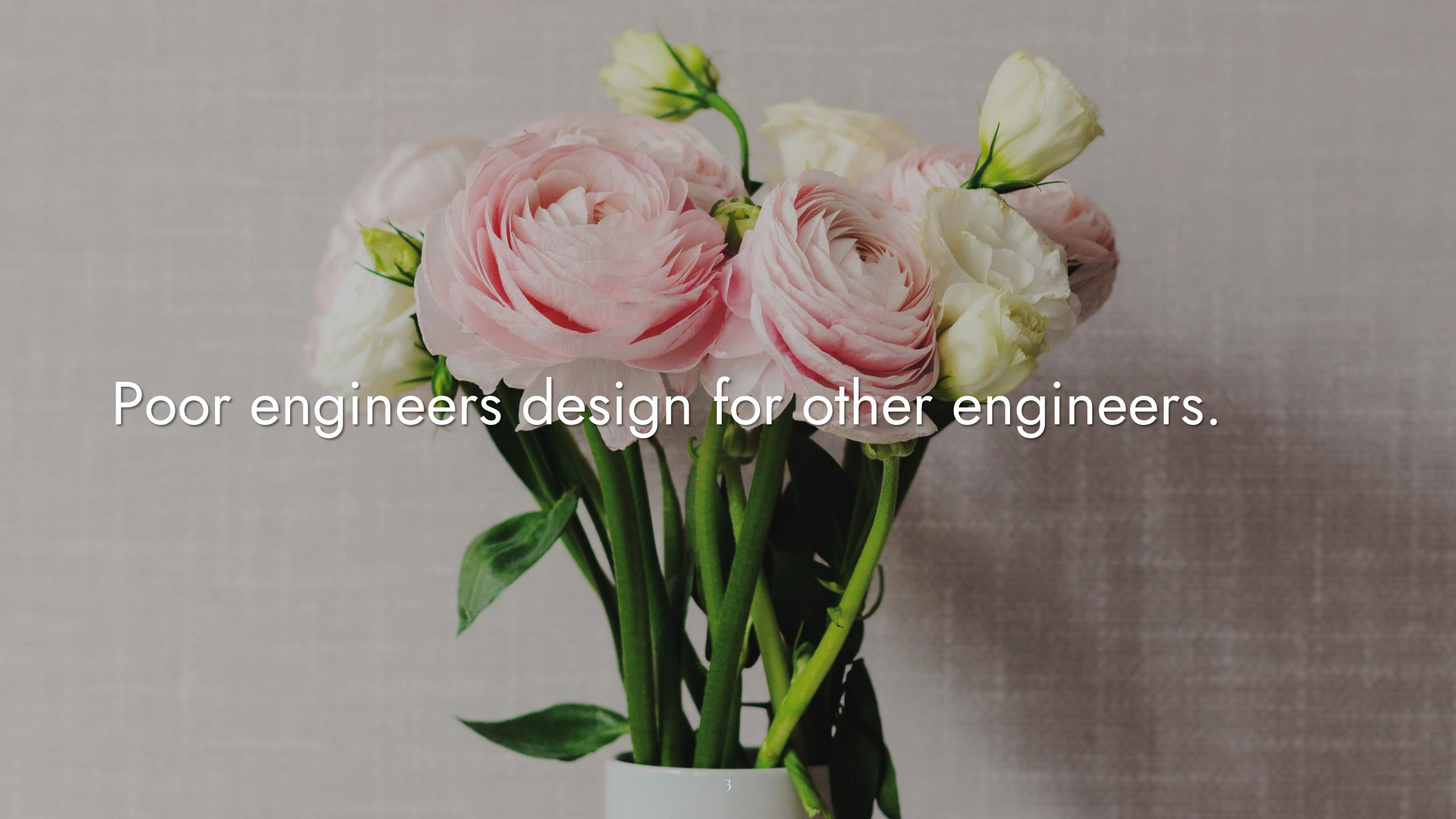
Practical Magic: The Resilience Potion and Security Chaos Engineering

Kelly Shortridge @swagitda_ | @shortridge

GOTO Chicago 2023


Everything is a system in transition. Our requirements will change. Contexts evolve.


Poor engineers design for other engineers.


Mediocre engineers design for the future they imagine.


Good engineers design for change.


If we pursue resilience, we can improve software quality at the same time as security.


What principles, practices, and tools should we adopt so we can sustain software resilience?

We can imbibe the Resilience Potion.


What is resilience?

Failure is inevitable; it's a natural part of complex systems as they operate

shortridge@hachyderm.io | @swagitda_

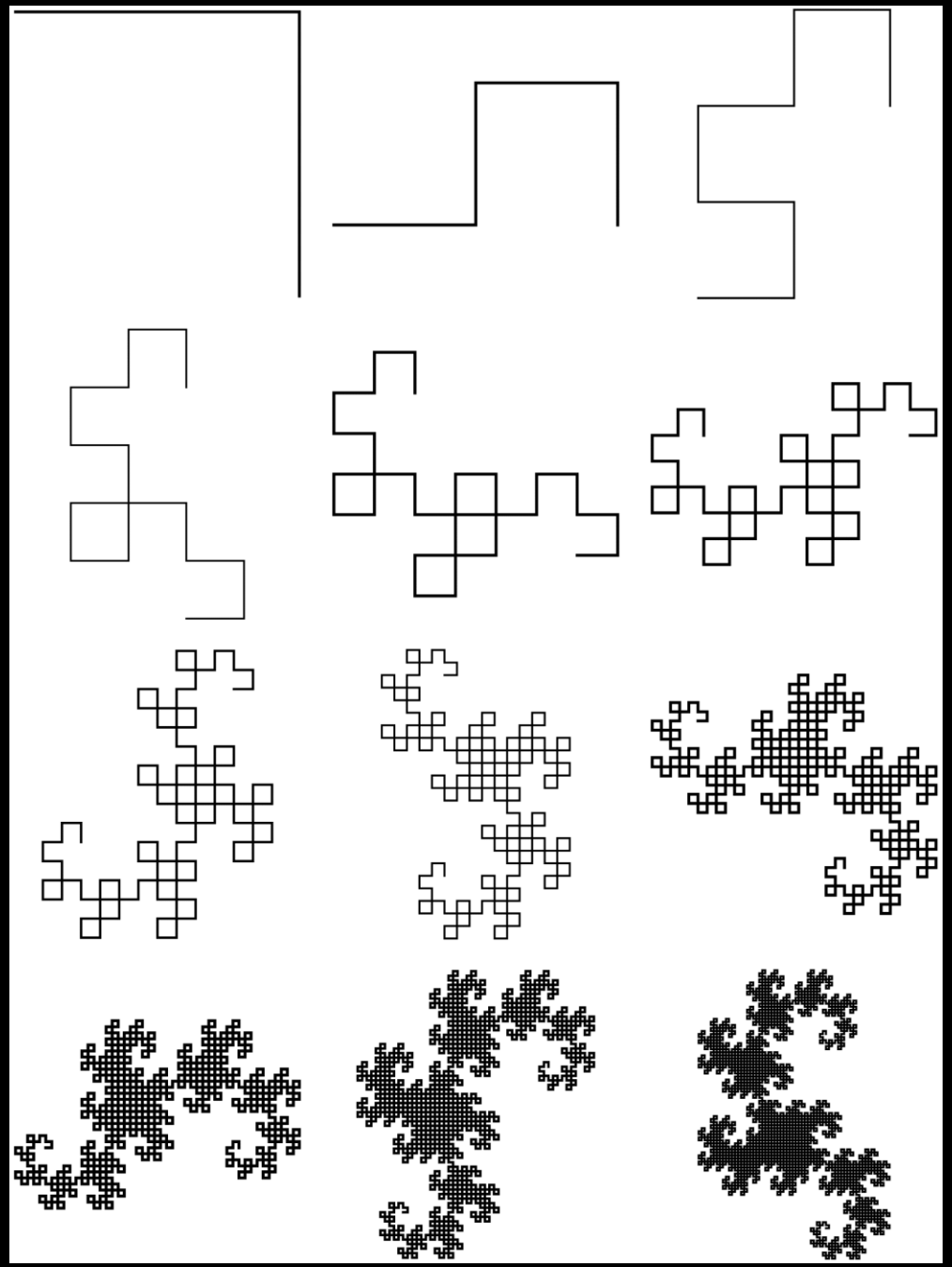
Our software is also a complex system; but our beliefs about it don't always align with reality

shortridge@hachyderm.io | @swagitda_


The software we design, build, and operate
reflects our mental models of reality.

Every time you encounter a bug, that is the diff between your mental model and reality

shortridge@hachyderm.io | @swagitda_

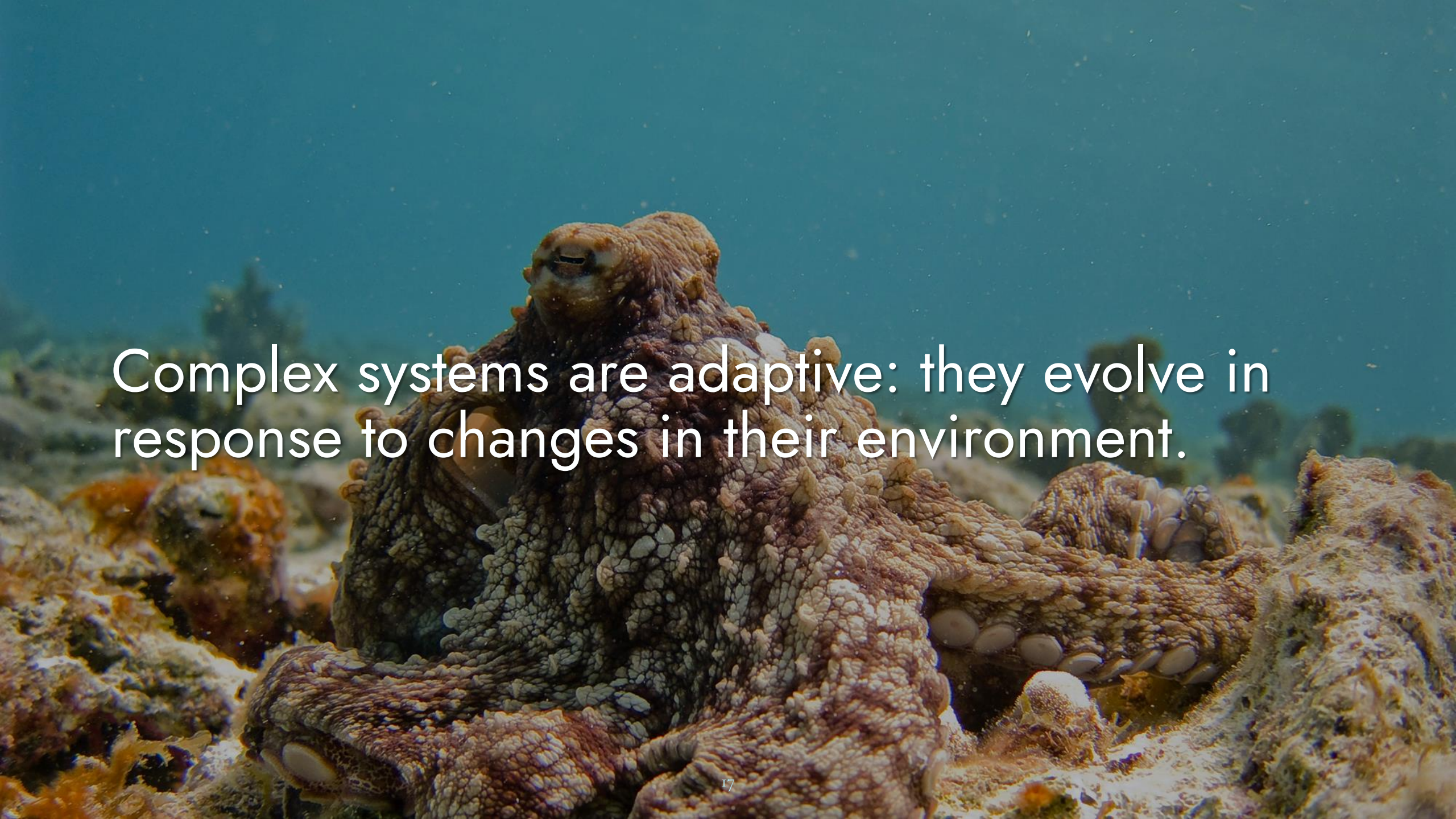


shortridge@hachyderm.io | @swagitda_

Surprise is “the revelation that a given phenomenon of the environment was, until this moment, misinterpreted.”

shortridge@hachyderm.io | @swagitda_

We must “prepare to be surprised.”


