Dependability through Redundancy

Prof. George Candea
School of Computer & Communication Sciences

How to achieve dependability?

- Use modularity ...
- ... and REDUNDANCY for ...
  - *fault tolerance*
  - *high reliability*
  - *high availability*

How to achieve dependability?

- Use modularity ...
- ... and REDUNDANCY for ...
  - *fault tolerance*
  - *high reliability*
  - *high availability*

Redundancy = duplication with the purpose of increasing dependability

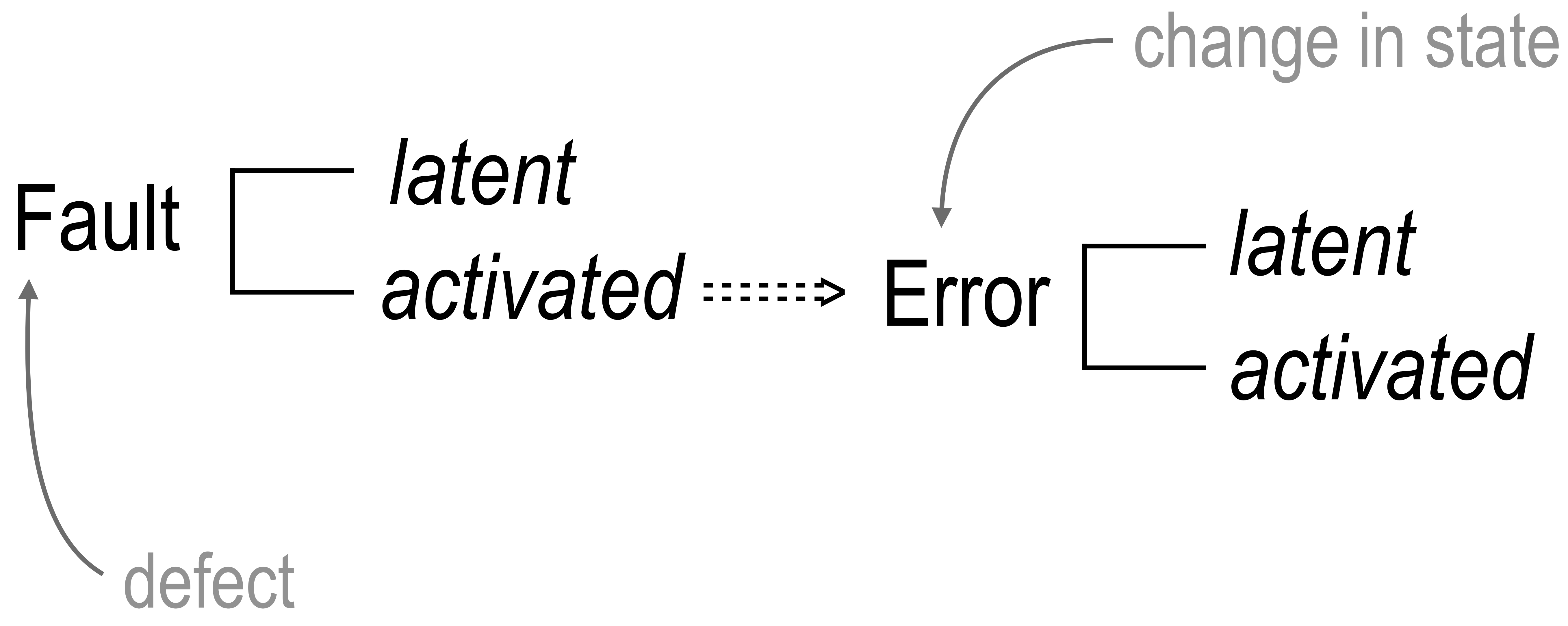
Fault tolerance

Fault tolerance

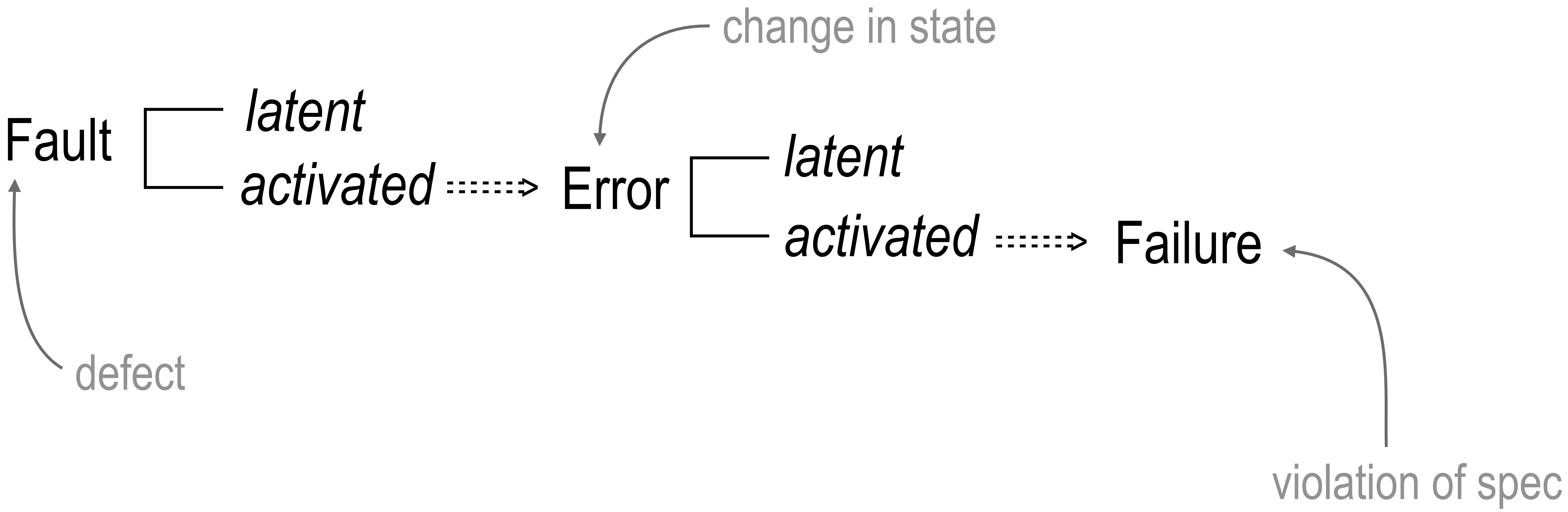
Fault
- *latent*
- *activated*

defect

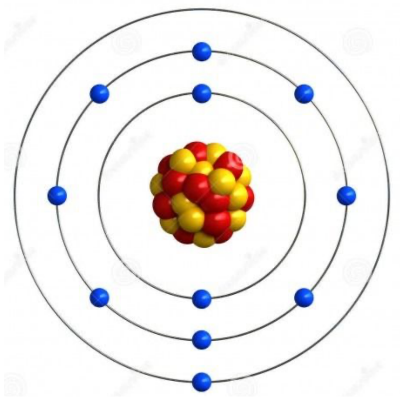
Fault tolerance



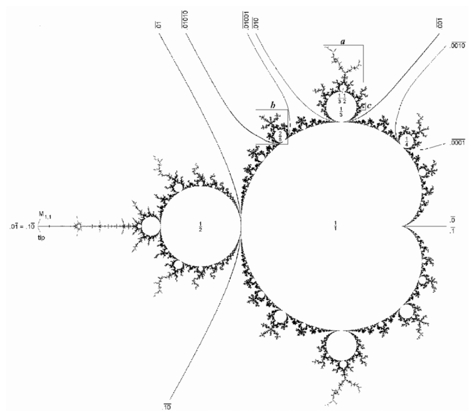
Fault tolerance



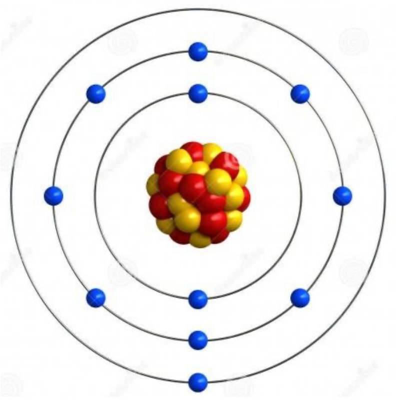
Types of software faults / defects



$$\Delta\chi\Delta\rho \geq \frac{\hbar}{2}$$



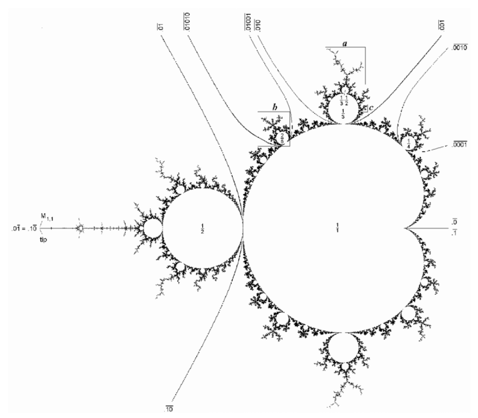
Types of software faults / defects



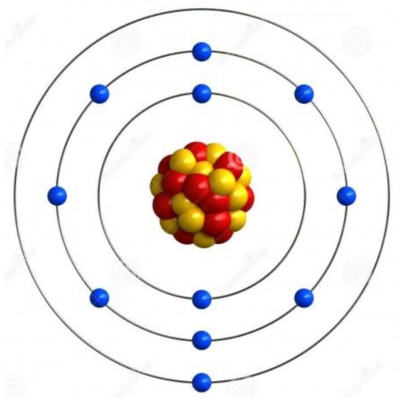
- Bohrbug
  - *clear + easy to reproduce => easy to fix*

$$\Delta\chi\Delta\rho \geq \frac{\hbar}{2}$$

- Heisenbug
  - *disappears when you attach with debugger*



Types of software faults / defects

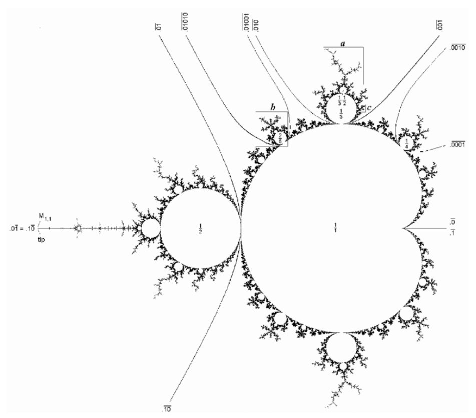


- Bohrbug
  - *clear + easy to reproduce => easy to fix*

$$\Delta\chi\Delta\rho \geq \frac{\hbar}{2}$$

- Heisenbug
  - *disappears when you attach with debugger*

- Schrödingbug
  - *starts causing failure once you realize it should*



- Mandelbug
  - *complex, obscure, chaotic, seemingly non-deterministic*

Using redundancy to tolerate faults

- "Tolerate faults" = cope with errors or the resulting failures
  - *the actual goal is to tolerate the consequences of faults*

Using redundancy to tolerate faults

- "Tolerate faults" = cope with errors or the resulting failures
  - *the actual goal is to tolerate the consequences of faults*
- Redundancy to cope with errors
  - *forward error correction*
  - *redundant copies/replicas (=coarse-grained ECC)*
  - *...*

Data/information redundancy

Geographic redundancy

Using redundancy to tolerate faults

- "Tolerate faults" = cope with errors or the resulting failures
  - *the actual goal is to tolerate the consequences of faults*
- Redundancy to cope with errors
  - *forward error correction*
  - *redundant copies/replicas (=coarse-grained ECC)*
  - *...*
- Redundancy to cope with failures
  - *server/service failover*
  - *Internet routing*
  - *…*

Data/information redundancy

Geographic redundancy

Processing redundancy
- Space
- Time

Functional redundancy


https://www.submarinecablemap.com
Arctic Ocean
Greenland Sea
Baffin Bay
Barents Sea
Kara Sea
East Siberian Sea
Norwegian Sea
Northwestern Passages
Greenland
Iceland
Sweden
Finland
Norway
Canada
Hudson Bay
Labrador Sea
Russia
Poland
Belarus
Germany
Ukraine
France
Romania
Kazakhstan
Mongolia
Sea of Okhotsk
Bering Sea
United States
North Atlantic Ocean
Spain
Italy
Turkey
Uzbekistan
Turkmenistan
China
Sea of Japan
South Korea
Morocco
Algeria
Libya
Egypt
Syria
Iraq
Iran
Afghanistan
Pakistan
Nepal
India
Myanmar (Burma)
Thailand
Saudi Arabia
Oman
Yemen
Gulf of Mexico
Mexico
Mauritania
Mali
Niger
Chad
Sudan
Ethiopia
Nigeria
South Sudan
Somalia
Kenya
Arabian Sea
Venezuela
Colombia
Suriname
DRC
Tanzania
Brazil
Peru
Bolivia
Paraguay
Chile
Argentina
Uruguay
Angola
Zambia
Namibia
Zimbabwe
Botswana
Madagascar
South Africa
Indian Ocean
South Atlantic Ocean
South Pacific Ocean
Australia
Great Australian Bight
Tasman Sea
Coral Sea
Banda Sea
Philippine Sea


https://www.submarinecablemap.com
Arctic Ocean
Greenland Sea
Baffin Bay
Barents Sea
Kara Sea
East Siberian Sea
Norwegian Sea
Northwestern Passages
Greenland
Iceland
Sweden
Finland
Norway
Canada
Hudson Bay
Labrador Sea
Russia
Bering Sea
Sea of Okhotsk
Poland
Belarus
Germany
Ukraine
Kazakhstan
Mongolia
France
Romania
Uzbekistan
Turkmenistan
Spain
Italy
Turkey
United States
North Atlantic Ocean
Syria
Iraq
Iran
Afghanistan
Pakistan
China
Sea of Japan
South Korea
Morocco
Algeria
Libya
Egypt
Saudi Arabia
Nepal
India
Myanmar (Burma)
Mexico
Gulf of Mexico
Mauritania
Mali
Niger
Chad
Sudan
Oman
Yemen
Thailand
Philippine Sea
Arabian Sea
Nigeria
South Sudan
Ethiopia
Venezuela
Colombia
Suriname
Somalia
Kenya
DRC
Tanzania
Peru
Brazil
Angola
Zambia
Bolivia
Namibia
Zimbabwe
Madagascar
Indian Ocean
Paraguay
Botswana
Chile
South Africa
Australia
Coral Sea
South Pacific Ocean
South Atlantic Ocean
Uruguay
Argentina
Great Australian Bight
Tasman Sea
New Zealand


https://www.submarinecablemap.com
Arctic Ocean
Greenland Sea
Baffin Bay
Barents Sea
Kara Sea
East Siberian Sea
Norwegian Sea
Northwestern Passages
Greenland
Iceland
Canada
Hudson Bay
Labrador Sea
Sweden
Finland
Norway
Russia
Bering Sea
Sea of Okhotsk
Poland
Belarus
Germany
Ukraine
France
Austria
Romania
Kazakhstan
Mongolia
United States
North Atlantic Ocean
Spain
Italy
Uzbekistan
Kyrgyzstan
Turkmenistan
Turkey
Syria
Iraq
Iran
Afghanistan
Pakistan
Nepal
China
Sea of Japan
South Korea
Morocco
Tunisia
Algeria
Libya
Egypt
Saudi Arabia
India
Myanmar (Burma)
Thailand
Mexico
Gulf of Mexico
Western Sahara
Mauritania
Mali
Niger
Chad
Sudan
Oman
Yemen
Arabian Sea
Philippine Sea
Venezuela
Colombia
Suriname
Nigeria
South Sudan
Ethiopia
Somalia
Kenya
DRC
Tanzania
Indonesia
Brazil
Peru
Bolivia
Paraguay
Angola
Zambia
Namibia
Zimbabwe
Botswana
Madagascar
Indian Ocean
South Pacific Ocean
South Atlantic Ocean
Chile
Uruguay
Argentina
South Africa
Australia
Great Australian Bight
Tasman Sea
Coral Sea
New Zealand
North Pacific Ocean

https://www.submarinecablemap.com
Arctic Ocean
Greenland Sea
Baffin Bay
Barents Sea
Kara Sea
East Siberian Sea
Norwegian Sea
Northwestern Passages
Greenland
Iceland
Canada
Hudson Bay
Labrador Sea
Sweden
Finland
Norway
Russia
Poland
Belarus
Germany
Ukraine
France
Romania
Kazakhstan
Mongolia
Uzbekistan
Turkmenistan
Spain
Italy
Turkey
Syria
Iraq
Iran
Afghanistan
Pakistan
China
India
Nepal
Myanmar (Burma)
Thailand
United States
North Atlantic Ocean
Morocco
Algeria
Libya
Egypt
Saudi Arabia
Oman
Yemen
Arabian Sea
Mexico
Gulf of Mexico
Mauritania
Mali
Niger
Chad
Sudan
Nigeria
Ethiopia
South Sudan
Kenya
Venezuela
Colombia
Suriname
Brazil
Peru
Bolivia
Paraguay
Chile
Uruguay
Argentina
South Pacific Ocean
South Atlantic Ocean
DRC
Tanzania
Angola
Zambia
Namibia
Zimbabwe
Botswana
South Africa
Madagascar
Indian Ocean
Australia
Great Australian Bight
Tasman Sea
Bering Sea
SPOF

