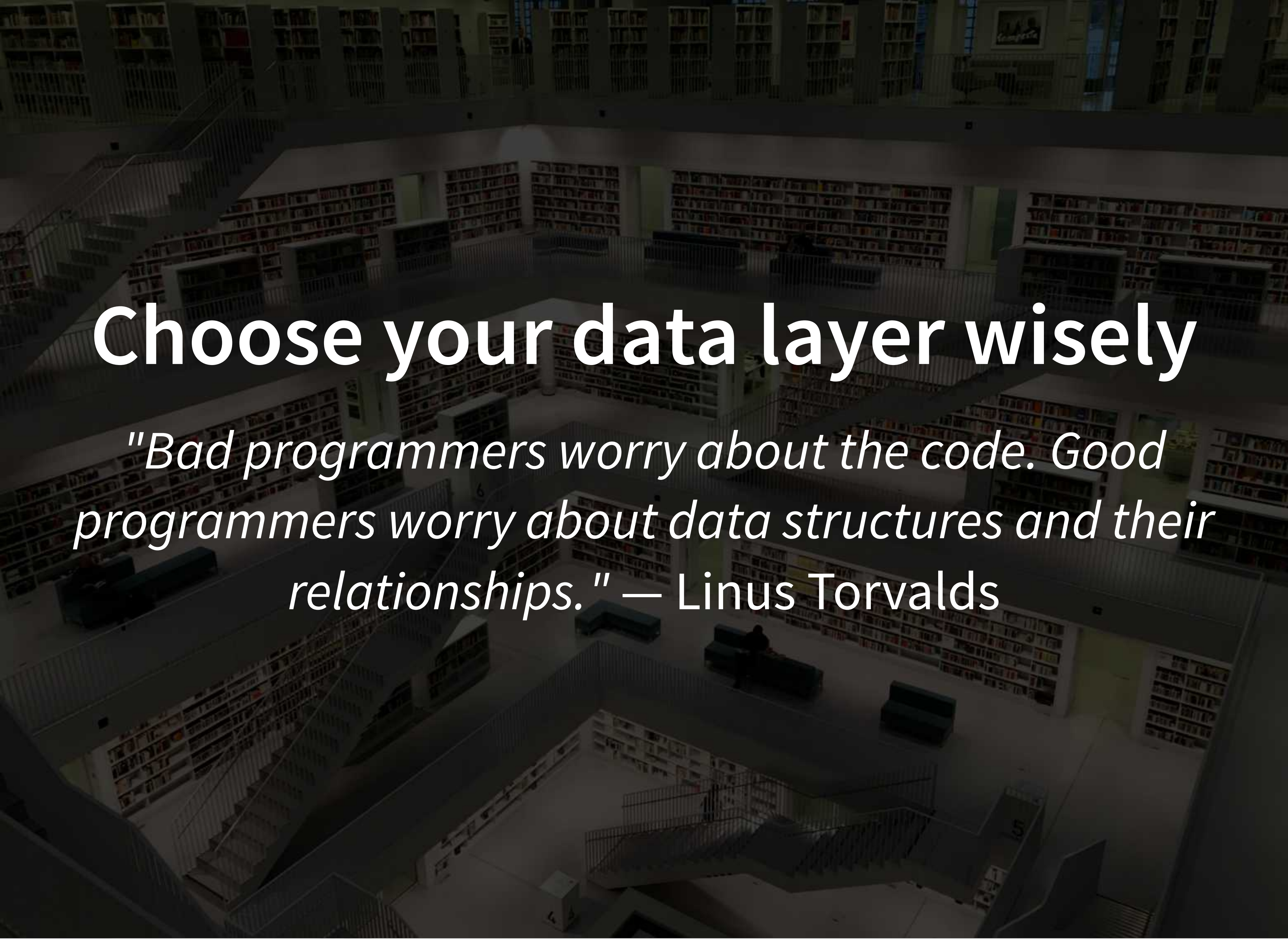
Manage API complexity

- Strive for simplicity, and learn to say no
- Be explicit , watch out for accidental flexibility
- Can't break API - if you build it, you're stuck with it
  - Try to plan for forward compatibility
- Don't leak implementation details



How do you deploy?

- SaaS? On prem? Both? Hybrid?
- Are you limited to AWS? GCP? Only PostgreSQL?
- Have any other runtime dependencies?



Choose your data layer wisely

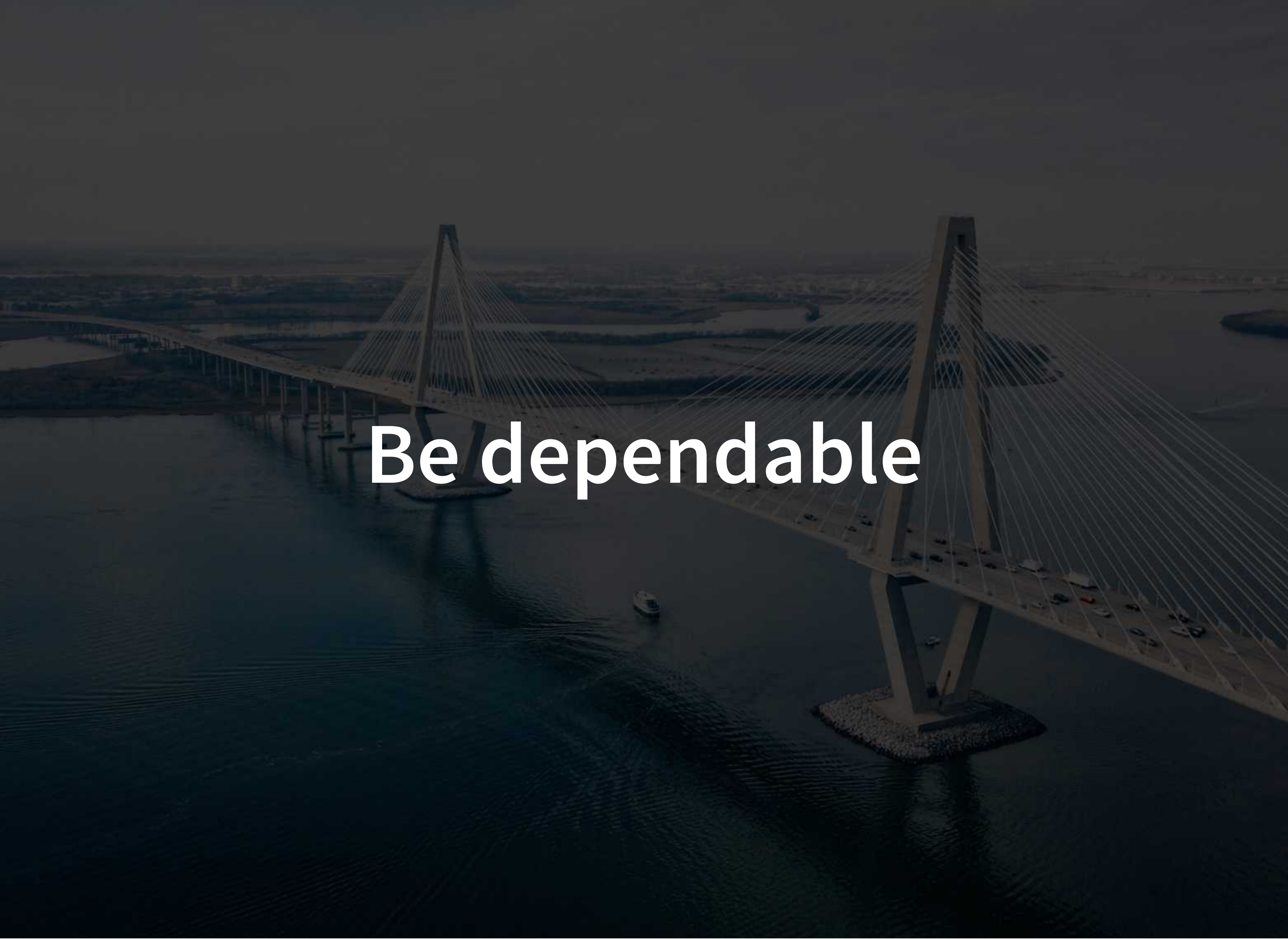
"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." — Linus Torvalds