Complex systems are adaptive: they evolve in response to changes in their environment.

Adaptive capacity: how poised a system is to change how it works based on context

shortridge@hachyderm.io | @swagitda_

Resilience is “the ability to prepare and plan for, absorb, recover from, and more successfully adapt to adverse events.”

shortridge@hachyderm.io | @swagitda_

Security Chaos Engineering (SCE):

a socio-technical transformation that enables the organizational ability to gracefully respond to failure and adapt to evolving conditions.

shortridge@hachyderm.io | @swagitda_

We can imbibe the Resilience Potion to help us on our journey to sustain software resilience...

shortridge@hachyderm.io | @swagitda_

The Resilience Potion

shortridge@hachyderm.io | @swagitda_

There are five ingredients to sustain resilience...

shortridge@hachyderm.io | @swagitda_

Define the system's critical functions

Define the system's safe boundaries


Observe system
interactions across
spacetime

Feedback loops and a learning culture

Flexibility and willingness to change

How can we brew this potion when we develop and deliver software? Opportunities abound...

shortridge@hachyderm.io | @swagitda_

I. Critical Functionality

Our aim is simplicity and understandability of critical functions when we develop code

shortridge@hachyderm.io | @swagitda_

The Airlock Approach

During dev, we need to define what we can "throw out of the airlock."

shortridge@hachyderm.io | @swagitda_

What parts would you *like* to be able to neglect during an incident?

shortridge@hachyderm.io | @swagitda_

If a non-critical component is compromised, the airlock approach allows you to shut it off

If processing transactions is your critical function, throw reporting “out the airlock”

shortridge@hachyderm.io | @swagitda_


Choose “boring” tech

Dan McKinley: boring is not inherently bad; it likely indicates well-understood capabilities


The end user doesn’t care if HackerNews thinks you did something super cool

End users want to use your service whenever they want, as quickly as they want, and with the functionality they want.

shortridge@hachyderm.io | @swagitda_


Sometimes solving business problems does require fancy tech as a market differentiator

P.S. attackers love when devs adopt tools that aren't well understood yet....

shortridge@hachyderm.io | @swagitda_

Optimize for the “least-worst” tools for as many non-differentiator problems as you can

shortridge@hachyderm.io | @swagitda_


Standardizing raw materials

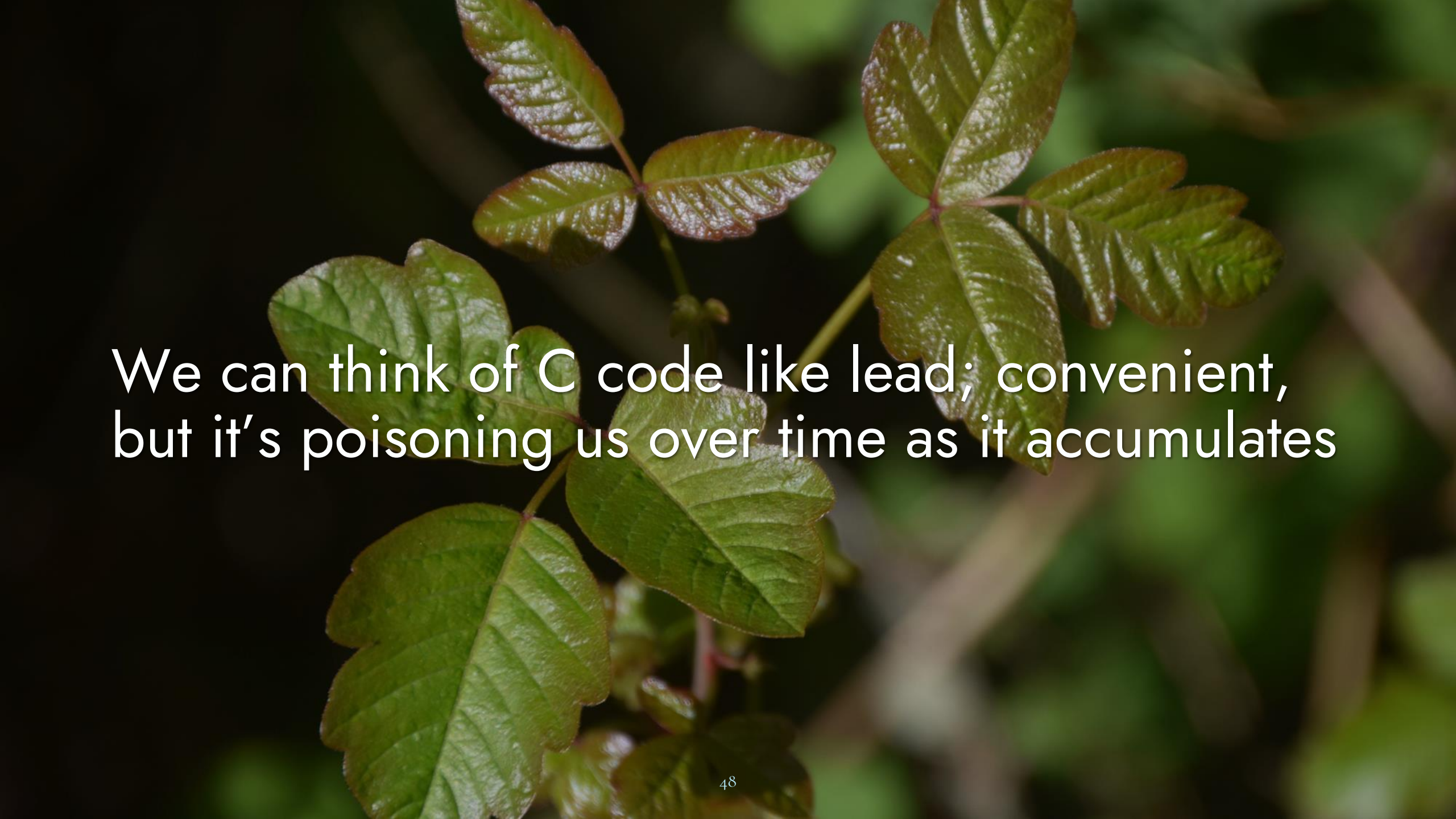
“Raw materials” in software systems: languages, libraries, and tooling

shortridge@hachyderm.io | @swagitda_


Yes, we're going to talk about memory safety,
the hottest software quality trend for S/S 2023

Memory safety: memory resource management is handled by the language and runtime itself

shortridge@hachyderm.io | @swagitda_


We can think of C code like lead; convenient,
but it's poisoning us over time as it accumulates

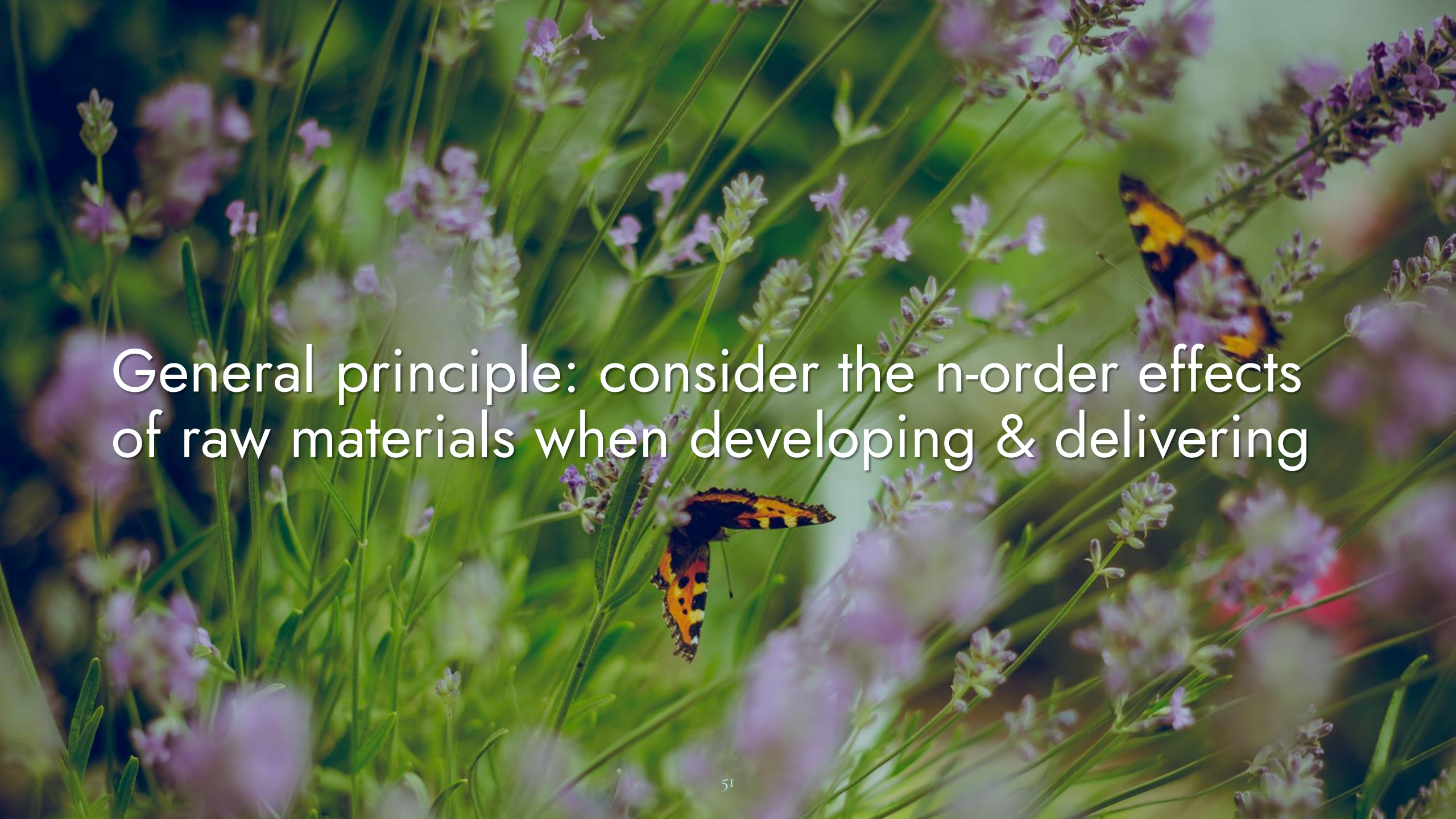
YOYD + YEET: You own our dependencies (YOYD) so yeet the hazardous ones away

shortridge@hachyderm.io | @swagitda_

```
<(curl -s https://stinkytool.io/bash)
```