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*
- Examples:

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*
- Examples:
  - *Crash fault model*

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*
- Examples:
  - *Crash fault model*
  - *Byzantine fault model*

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*
- Examples:
  - *Crash fault model*
  - *Byzantine fault model*
  - *N-version programming*

Safety-critical systems

- Safety critical = system whose failure may result in "bad" outcomes
  - *SCADA, aviation, space, automotive, healthcare, …*

Safety-critical systems

- Safety critical = system whose failure may result in "bad" outcomes
  - *SCADA, aviation, space, automotive, healthcare, …*
- Fail-safe = failure does not have "bad" consequences
  - *safety-critical* $\not\Rightarrow$ *fail-safe*

Dependable systems

Dependable systems

- Availability = readiness for correct service

Dependable systems

- Availability = readiness for correct service
- Reliability = continuity of correct service

Dependable systems

- Availability = readiness for correct service
- Reliability = continuity of correct service
- Safety = absence of catastrophic consequences

Dependable systems

- Availability = readiness for correct service
- Reliability = continuity of correct service
- Safety = absence of catastrophic consequences
- Confidentiality = absence of unauthorized disclosure of information

Dependable systems

- Availability = readiness for correct service
- Reliability = continuity of correct service
- Safety = absence of catastrophic consequences
- Confidentiality = absence of unauthorized disclosure of information
- Integrity = absence of improper system state alterations

Dependable systems

- Availability = readiness for correct service
- Reliability = continuity of correct service
- Safety = absence of catastrophic consequences
- Confidentiality = absence of unauthorized disclosure of information
- Integrity = absence of improper system state alterations
- Maintainability = ability to undergo repairs and modifications

Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

R(t) = P(module operates correctly at time t | it was operating correctly at $t=0$)

Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

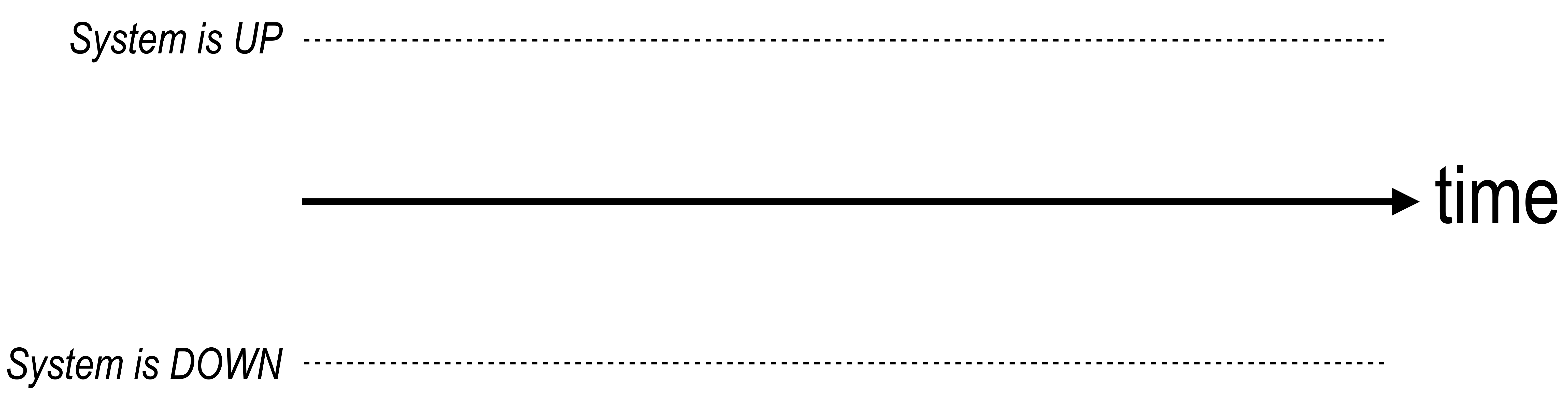
R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)

time

Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

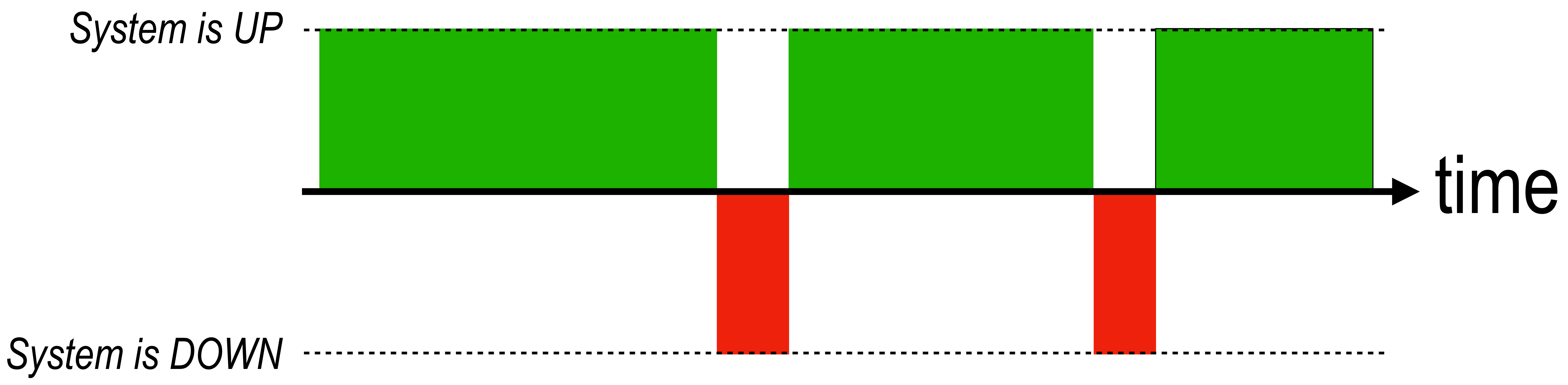
$$R(t) = P(\text{module operates correctly at time } t \mid \text{it was operating correctly at } t=0)$$



Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

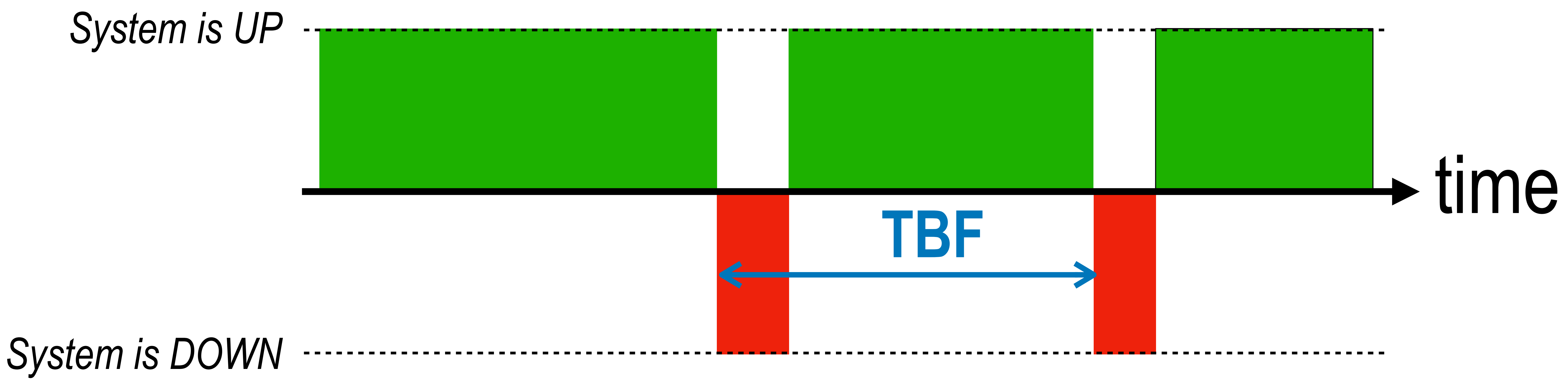
R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

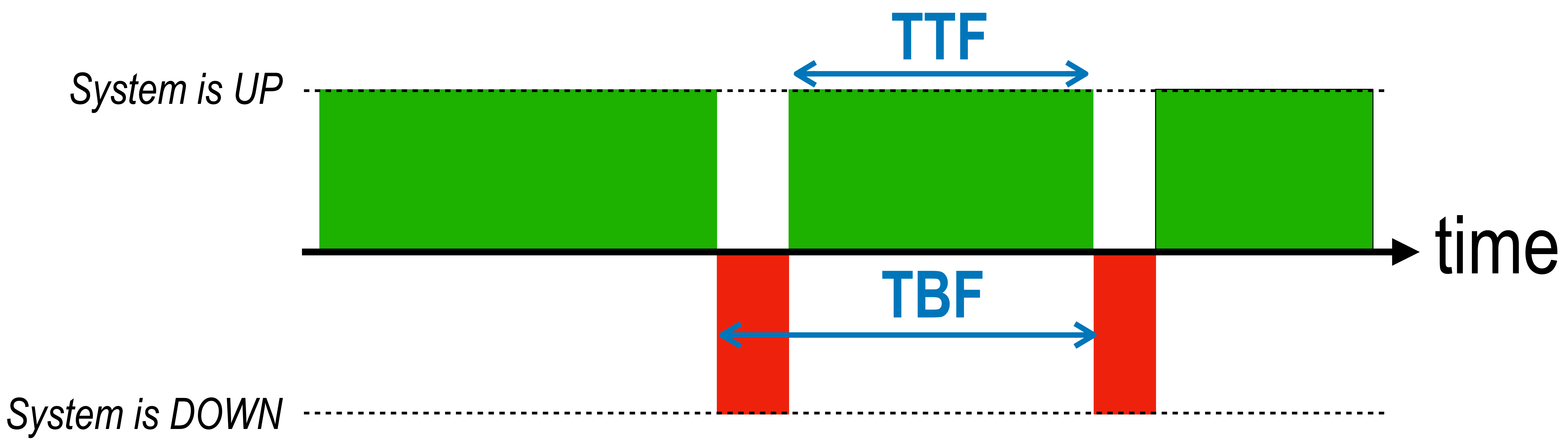
R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

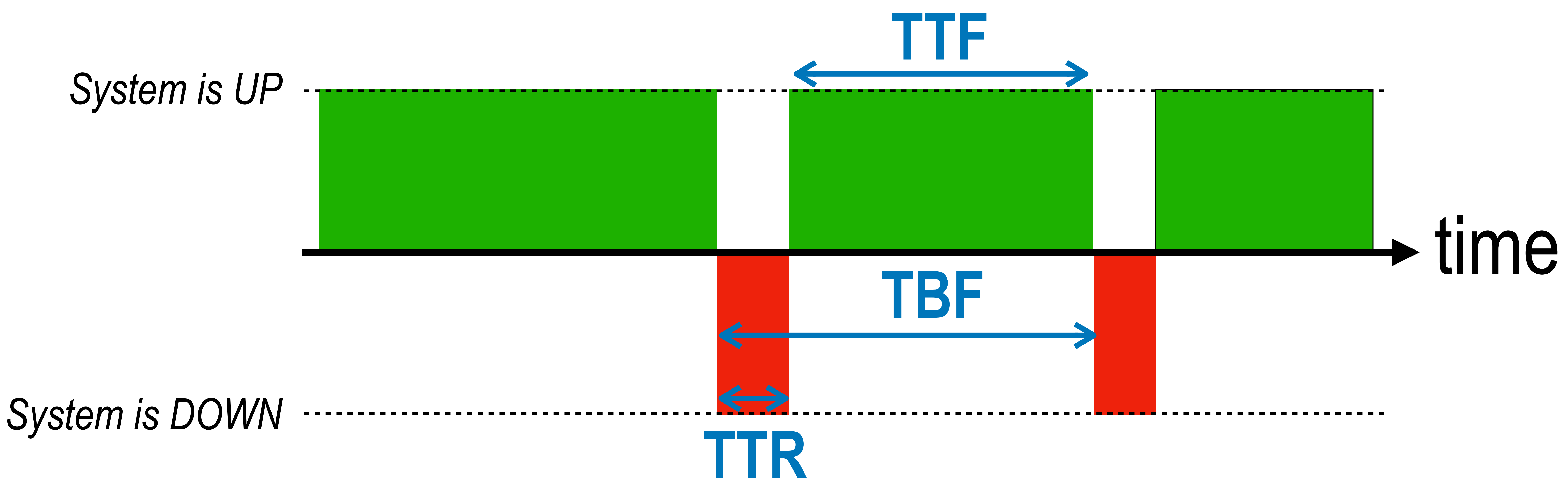
R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

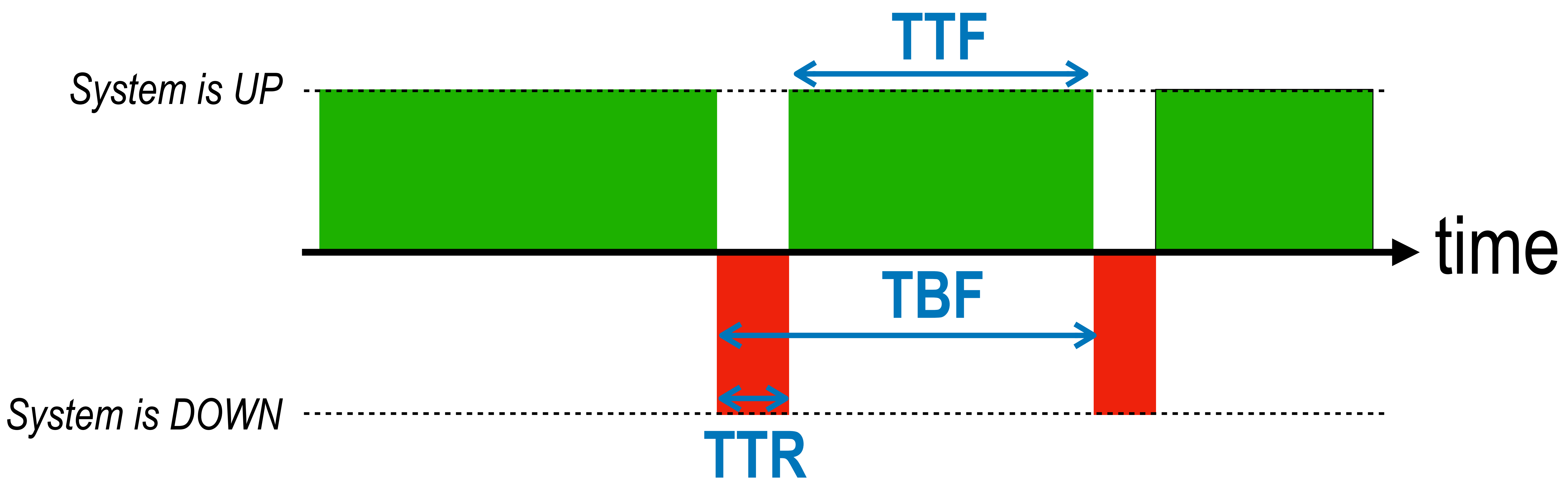
R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

R(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



MTBF = MTTF + MTTR

Measuring reliability

- In general MTBF or MTTF (MTBF = MTTF + MTTR)
  - *Specifics: Example from SSD spec sheet: P/E cycles, TBW, GB/day, DWPD, MTBF ...*
- Example: Samsung SSD 850 Pro SATA
  - *Warranty period = 10 years*
  - *MTBF = 2M hours (228 years)*

Measuring reliability

- In general MTBF or MTTF (MTBF = MTTF + MTTR)
  - *Specifics: Example from SSD spec sheet: P/E cycles, TBW, GB/day, DWPD, MTBF ...*
- Example: Samsung SSD 850 Pro SATA
  - *Warranty period = 10 years*
  - *MTBF = 2M hours (228 years)*

Why different???