People will still use it wrong

- Make it easy to use right
- Make it hard to use wrong
- Expect it to be used wrong and be ready

Find your north star

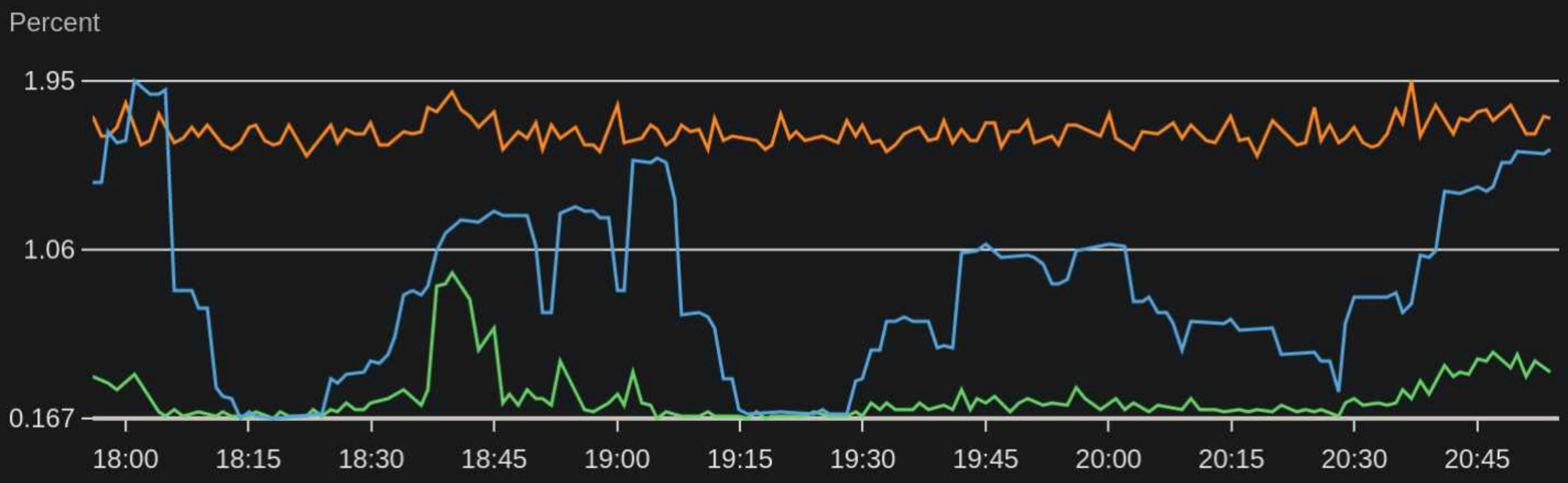
- For us it's RELIABILITY, so:
  - Never lose a message once accepted
  - Avoid downtime at all costs
  - High speed and low latency