You can feel its radioactive heat from here.

shortridge@hachyderm.io | @swagitda_

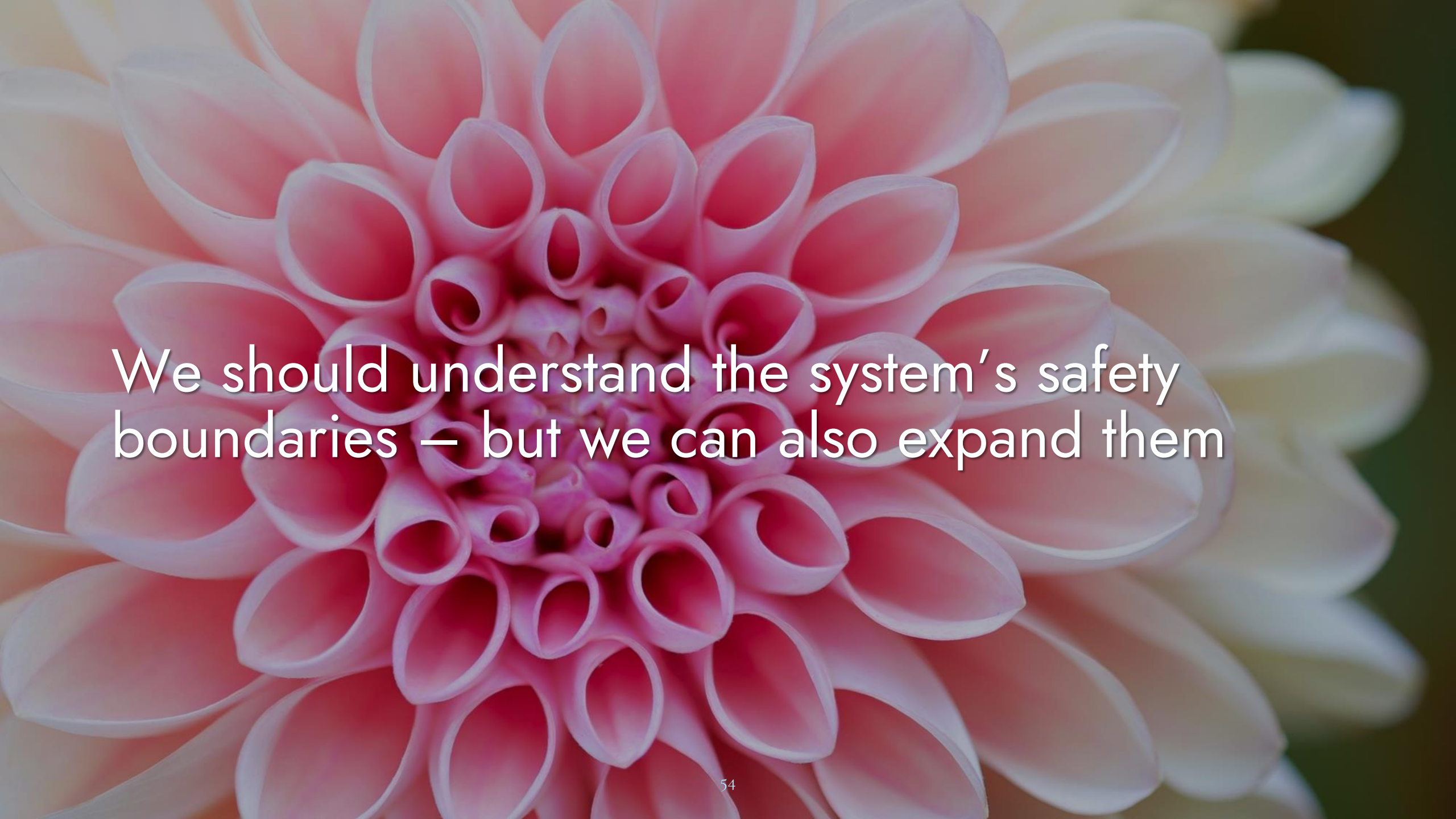


General principle: consider the n-order effects of raw materials when developing & delivering

Data can also be a hazardous material; we can and should isolate access to sensitive data

shortridge@hachyderm.io | @swagitda_

II. Expanding Safety Boundaries


We should understand the system's safety boundaries – but we can also expand them

Rather than relying on everything being perfect pre-deploy, we can cope well with mistakes

shortridge@hachyderm.io | @swagitda_

A lot of getting security “right” is just solid engineering. Security is a facet of quality.

shortridge@hachyderm.io | @swagitda_


Anticipating scale

How might operating conditions evolve?
Where do the system's safety boundaries lie?

shortridge@hachyderm.io | @swagitda_


Challenging our “this will always be true”
assumptions can expose scalability issues

“On every incoming request, we first need to correlate it with the user’s prior shopping cart – which means making a query to this other thing.”

We must anticipate what ops / SRE will need when responding to incidents

Attackers also target our “this will always be true” assumptions that exist all over our stack.

shortridge@hachyderm.io | @swagitda_

Parsing this string will always be fast

Messages on this port will always be post-authentication


An alert will always fire if a malicious executable appears

Standardizing patterns and tools

Standardization: ensuring work produced is consistent with preset guidelines

shortridge@hachyderm.io | @swagitda_


Prioritize patterns for parts of the system with the biggest security implications

Hazardous methods can look like roll-your-own: crypto, database, logging pipeline, etc.

shortridge@hachyderm.io | @swagitda_

SQLi can be characterized as the result of rolling your own database query builder

Don't DIY middleware.

shortridge@hachyderm.io | @swagitda_

Give teams a list of well-vetted libraries & service providers they should choose from


Paved roads: well-integrated, supported solutions to common problems that allow humans to focus on their unique value creation


Understanding dependencies

Understand faults in our tools so we can fix or mitigate them — or even consider better tools

shortridge@hachyderm.io | @swagitda_


When should you care about a security vuln?

1. How easy is the attack to automate & scale?
2. How many steps away is the attack from the attacker's goal outcome?

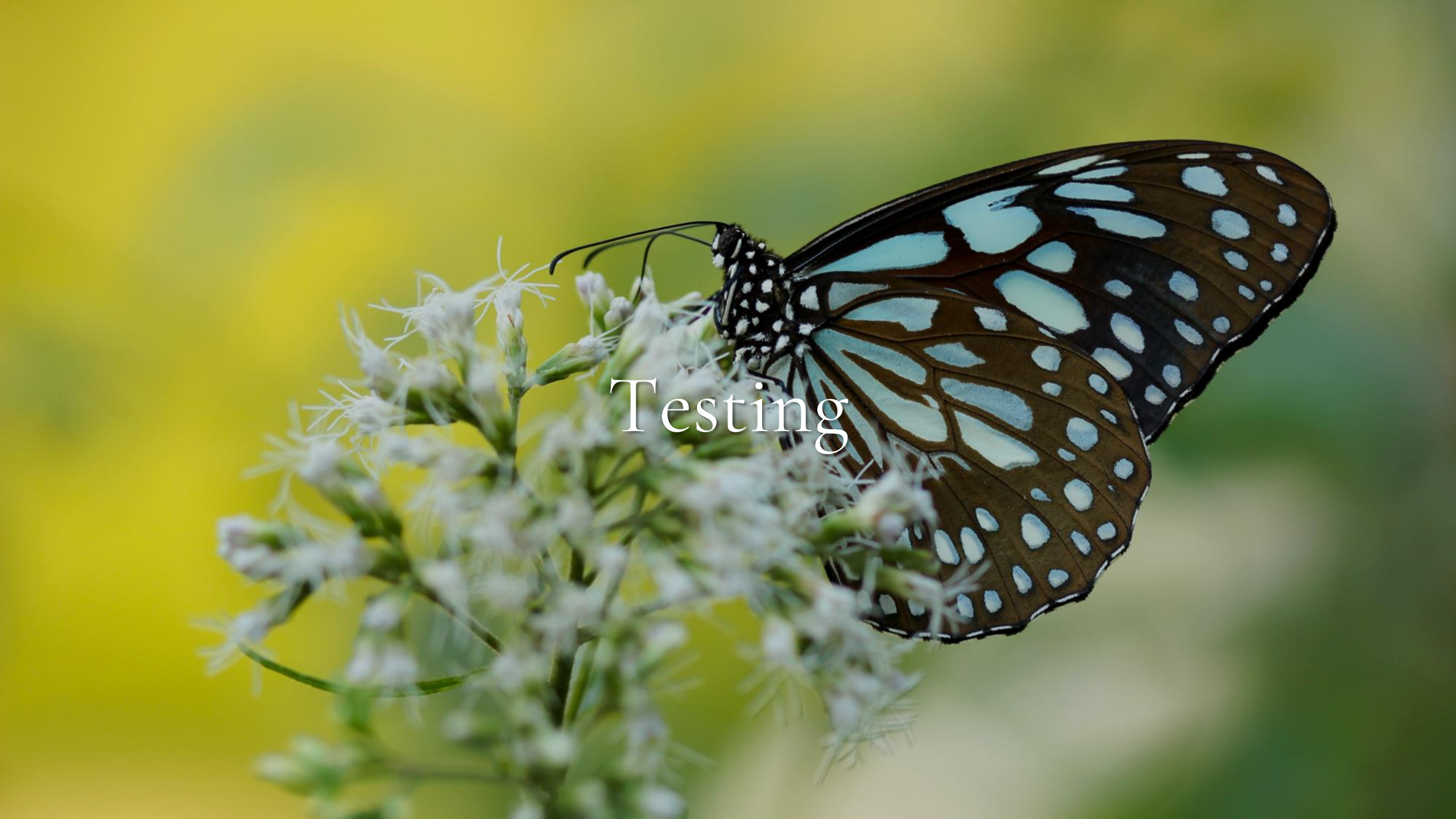
shortridge@hachyderm.io | @swagitda_

III. Observing System Interactions across Spacetime

We must observe how our systems' behaviors unfold over time and across their topology