Measuring reliability

- In general MTBF or MTTF (MTBF = MTTF + MTTR)
  - *Specifics: Example from SSD spec sheet: P/E cycles, TBW, GB/day, DWPD, MTBF ...*
- Example: Samsung SSD 850 Pro SATA
  - *Warranty period = 10 years*
  - *MTBF = 2M hours (228 years)*
    - assumes operation of 8 hrs/day

Why different???

Measuring reliability

- In general MTBF or MTTF (MTBF = MTTF + MTTR)
  - *Specifics: Example from SSD spec sheet: P/E cycles, TBW, GB/day, DWPD, MTBF ...*
- Example: Samsung SSD 850 Pro SATA
  - *Warranty period = 10 years*
  - *MTBF = 2M hours (228 years)*
    - assumes operation of 8 hrs/day
    - 2.5K SSDs => you'd experience 1 failure every ~100 days (2M / 8 / 2500)

Why different???

Availability

- Availability = probability of producing (correct) outputs in response to inputs

Availability

- Availability = probability of producing (correct) outputs in response to inputs

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

Availability

- Availability = probability of producing (correct) outputs in response to inputs

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

Availability

- Availability = probability of producing (correct) outputs in response to inputs

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min. ↑ ×10	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec. ↓ ÷10	8.6 msec

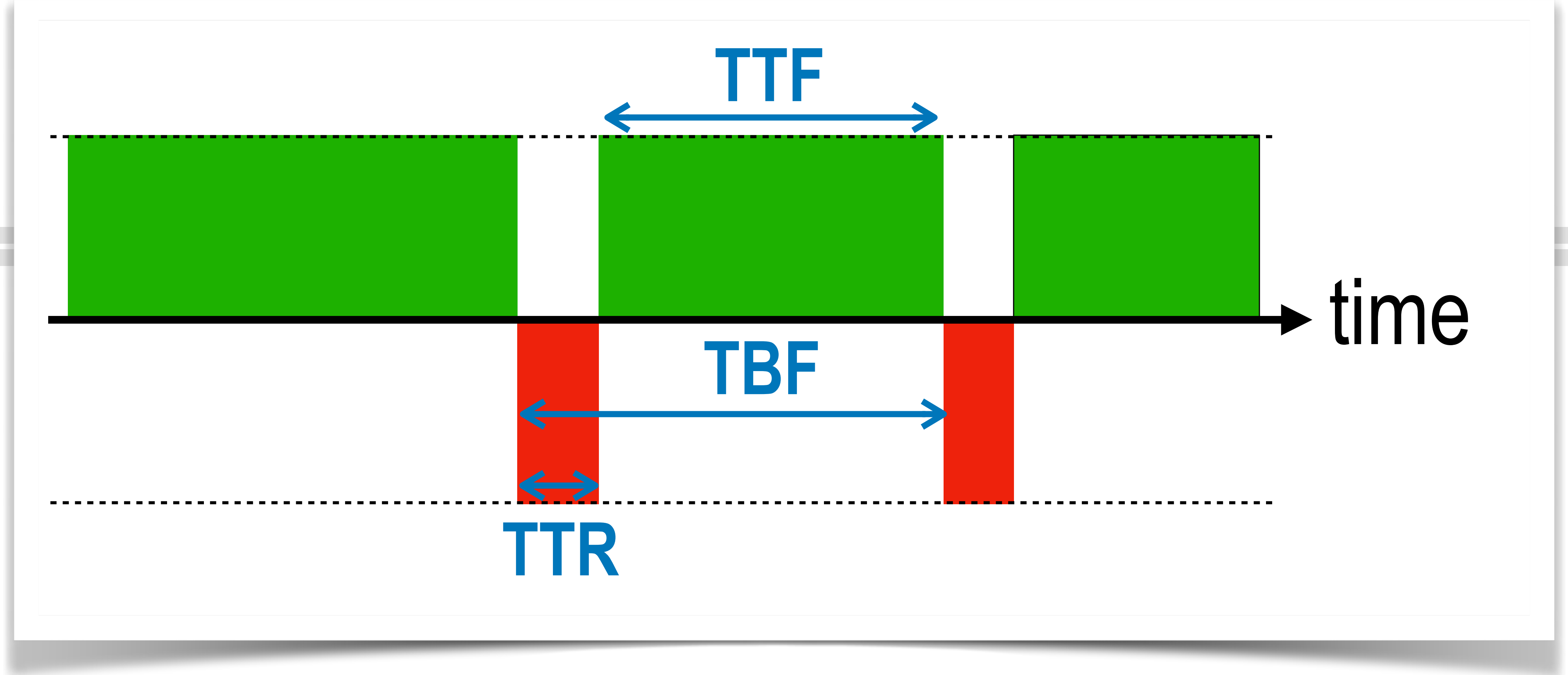
Availability vs. Reliability

- Continuity of service does not matter (unlike reliability)
  - *In theory: uptime is too strict a measure of availability*
  - *In practice: what's the difference?*
- Uptime => availability but Availability ⇏ uptime
- Examples of …
  - *Highly available systems with poor reliability (and how is redundancy used)*

    ...
  - *Highly reliable systems with poor availability (and how is redundancy used)*

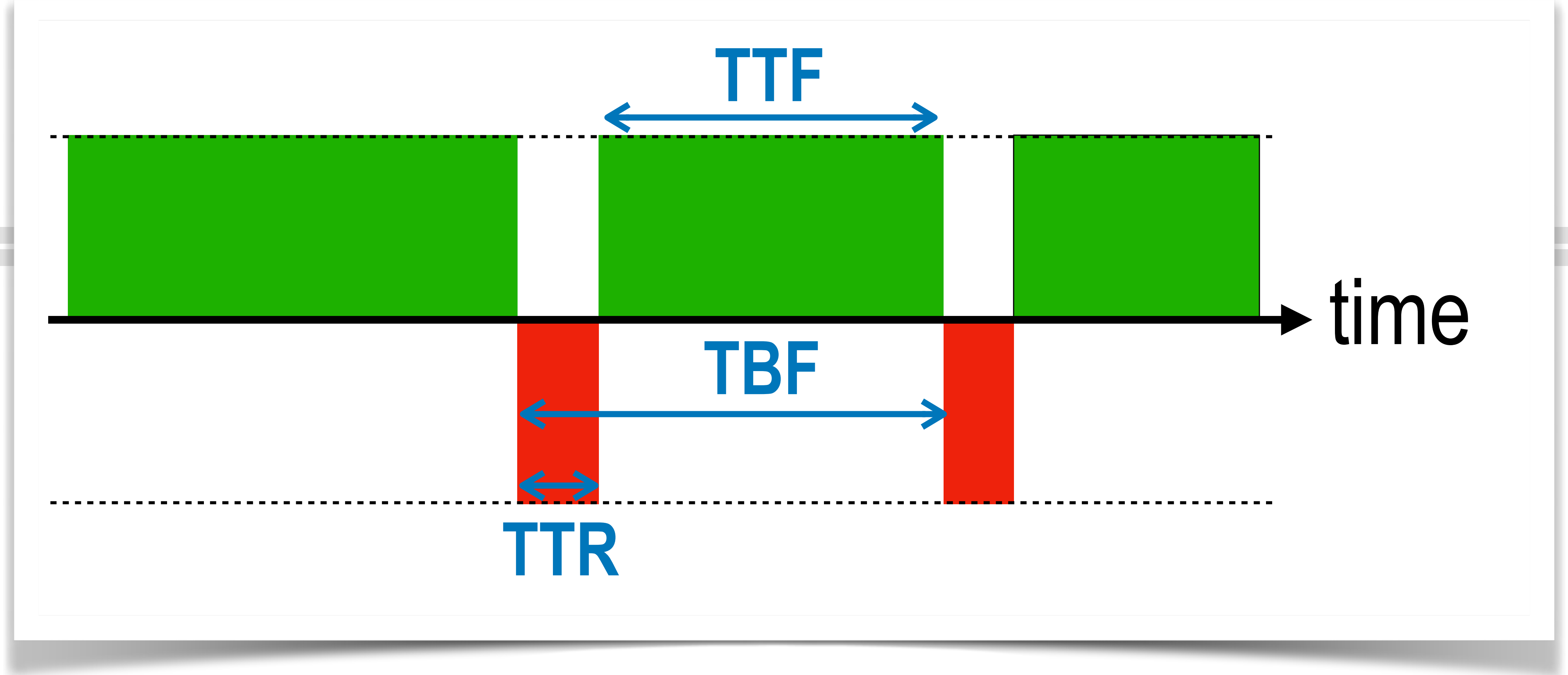
    ...

System availability

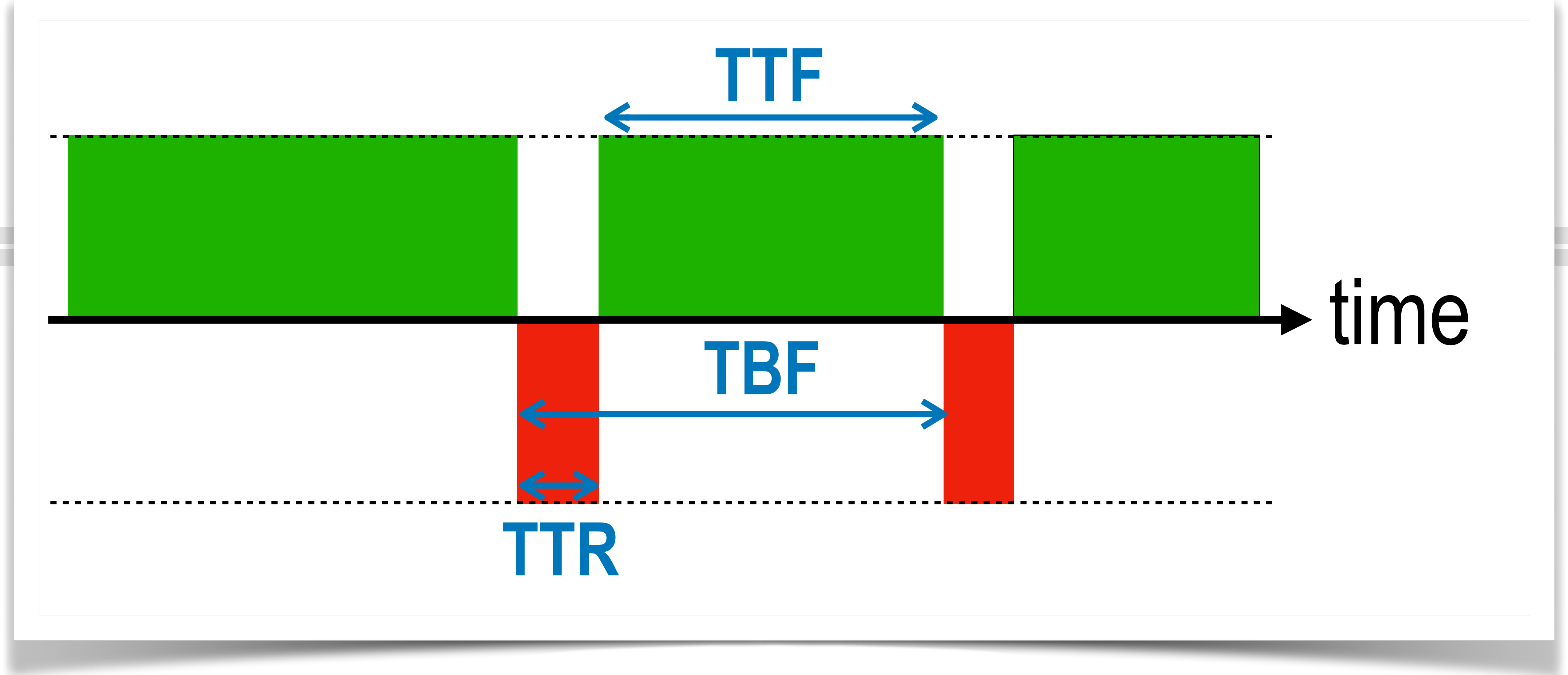


System availability

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$



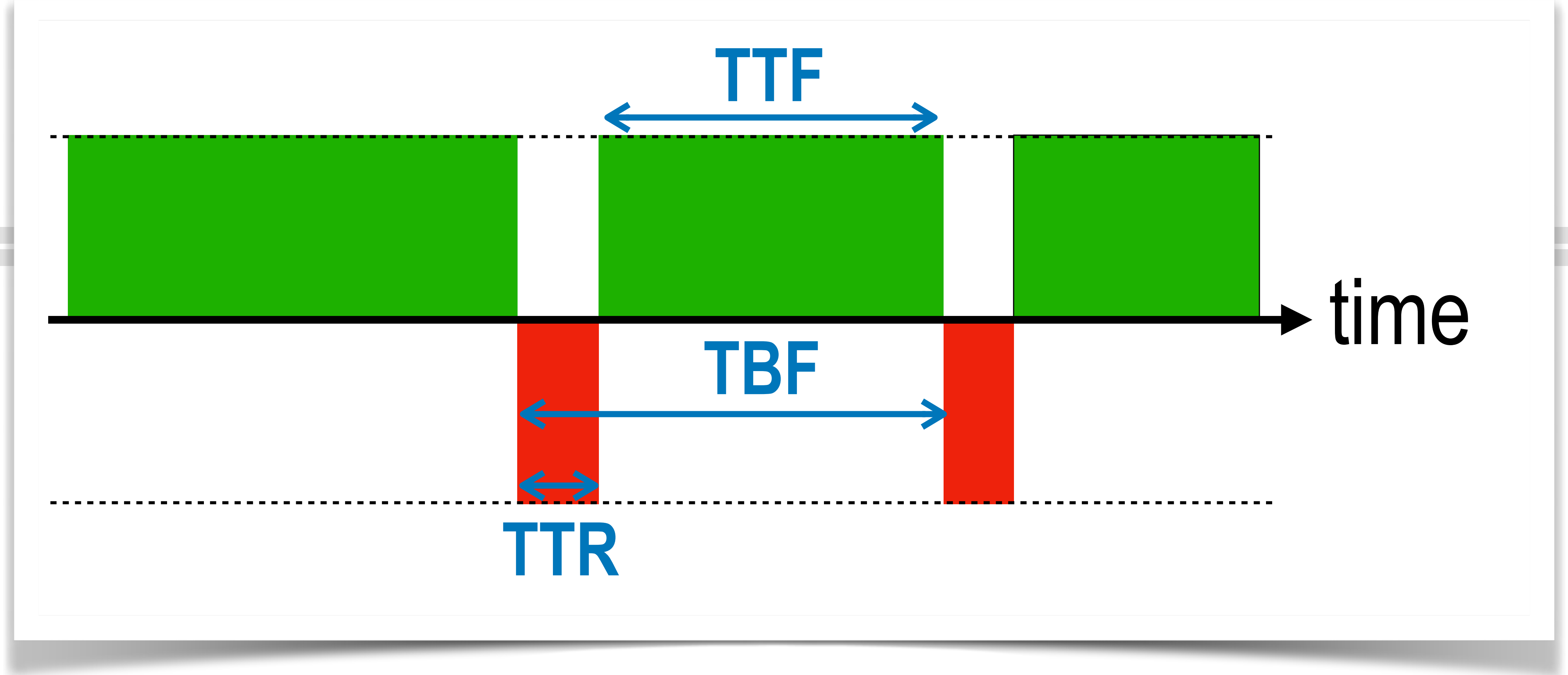
System availability



$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$

$$\text{Unavailability} = 1 - \text{Availability} = \frac{\text{MTTR}}{\text{MTBF}}$$