Be dependable

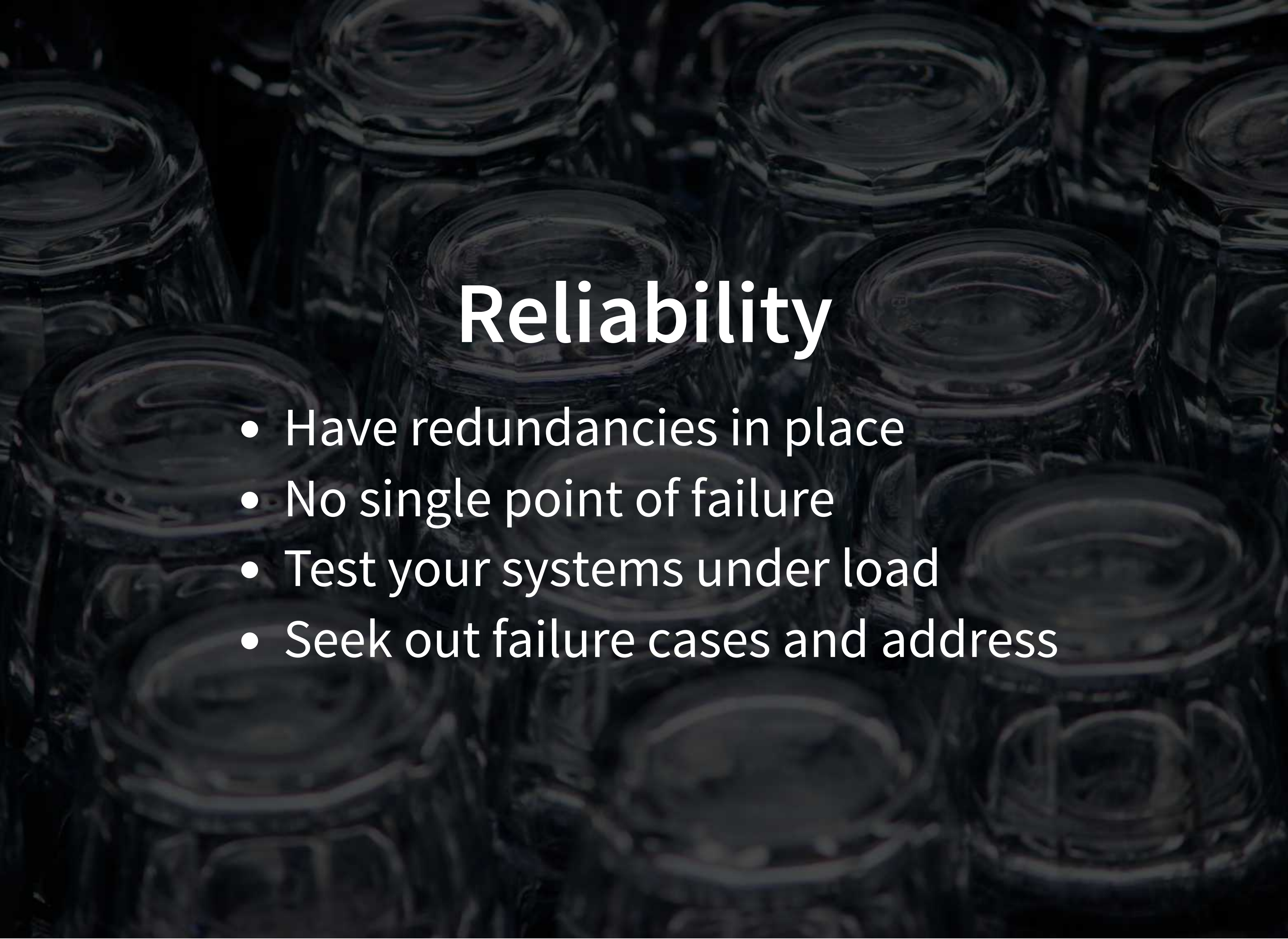
Monitor everything

Have fully visibility into your systems



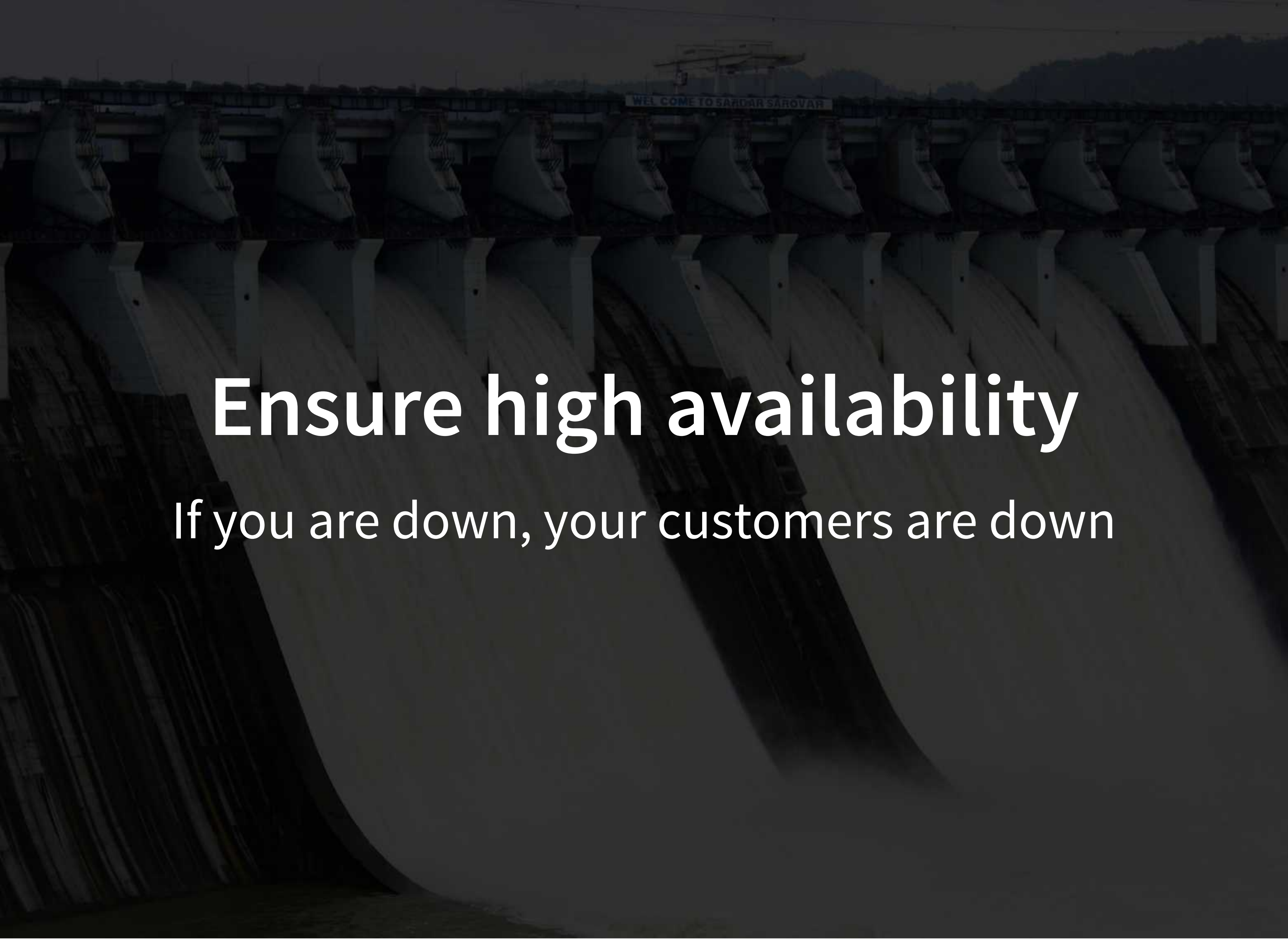
Logs everywhere

- You want to be able to detect anomalies
- Have information when something goes wrong
- Watch out not to log:
  - PII & PHI
  - Sensitive information



Reliability

- Have redundancies in place
- No single point of failure
- Test your systems under load
- Seek out failure cases and address

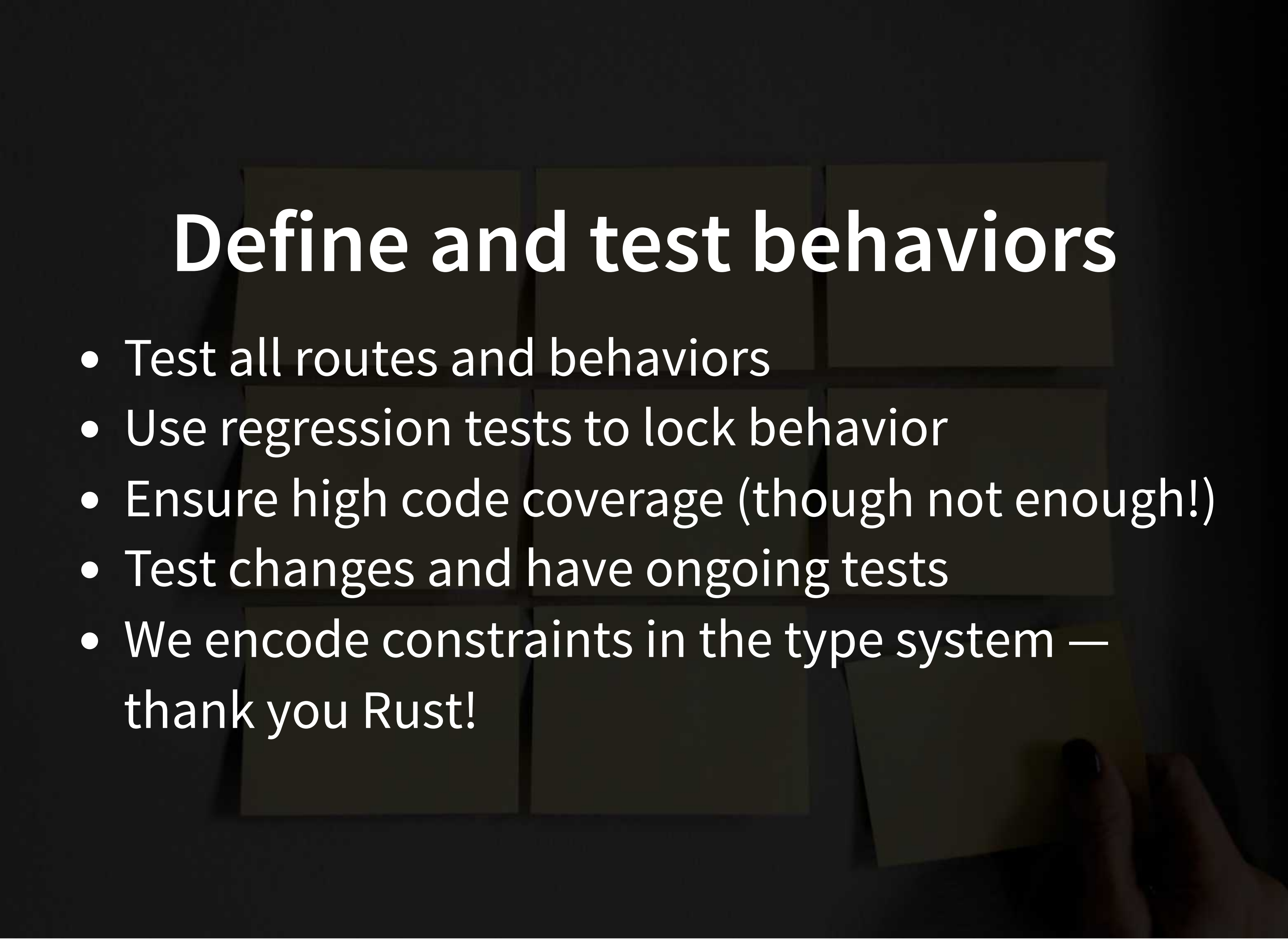


Ensure high availability

If you are down, your customers are down

Accountability

- Have a public status page
- Be quick and transparent about failures
- Reach out to affected parties immediately
- Do post-mortems to understand failures
- Hold yourself to the highest standards



