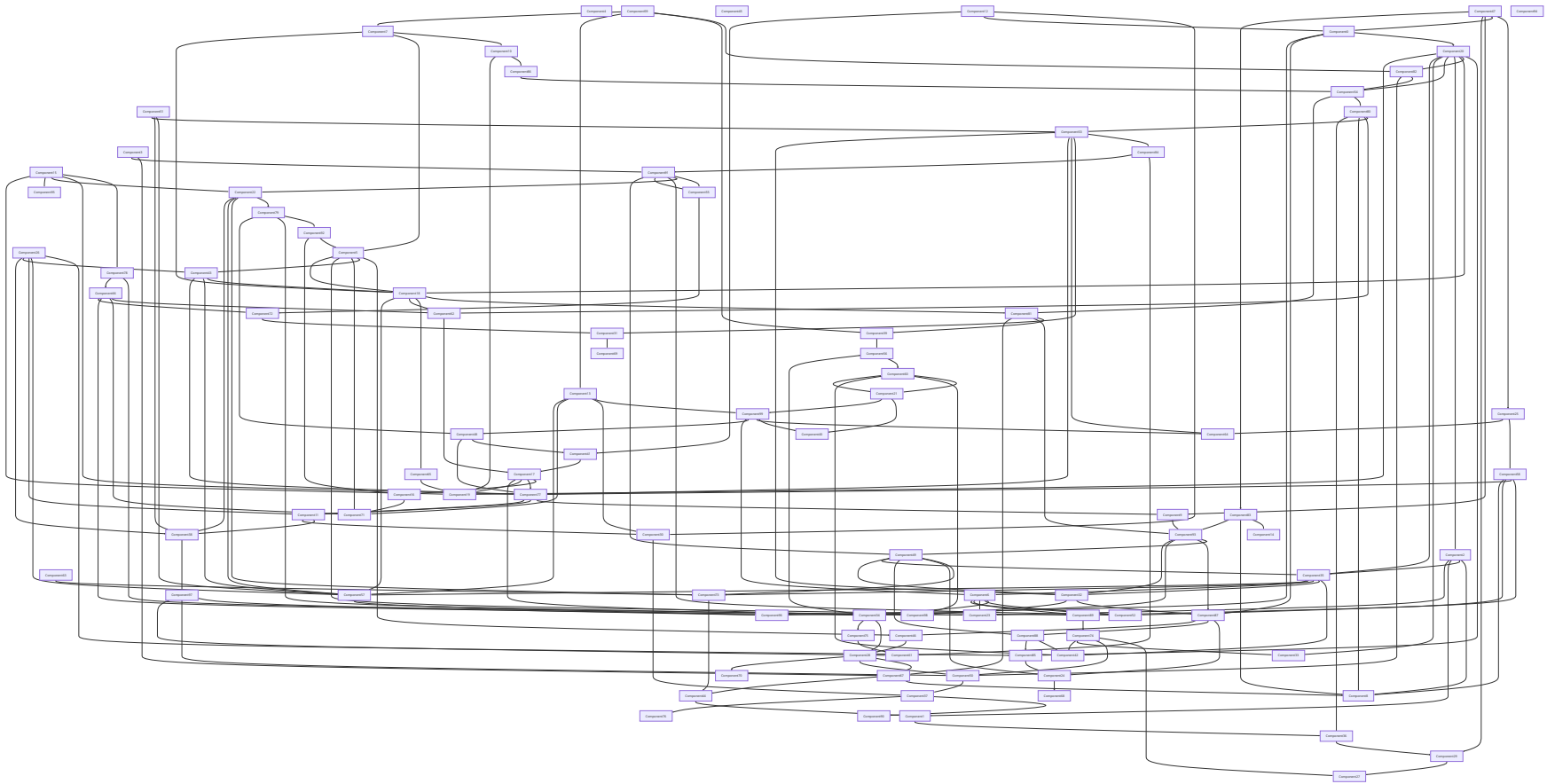
Strangling the dragon

Modernising legacy systems using the Strangler Fig pattern

Daniel Raniz Raneland
Coding Architect @ factor10

factor10.com
raniz@factor10.com
raniz.blog
raniz@mastodon.online
/in/raneland

Is this your system?



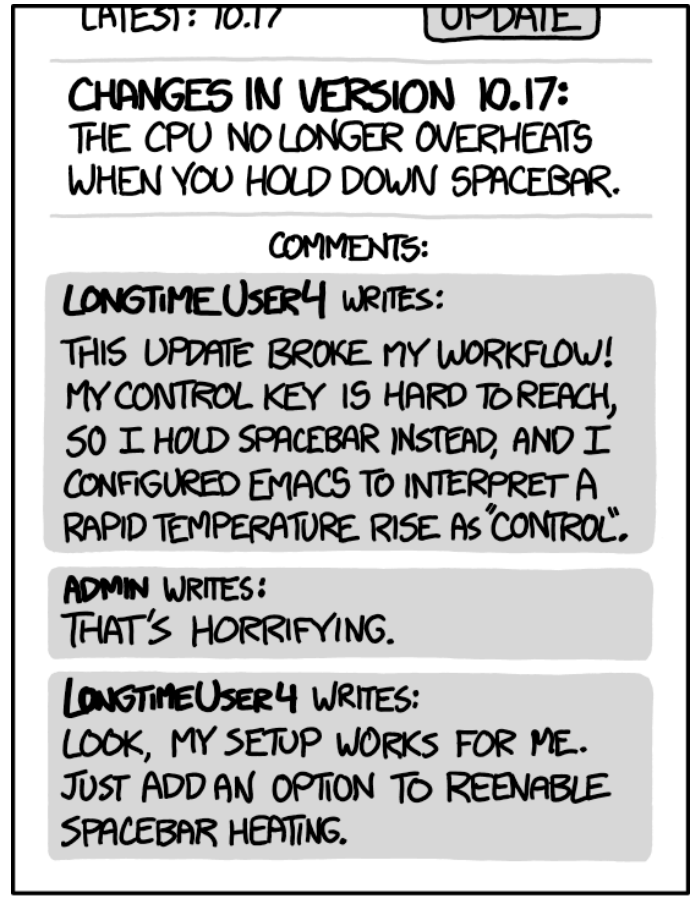
Nikola Jovanovic via Unsplash

However …

Most rewrites fail

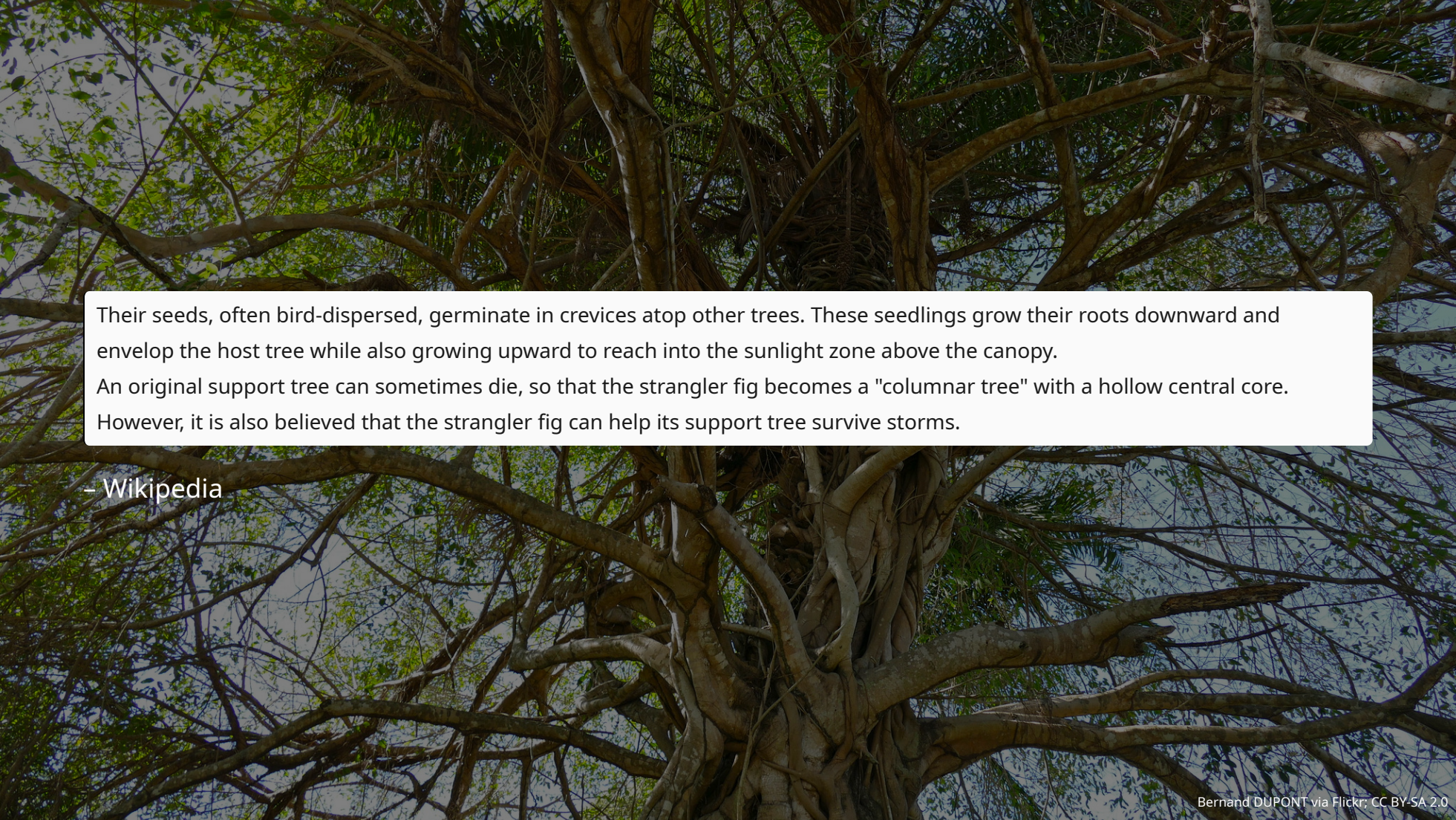
Most rewrites fail

- Chasing a moving target
- Poorly documented requirements
- Second system effect
- Complexity underestimation
  - Old edge cases show themselves
  - Complex requirements that were forgotten surface again
- Old bugs
  - Some resurface
  - Should we keep the existing ones?



