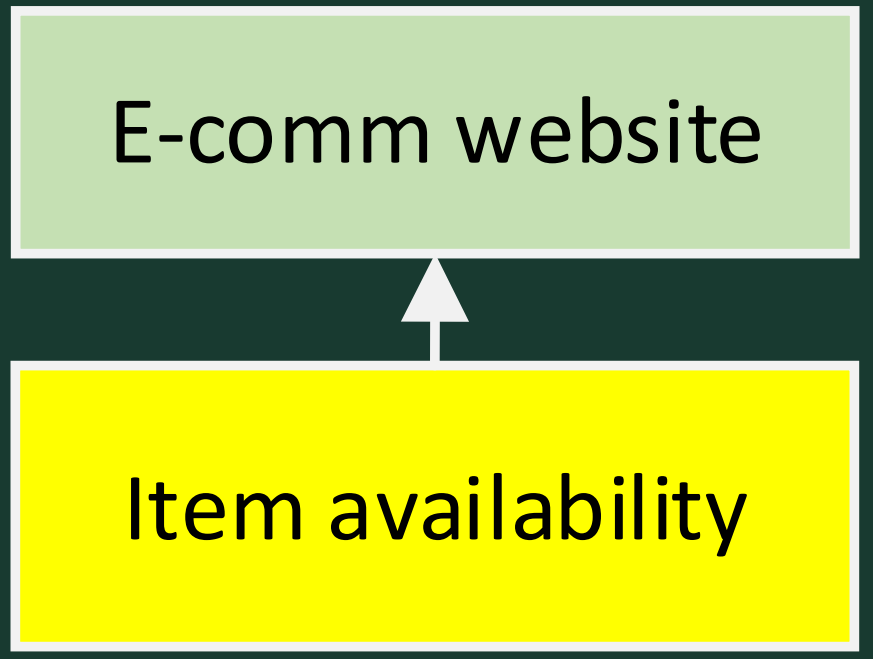
Fabulous Fortunes, Fewer Failures, and Faster Fixes from Functional Fundamentals

Scott Havens
Senior Director, Head of Supply Chain Technology
MODA OPERANDI


jet

Item availability

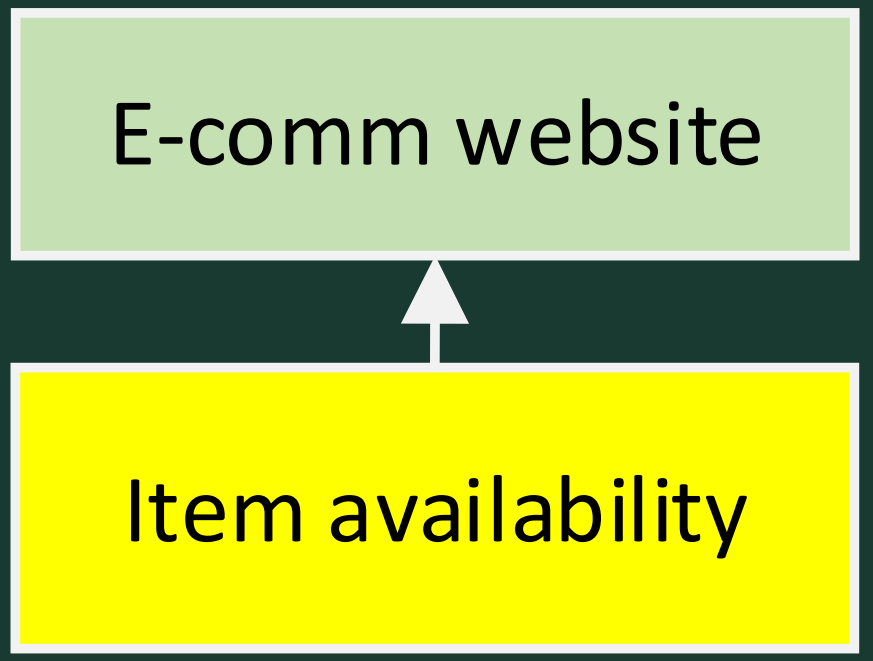
SLA: **99.98%** uptime @ **300ms**



Item availability

What affects item availability?

SLA: **99.98%** uptime @ **300ms**

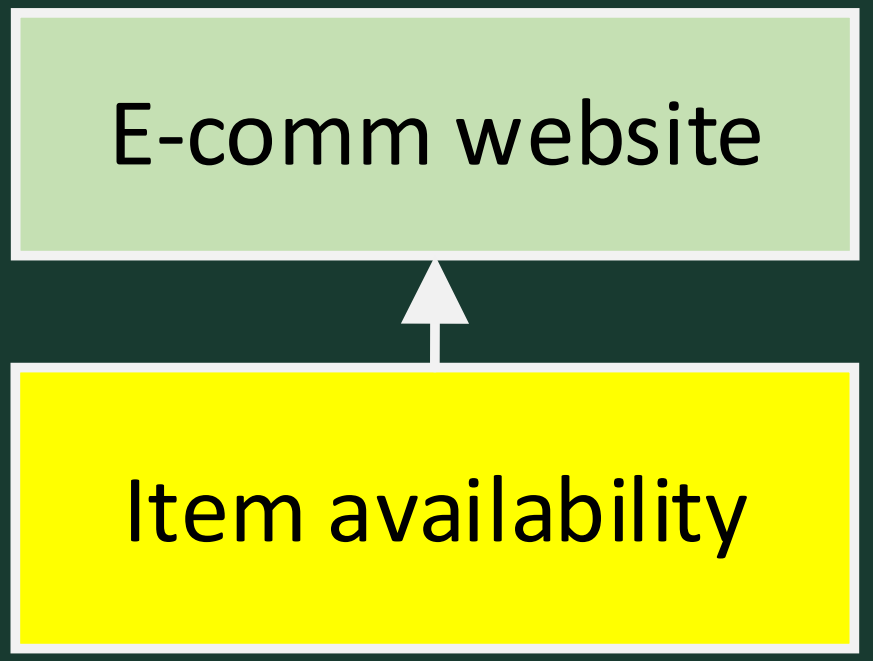


Item availability

What affects item availability?

- Warehouse inventory
- Reservations & orders

SLA: **99.98%** uptime @ **300ms**

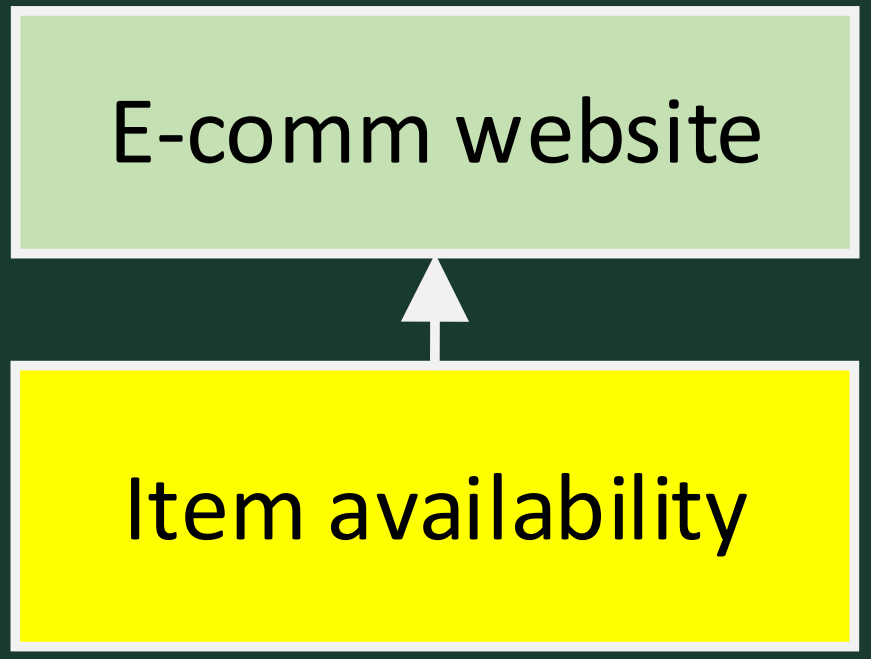


Item availability

What affects item availability?