Define and test behaviors

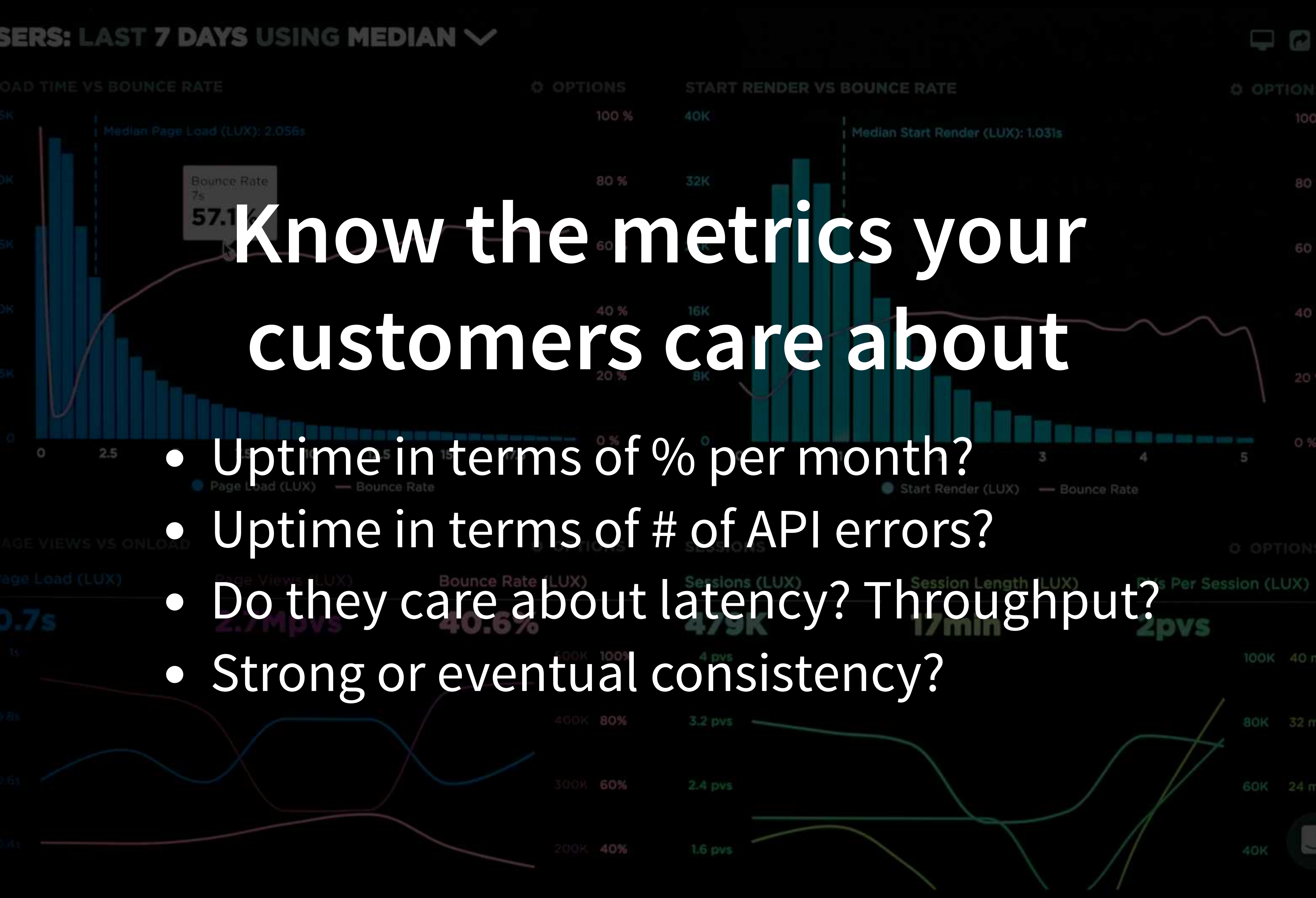
- Test all routes and behaviors
- Use regression tests to lock behavior
- Ensure high code coverage (though not enough!)
- Test changes and have ongoing tests
- We encode constraints in the type system — thank you Rust!

Disaster recovery

- Hope for the best, prepare for the worst
- Have off-site backups and point-in-time recovery
- List what can go wrong and how to address
- Train staff on recovering from failures

Know your dependencies

- You're only as strong as your weakest link
- Only use trusted tools from trusted sources
- Make sure your tools are dependable
- Evaluate your dependencies yourself
- It's OK to be bold, if you are also prudent



Know the metrics your customers care about

- Uptime in terms of % per month?
- Uptime in terms of # of API errors?
- Do they care about latency? Throughput?
- Strong or eventual consistency?

Application security

- OWASP Top 10 and best practices
- Correct password handling
- Strict authorization and restrictions

Operational security

- Vulnerability handling - patching systems
- Security and dependency scanners
- Security training for the team
- Encryption in transit and at rest

Strict and locked down

- Least privilege access control
- No one should have direct access to prod
- Limit capabilities of accounts with access
- Paper trail - log all access and operations

Isolating data between customers

- Multi tenant? Single tenant?
- Can one customer access another's data?
- Have strong enforcement for that

Make security easy for your customers

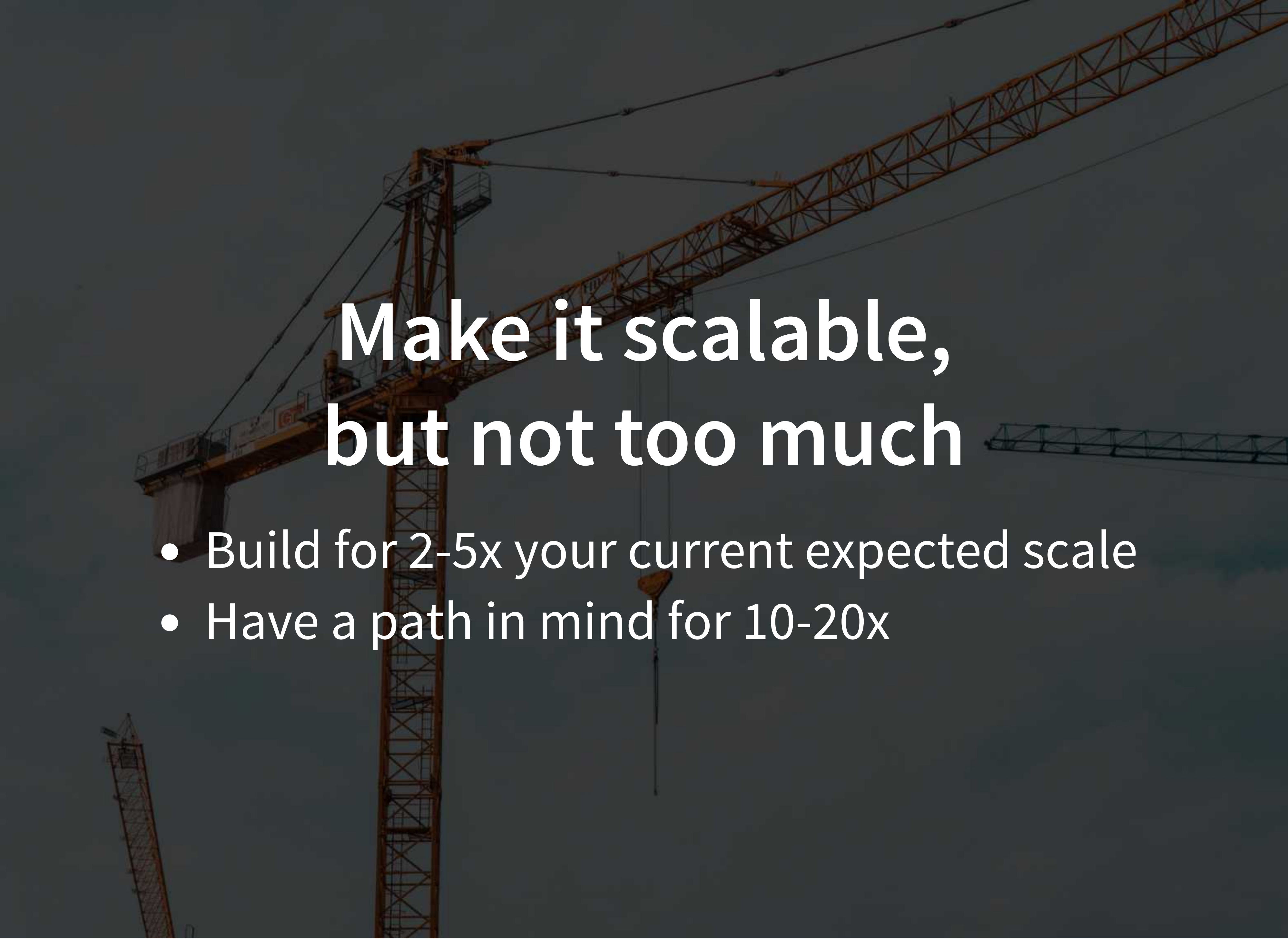
- Role based access control
- Enable key rotation and scoped keys
- Educate them about security with your service
- Make it hard to get security wrong