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

The Strangler Fig pattern



Their seeds, often bird-dispersed, germinate in crevices atop other trees. These seedlings grow their roots downward and envelop the host tree while also growing upward to reach into the sunlight zone above the canopy.
An original support tree can sometimes die, so that the strangler fig becomes a "columnar tree" with a hollow central core.
However, it is also believed that the strangler fig can help its support tree survive storms.

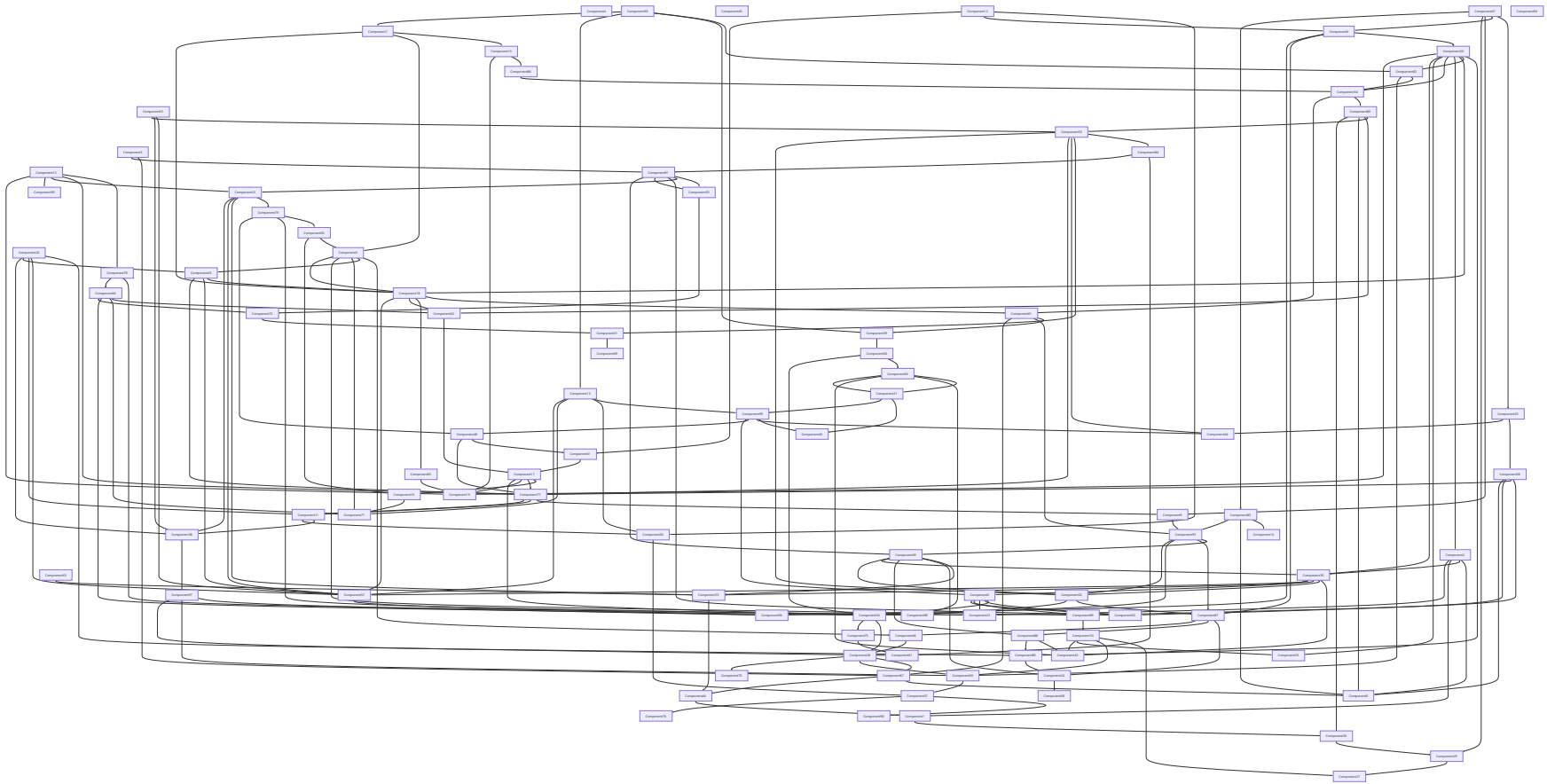
– Wikipedia

The Strangler Fig pattern

1. Identify one (small) part of the system
2. Break it out and/or replace it.
3. Repeat...

Sounds easy?

Is this your system?



Story time!

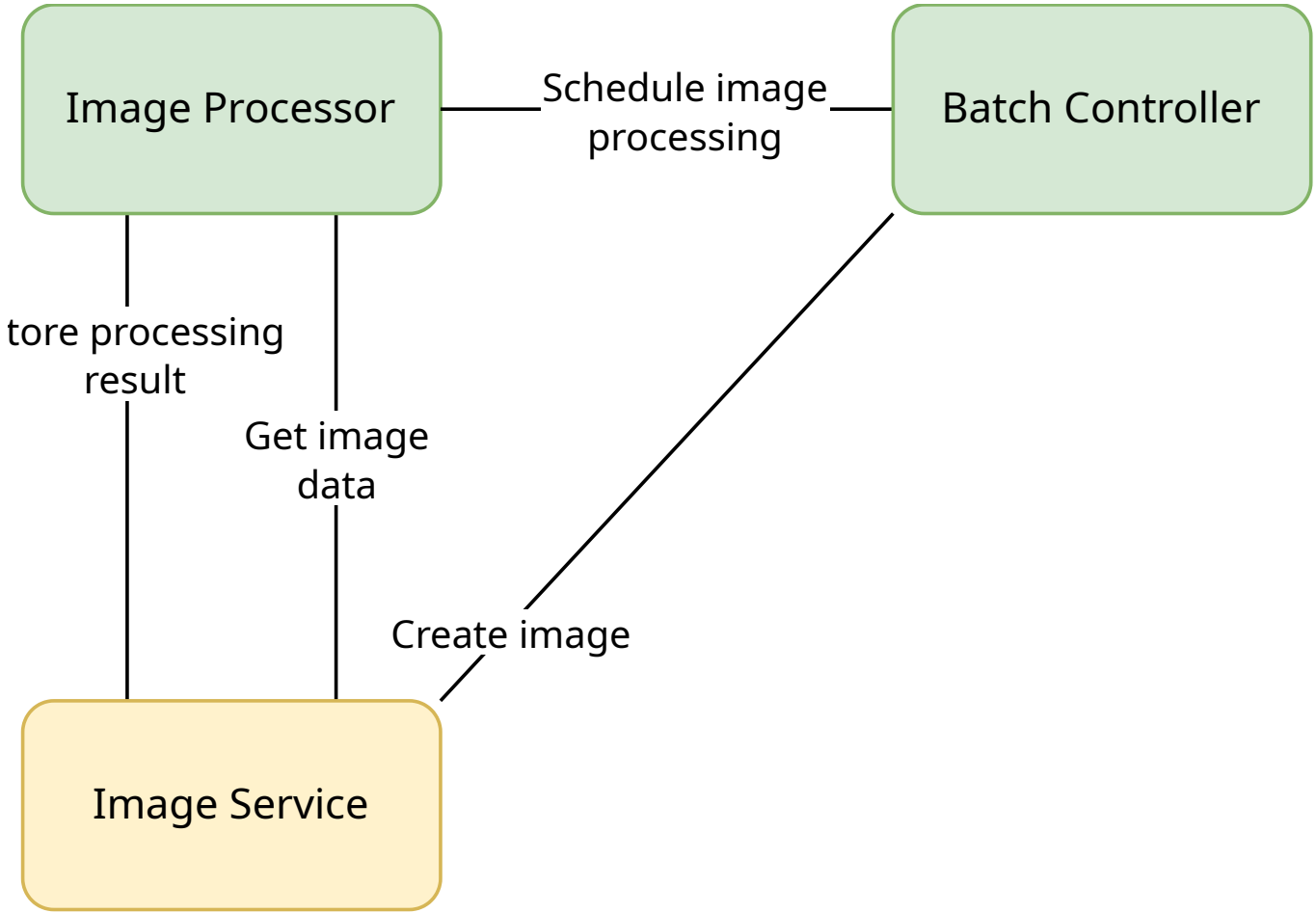


Image Processor
Schedule image processing
Batch Controller
tore processing result
Get image data
Create image
Image Service