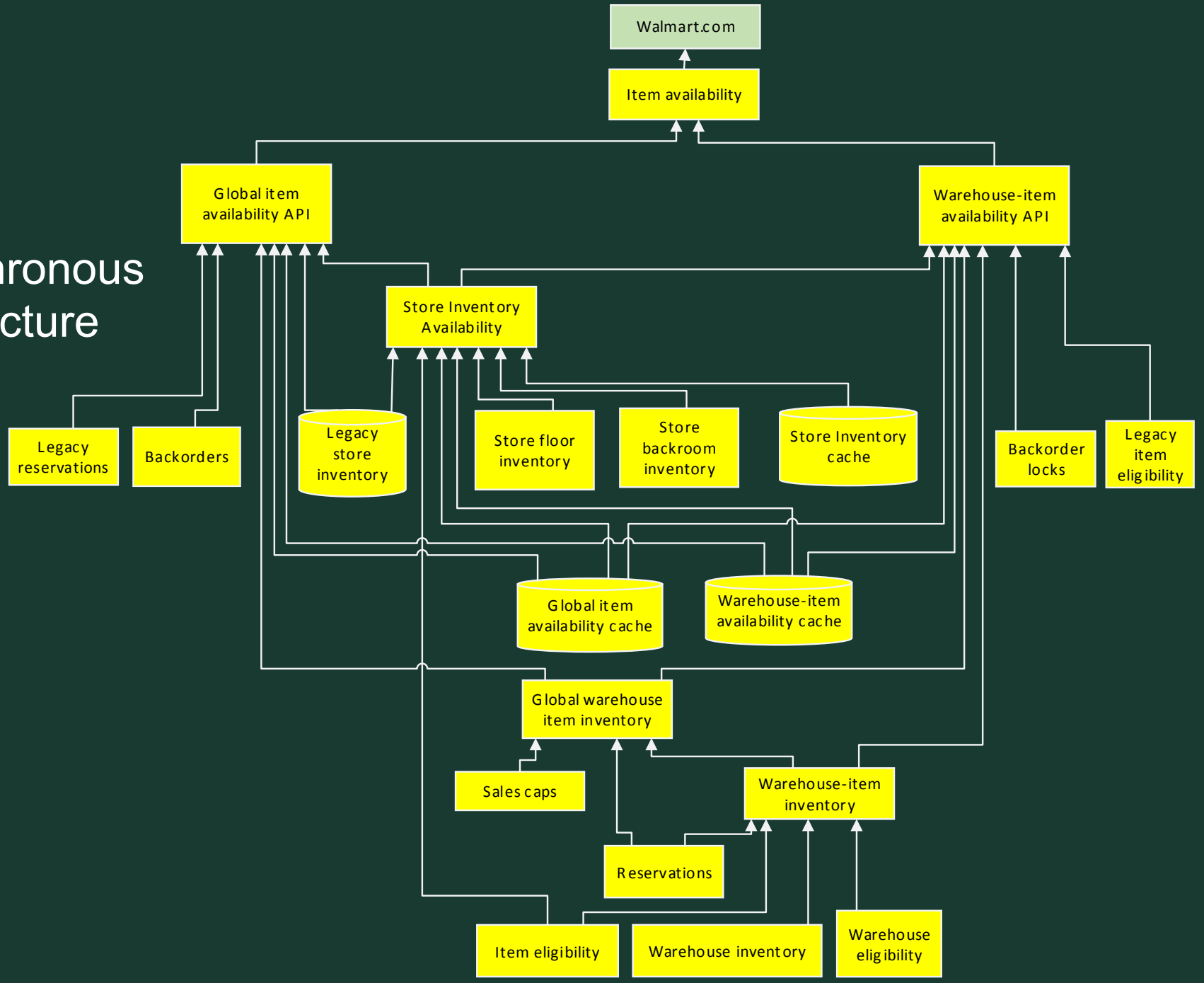
- Warehouse inventory
- Reservations & orders
- Store floor inventory
- Store backroom inventory
- 3rd parties' inventory
- Item eligibility
- Warehouse eligibility
- Sales caps
- Backorders
- Legacy systems