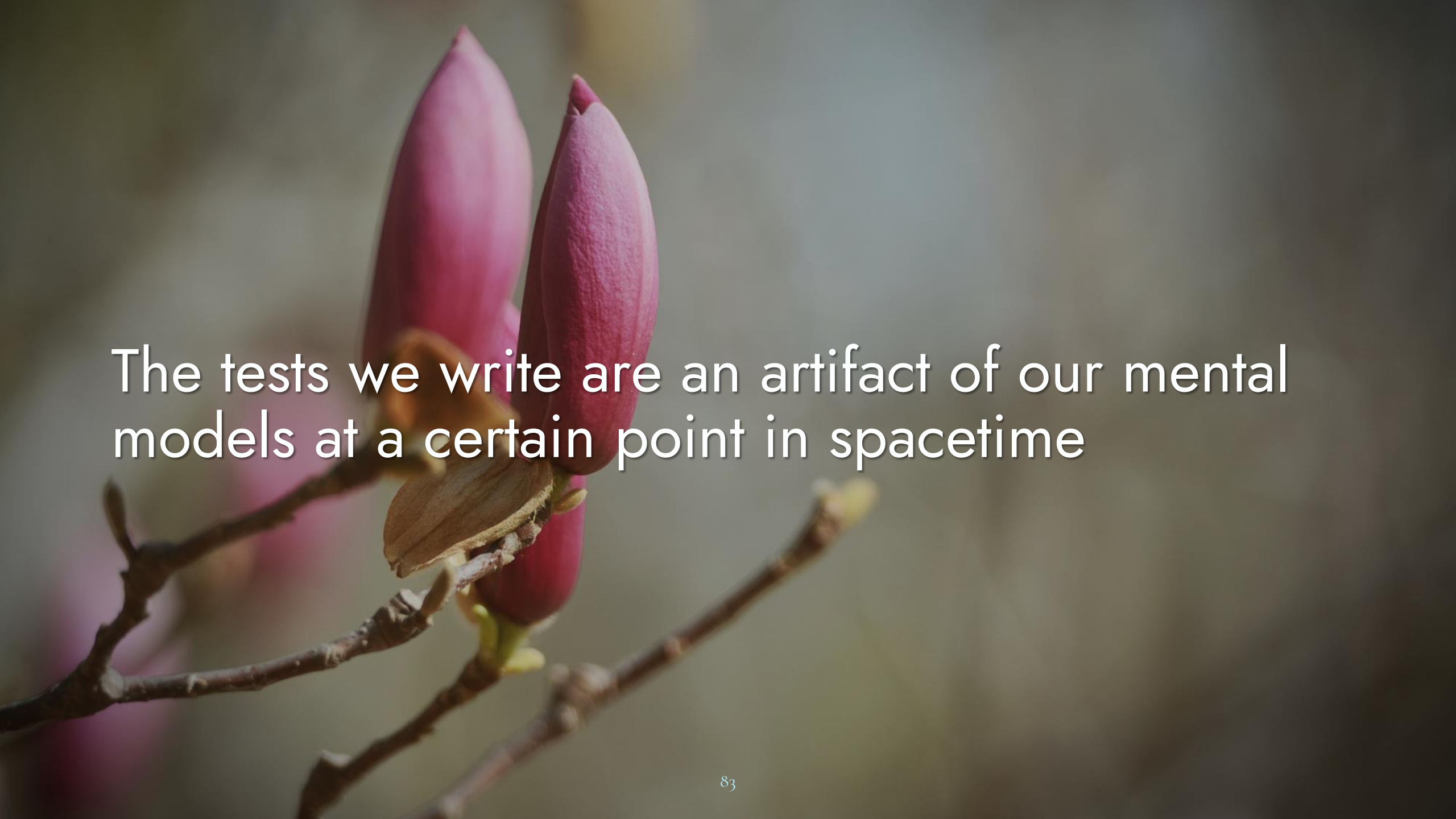
shortridge@hachyderm.io | @swagitda_

We can also make interactions more linear – curtailing the number of “baffling” interactions


Testing

Are we testing for resilience or quality over time, or just to say that we did testing?

shortridge@hachyderm.io | @swagitda_


The tests we write are an artifact of our mental models at a certain point in spacetime

Prioritize tests that refine our mental models and can adapt as system context evolves

shortridge@hachyderm.io | @swagitda_

Integration tests can be a valuable first pass at uncovering “baffling” interactions

The AttachMe vuln is an example of what we hope to uncover with integration tests


Integration tests for attaching a disk to a VM in another account, spikes in resource nom nom…

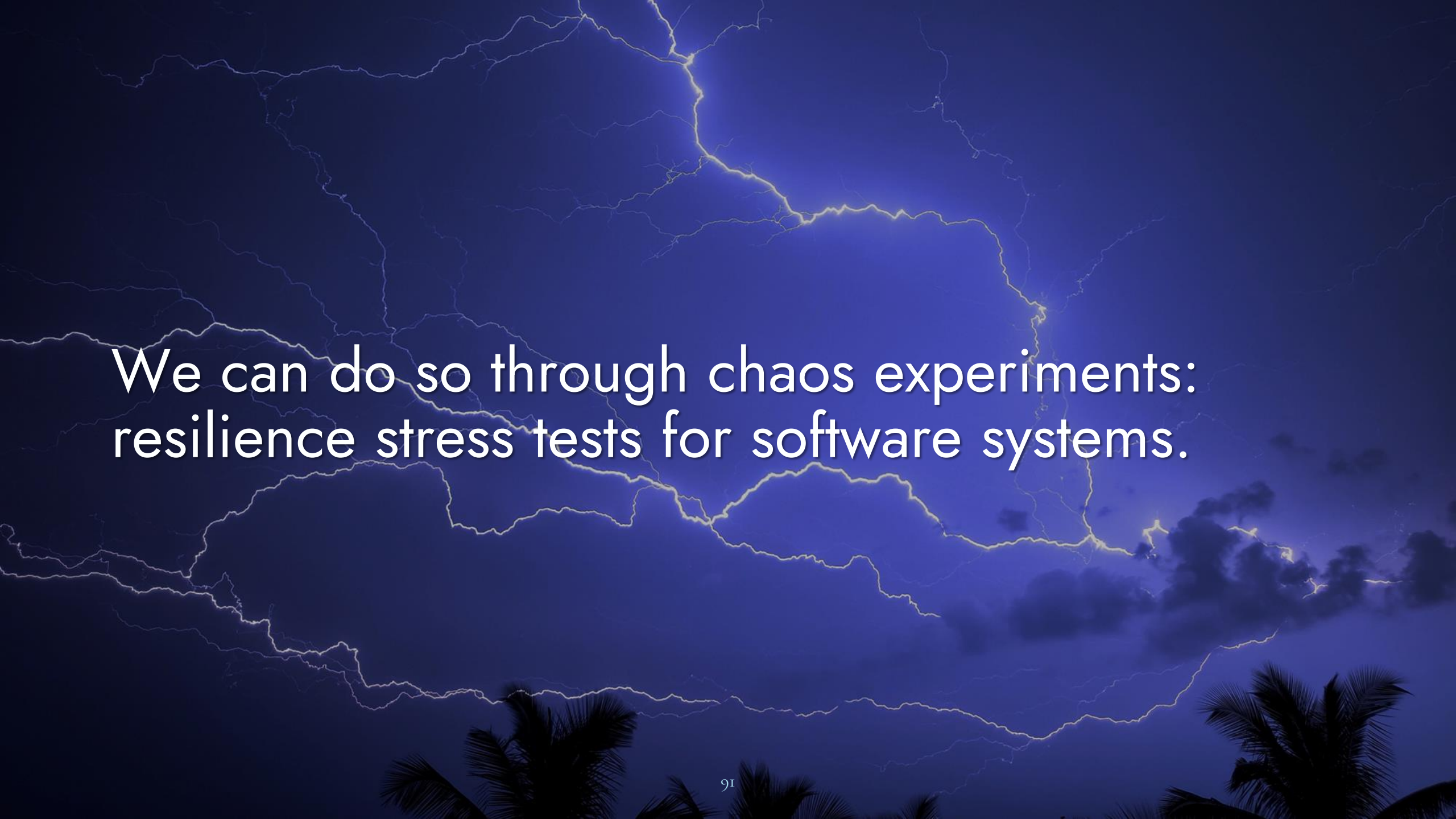
A single input in one component is insufficient for reproducing catastrophic failures in tests

shortridge@hachyderm.io | @swagitda_


(Security) chaos experiments

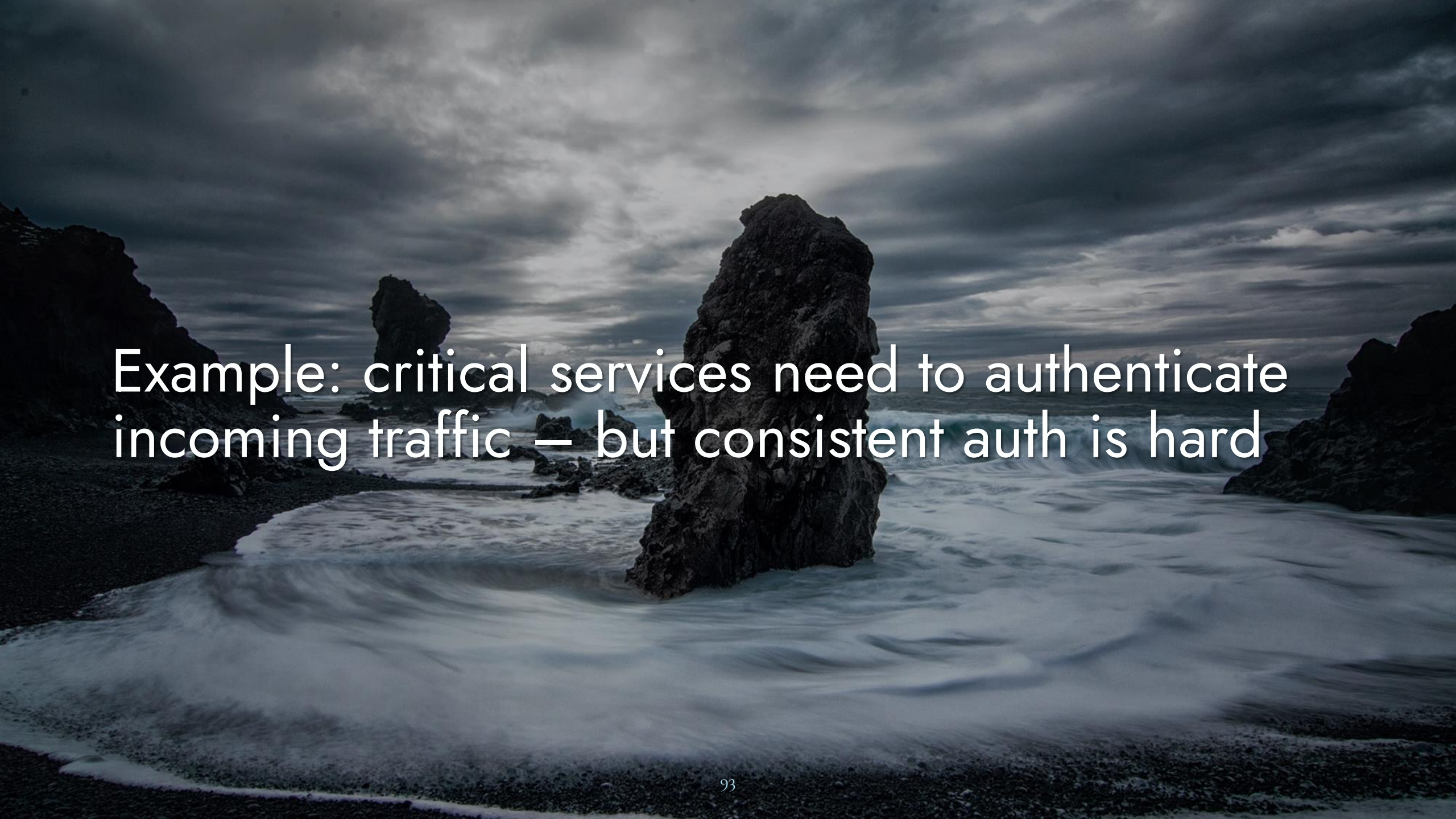
Our goal is to uncover “baffling interactions” in our systems that defy our expectations.

shortridge@hachyderm.io | @swagitda_


We can do so through chaos experiments:
resilience stress tests for software systems.