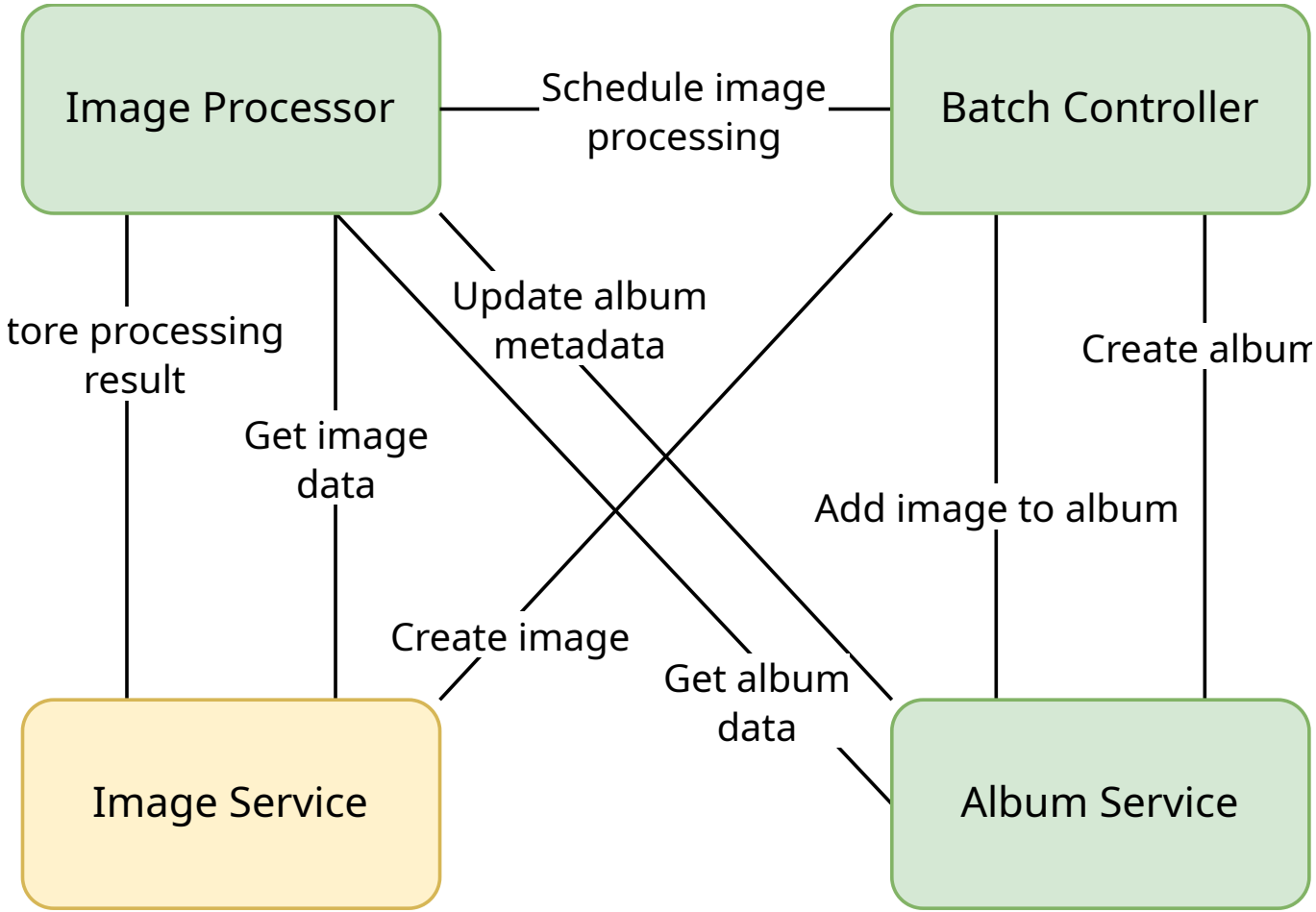
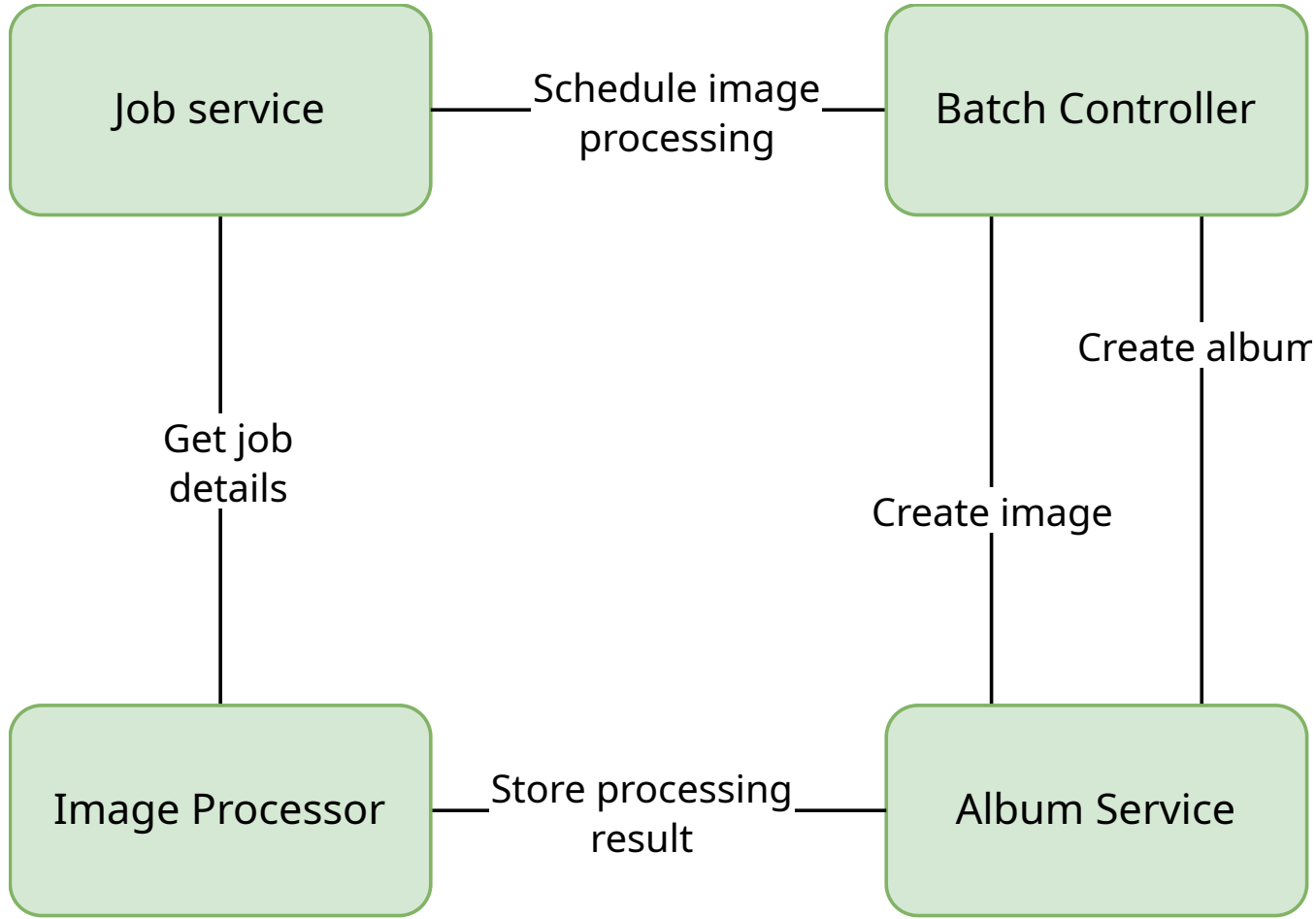


Image Processor
Schedule image processing
Batch Controller
tore processing result
Update album metadata
Create album
Get image data
Add image to album
Create image
Get album data
Image Service
Album Service


Job service
Schedule image processing
Batch Controller
Create album
Get job details
Create image
Image Processor
Store processing result
Album Service