SLA: **99.98%** uptime @ **300ms**



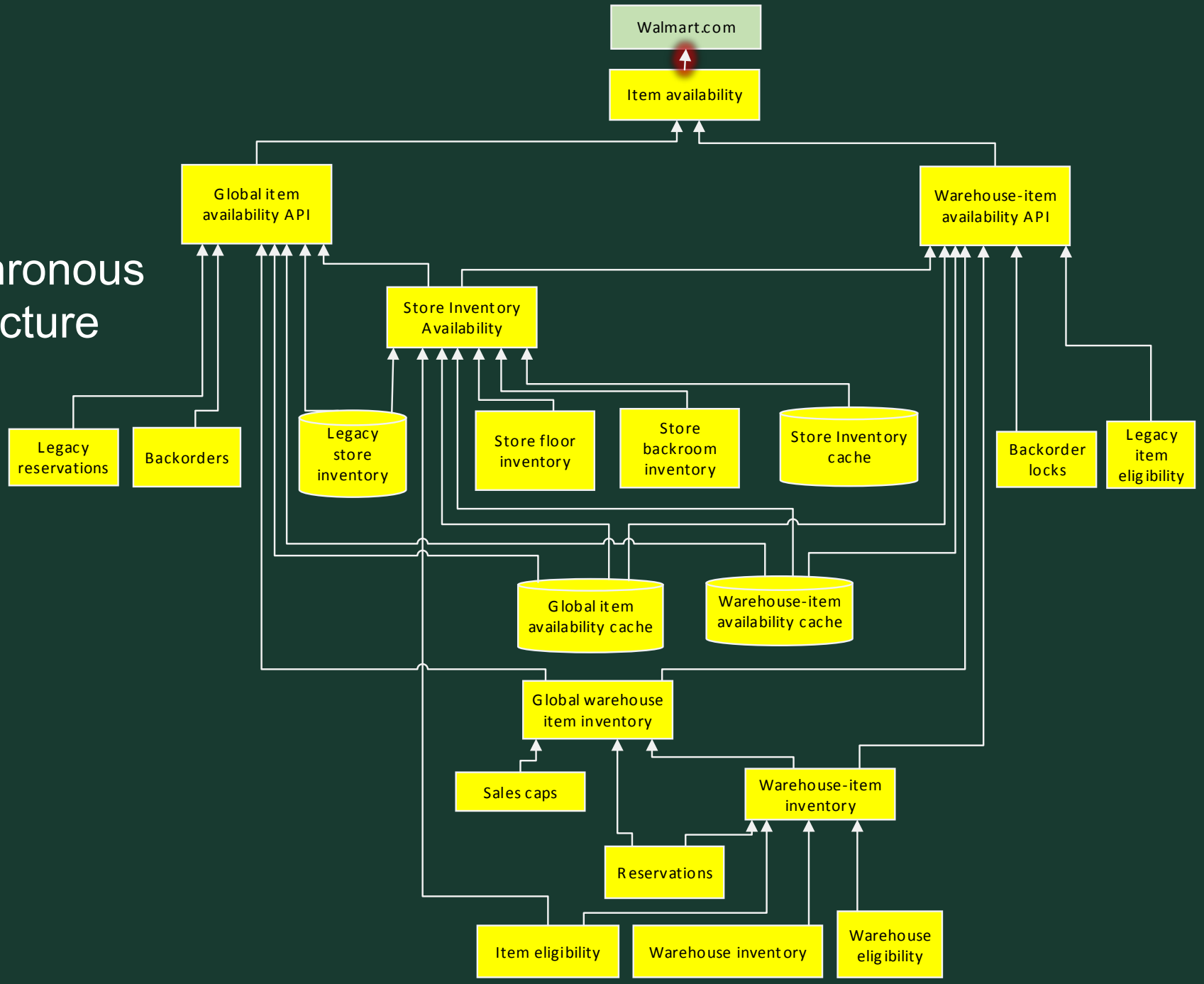
Item availability

Item availability via synchronous **Service-oriented** architecture



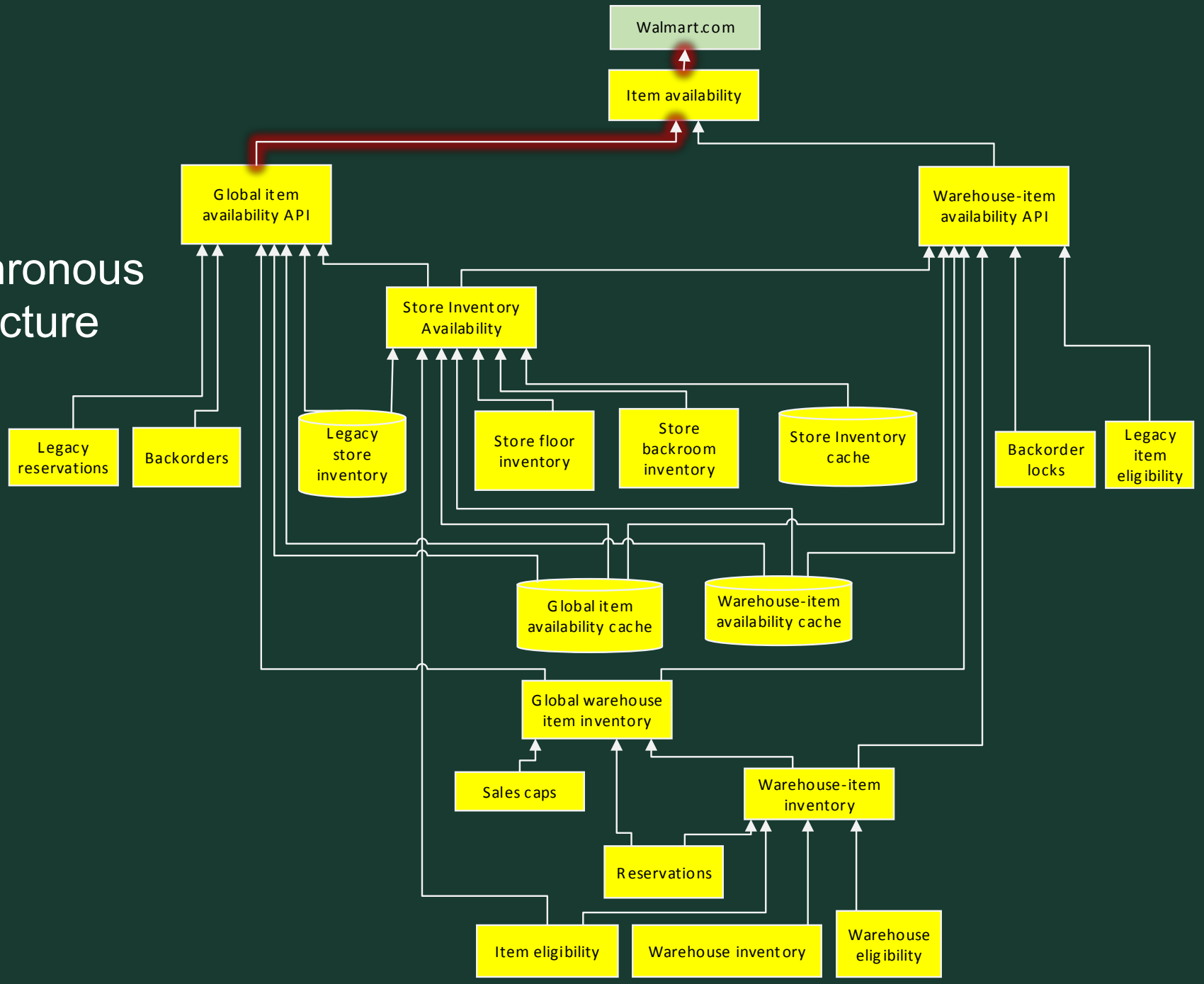
Item availability

Item availability via synchronous **Service-oriented** architecture



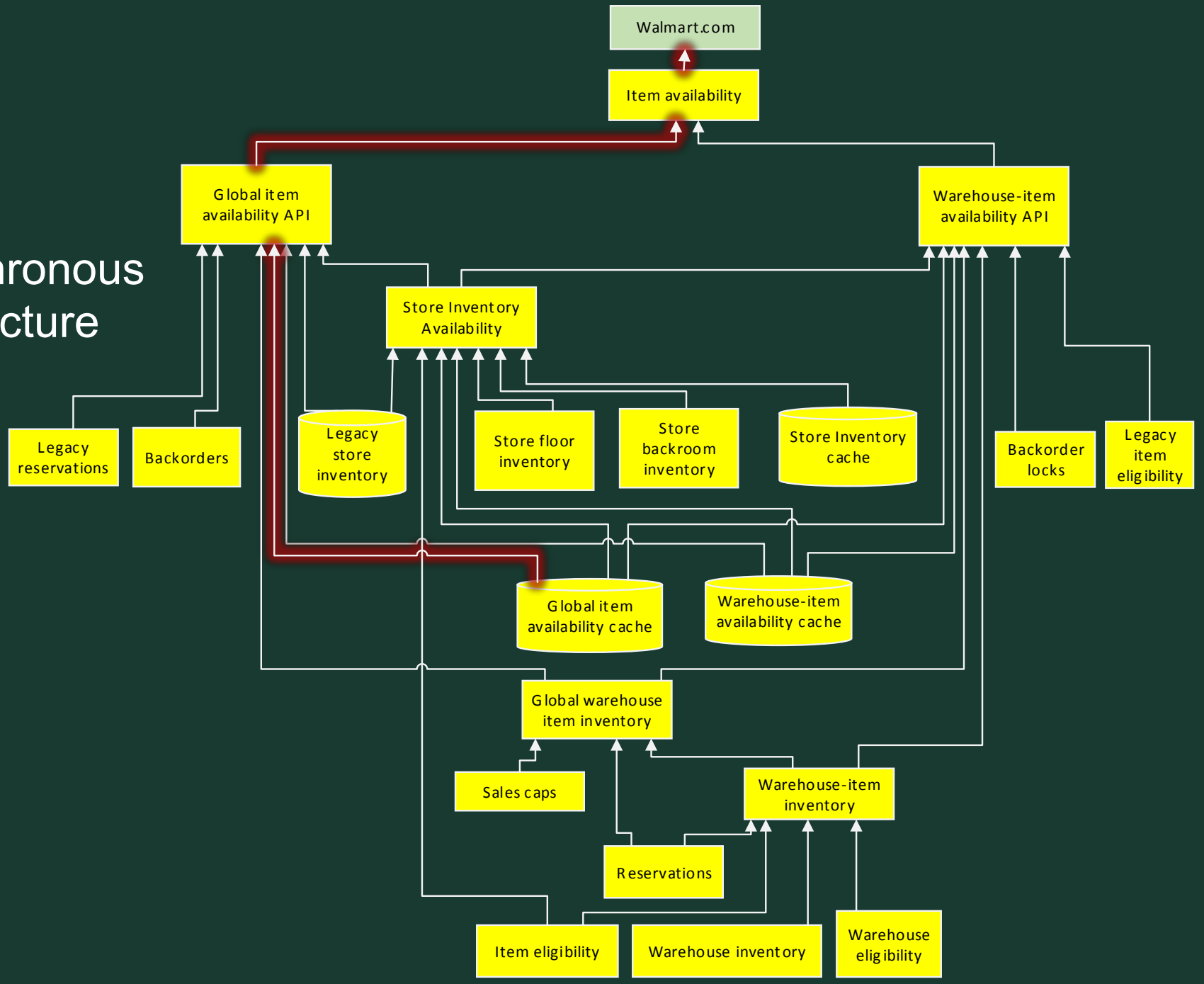
Item availability

Item availability via synchronous **Service-oriented** architecture



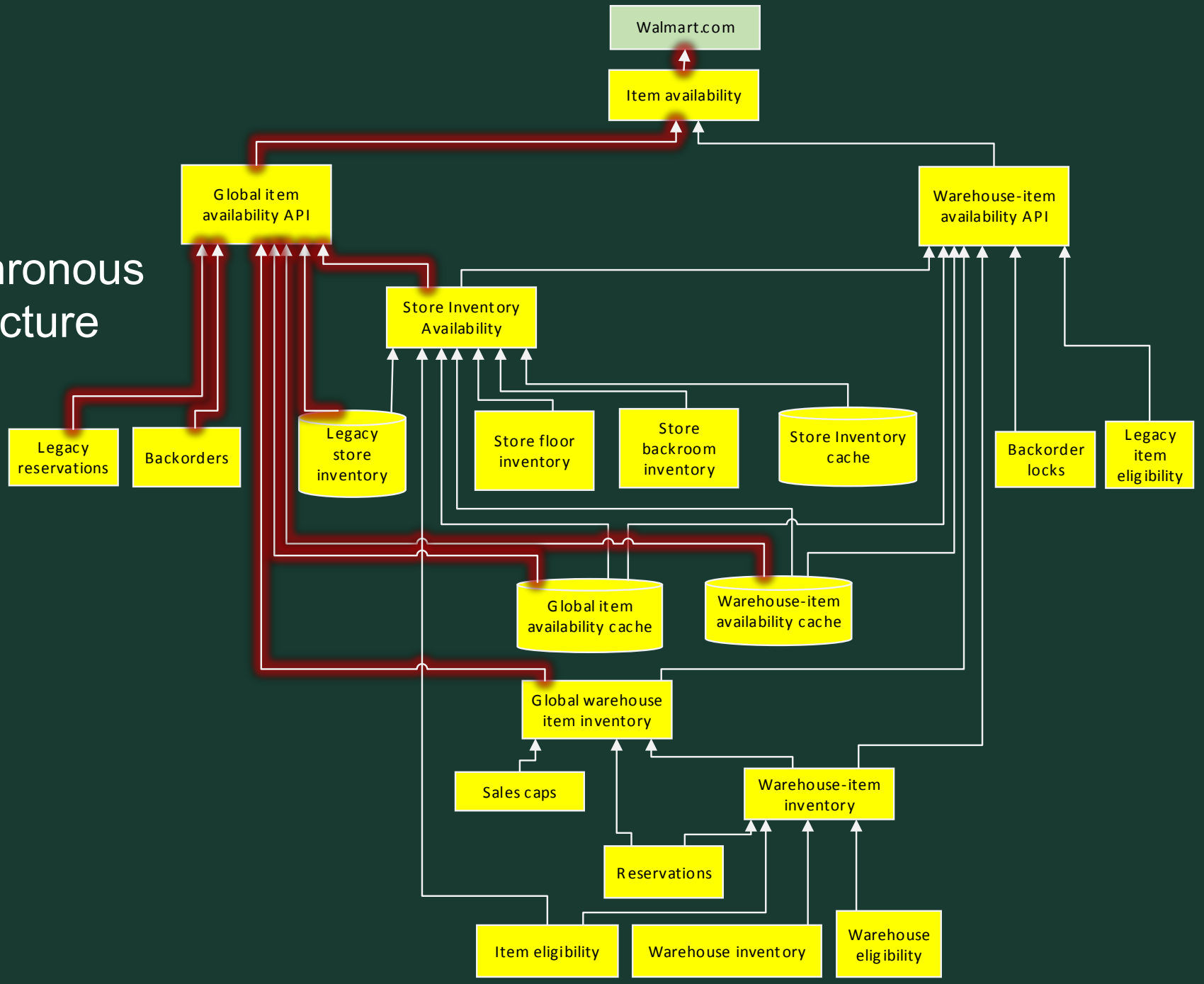
Item availability

Item availability via synchronous **Service-oriented** architecture



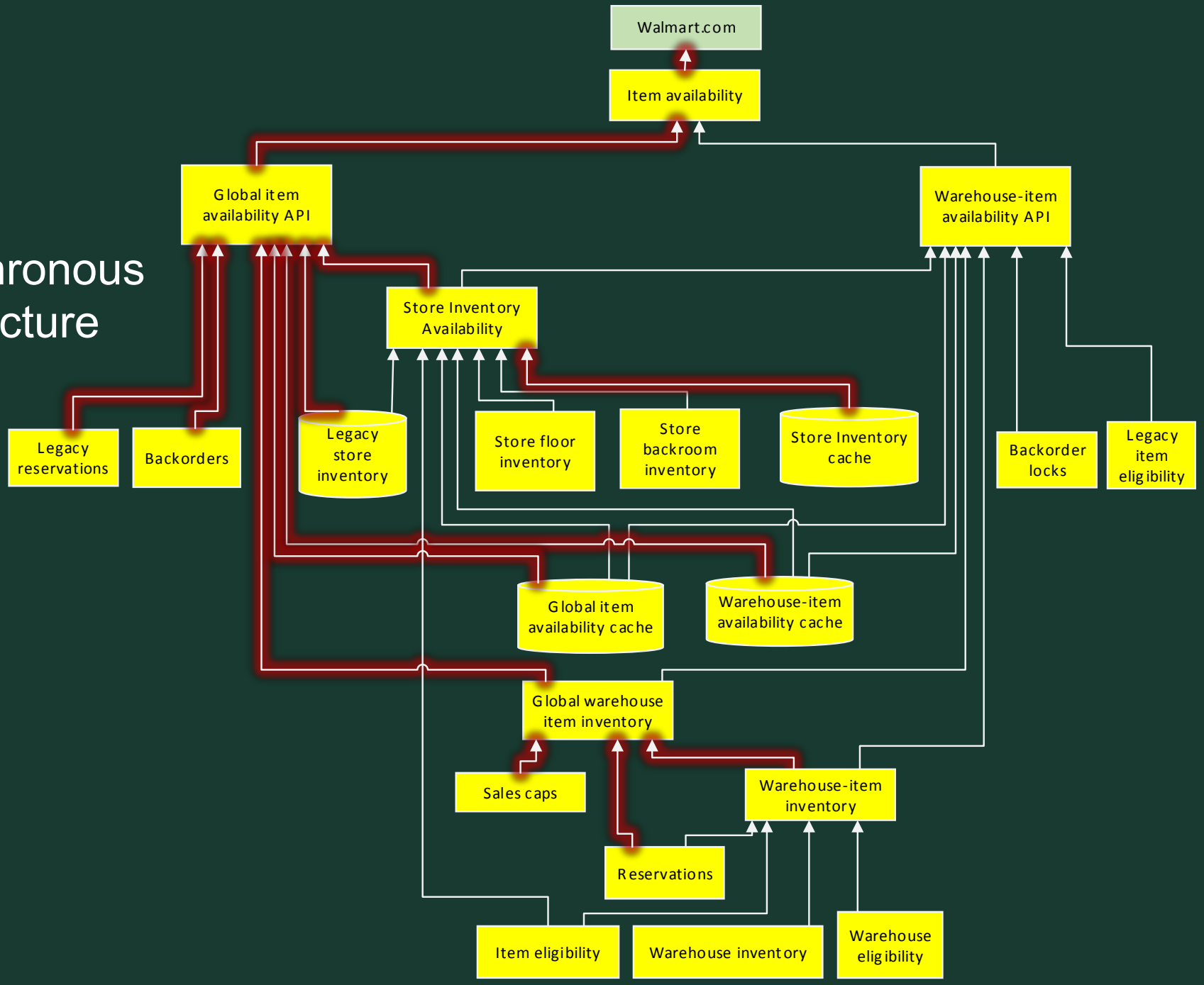
Item availability

Item availability via synchronous **Service-oriented** architecture