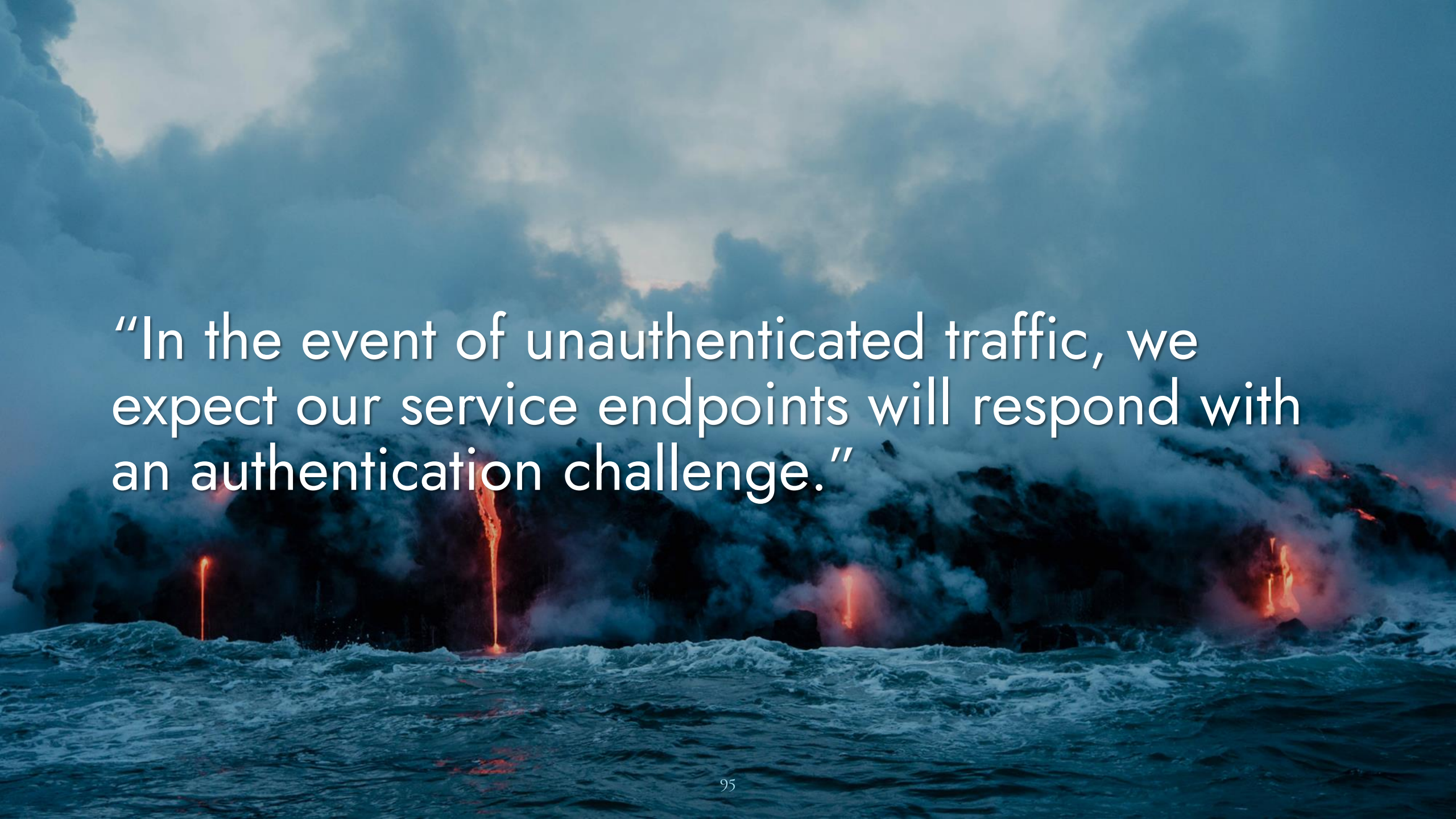
Chaos experiments help us more quickly learn about system behavior and its context.

shortridge@hachyderm.io | @swagitda_


Example: critical services need to authenticate incoming traffic – but consistent auth is hard

Create an experiment for evidence of which services automatically require authN

shortridge@hachyderm.io | @swagitda_


“In the event of unauthenticated traffic, we expect our service endpoints will respond with an authentication challenge.”

Is our org's chosen authN middleware present in everything we deploy? Collect evidence!

Hypothesis proven incorrect: authN is not validated properly everywhere + no alerts

shortridge@hachyderm.io | @swagitda_


Evidence informs design changes to our middleware and observability pipelines

But more questions remain: were there other failures associated with this scenario? Did we receive alerts elsewhere? Any reported issues?

IV. Feedback Loops and Learning

We must learn from system behavior during adverse events and use it to inform change

shortridge@hachyderm.io | @swagitda_

We need ways to summon, preserve, and learn from these memories for a feedback loop

shortridge@hachyderm.io | @swagitda_


Distributed tracing

It's difficult to look at breadcrumbs left by the system that aren't brought together in a story

shortridge@hachyderm.io | @swagitda_


You can’t form a feedback loop without being able to see what’s going on over time.

We should plan for and build this feedback into our services through tracing and logging.

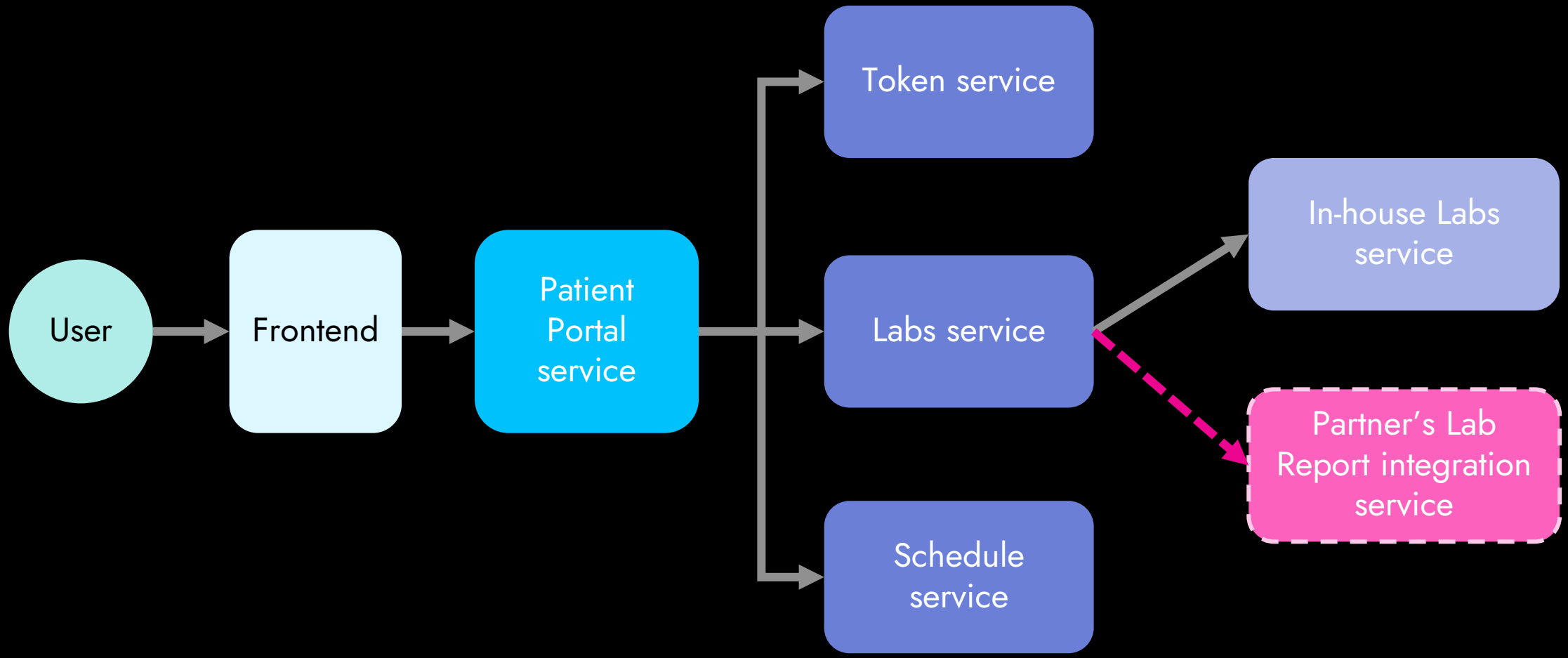
shortridge@hachyderm.io | @swagitda_


Distributed tracing lets us observe the flow of data as it pours through a distributed system

We can assign a trace ID at the point of traffic ingress and follow the event as it flows through

shortridge@hachyderm.io | @swagitda_

Case study: an attacker exfiltrating data from a hospital's patient portal


User
Frontend
Patient Portal service
Token service
Labs service
In-house Labs service
Partner's Lab Report integration service
Schedule service


How do we trace the data flows from all the requests from the Labs service and beyond?