2016: Determine that the current architecture was not ideal for future use-cases

2016: Add Job Service

2016–2019: Gradually move functionality out of *Image Service*

2019: Replace the last piece of functionality of *Image Service*

2019: Decomission *Image Service* cluster

2016: Determine that the current architecture was not ideal for future use-cases

2016: Add Job Service

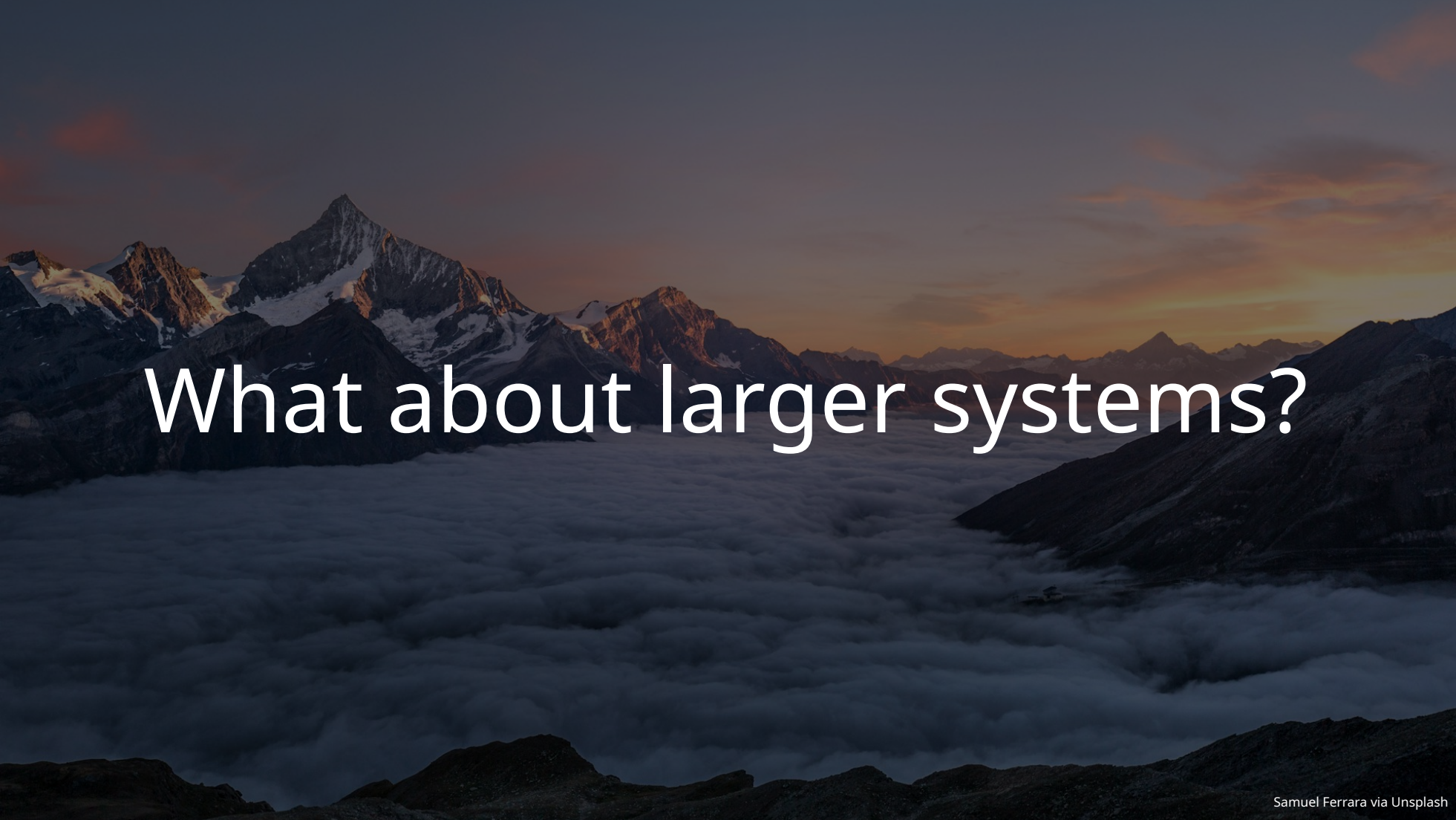
2016–2019: Gradually move functionality out of *Image Service*