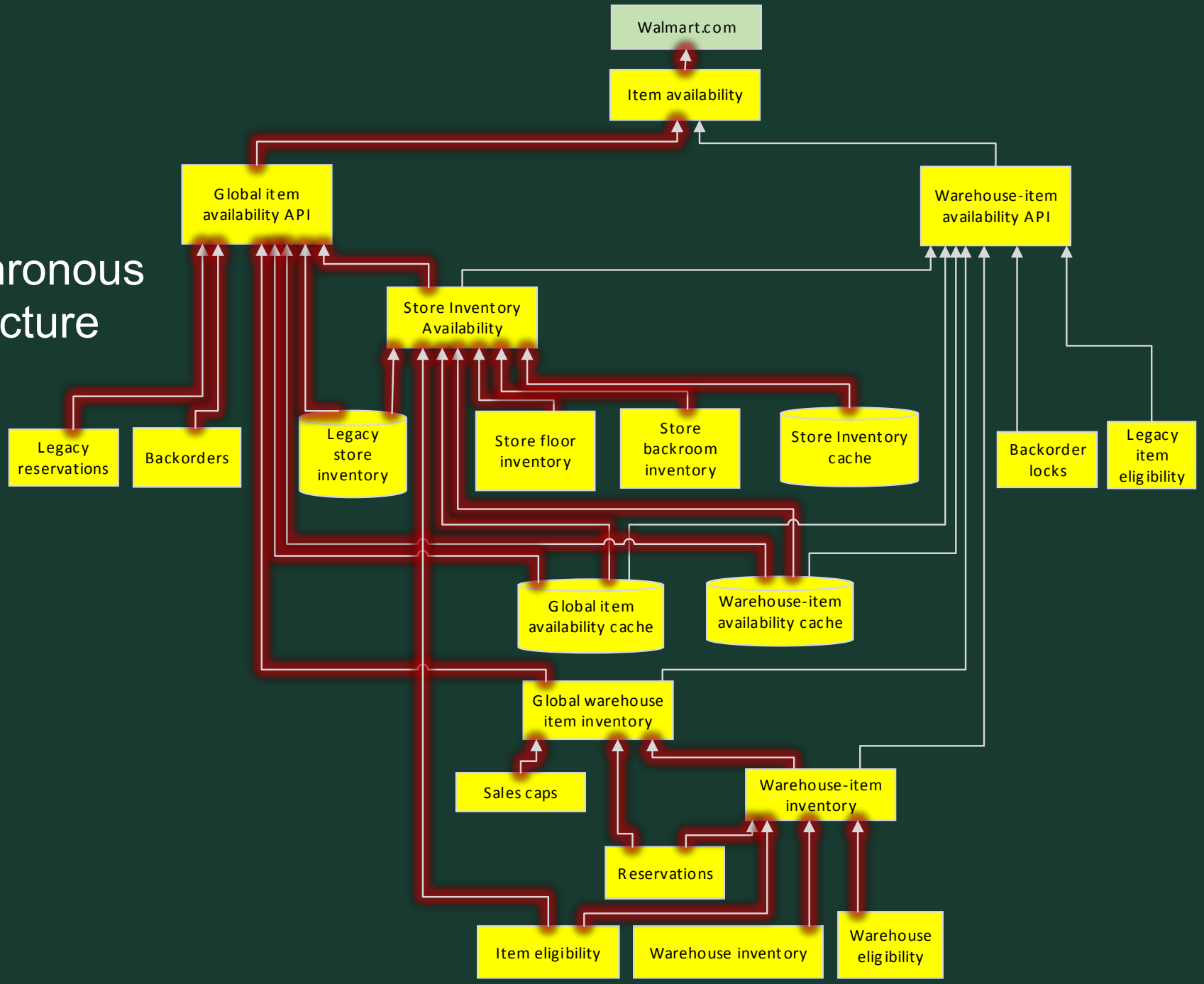
Item availability

Item availability via synchronous **Service-oriented** architecture



Item availability

Item availability via synchronous **Service-oriented** architecture



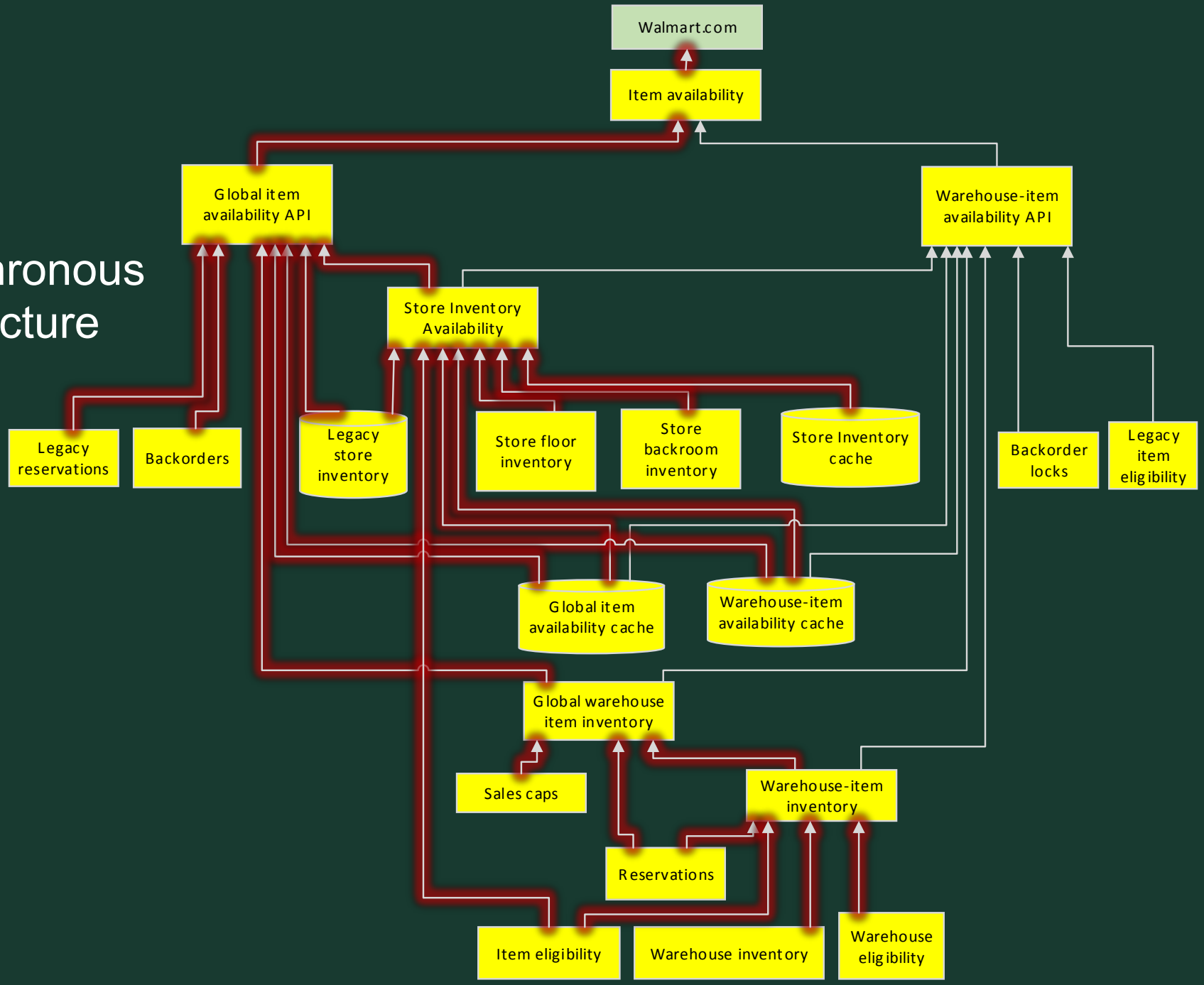
Item availability

Item availability via synchronous **Service-oriented** architecture

SLA: **99.98%** uptime @ **300ms**

requires

23 service calls
99.999% uptime
@ **50ms** mean processing SLO



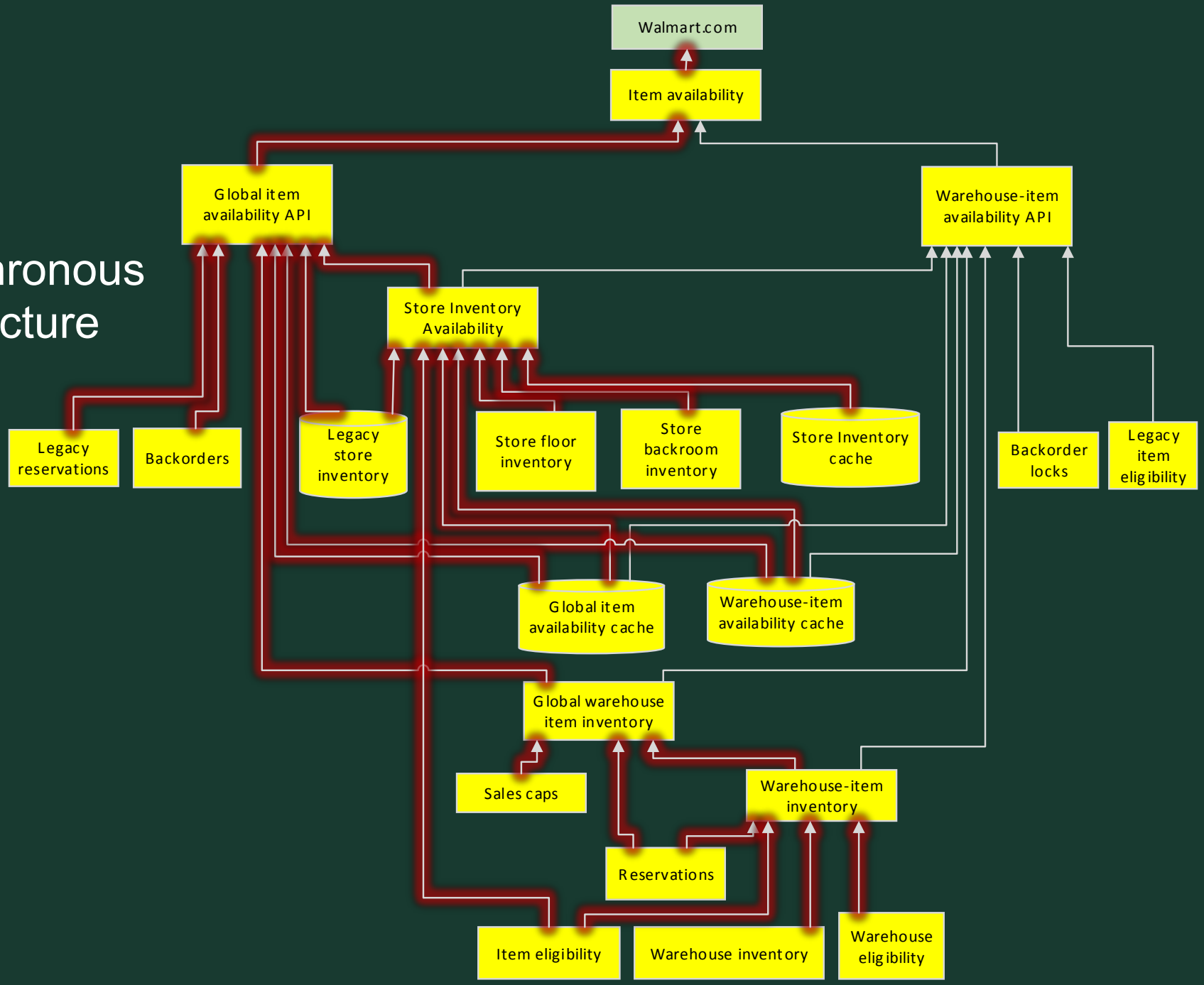
Item availability

Item availability via synchronous **Service-oriented** architecture

SLA: **99.98%** uptime @ **300ms**

requires

23 service calls
99.999% uptime
@ **50ms** mean processing SLO



How can we solve these problems?

How can we solve these problems?

Functional programming

Principles of functional programming

- Immutability

Principles of functional programming

- Immutability
- Purity

Principles of functional programming

- Immutability
- Purity
- Duality of code and data

What is designing functionally?

Immutability →

- Message-based, log-driven communication

What is designing functionally?

Immutability →

- Message-based, log-driven communication
- Event sourcing

What is designing functionally?

Immutability →

- Message-based, log-driven communication
- Event sourcing

Purity →

- Isolate computations from the real world

What is designing functionally?