Educate yourself with security best practices

It's more than just code

- There is a lot of devops
- There is a lot of infrastructure
- Use infra-as-code - don't touch the UI!
- Build or buy? I prefer buy

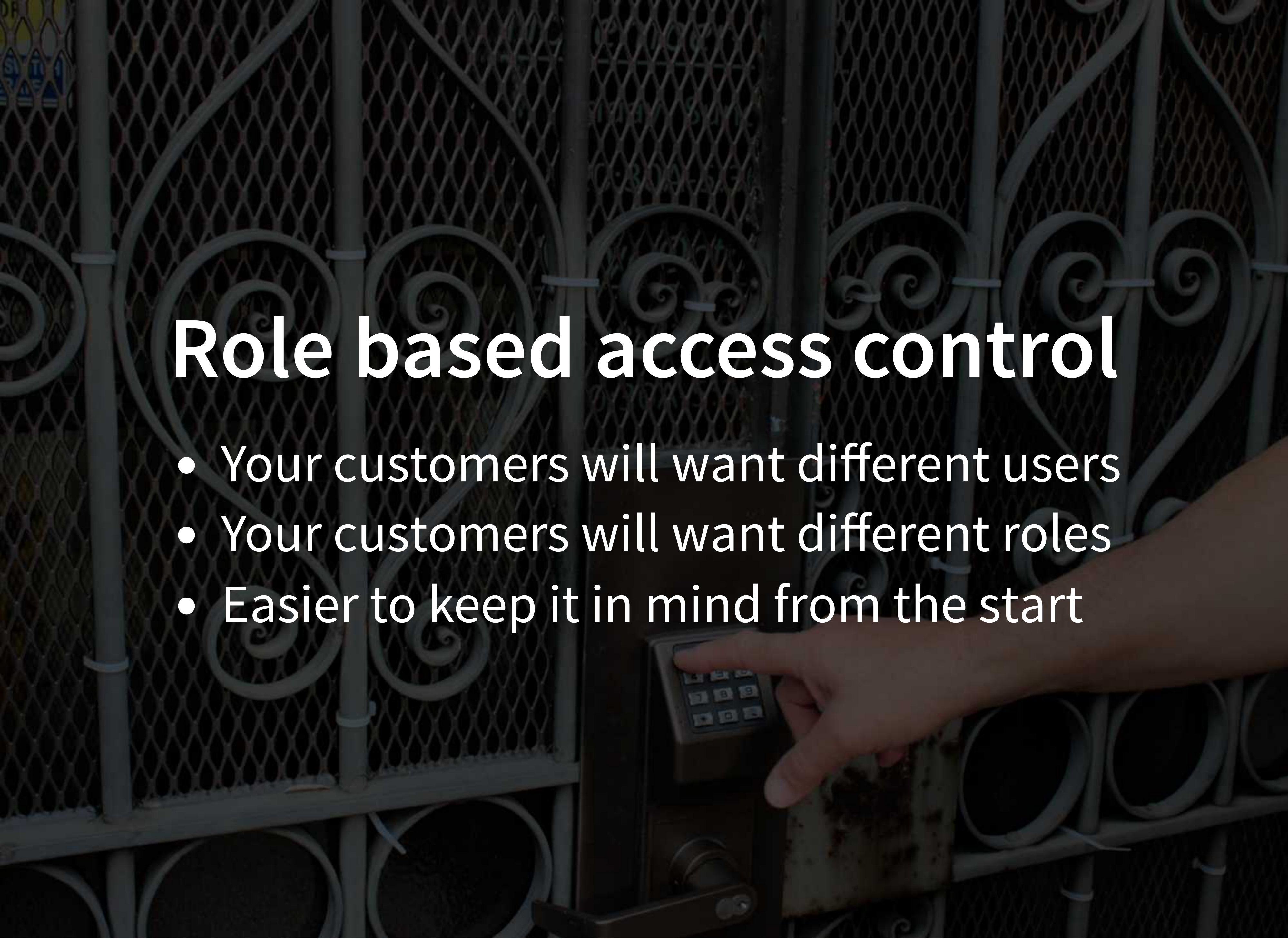
Prepare for the future, but build for the present