Distributed tracing dissipates this nightmare by assigning a trace ID at the traffic ingress point

shortridge@hachyderm.io | @swagitda_


Distributed tracing also helps us refine system design and design new, better versions

We need to understand the impact a potential design change has on our tree of consumers

shortridge@hachyderm.io | @swagitda_


Dist tracing helps us refine that mental model
by learning about real interactions in the system

It makes the statement that we want to correlate data across systems – that we want that trace ID

V. Flexibility and Willingness to Change

We must remain flexible in the face of failures and evolving conditions

shortridge@hachyderm.io | @swagitda_


Nature is a patient architect, allowing evolution
to bloom over generational cycles

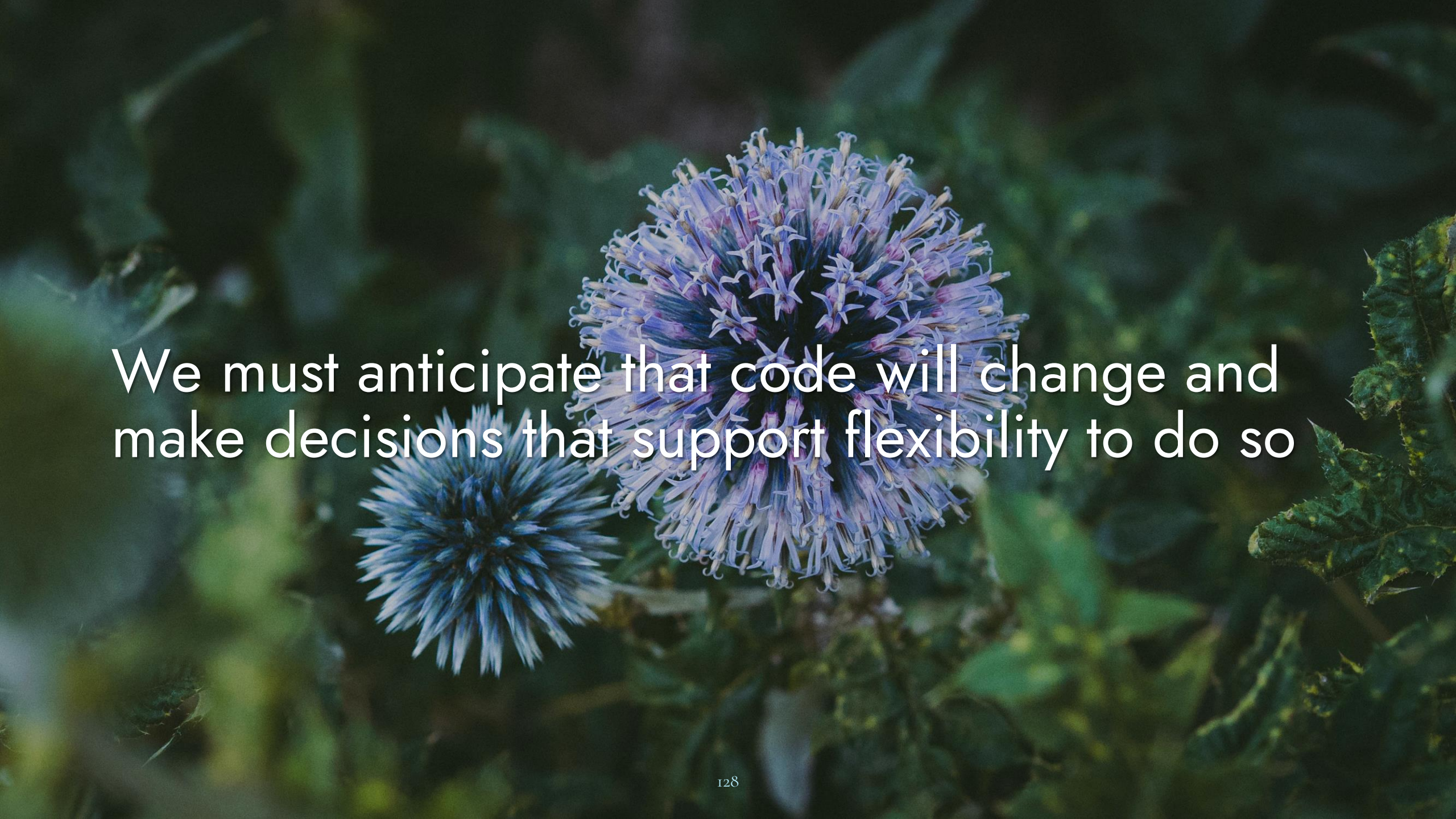
We need strategies that promote the speed on which our graceful adaptability depends

shortridge@hachyderm.io | @swagitda_


Preserving possibilities for refactoring

No one thinks about the remake when they film the original – same with code and refactoring

shortridge@hachyderm.io | @swagitda_


We must anticipate that code will change and make decisions that support flexibility to do so

We need an easy path to safely restructure abstractions, data models, and approaches

shortridge@hachyderm.io | @swagitda_


Type systems are often thought of as a way to resist change, but they can facilitate change

Type declarations can help us preserve possibilities when developing code

shortridge@hachyderm.io | @swagitda_

A type is a set of requirements declaring what operations can be performed on values that are considered to conform to the type.


Static typing can make it easier to refactor since type errors help guide the migration.

If we pass around Int64s to represent a time-stamp, then call them “Timestamp” for clarity

shortridge@hachyderm.io | @swagitda_

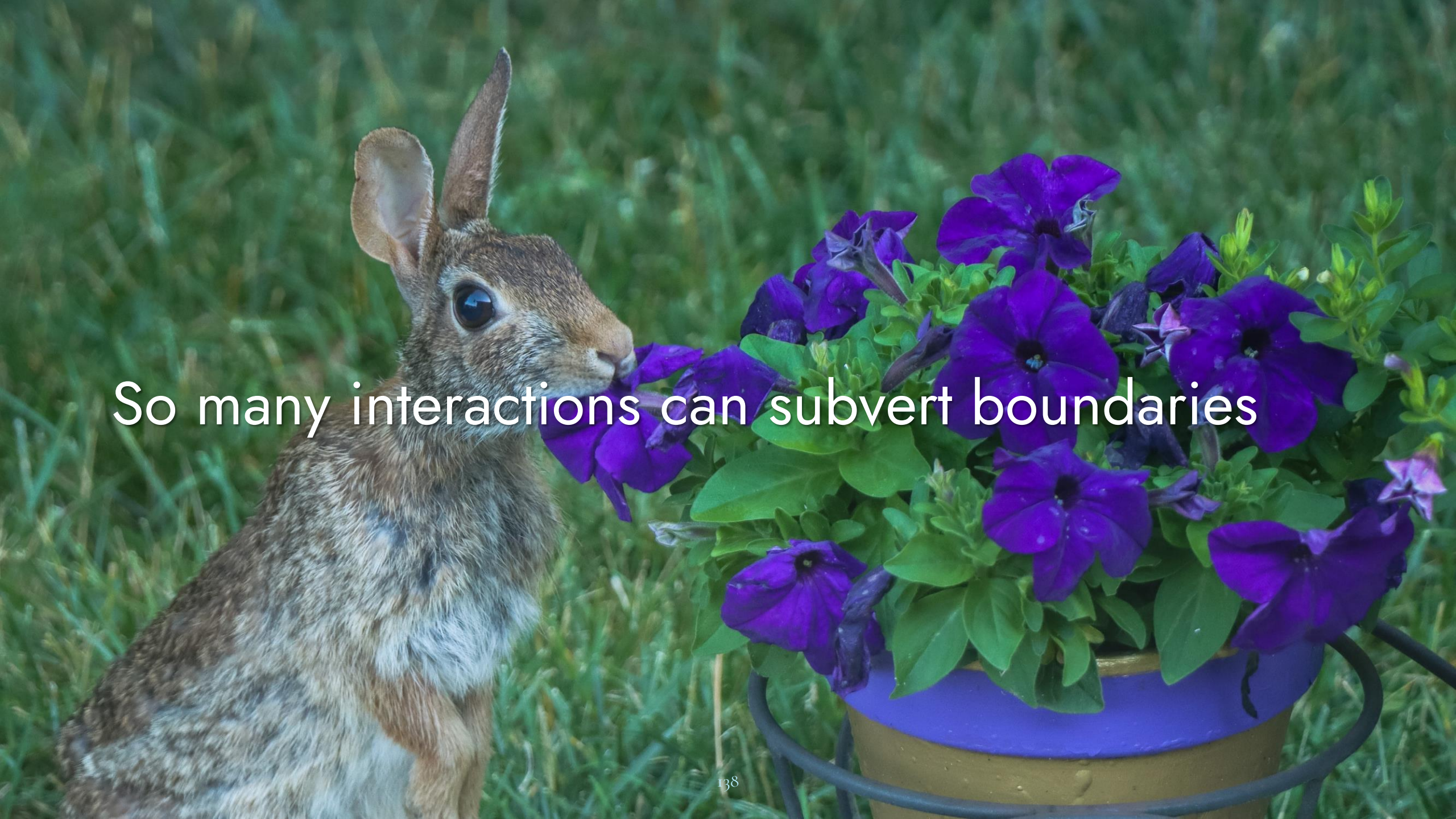
The more clarity we can crystallize around the system's functions, the more we can adapt

shortridge@hachyderm.io | @swagitda_


Modularity

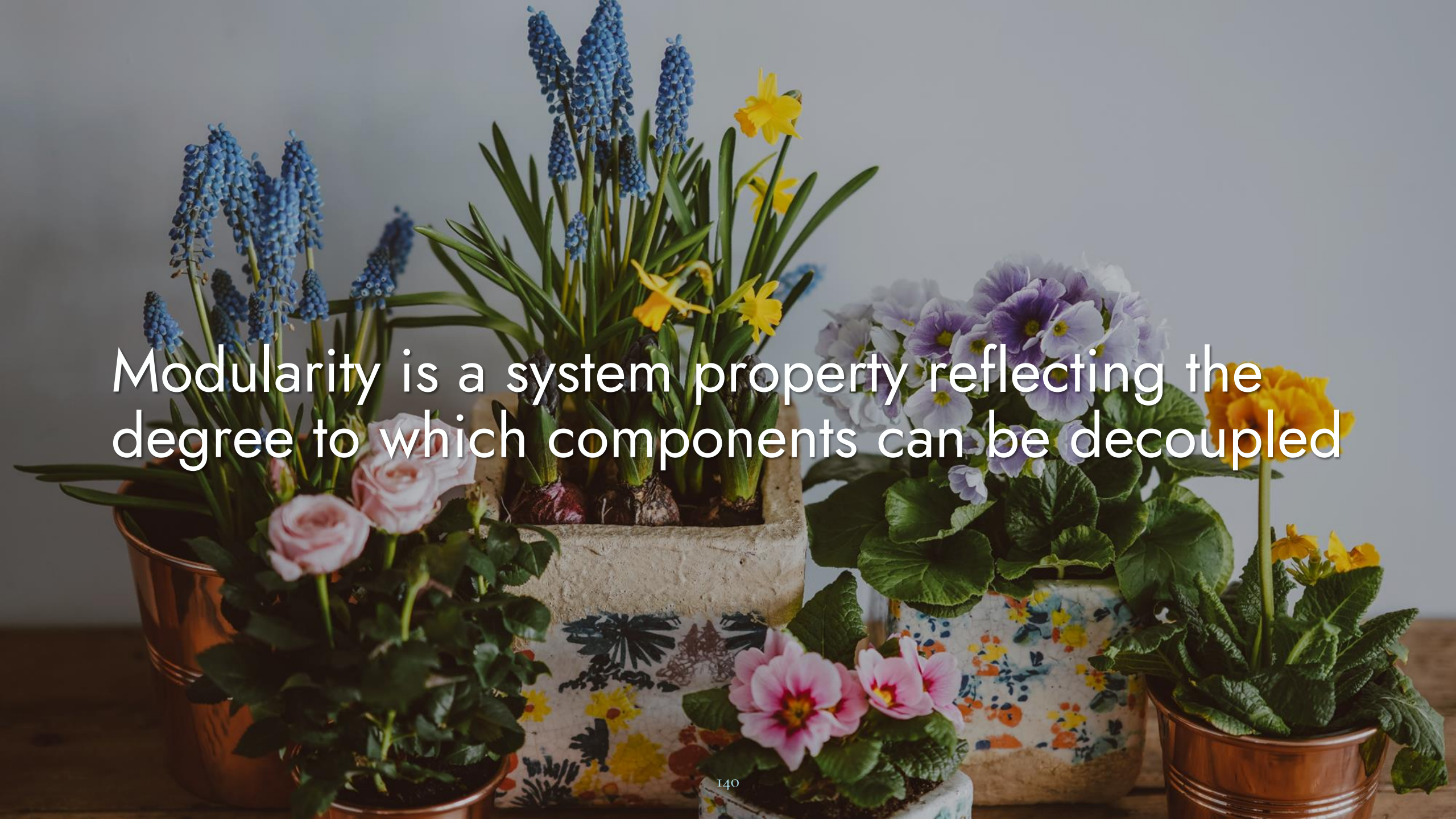
Engineers fundamentally misunderstand modularity with respect to resilience

shortridge@hachyderm.io | @swagitda_


So many interactions can subvert boundaries

Modularity allows “structurally or functionally distinct parts to retain autonomy during a period of stress and allows for easier recovery.”