Immutability →

- Message-based, log-driven communication
- Event sourcing

Purity →

- Isolate computations from the real world
- No dual writes – change data capture (CDC) instead

What is designing functionally?

Immutability →

- Message-based, log-driven communication
- Event sourcing

Purity →

- Isolate computations from the real world
- No dual writes – change data capture (CDC) instead

What is designing functionally?

Immutability →

- Message-based, log-driven communication
- Event sourcing

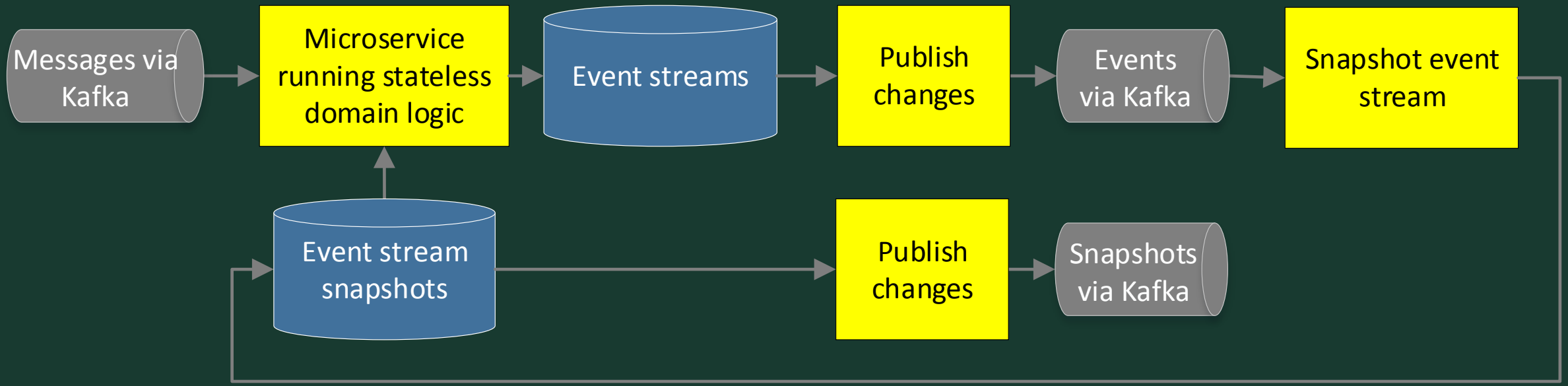
Purity →

- Isolate computations from the real world
- No dual writes – change data capture instead

Duality of code and data →

- Replace realtime compute with data lookup wherever feasible

What is designing functionally?