Make it scalable, but not too much

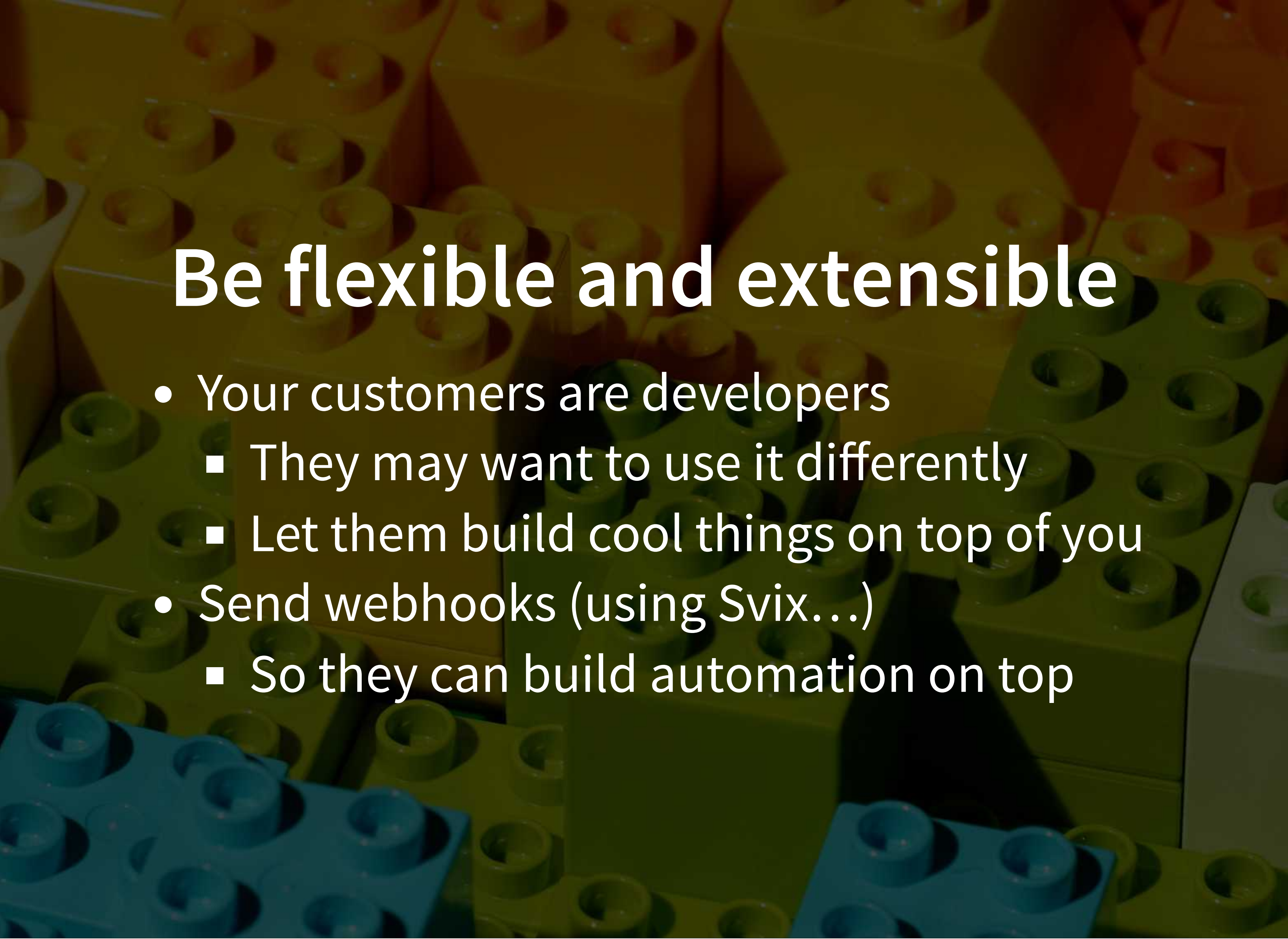
- Build for 2-5x your current expected scale
- Have a path in mind for 10-20x

Multi region, multi environment



Role based access control

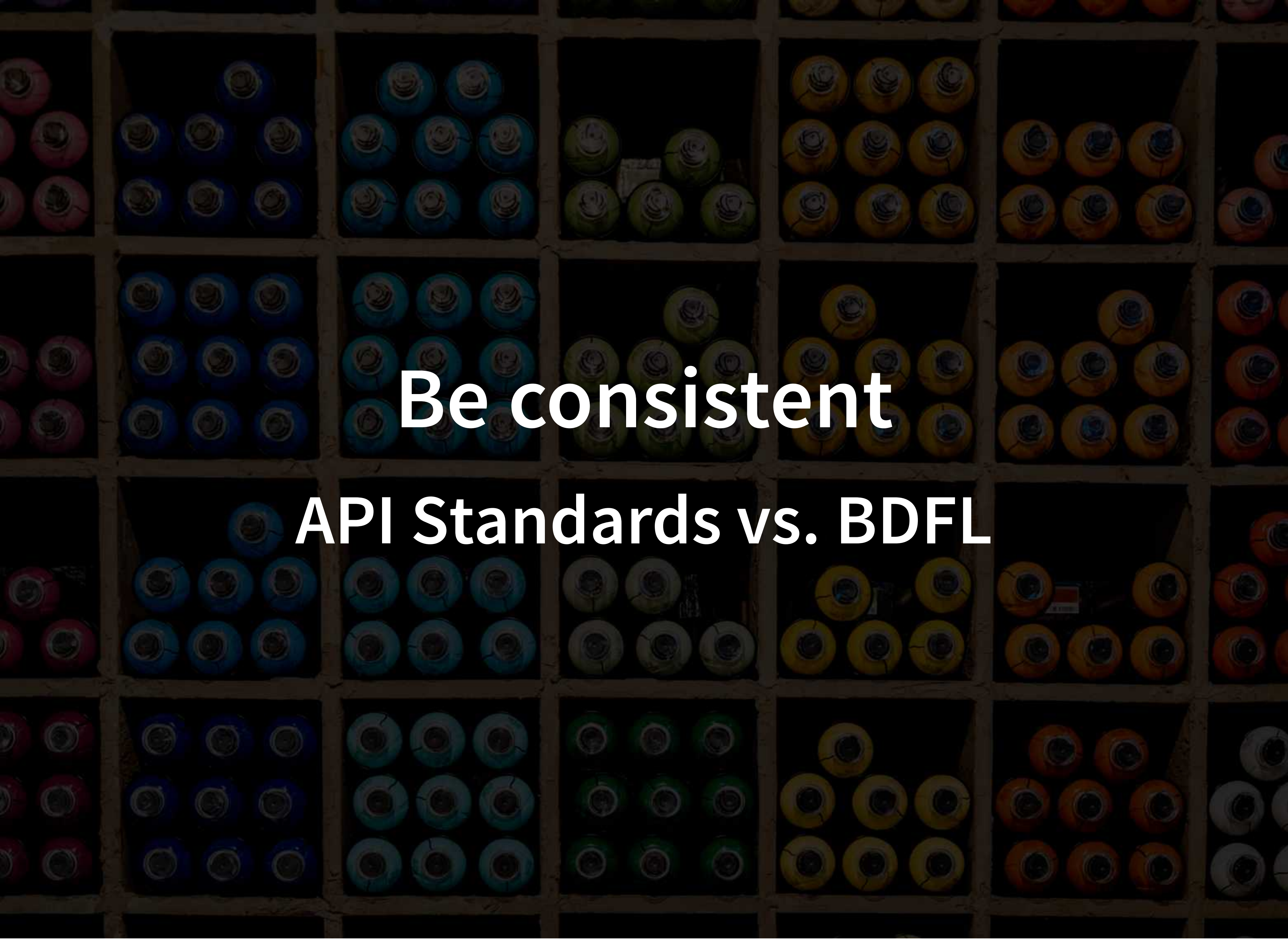
- Your customers will want different users
- Your customers will want different roles
- Easier to keep it in mind from the start