System availability

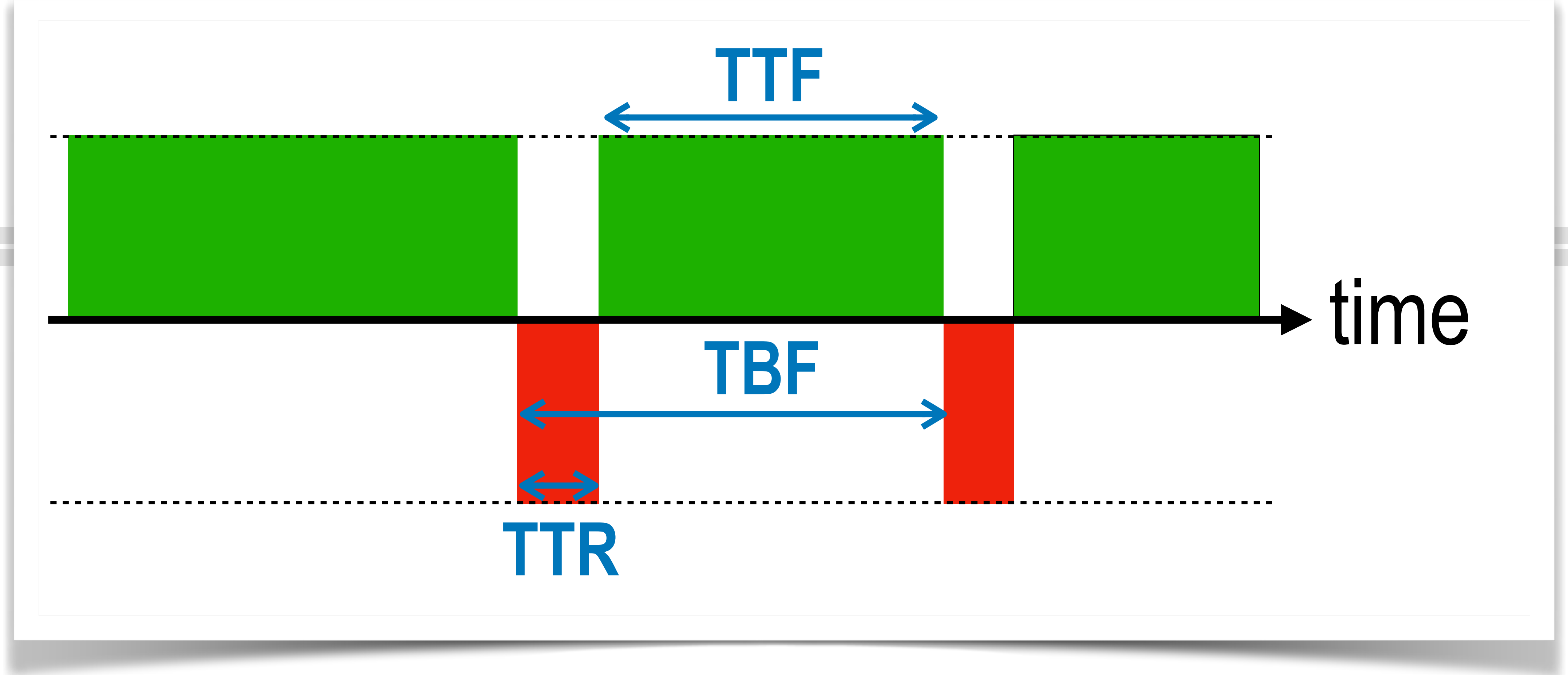


$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$

$$\text{Unavailability} = 1 - \text{Availability} = \frac{\text{MTTR}}{\text{MTBF}}$$

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \cong \text{MTTF} \quad (\text{if MTTF} \gg \text{MTTR})$$

System availability



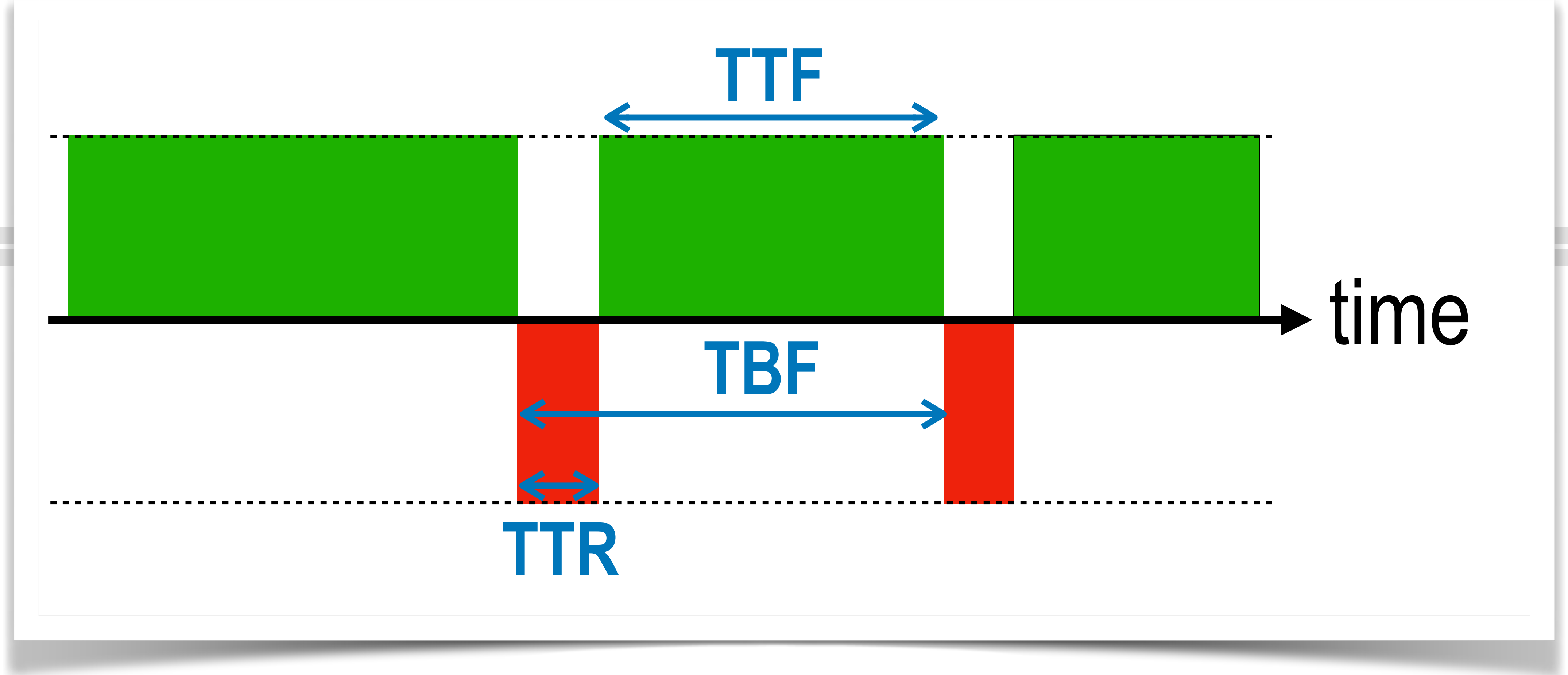
$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$

$$\text{Unavailability} = 1 - \text{Availability} = \frac{\text{MTTR}}{\text{MTBF}}$$

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \cong \text{MTTF} \quad (\text{if MTTF} \gg \text{MTTR})$$

$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF}}$$

System availability



$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$

$$\text{Unavailability} = 1 - \text{Availability} = \frac{\text{MTTR}}{\text{MTBF}}$$

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \cong \text{MTTF} \quad (\text{if MTTF} \gg \text{MTTR})$$

$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF}}$$

- Increase availability by
  - *increasing MTTF (higher reliability)*
  - *reducing MTTR (faster recovery)*

Failure modes

Failure modes

- Definition:

  When a system fails, how does that failure appear at the interface of a component?

- Four kinds
  - *fail-stop*
  - *fail-fast*
  - *fail-safe*
  - *fail-soft*

Failure mode 1: Fail-stop

Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*

Failure mode 1: Fail-stop

Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)

Failure mode 1: Fail-stop

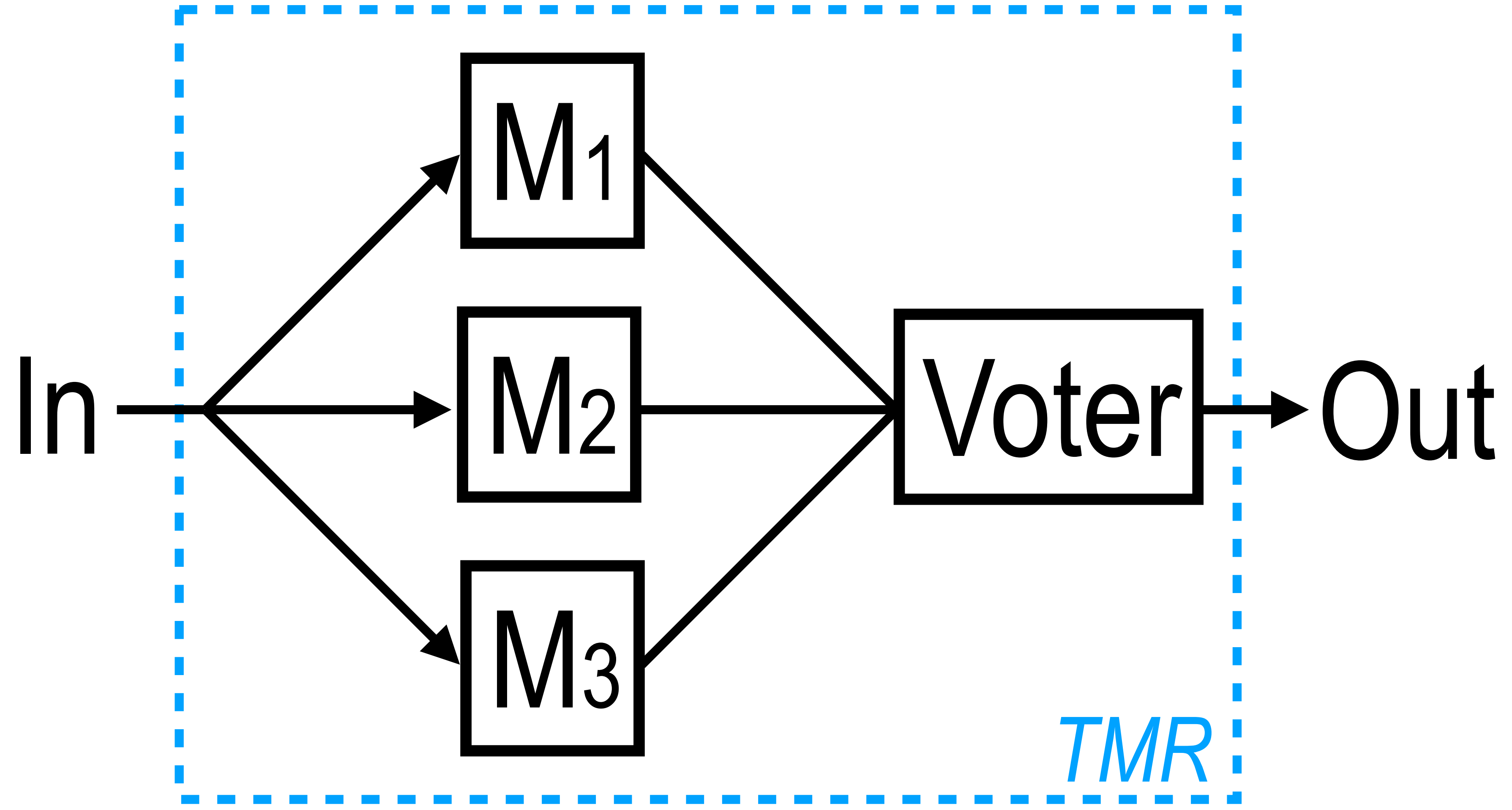
Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)
  - *Strict fault model: voter is reliable*

Failure mode 1: Fail-stop

> Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

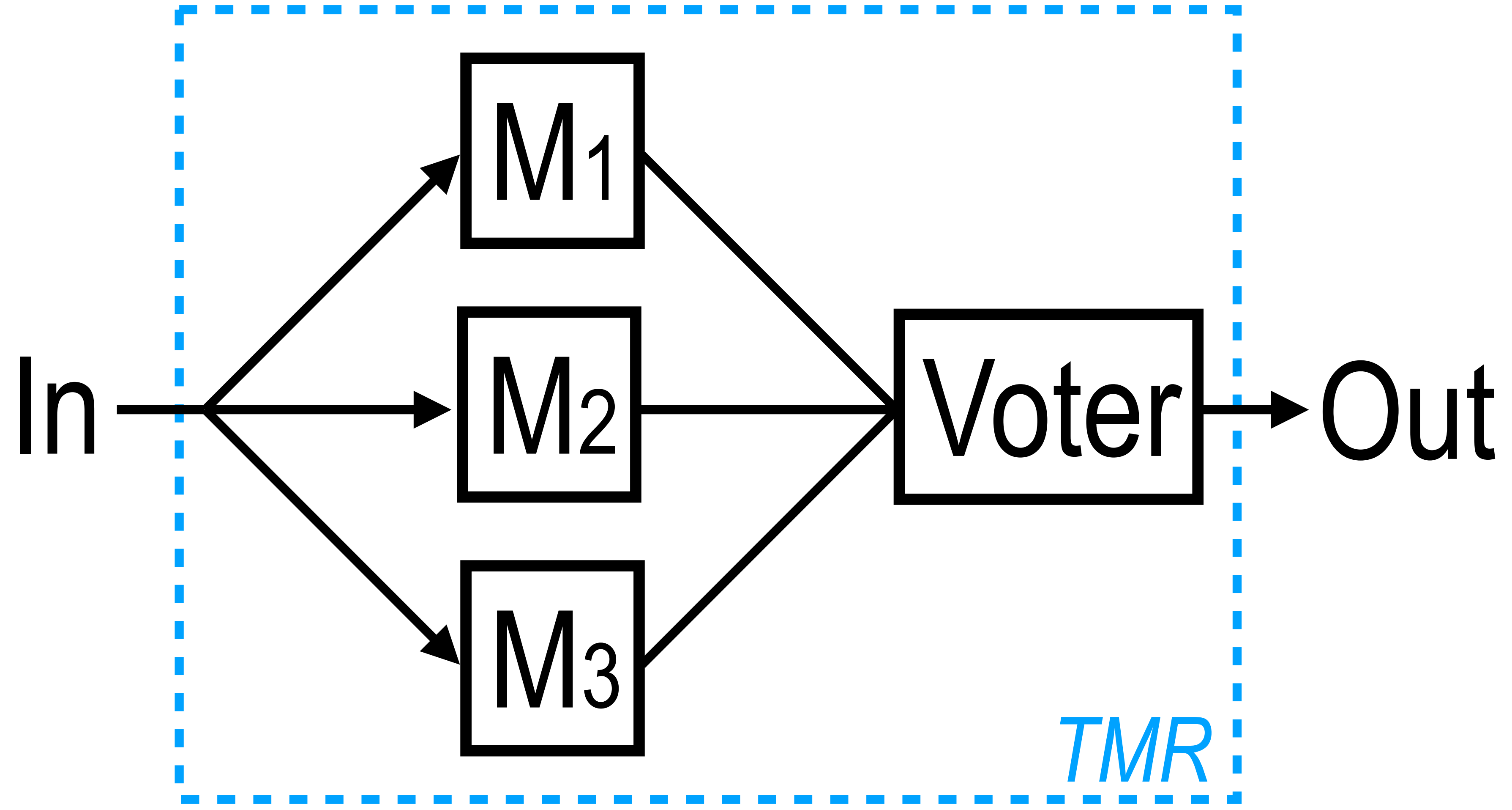
- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)
  - *Strict fault model: voter is reliable*
  - *Tolerate 1 arbitrary failure*
    - double modular redundancy → fail-stop behavior (f+1 modules)