Panther

Inventory tracking

Reservation management

Panther functional goals

Maximize availability

Minimize reject rates

Improve customer experience

Enhance insights

Unify inventory mgmt

Panther nonfunctional goals

High availability

Georedundancy

SLA-backed performance

Panther results

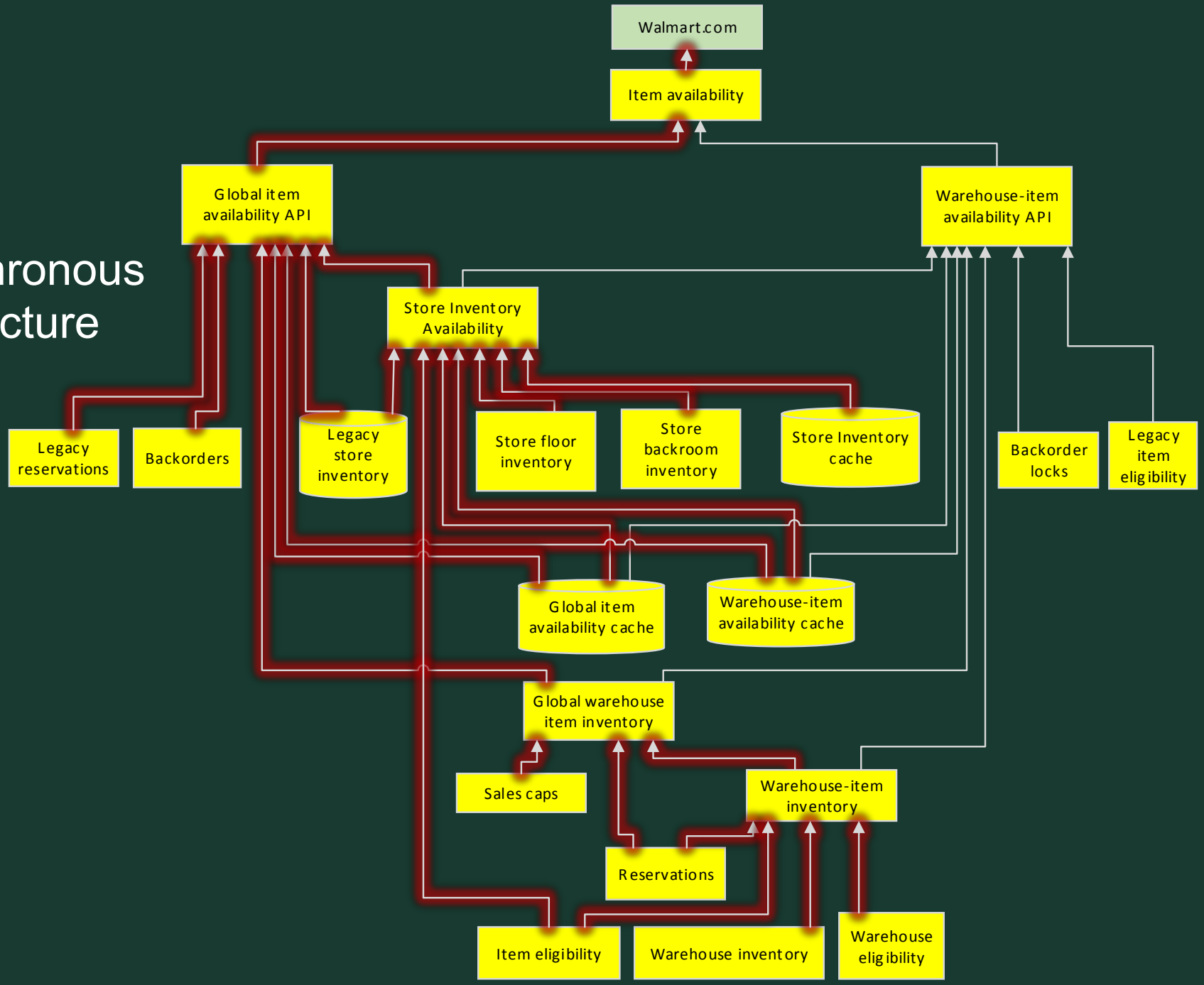
Time to initial delivery

Team size

Time to deliver features

Item Availability

Item availability via synchronous **Service-oriented** architecture



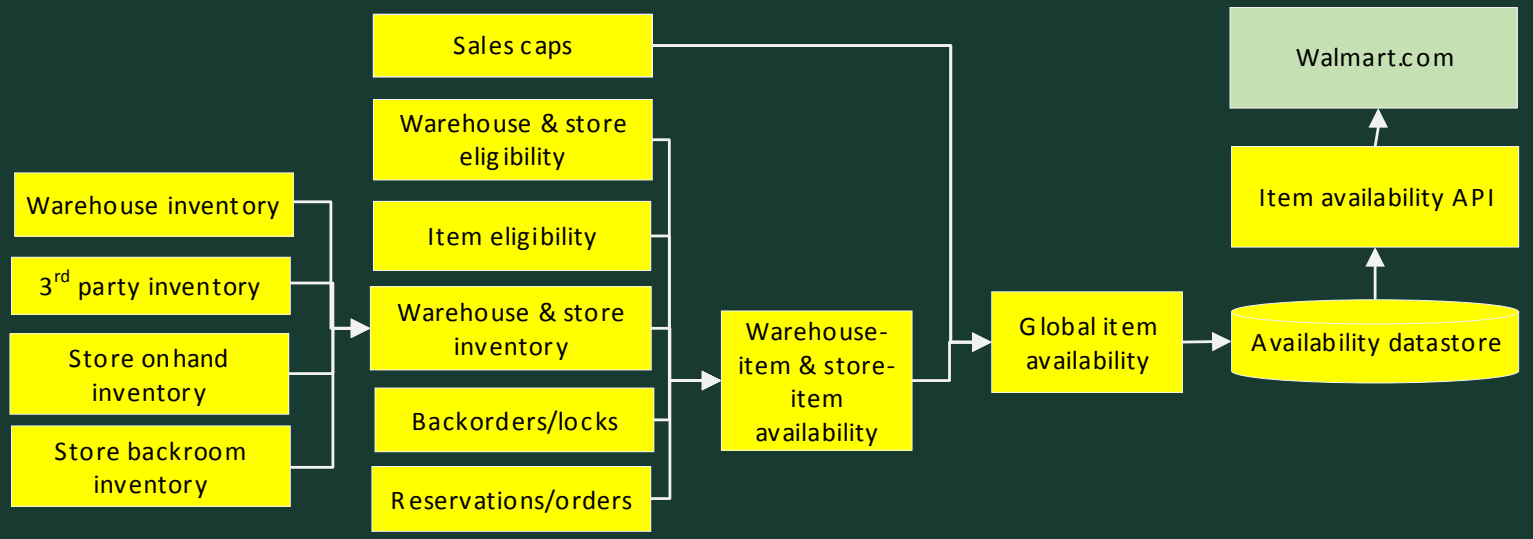
SLA: **99.98%** uptime @ **300ms**

requires

23 service calls
99.999% uptime
@ **50ms** mean processing SLO

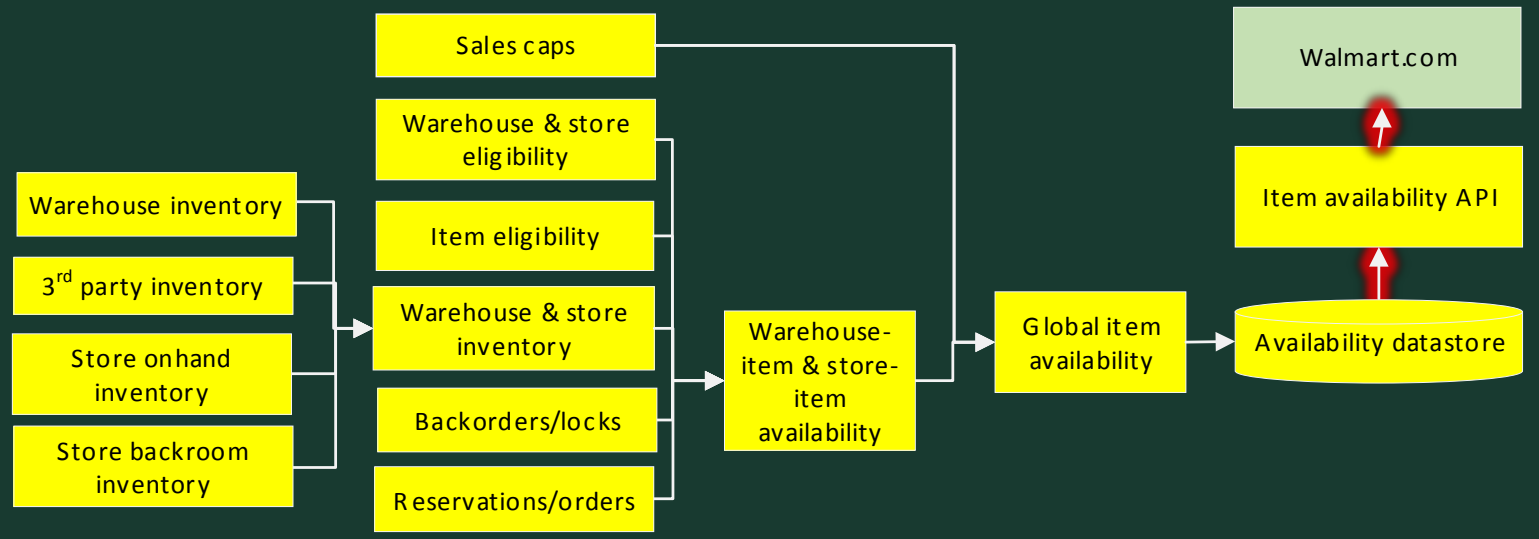
Item Availability

Item availability via
Event-driven architecture



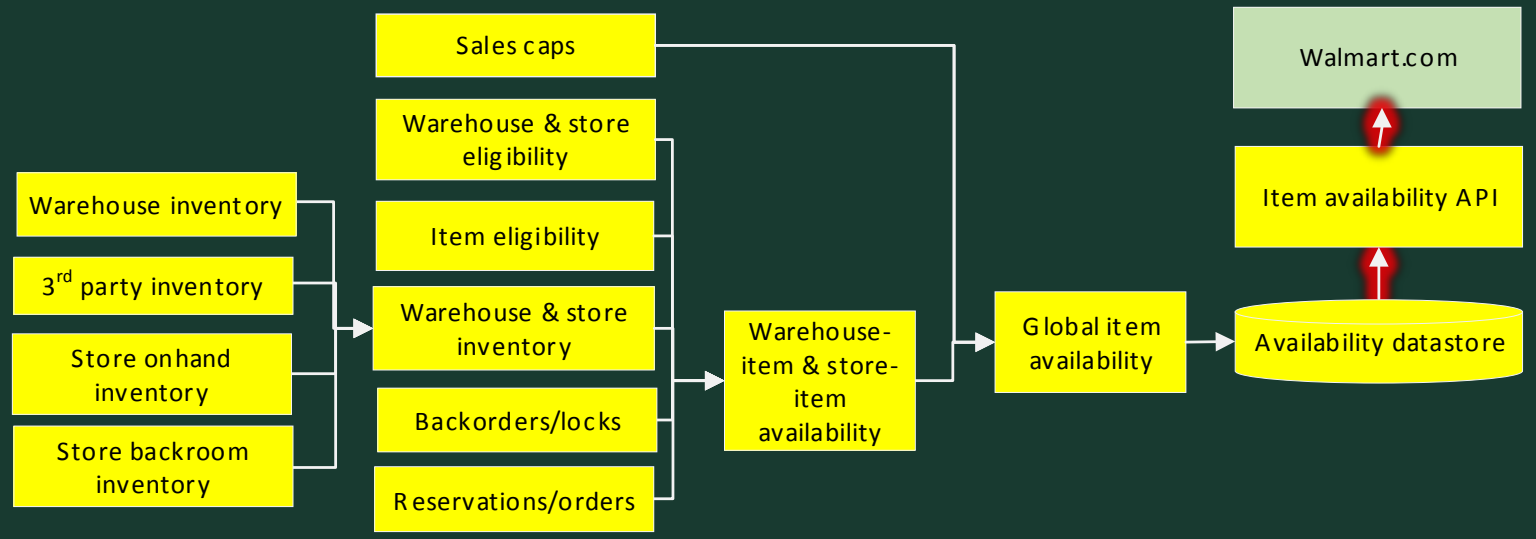
Item Availability

Item availability via
Event-driven architecture



Item Availability

Item availability via
Event-driven architecture



SLA: **99.98%** uptime @ **300ms**

requires

2 service calls
99.99% uptime
@ **150ms** mean processing SLO

Operational costs

Service uptime	Processing latency	Operational cost
99.9%	1 min	$

Operational costs

Service uptime	Processing latency	Operational cost
99.9%	1 min	$
99.99%	150 ms	

Operational costs

Service uptime	Processing latency	Operational cost
99.9%	1 min	$
99.99%	150 ms	$$$$$$$$$$

Operational costs

Service uptime	Processing latency	Operational cost
99.9%	1 min	$
99.99%	150 ms	$$$$$$$$$$
99.999%	50 ms	

Operational costs

Service uptime	Processing latency	Operational cost
99.9%	1 min	$
99.99%	150 ms	$$$$$$$$$$
99.999%	50 ms	$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$