2019: Replace the last piece of functionality of *Image Service*

2019: Decomission *Image Service* cluster

2019: Celebrate with champagne

Alexander Naglestad via Unsplash



Samuel Ferrara via Unsplash

DDD, TDD and Conway’s law

The Strangler Fig’s best friends

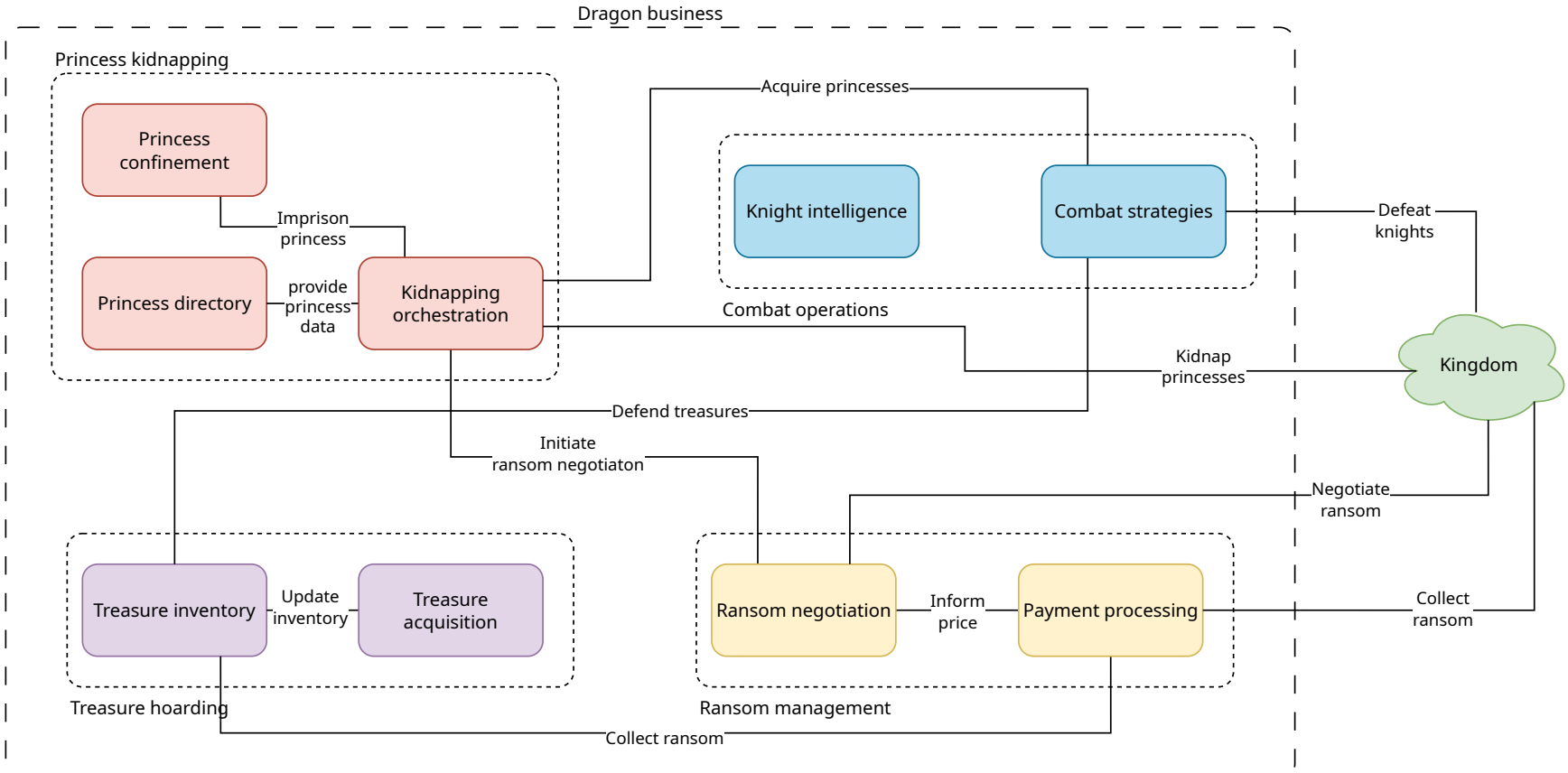
Domain-Driven Design

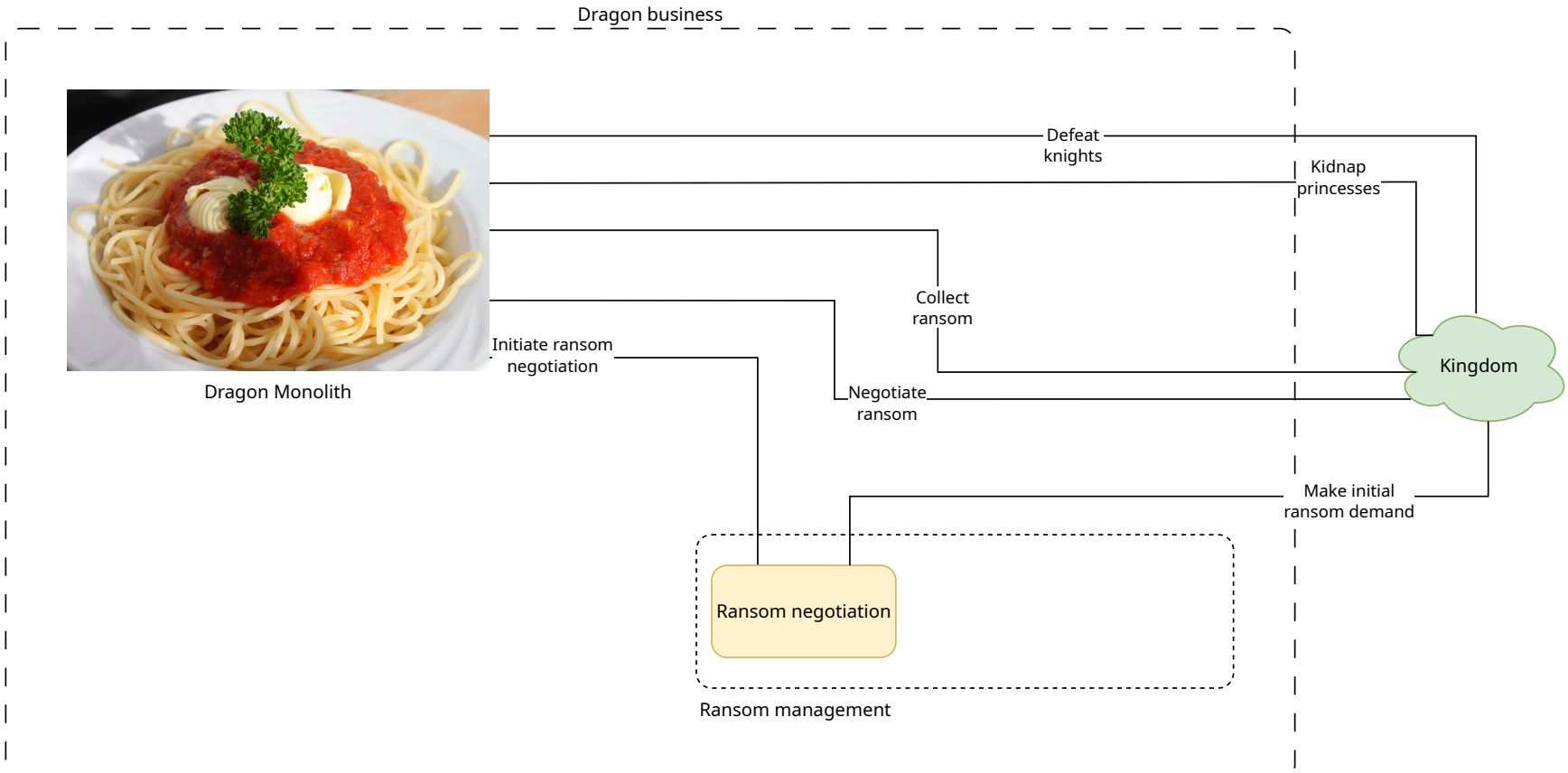
Developing the future vision

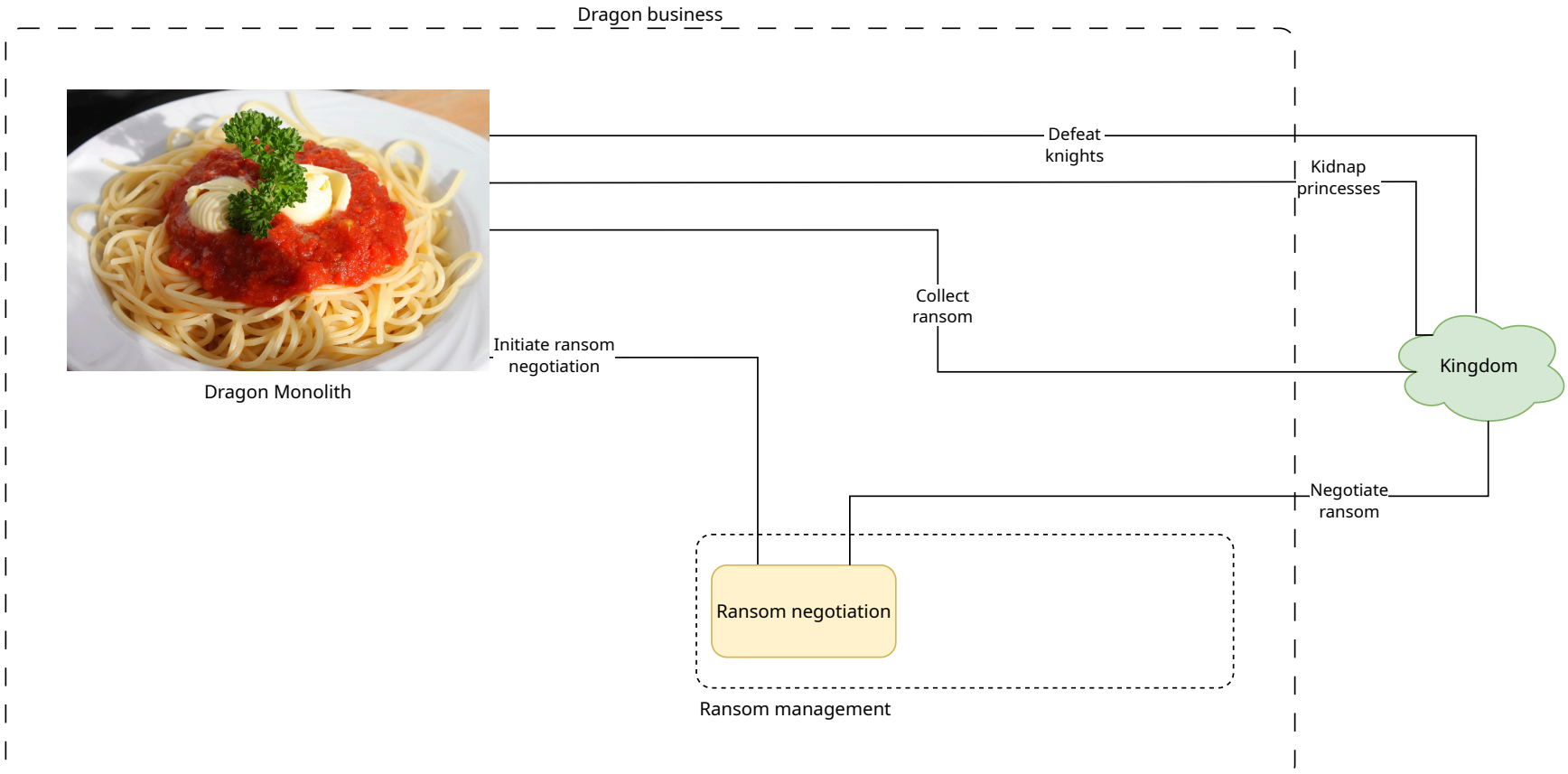
What is Domain-Driven Design?

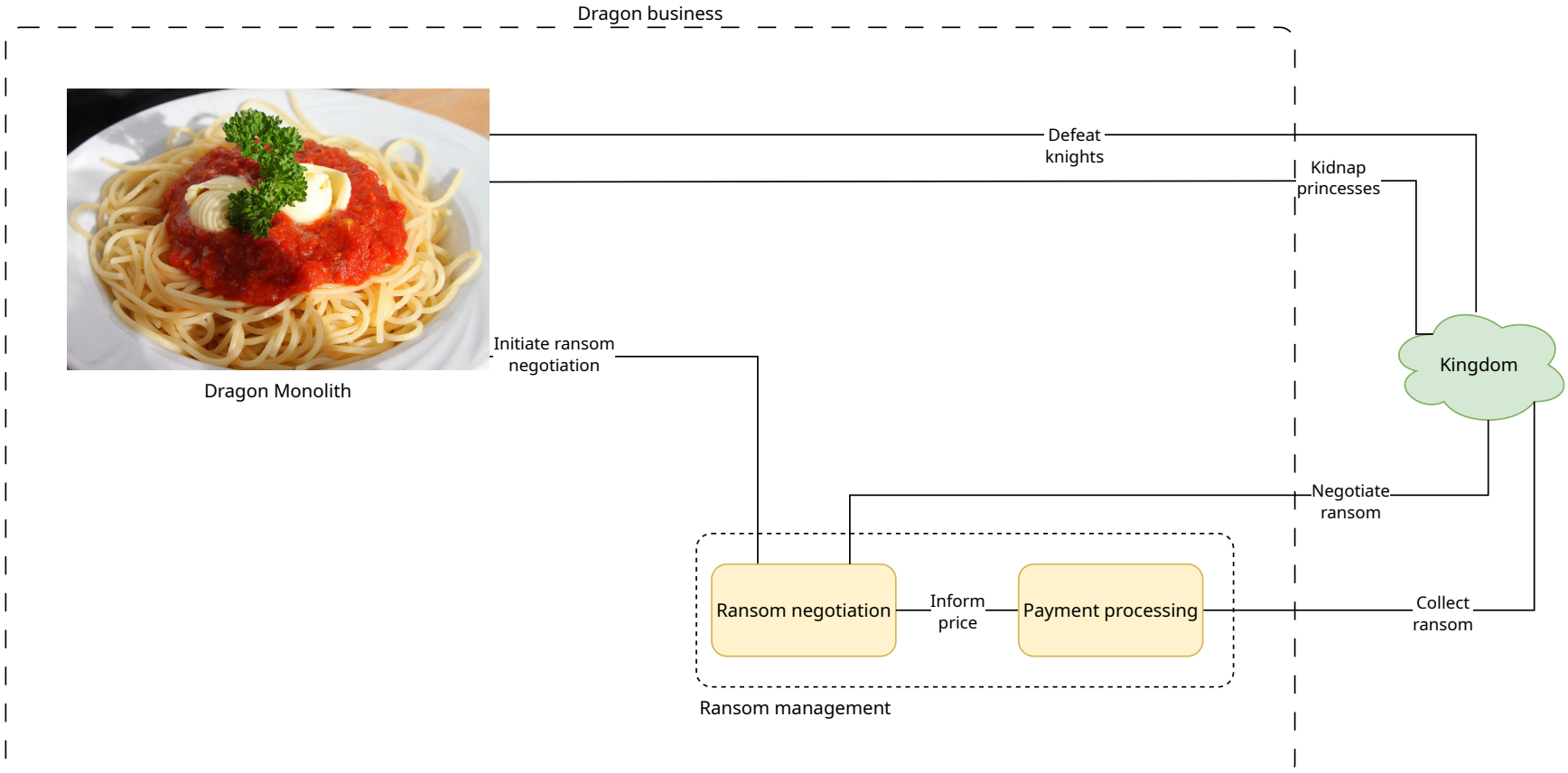
> a major software design approach, focusing on modeling software to match a domain according to input from that domain's experts.