Failure mode 1: Fail-stop

Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

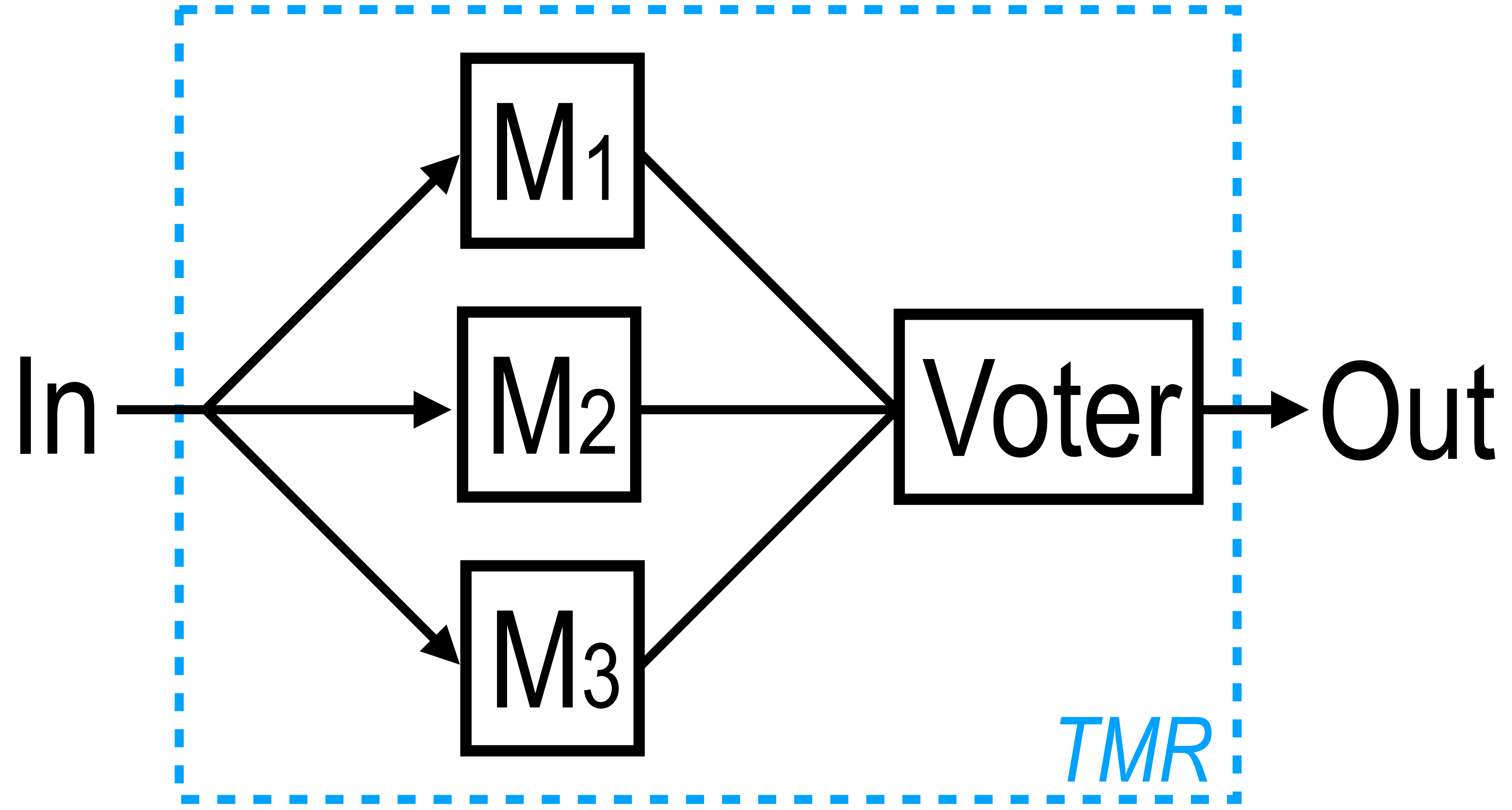
- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)
  - *Strict fault model: voter is reliable*
  - *Tolerate 1 arbitrary failure*
    - double modular redundancy → fail-stop behavior (f+1 modules)
    - triple modular redundancy → full fault tolerance (2f+1 modules)



Failure mode 1: Fail-stop

Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

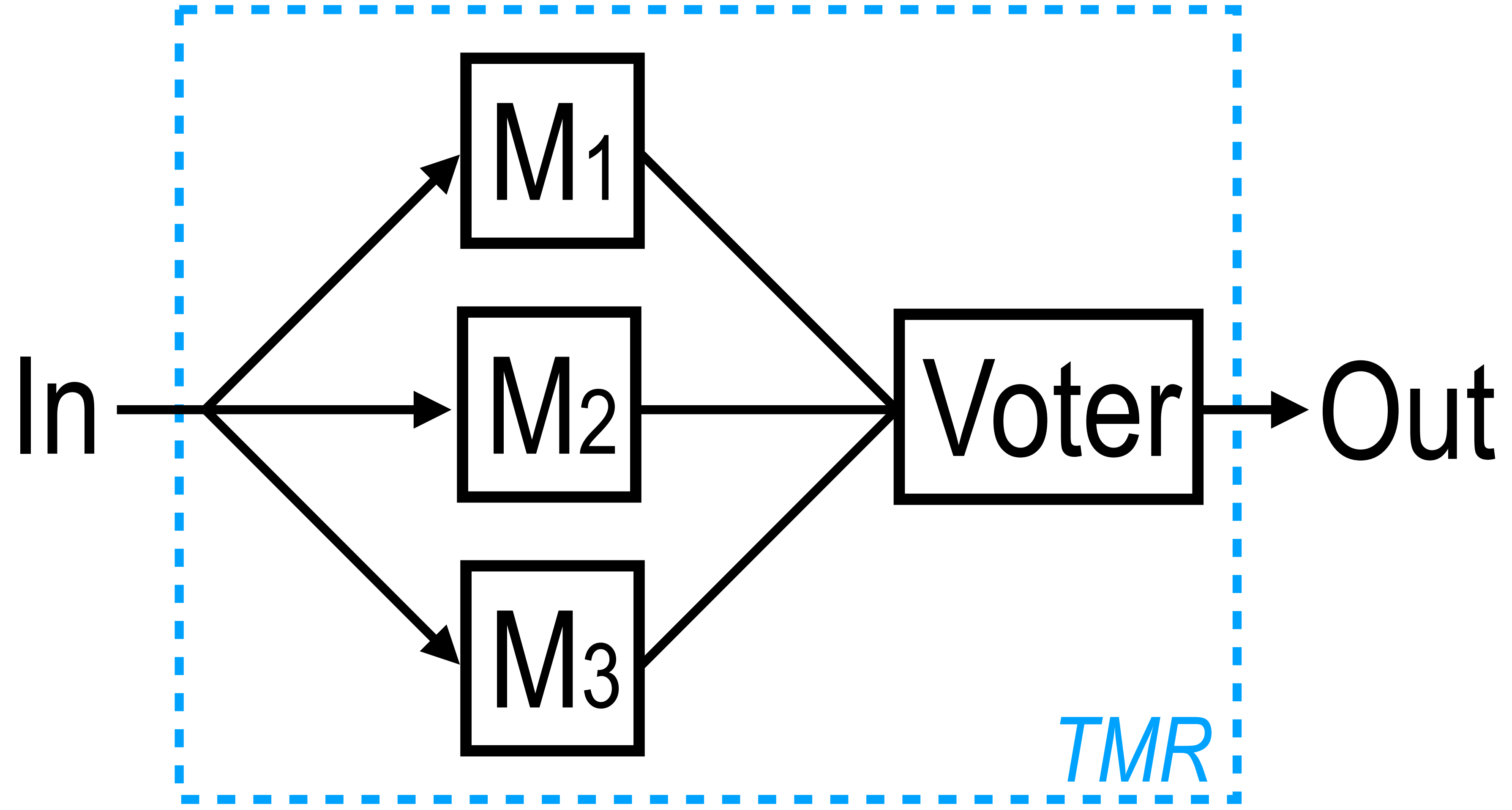
- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)
  - *Strict fault model: voter is reliable*
  - *Tolerate 1 arbitrary failure*
    - double modular redundancy → fail-stop behavior (f+1 modules)
    - triple modular redundancy → full fault tolerance (2f+1 modules)
  - *Achilles's heel: trusted voter*



Failure mode 1: Fail-stop

Different components/subsystems have their own failure mode, and the composition of failure modes results in the system's overall failure mode

- a.k.a. "crash failure" mode
- *Definition*: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with modular redundancy (MR)
  - *Strict fault model: voter is reliable*
  - *Tolerate 1 arbitrary failure*
    - double modular redundancy → fail-stop behavior (f+1 modules)
    - triple modular redundancy → full fault tolerance (2f+1 modules)
  - *Achilles's heel: trusted voter*
    - would need 3f+1 nodes if want consensus among peers



Failure mode 2: Fail-fast

Failure mode 2: Fail-fast

- *Definition*: immediately report at interface any situation that could lead to failure
  - *Can stop immediately after detection or delay (if expect recovery)*
  - *Must stop before failure manifests externally*

Failure mode 2: Fail-fast

- *Definition*: immediately report at interface any situation that could lead to failure
  - *Can stop immediately after detection or delay (if expect recovery)*
  - *Must stop before failure manifests externally*
- Requires frequent checks of state invariants

Failure mode 2: Fail-fast

- *Definition*: immediately report at interface any situation that could lead to failure
  - *Can stop immediately after detection or delay (if expect recovery)*
  - *Must stop before failure manifests externally*
- Requires frequent checks of state invariants
- Get auditability of error propagation

Failure mode 3: Fail-safe

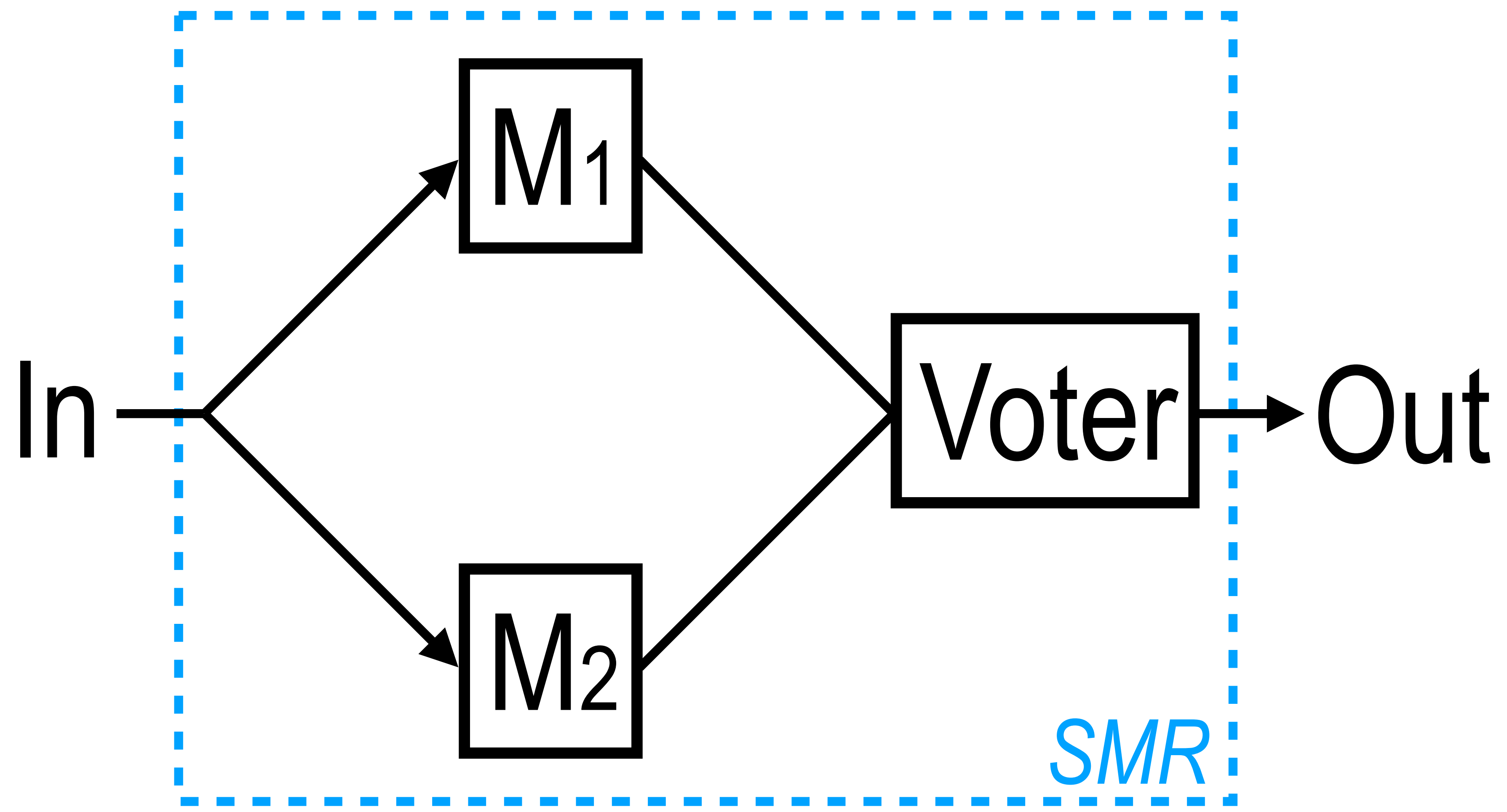
- *Definition*: the component remains safe in the face of failure
  - *but possibly degraded functionality or performance*

Failure mode 3: Fail-safe

- *Definition*: the component remains safe in the face of failure
  - *but possibly degraded functionality or performance*
- "Safety" is context-dependent

Failure mode 3: Fail-safe

- *Definition*: the component remains safe in the face of failure
  - *but possibly degraded functionality or performance*
- "Safety" is context-dependent
- "Controlled" failure