shortridge@hachyderm.io | @swagitda_



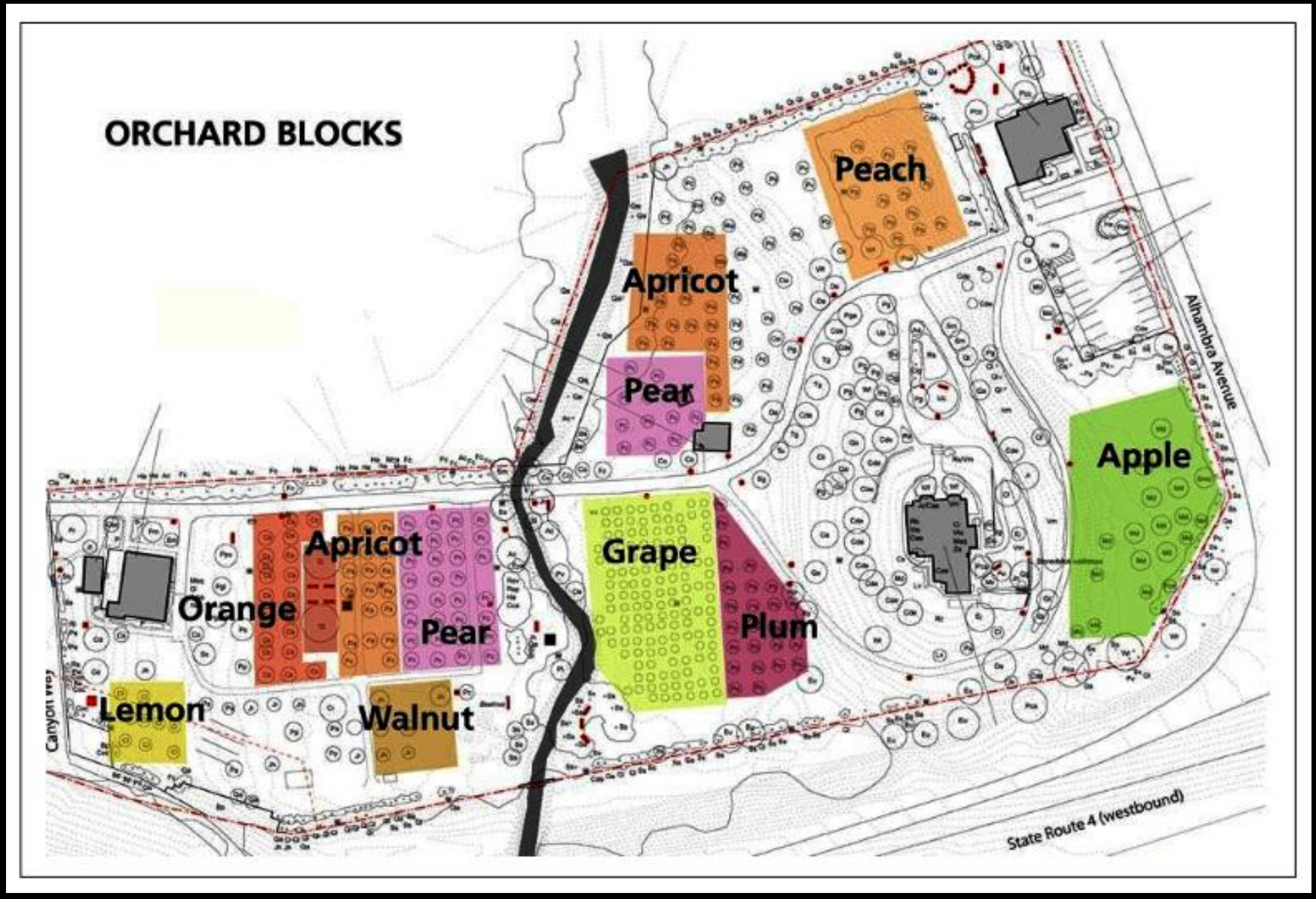
Modularity is a system property reflecting the degree to which components can be decoupled

Humans have intuitively grasped how modularity supports resilience for millenia

shortridge@hachyderm.io | @swagitda_

During a disturbance, a modular feature can function independently of other features

shortridge@hachyderm.io | @swagitda_



shortridge@hachyderm.io | @swagitda_

When there's low modularity, failure cascades pervade – it enables contagion effects

shortridge@hachyderm.io | @swagitda_

Ransomware's success relies on low modularity

A system with high modularity can contain or "buffer" stressors and surprises


Modules create a local boundary for isolation

Isolation is a core property that supports software and systems resilience

shortridge@hachyderm.io | @swagitda_


In software, we're lucky that we can isolate failure to handle unexpected interactions

Start “boring”: set AWS security groups — or use serverless functions, containers, or VMs

RLBox: trap C code in a WebAssembly (Wasm) sandbox to isolate hazardous subcomponents

shortridge@hachyderm.io | @swagitda_


If a vulnerable component is in a sandbox, the attacker faces a challenge to reach their goal

Modularity makes navigating and updating the system easier, too

Chaos experiments show us to what extent our modular boundaries are useful for resilience

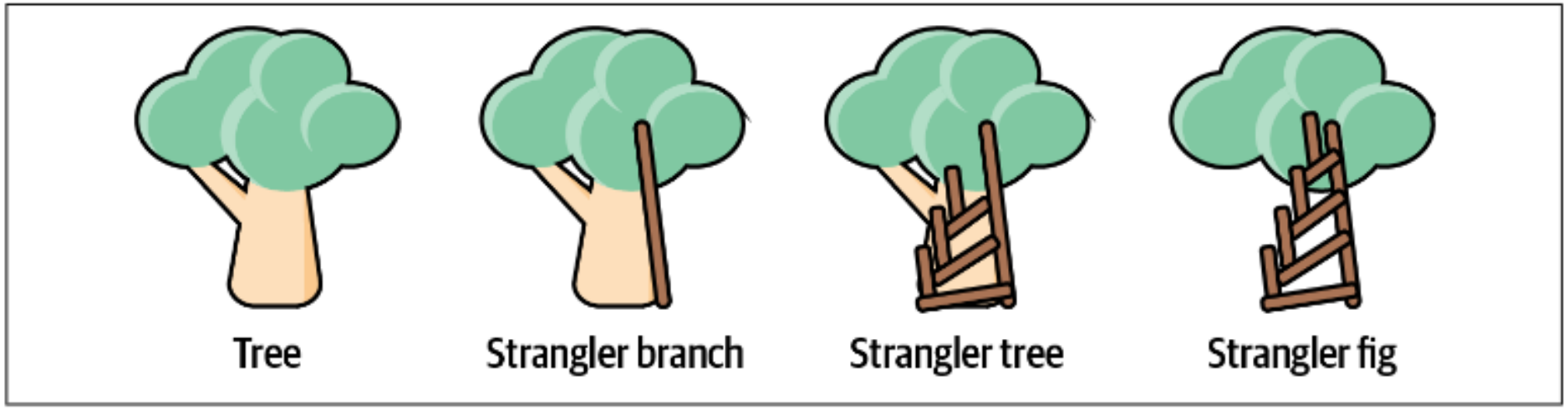
shortridge@hachyderm.io | @swagitda_

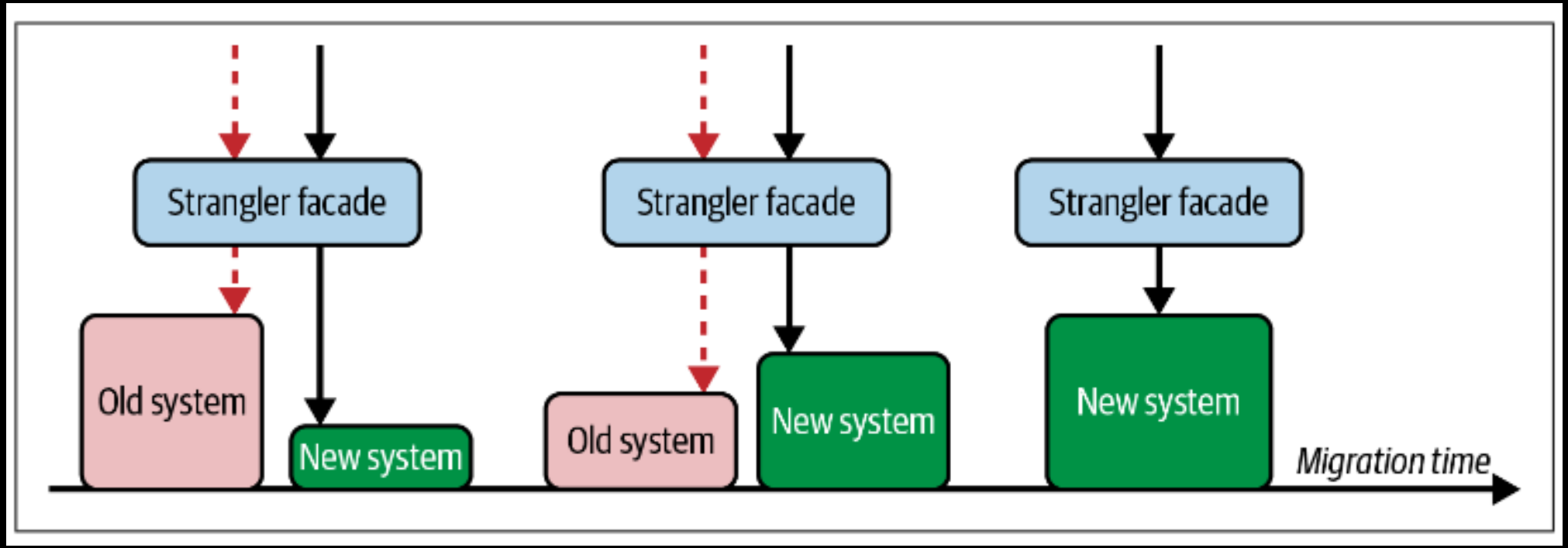

Strangler Fig

How can we change our system without contaminating critical functionality?

shortridge@hachyderm.io | @swagitda_

The Strangler Fig pattern supports our capacity to change and helps us maintain flexibility


Tree
Strangler branch
Strangler tree
Strangler fig


Strangler facade
Old system
New system
Strangler facade
Old system
New system
Strangler facade
New system
Migration time

Stranger Fig is the conservative approach – but usually also the faster and sustainable one

shortridge@hachyderm.io | @swagitda_

We also need to transform the socio part; humans' mental models are often sticky.