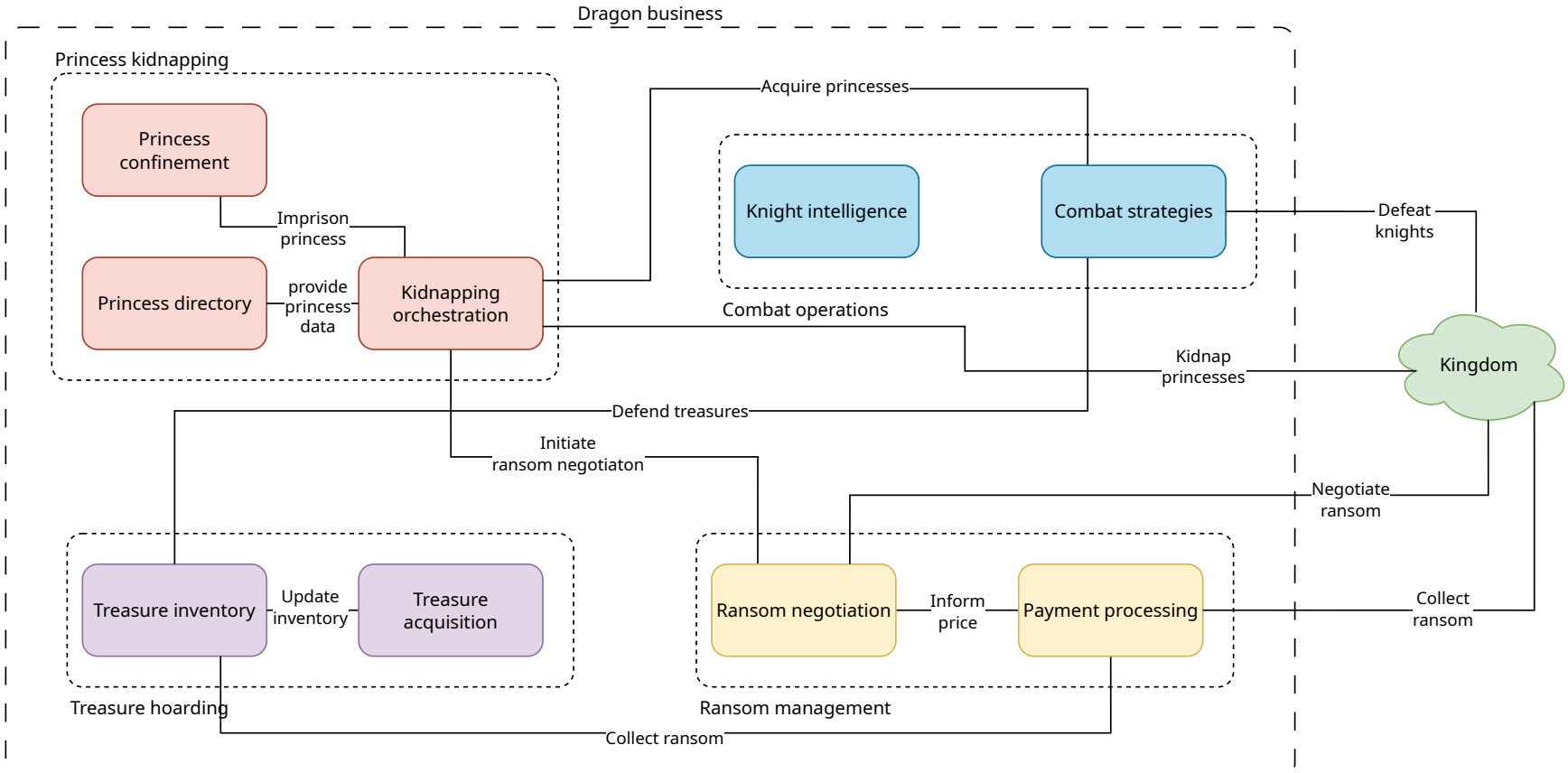
– Wikipedia


Dragon business
Princess kidnapping
Princess confinement
Imprison princess
Princess directory
provide princess data
Kidnapping orchestration
Acquire princesses
Knight intelligence
Combat strategies
Defeat knights
Combat operations
Kidnap princesses
Kingdom
Defend treasures
Initiate ransom negotiaton
Negotiate ransom
Treasure inventory
Update inventory
Treasure acquisition
Treasure hoarding
Ransom negotiation
Inform price
Payment processing
Collect ransom
Ransom management
Collect ransom


Dragon business
Dragon Monolith
Defeat knights
Kidnap princesses
Collect ransom
Initiate ransom negotiation
Negotiate ransom
Kingdom
Make initial ransom demand
Ransom negotiation
Ransom management


Dragon business
Defeat knights
Kidnap princesses
Collect ransom
Initiate ransom negotiation
Kingdom
Negotiate ransom
Ransom negotiation
Ransom management
Dragon Monolith


Dragon business
Dragon Monolith
Defeat knights
Kidnap princesses
Initiate ransom negotiation
Kingdom
Negotiate ransom
Ransom negotiation
Inform price
Payment processing
Collect ransom
Ransom management


Dragon business
Princess kidnapping
Princess confinement
Imprison princess
Princess directory
provide princess data
Kidnapping orchestration
Acquire princesses
Knight intelligence
Combat strategies
Defeat knights
Combat operations
Kidnap princesses
Kingdom
Defend treasures
Initiate ransom negotiaton
Negotiate ransom
Treasure inventory
Update inventory
Treasure acquisition
Treasure hoarding
Ransom negotiation
Inform price
Payment processing
Collect ransom
Ransom management
Collect ransom

Conway’s law

Inverse Conway Manoeuvring your teams to align with the vision

Conway’s law

> Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.

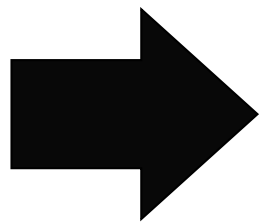
– Melvin E. Conway, How Do Committees Invent?