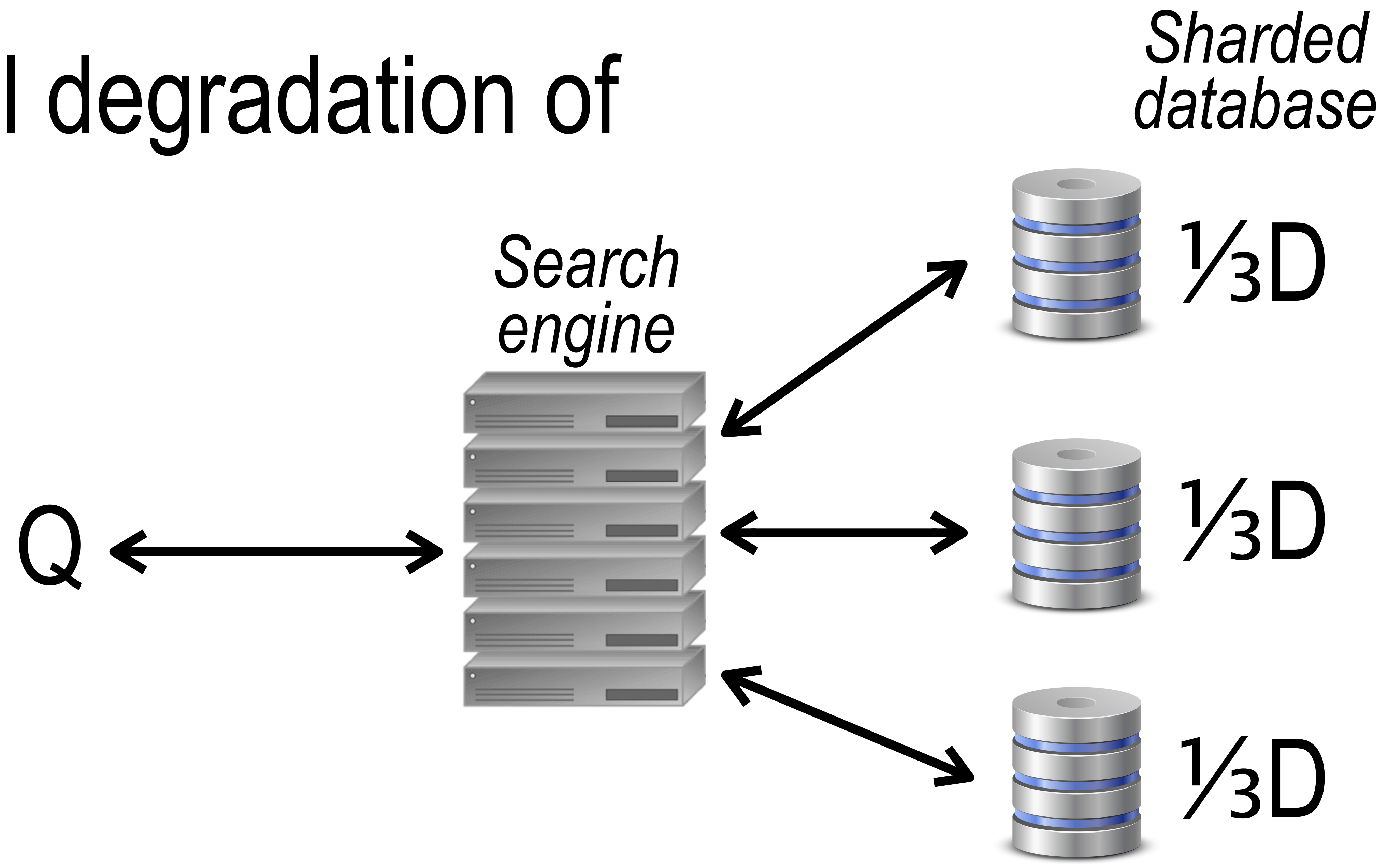


Failure mode 4: Fail-soft

- Definition: internal failures lead to graceful degradation of functionality instead of outright failure

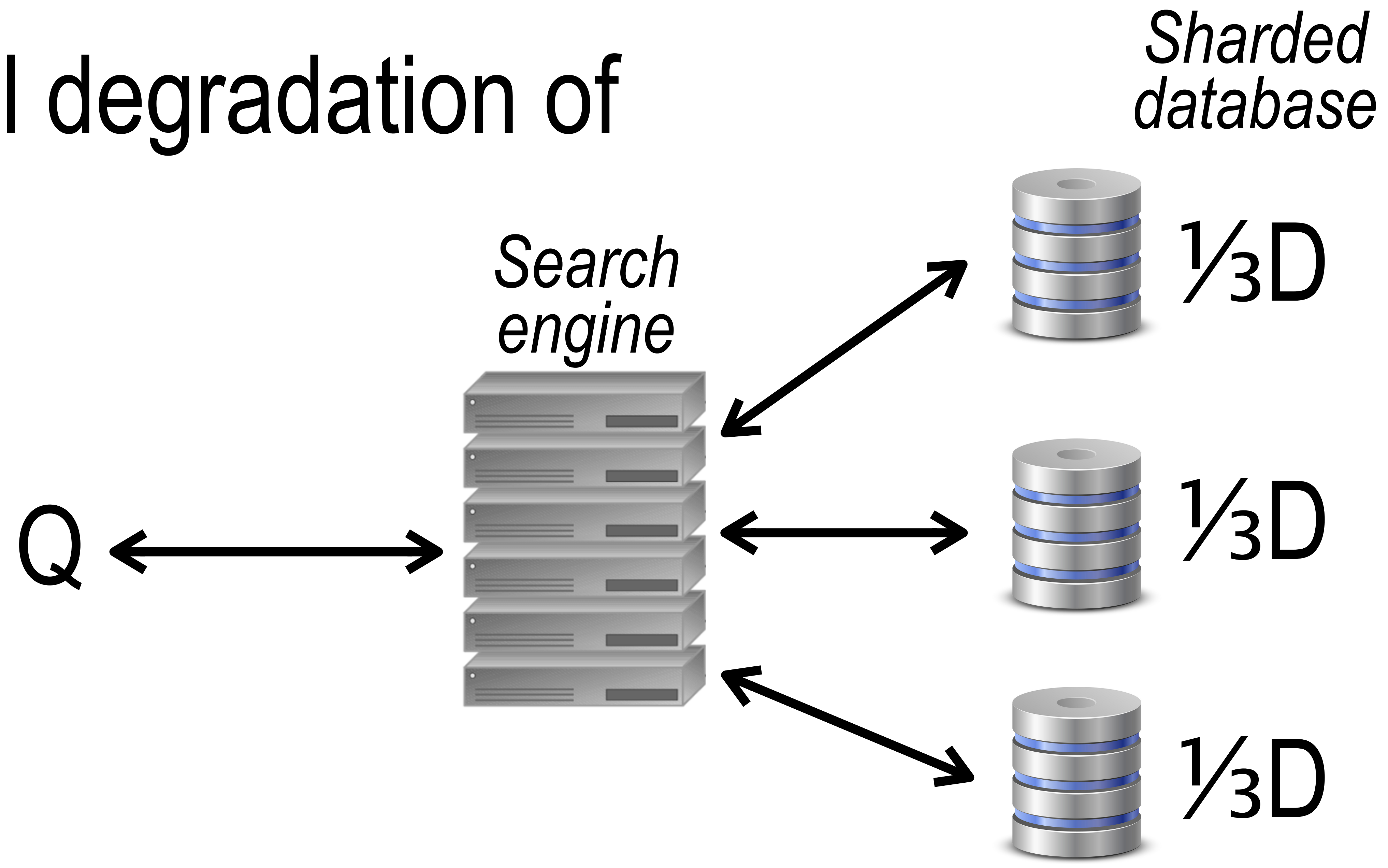
Failure mode 4: Fail-soft

- Definition: internal failures lead to graceful degradation of functionality instead of outright failure
- Example: simple search engine
  - *system has redundancy at every level*



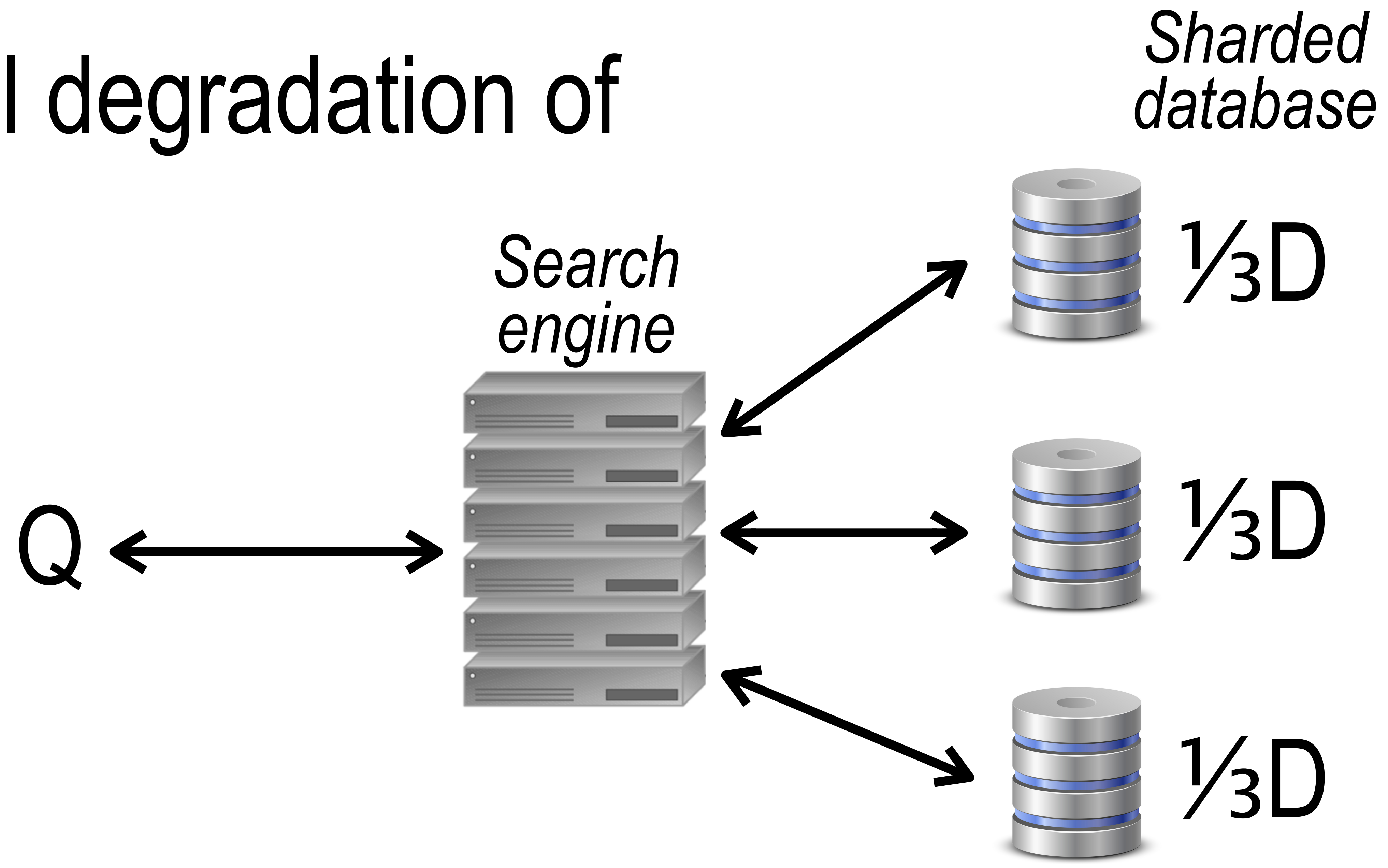
Failure mode 4: Fail-soft

- Definition: internal failures lead to graceful degradation of functionality instead of outright failure
- Example: simple search engine
  - *system has redundancy at every level*
- Intuition
  - *Functionality is typically bottlenecked on data movement (disks, network switches)*
    - => Functionality tied to how much data can be moved per unit of time



Failure mode 4: Fail-soft

- Definition: internal failures lead to graceful degradation of functionality instead of outright failure
- Example: simple search engine
  - *system has redundancy at every level*
- Intuition
  - *Functionality is typically bottlenecked on data movement (disks, network switches)*
    - => Functionality tied to how much data can be moved per unit of time
  - ***Harvest** (completeness of responses) vs. **yield** (fraction of requests served)*



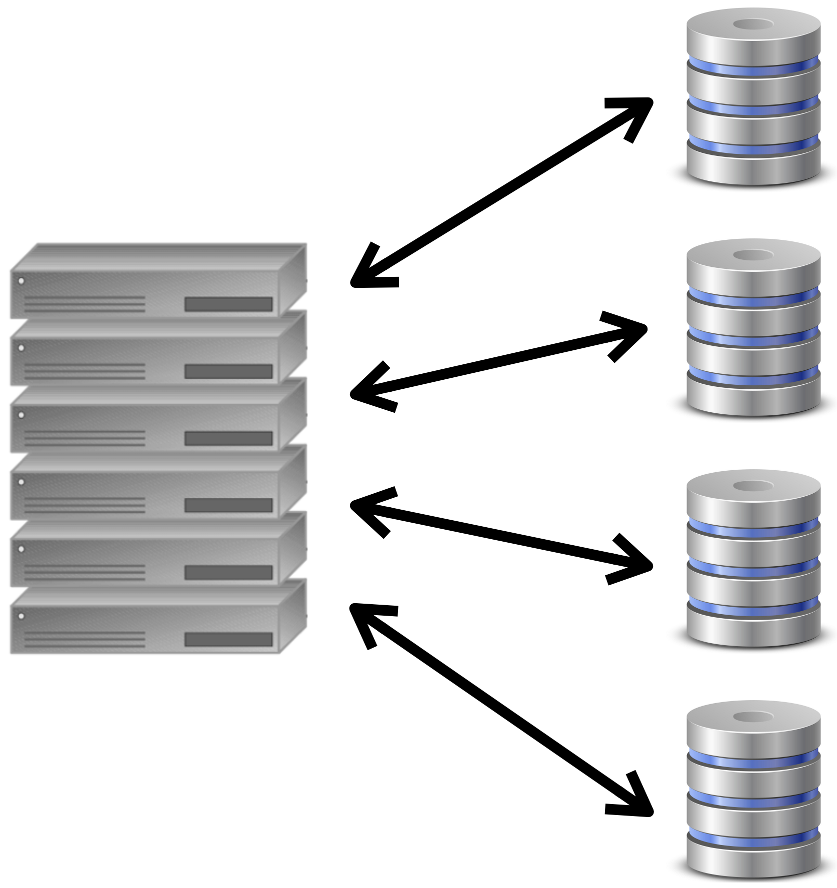
Failure mode 4: Fail-soft: DQ Principle

D = data/query DQ Principle: "D×Q is constant"
Q = queries/sec (DQ value *ρ* determined by system configuration)

Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)