The new principles and practices we adopt when changing need incremental iteration, too

shortridge@hachyderm.io | @swagitda_


Savoring our Potion

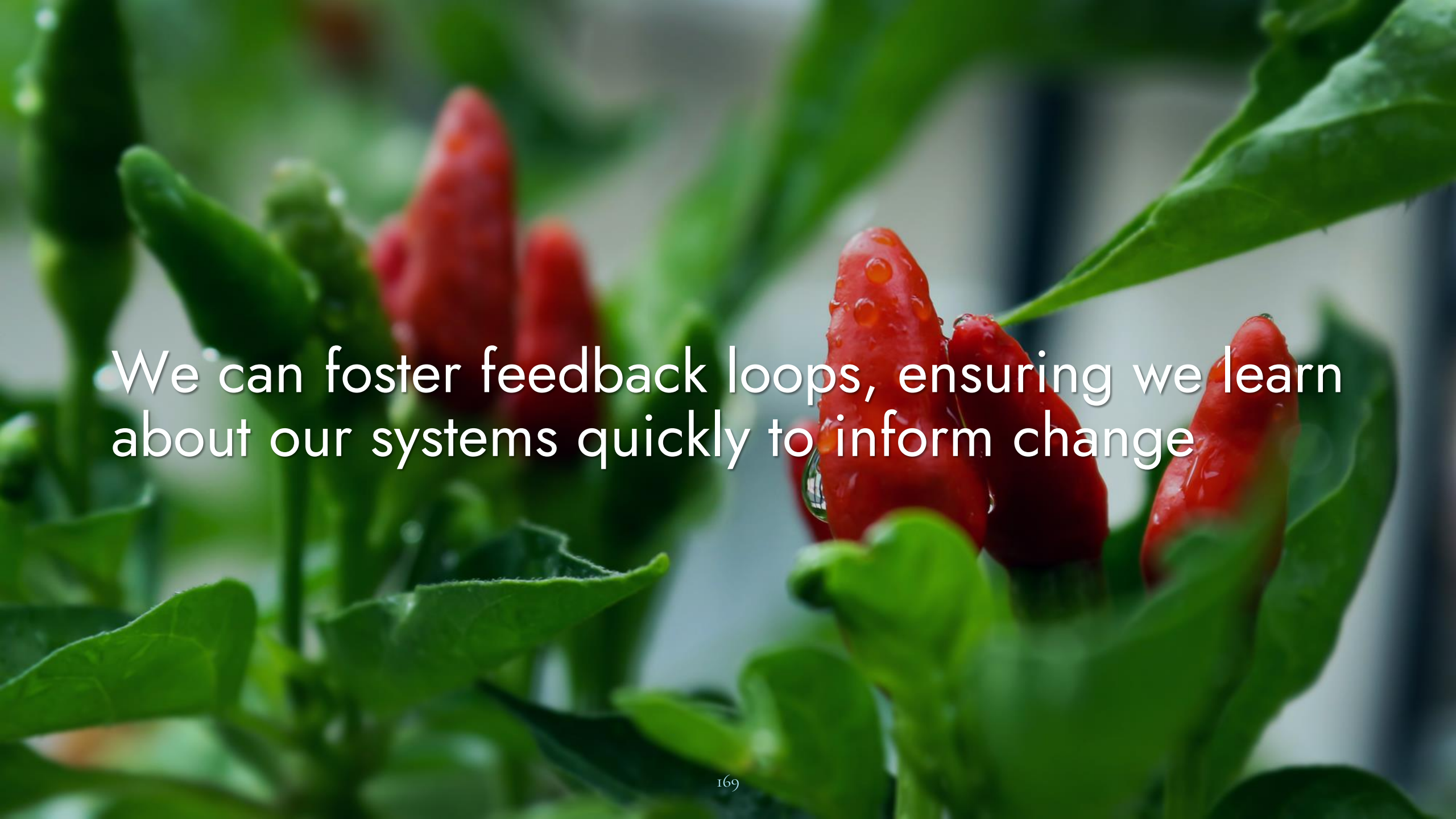

Resilience means organizations respond to failure & adapt to evolving conditions with grace

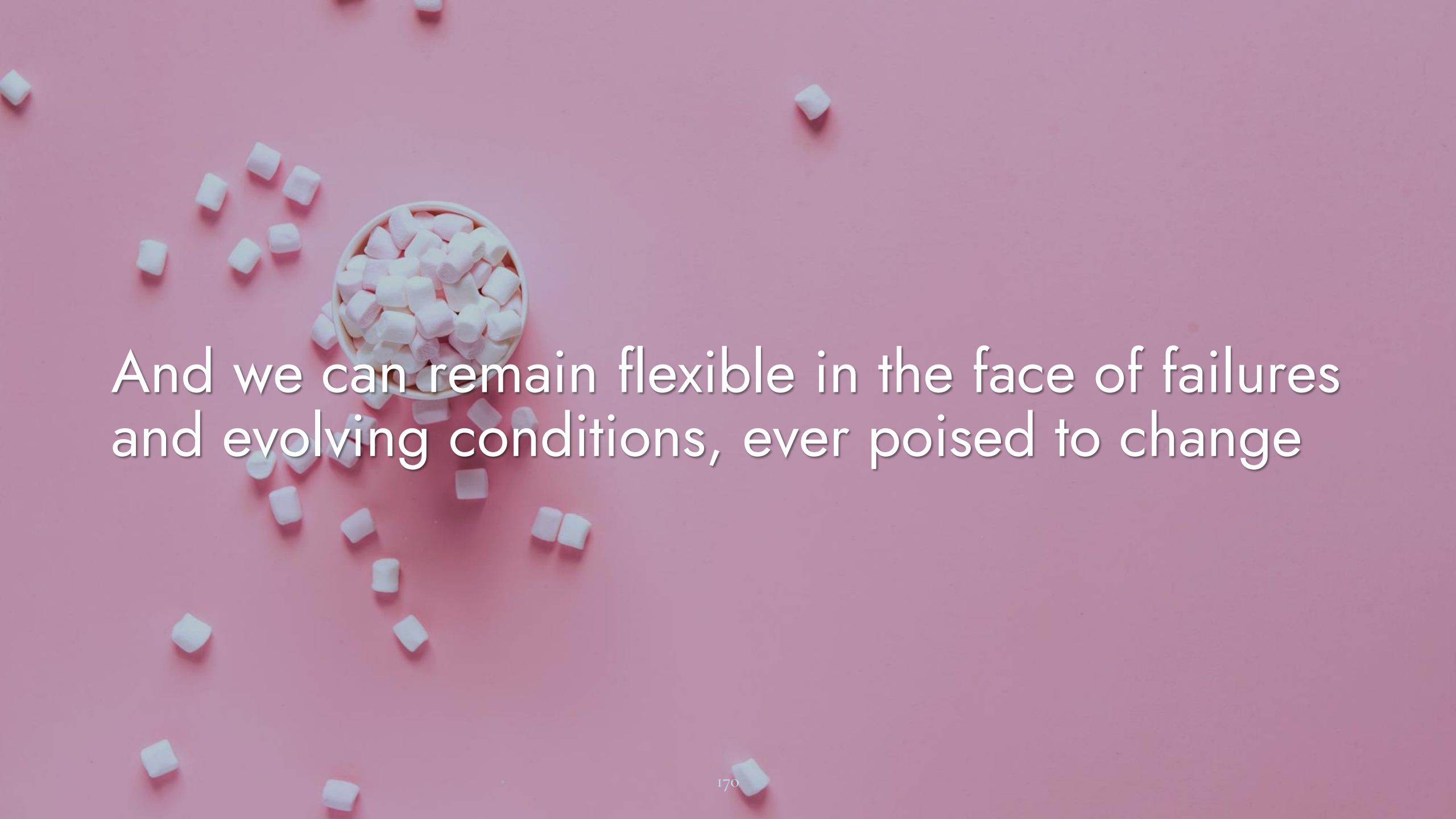
We can foster the five key ingredients we need to brew the Resilience Potion during dev

We can define our critical functions and prioritize preserving them in adverse conditions

We can understand and expand our system's boundaries of safe operation

We can observe system interactions across space-time and make them more understandable


We can foster feedback loops, ensuring we learn about our systems quickly to inform change



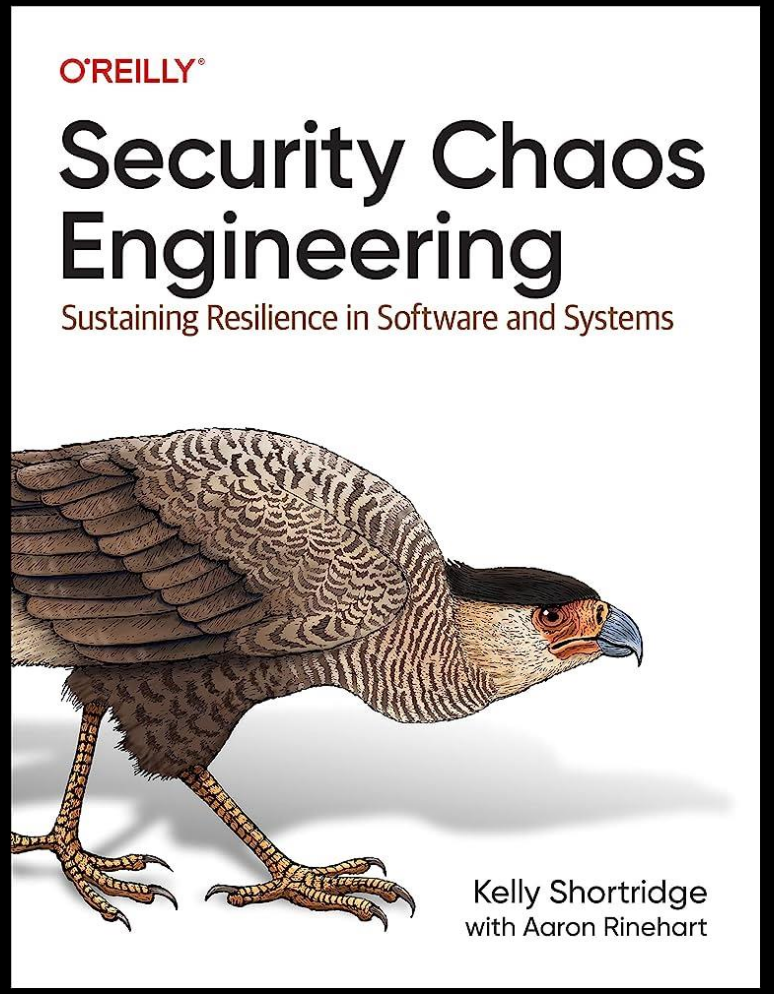
And we can remain flexible in the face of failures and evolving conditions, ever poised to change

Order the book today:

Amazon

Bookshop

& other major retailers



shortridge@hachyderm.io | @swagitda_

/in/kellyshortridge

@swagitda_

shortridge@hachyderm.io

@shortridge.bsky.social

chat@shortridge.io