rontend team

Finance team

Combat team

Database team

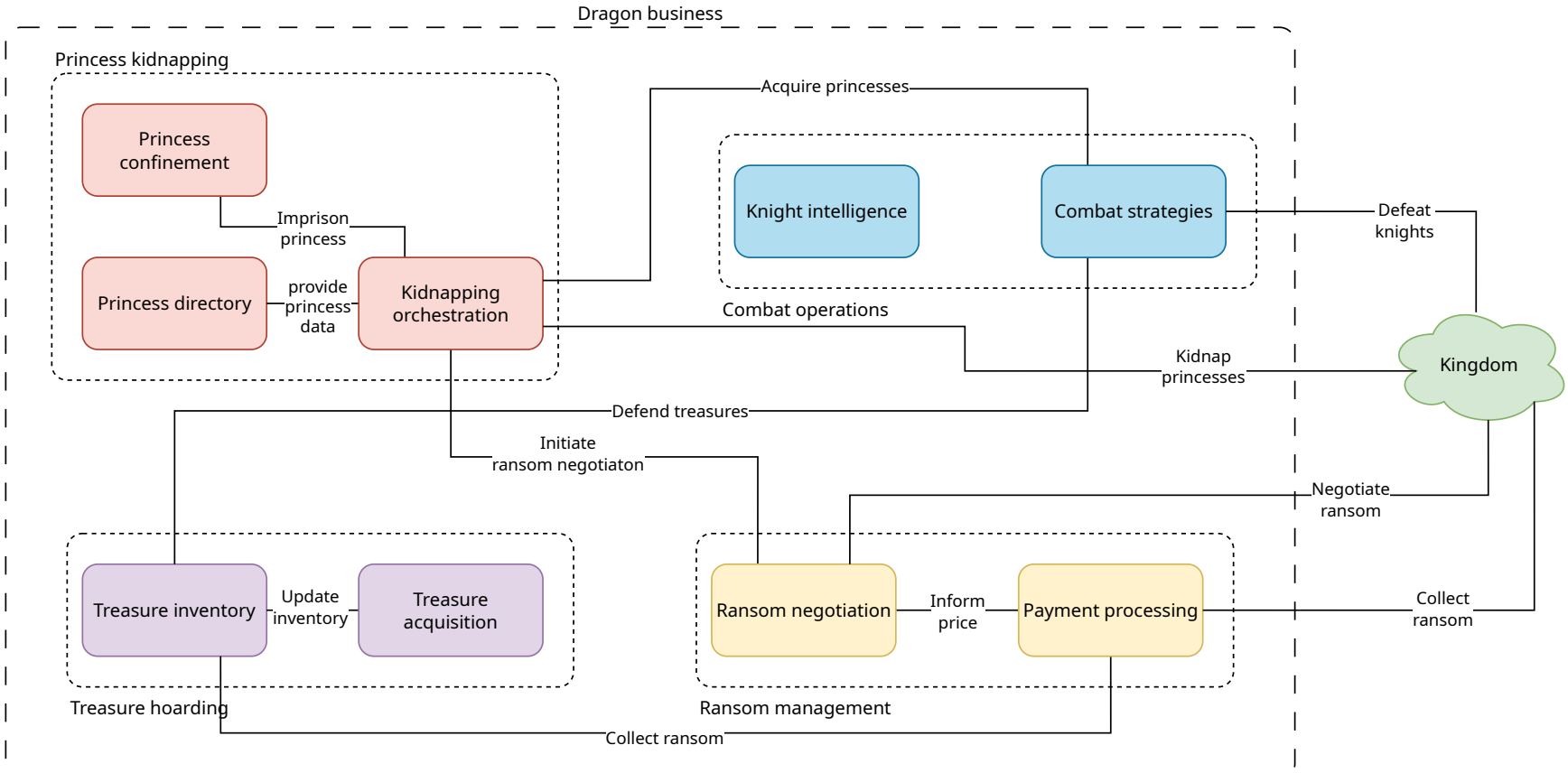


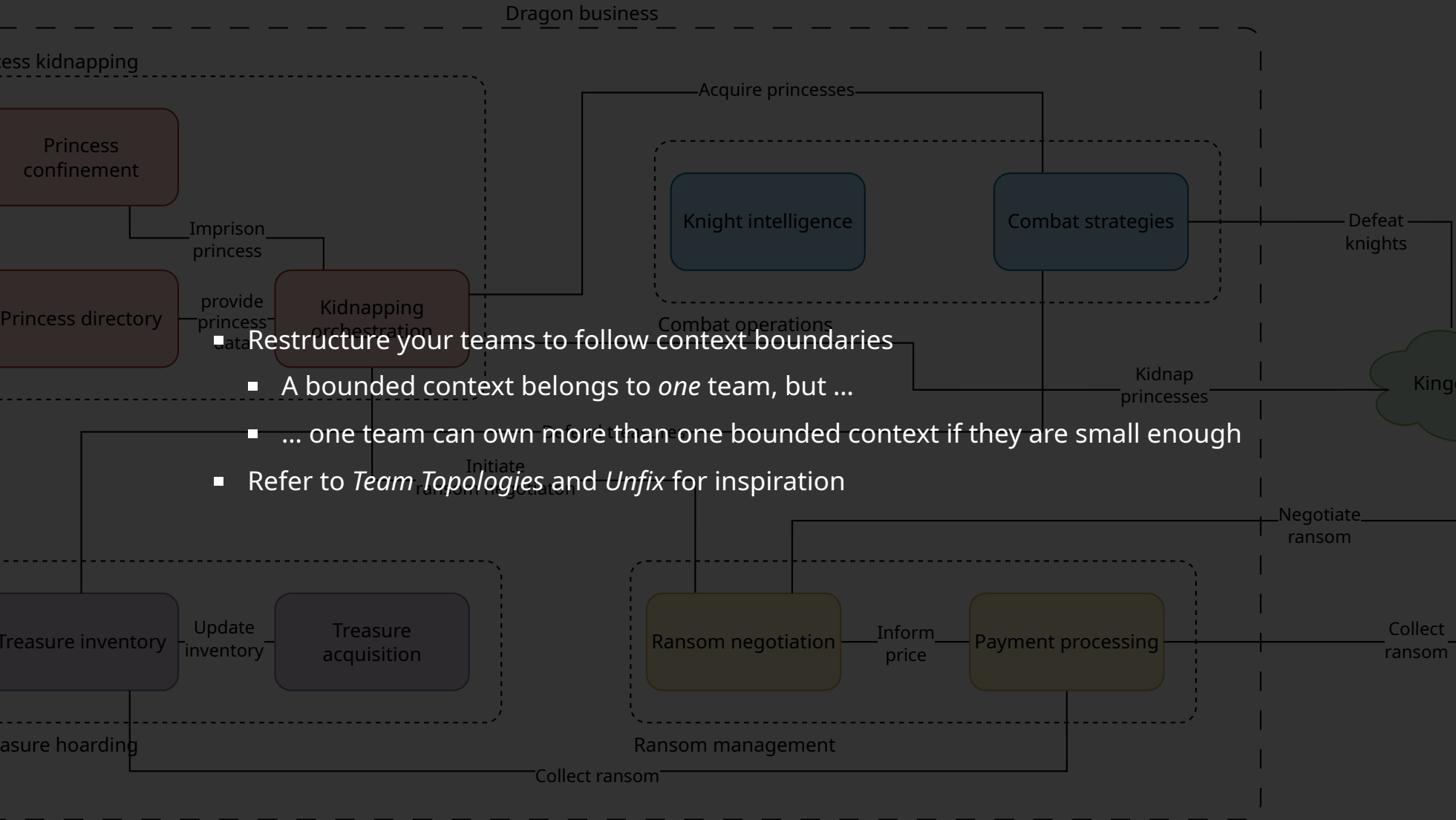
Single Page Application

Finance service

Combat service

Central databas


Dragon business
Princess kidnapping
Princess confinement
Imprison princess
Princess directory
provide princess data
Kidnapping orchestration
Acquire princesses
Knight intelligence
Combat strategies
Defeat knights
Combat operations
Kidnap princesses
Kingdom
Defend treasures
Initiate ransom negotiaton
Negotiate ransom
Treasure inventory
Update inventory
Treasure acquisition
Treasure hoarding
Ransom negotiation
Inform price
Payment processing
Collect ransom
Ransom management
Collect ransom



- Restructure your teams to follow context boundaries
  - A bounded context belongs to *one* team, but …
  - … one team can own more than one bounded context if they are small enough
- Refer to *Team Topologies* and *Unfix* for inspiration

Test-Driven Development

Ensuring you don’t break your business

TDD

1. Write a test
2. See it fail
3. Make the smallest change that can possibly make it pass
4. Refactor
5. Repeat

TDD is not vital to transformation, but proper tests are

How TDD helps with the Strangler Fig pattern

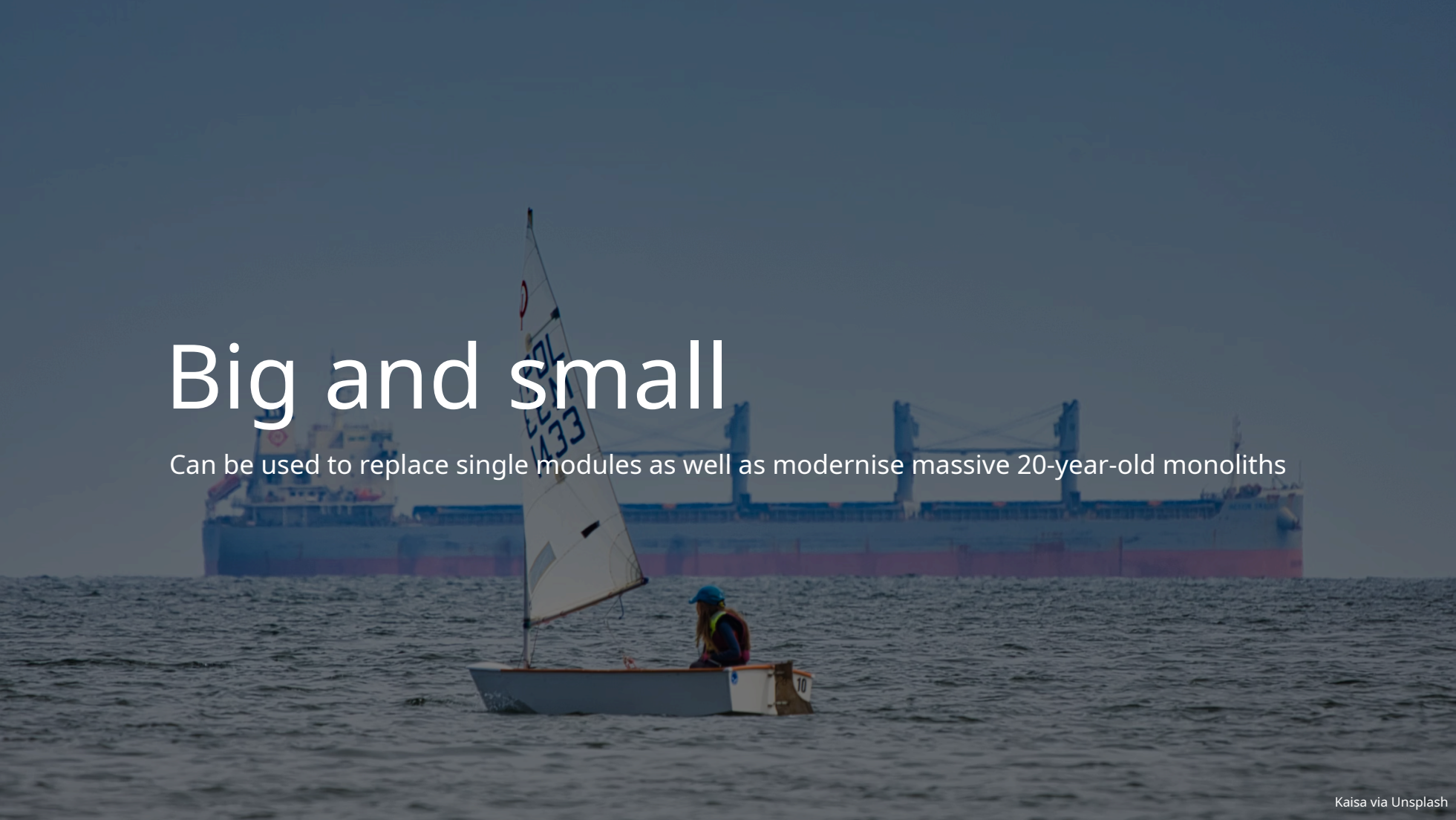
- Before making any changes to the existing system, fixate the behaviour so that we can ensure that we don’t change it
- If functionality is moved, move the tests along with the implementation
- Test-*driving* is optional, but is by far the best way of ensuring that the new system is properly tested and maintainable

Advantages over a rewrite

- Significantly lower risk as changes are small and incremental
- Low risk of repeating old mistakes
- System can keep evolving
  - Evolution can be a primary driver of the modernisation — i.e. modernise where changes are needed
  - New business opportunities are *great* drivers for evolution
- Second-system effect can be contained to the first few breakouts
- Can be paused at any time
  - A half-modernised system is still functional — a half-rewritten is not