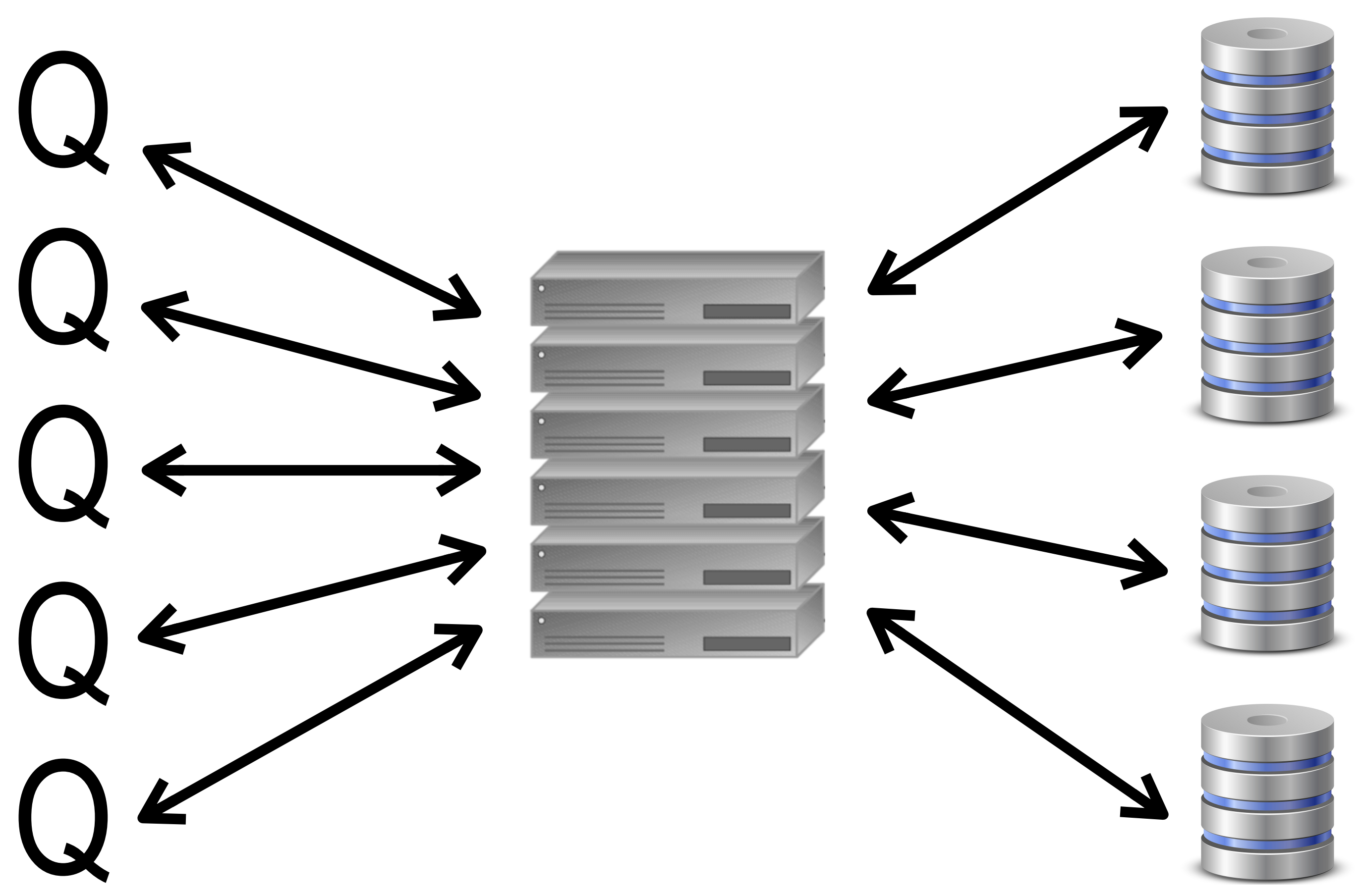
Failure mode 4: Fail-soft: DQ Principle

D = data/query

Q = queries/sec

DQ Principle: "D×Q is constant"

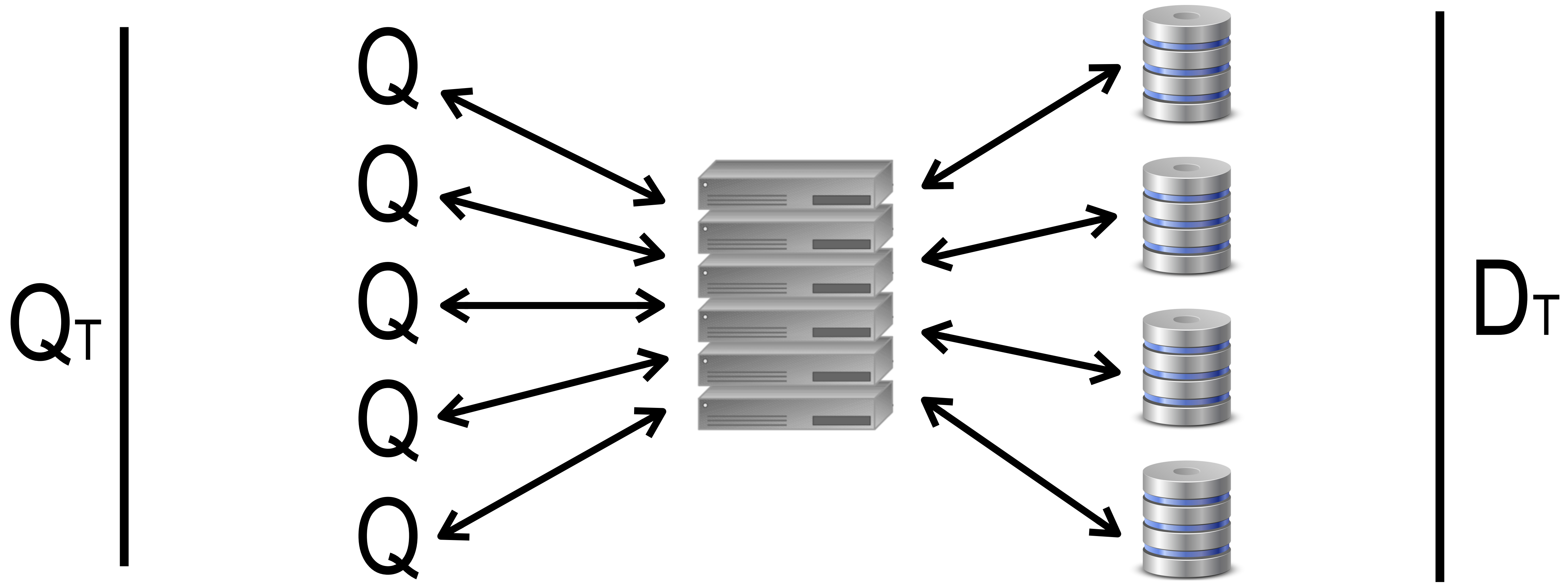
(DQ value ρ determined by system configuration)



Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

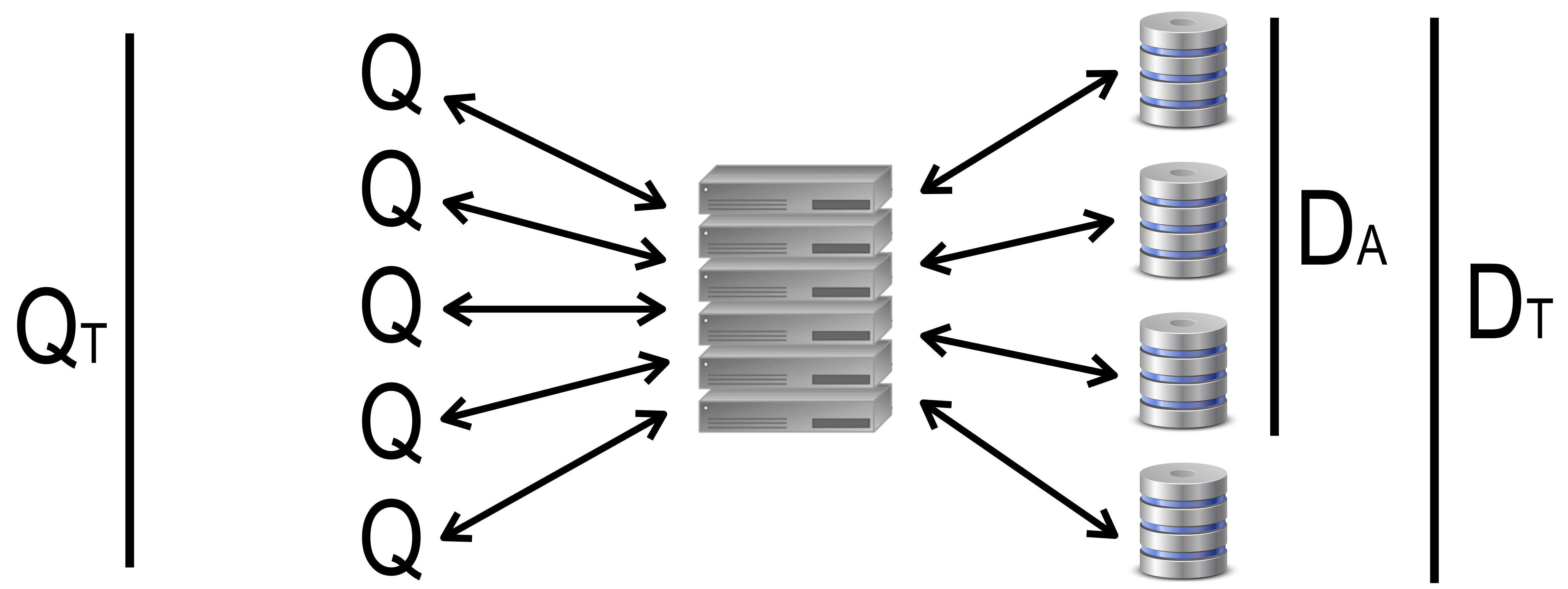
DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)



Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)



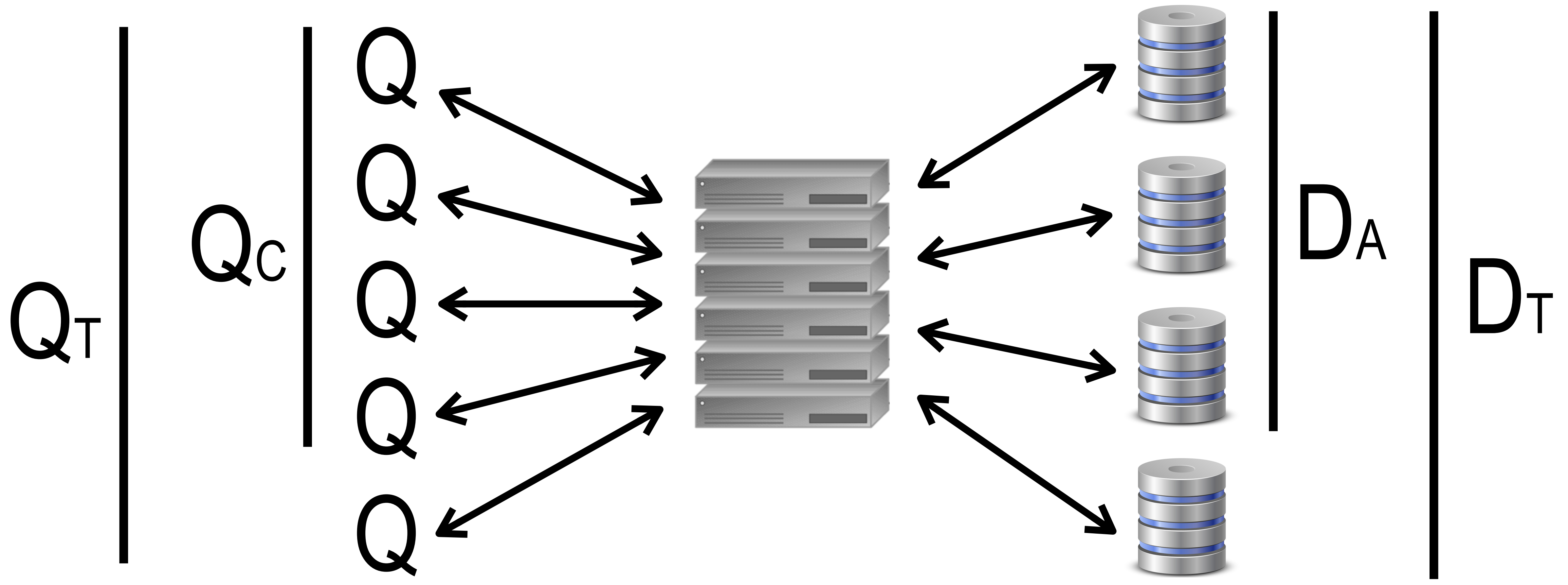
Failure mode 4: Fail-soft: DQ Principle

D = data/query

Q = queries/sec

DQ Principle: "D×Q is constant"

(DQ value ρ determined by system configuration)

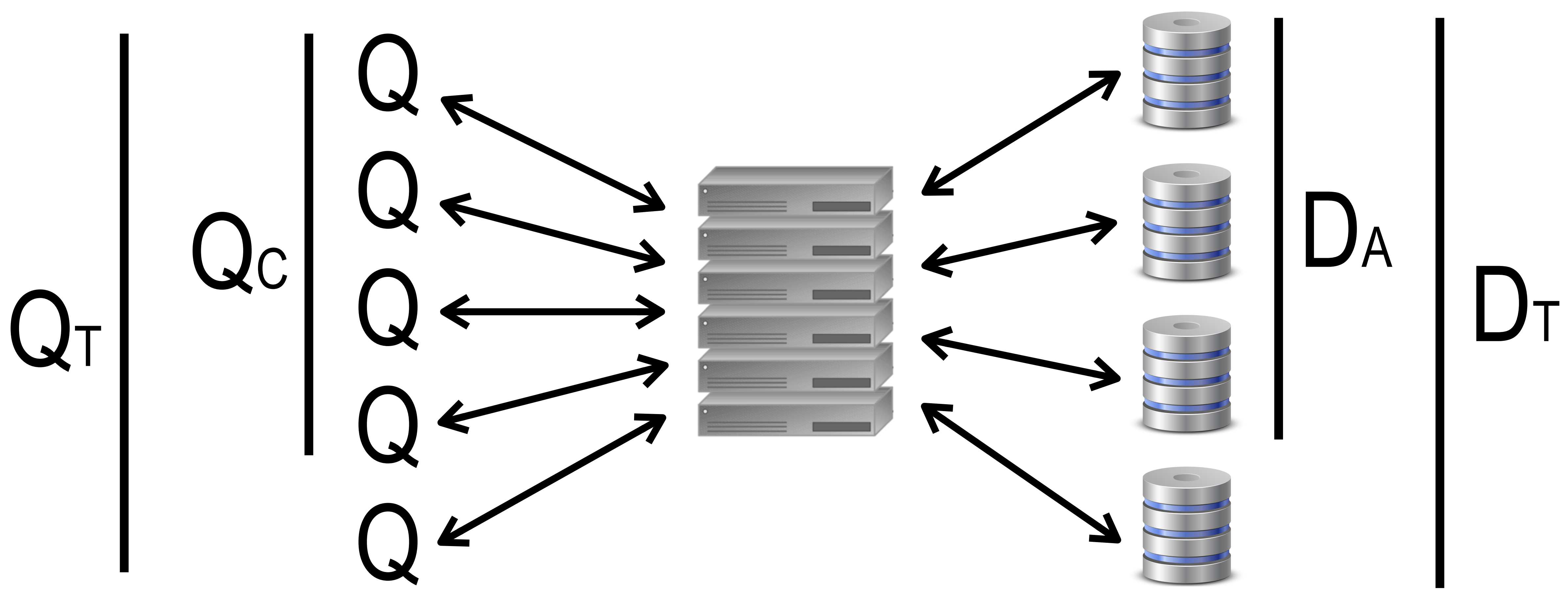


Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)