Operational costs

Item availability via
Event-driven architecture

Item availability via synchronous
Service-oriented architecture

Operational costs

Item availability via
Event-driven architecture

$$$$$$$

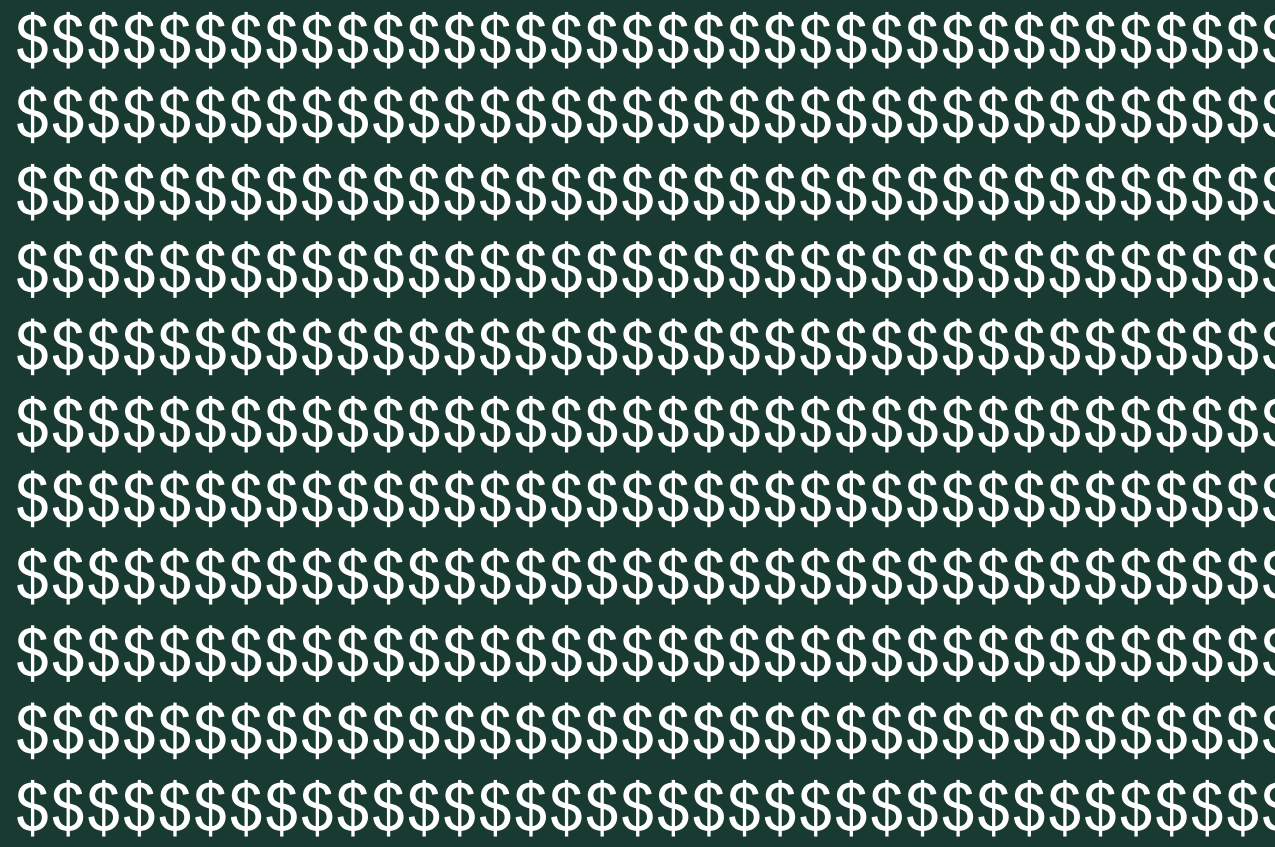
Item availability via synchronous
Service-oriented architecture

Operational costs

Item availability via
Event-driven architecture

$$$$$$$

Item availability via synchronous
Service-oriented architecture



“I object!”

“I object!

…My dev team isn’t skilled enough!”

“I object!

…We don’t have the technology to do this!”

“I object!

…It’ll make my app too complex!”

Gene testimonial on decoupling components via pub/sub

“I asked Scott Havens for his advice on how I decouple a bunch of components that write to a database through a terrible Ruby/ ActiveRecord app I wrote. He told me to move it to a event sourcing with pub/sub pattern.

At first, I couldn’t quite believe this was practical. Pub/sub for such a simple app?

But I was blown away by the results. Suddenly, two components that used to be tightly coupled together, with all sorts of weird failure modes where things wouldn’t be written out correctly just disappeared.

In my very simple case, I managed to reduce code size by 90%, and delete the portions written in Python and Ruby. All that remains is a small portion written in Clojure.”

“I object!

…It’ll cost too much!”

“I object!

…It’ll take too long!”

“I object!

…It’s too dangerous!”

“It’s just too much!”

What can you do?

What can you do?

Eliminate one dual write

What can you do?

Eliminate one dual write

Try property based testing

What can you do?

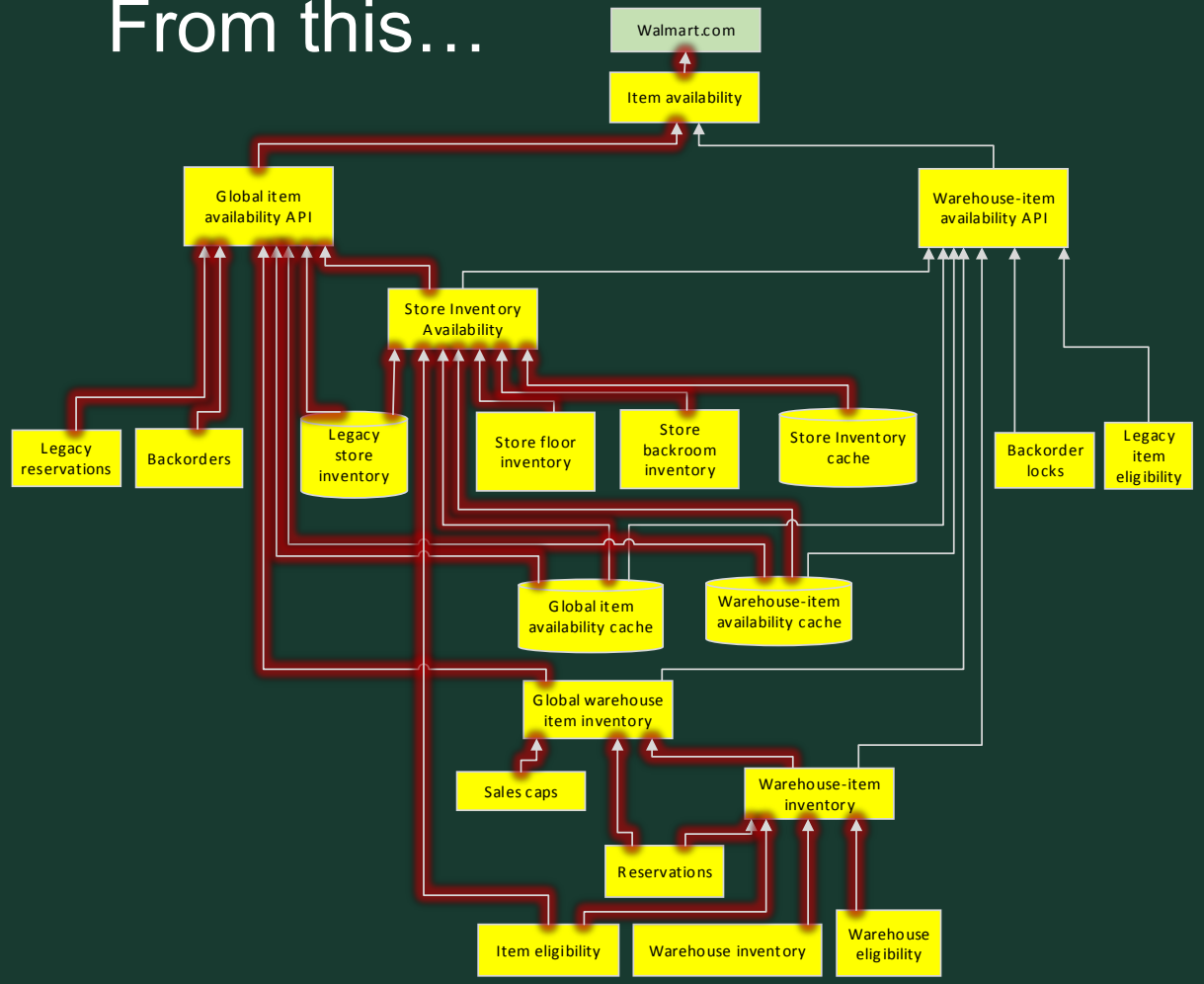
Eliminate one dual write

Try property based testing

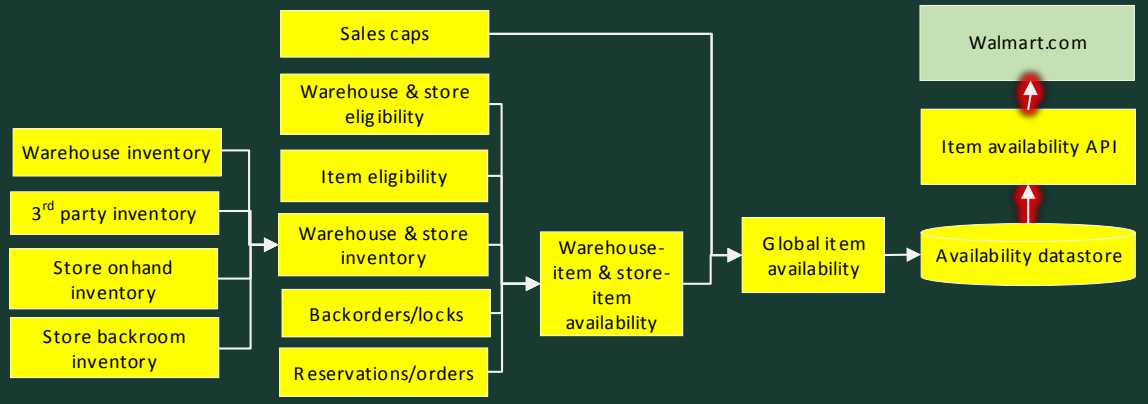
Switch one web service to also publish events

What can you do?

From this…



…to this



Contact me!

scott.havens@modaoperandi.com

@ScottHavens

Contact me!

scott.havens@modaoperandi.com
@ScottHavens

We’re hiring!