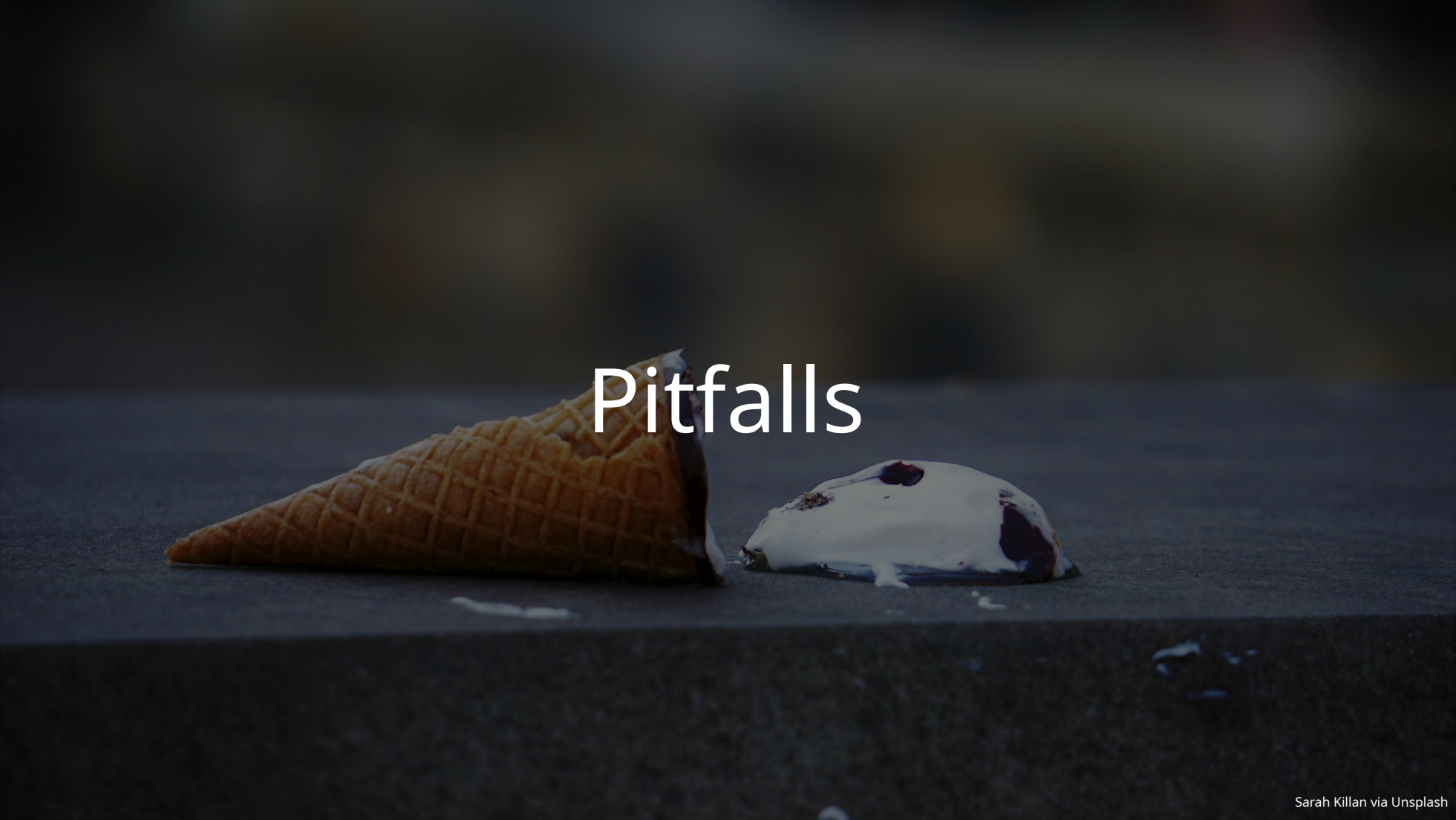
Big and small

Can be used to replace single modules as well as modernise massive 20-year-old monoliths



Kaisa via Unsplash

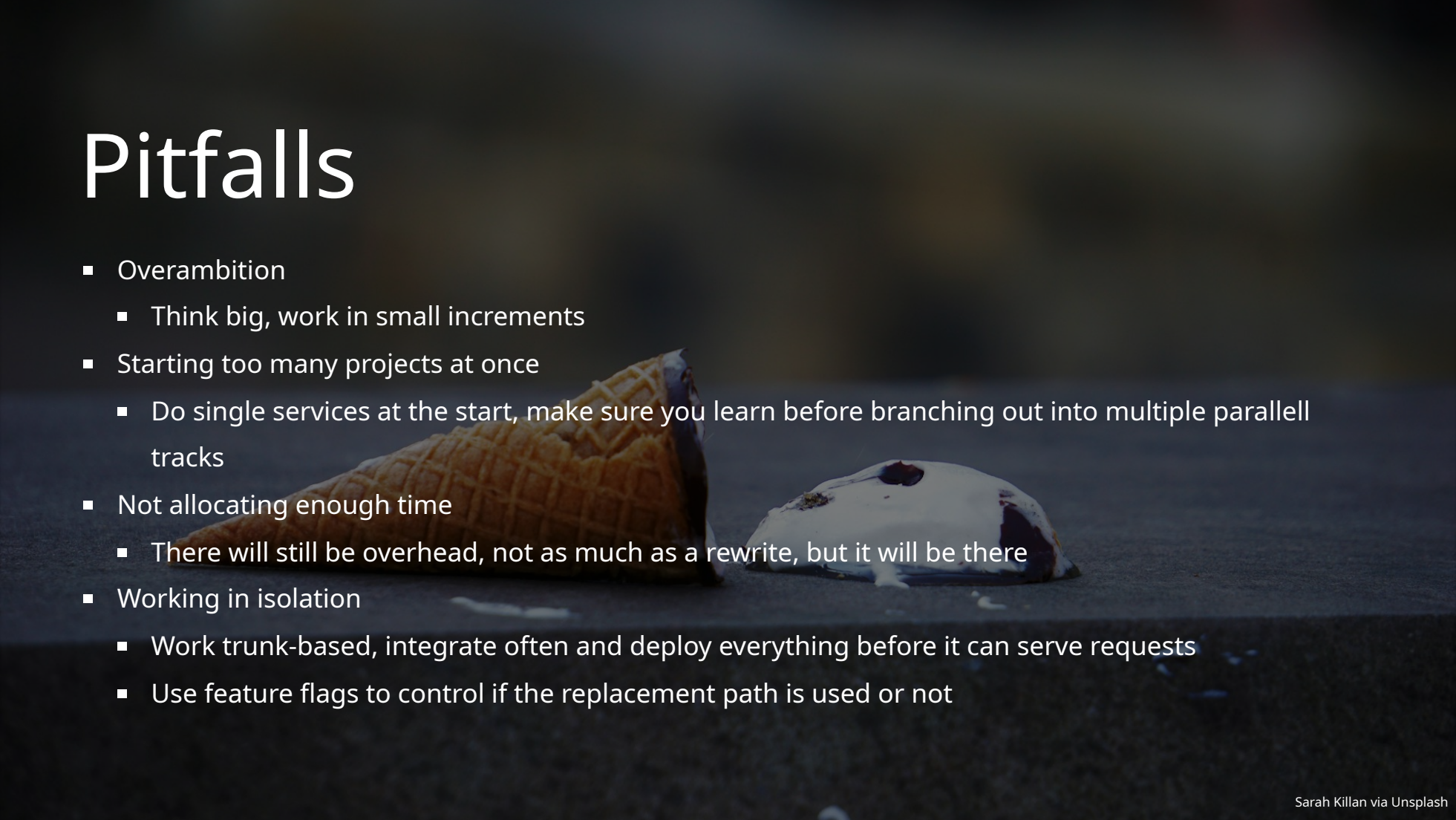
Pitfalls



Sarah Killan via Unsplash

Pitfalls

- Overambition
  - Think big, work in small increments
- Starting too many projects at once
  - Do single services at the start, make sure you learn before branching out into multiple parallell tracks
- Not allocating enough time
  - There will still be overhead, not as much as a rewrite, but it will be there
- Working in isolation
  - Work trunk-based, integrate often and deploy everything before it can serve requests
  - Use feature flags to control if the replacement path is used or not



Sarah Killan via Unsplash

How to start?

- Perform a Context mapping (DDD) to develop the architectural direction
- Create new team configurations (Team topologies/Unfix, Inverse Conway Manoeuvre) that conforms to the target architecture, **don’t** execute yet
- Select one (small) project to serve as a shining beacon
- Implement one tiny feature in the new project

Continued success

- Gradually add more projects and slowly transition both architecture and team structure towards the target
- Continuously reevaluate the target architecture and team configuration to ensure you’re still on the right path
- Don’t forget to celebrate!


factor10
Thanks!
Daniel Raniz Raneland
Coding Architect @ factor10
factor10.com
raniz@factor10.com
raniz.blog
raniz@mastodon.online
/in/raneland
Kelley Verdeck via Flicker; CC BY-ND 2.0


factor10
Need a hand?
Daniel Raniz Raneland
Coding Architect @ factor10
factor10.com
raniz@factor10.com
raniz.blog
raniz@mastodon.online
/in/raneland
Kelley Verdeck via Flicker; CC BY-ND 2.0