Be flexible and extensible

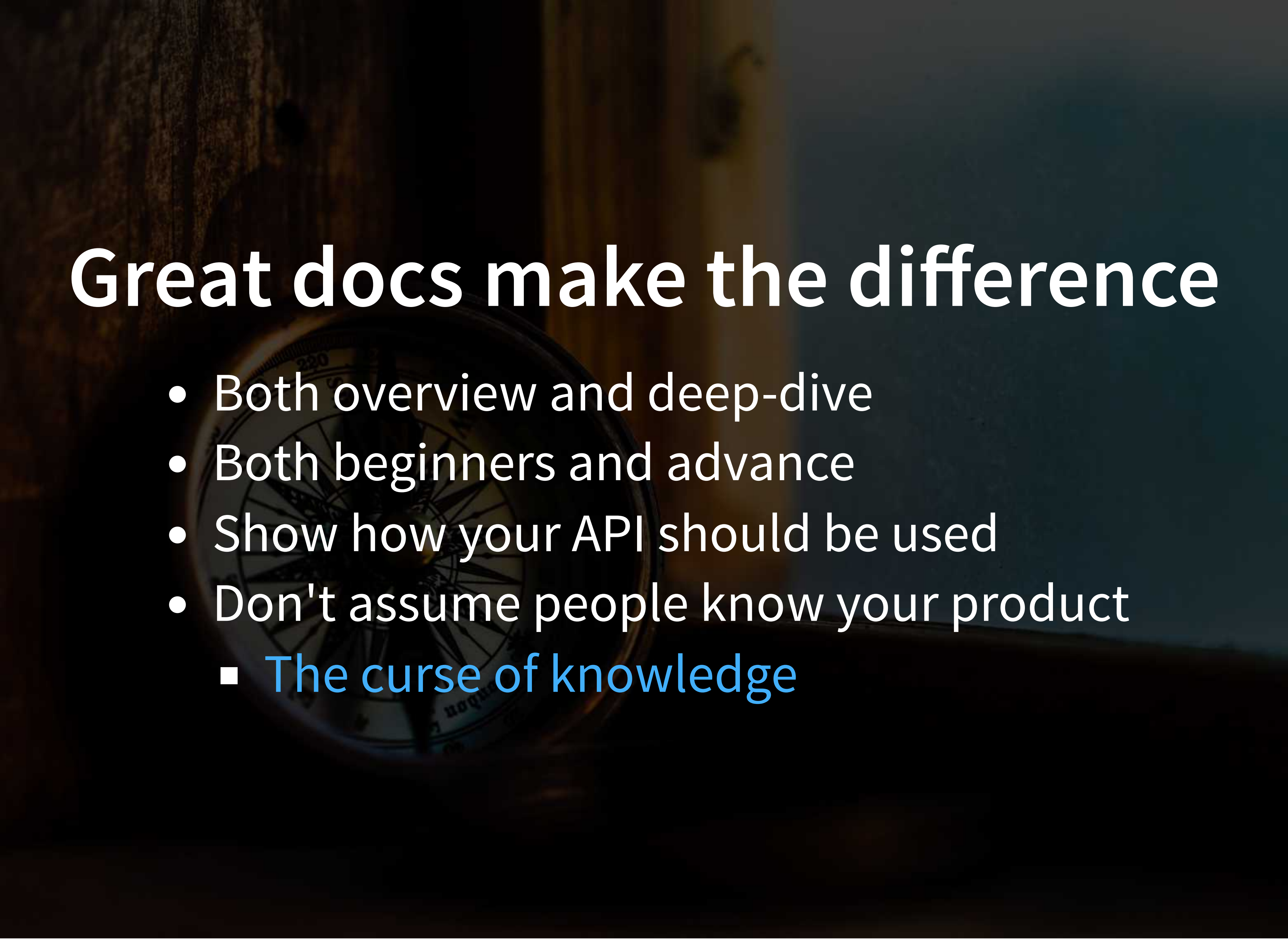
- Your customers are developers
  - They may want to use it differently
  - Let them build cool things on top of you
- Send webhooks (using Svix…)
  - So they can build automation on top

The devtool equivalent of UX



Be consistent

API Standards vs. BDFL

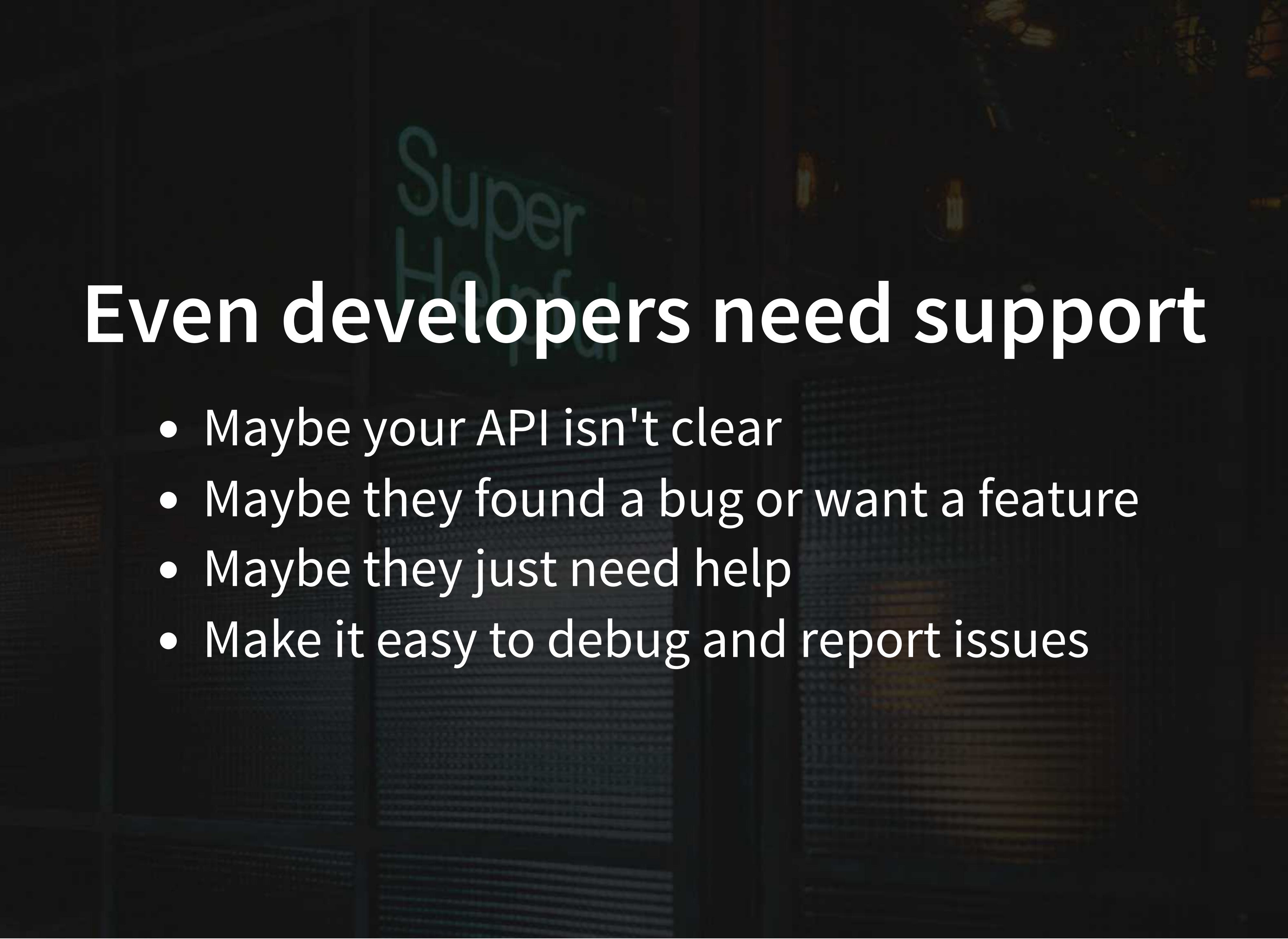


Great docs make the difference

- Both overview and deep-dive
- Both beginners and advance
- Show how your API should be used
- Don't assume people know your product
  - The curse of knowledge

Don't forget about tutorials

- Guide developers through common uses
- Get them started quickly - easy onboarding
- Highlight cool features



Even developers need support

- Maybe your API isn't clear
- Maybe they found a bug or want a feature
- Maybe they just need help
- Make it easy to debug and report issues

Have a great API

Good

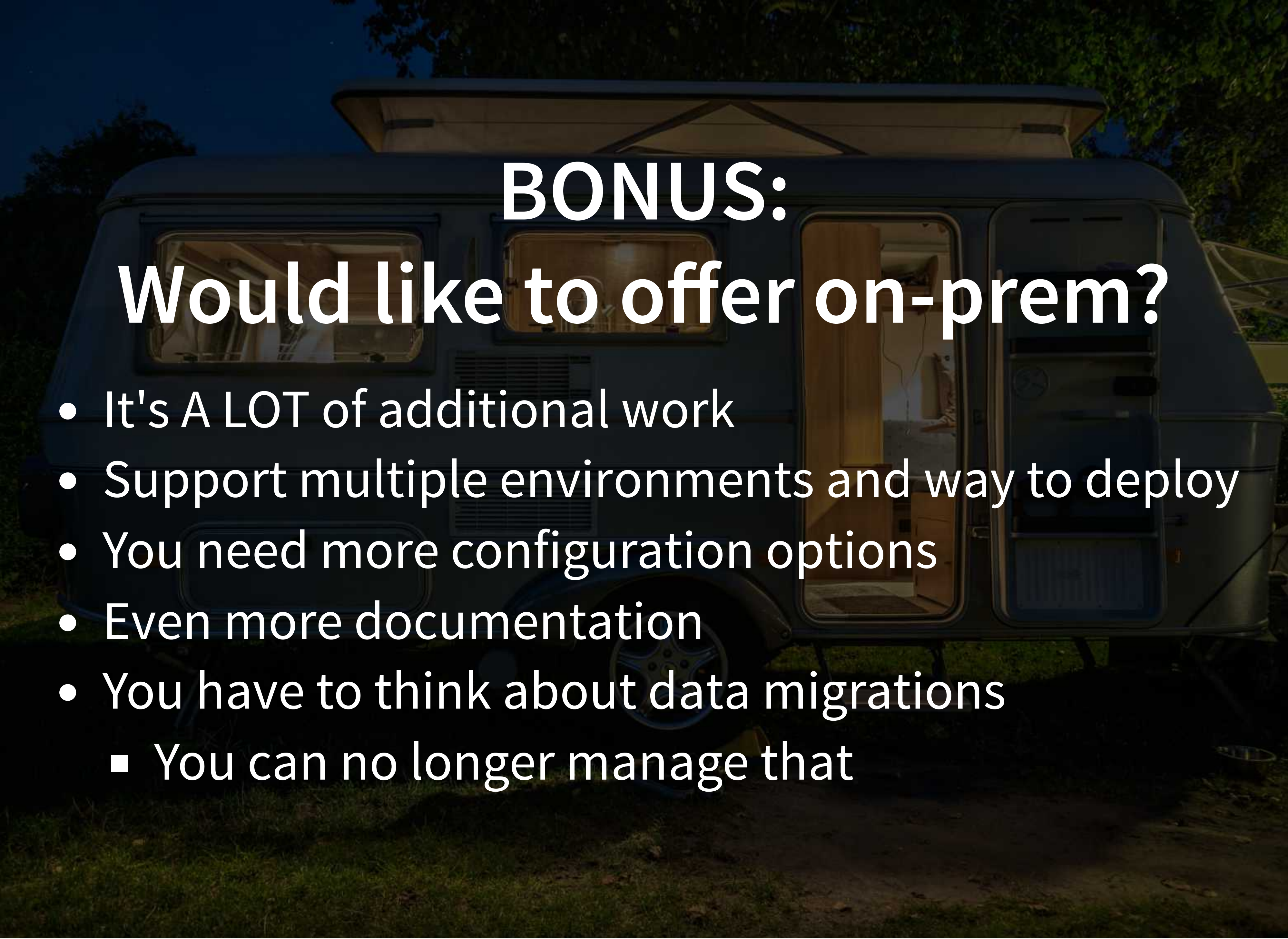
```
await fetch('https://api.svix.com/api/v1/app/', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
    'Accept': 'application/json'
    'Authorization': 'Bearer ' + token,
  },
  body: '{"name": "some name", "uid": "app_234"}',
});
```

Have an awesome SDK

Better

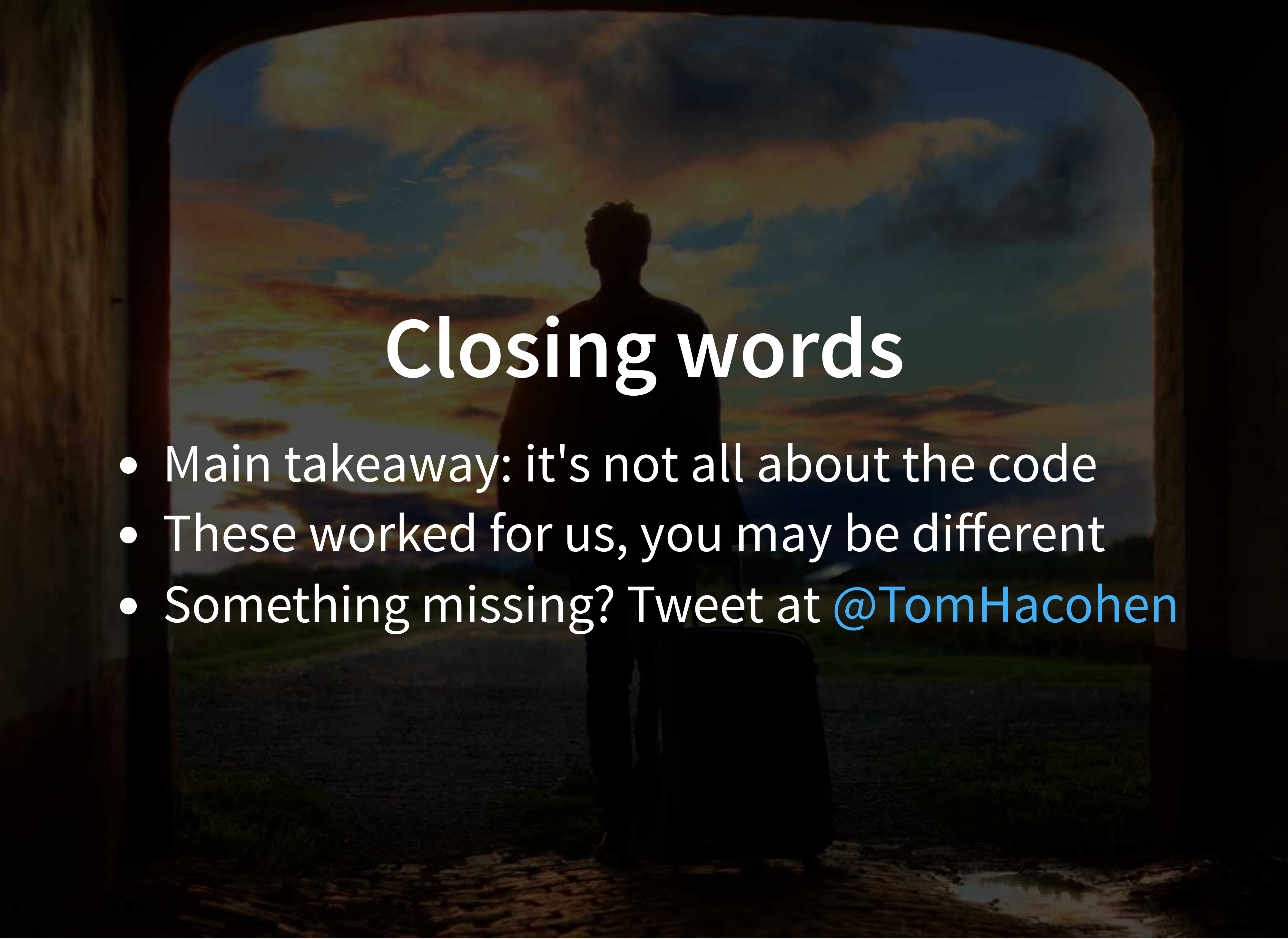
```
const svix = new Svix(token);

await svix.application.create({
  name: "some name",
  uid: "app_234",
});
```



BONUS:
Would like to offer on-prem?

- It's A LOT of additional work
- Support multiple environments and way to deploy
- You need more configuration options
- Even more documentation
- You have to think about data migrations
  - You can no longer manage that



Closing words

- Main takeaway: it's not all about the code
- These worked for us, you may be different
- Something missing? Tweet at @TomHacohen

Questions?

svix

Website: www.svix.com

Twitter: @SvixHQ

Github: github.com/svix/svix-webhooks