Harvest $H = \frac{D_A}{D_T}$



Failure mode 4: Fail-soft: DQ Principle

D = data/query

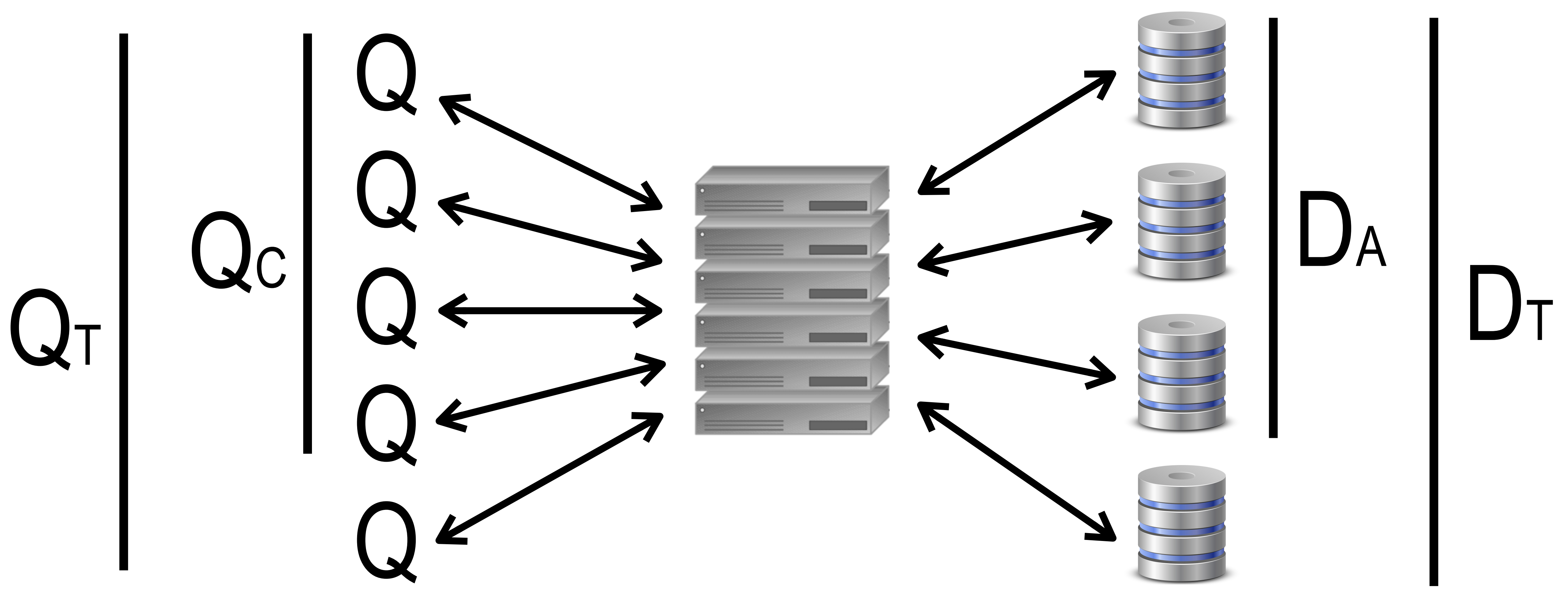
Q = queries/sec

DQ Principle: "D×Q is constant"

(DQ value ρ determined by system configuration)

$$\text{Harvest } H = \frac{D_A}{D_T}$$

$$\text{Yield } Y = \frac{Q_C}{Q_T}$$



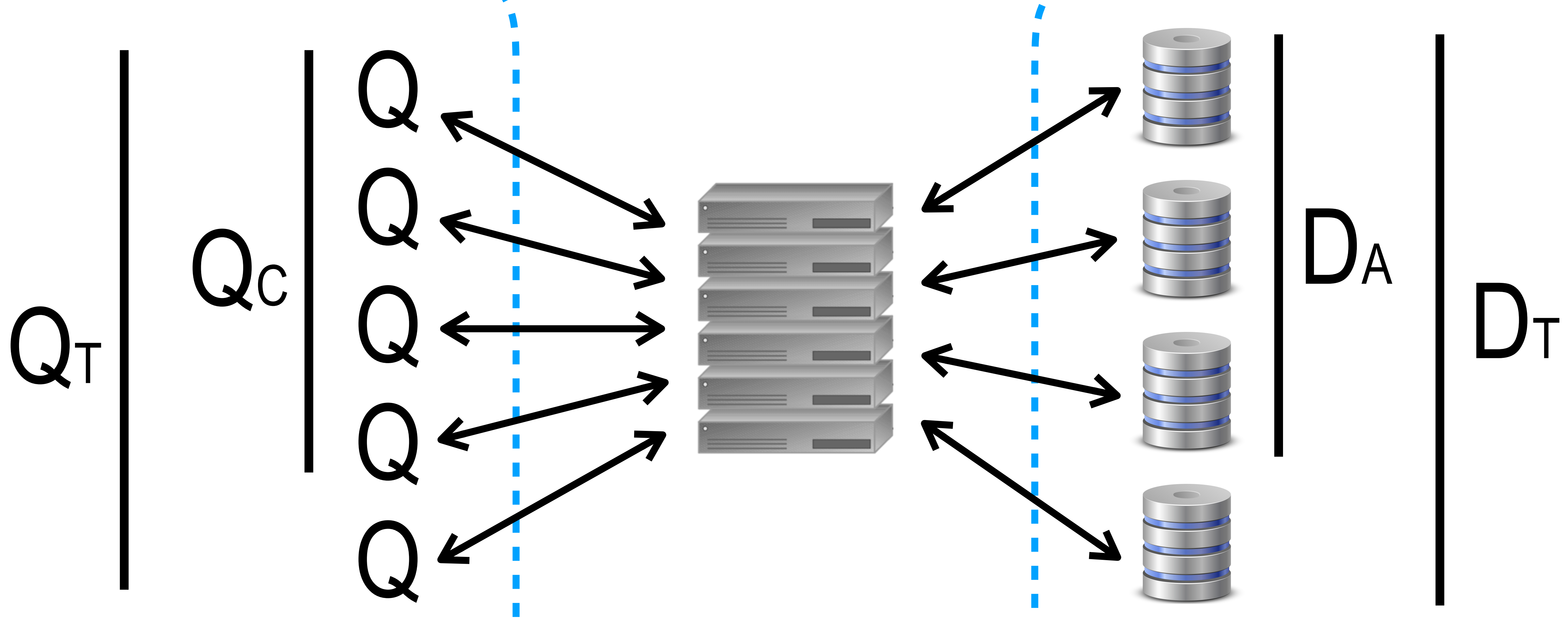
Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)

Harvest $H = \frac{D_A}{D_T}$

Yield $Y = \frac{Q_C}{Q_T}$



DQ Principle: $H \times Y = \rho$ $\dashrightarrow$ $Y = Q_C / Q_T$ $\times$ $H = D_A / D_T$ $= \rho$

[1] Assumption: Disk is the only bottleneck. All the other resources (e.g., network bandwidth) are plentiful.

Failure mode 4: Fail-soft: DQ Principle

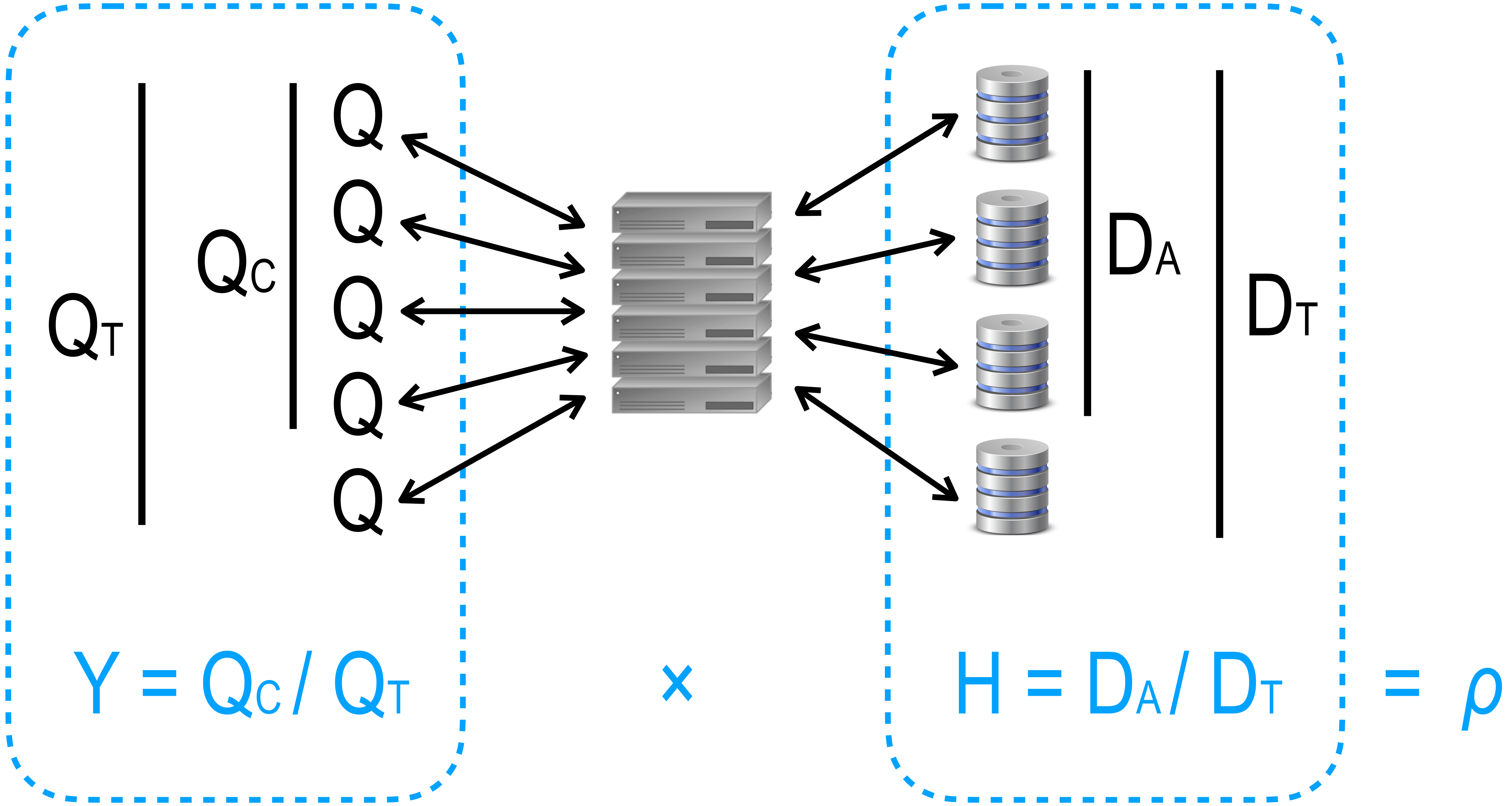
D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)

Harvest $H = \dfrac{D_A}{D_T}$

Yield $Y = \dfrac{Q_C}{Q_T}$

DQ Principle: $H \times Y = \rho$



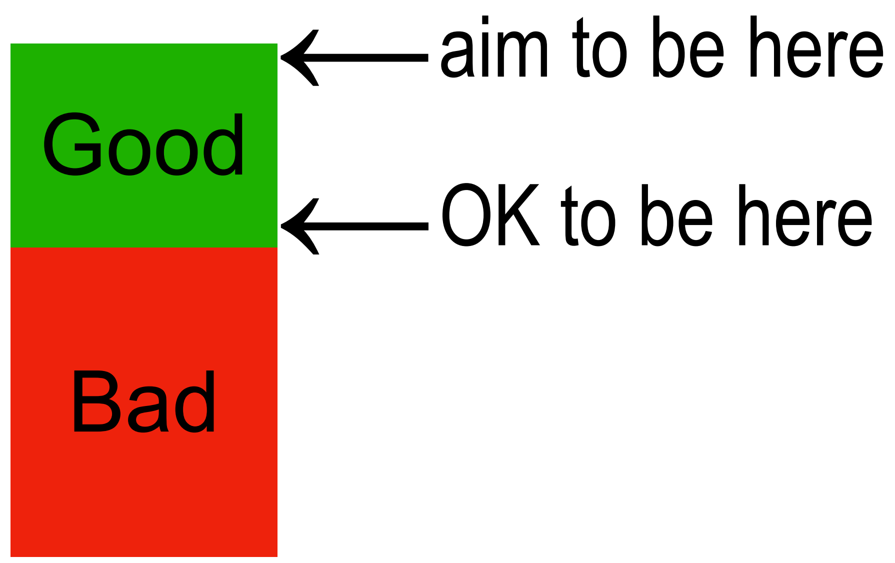
[1] Assumption: Disk is the only bottleneck. All the other resources (e.g., network bandwidth) are plentiful.

Recap: Failure modes

- Fail-stop (TMR)
- Fail-fast (Redundant invariant checks)
- Fail-safe
  - *OK to fail, as long as safety is not compromised*
- Fail-soft (Weaker spec)
  - *Redundant resources for top band of acceptable system behavior*
  - *Harvest/yield and the DQ principle in data-intensive parallel systems*

Recap: Failure modes

- Fail-stop (TMR)
- Fail-fast (Redundant invariant checks)
- Fail-safe
  - *OK to fail, as long as safety is not compromised*
- Fail-soft (Weaker spec)
  - *Redundant resources for top band of acceptable system behavior*
  - *Harvest/yield and the DQ principle in data-intensive parallel systems*



How to reduce unavailability by 10× ?

How to reduce unavailability by 10× ?

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$$

How to reduce unavailability by 10× ?

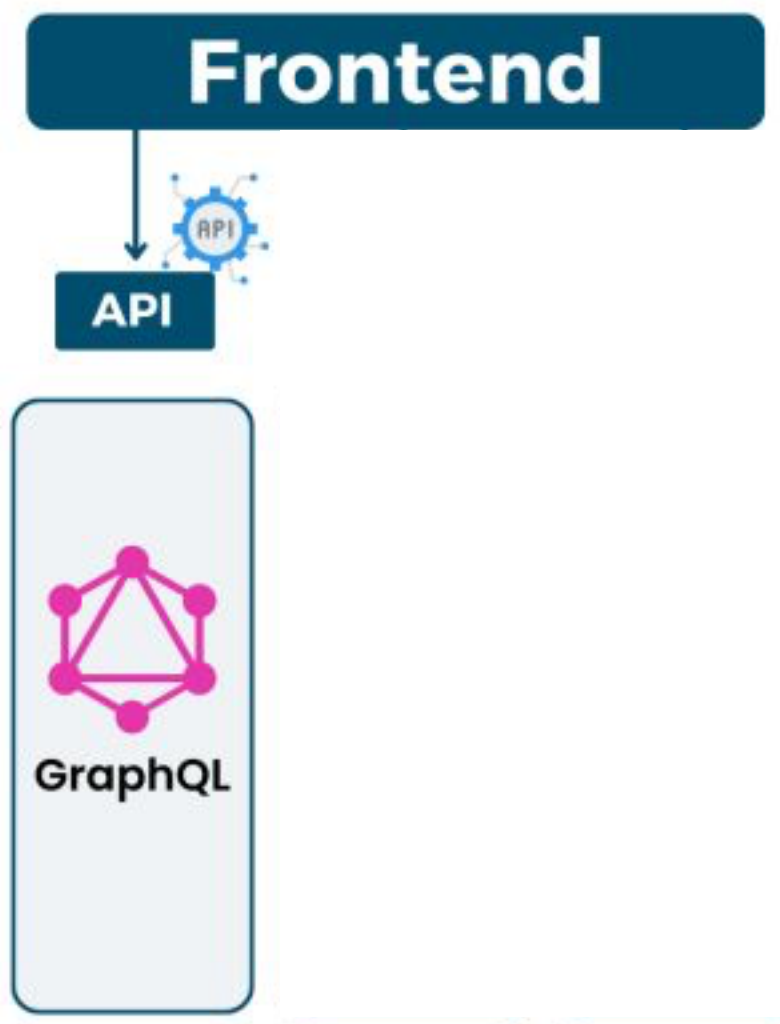
$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF}}$$

How to reduce unavailability by 10× ?

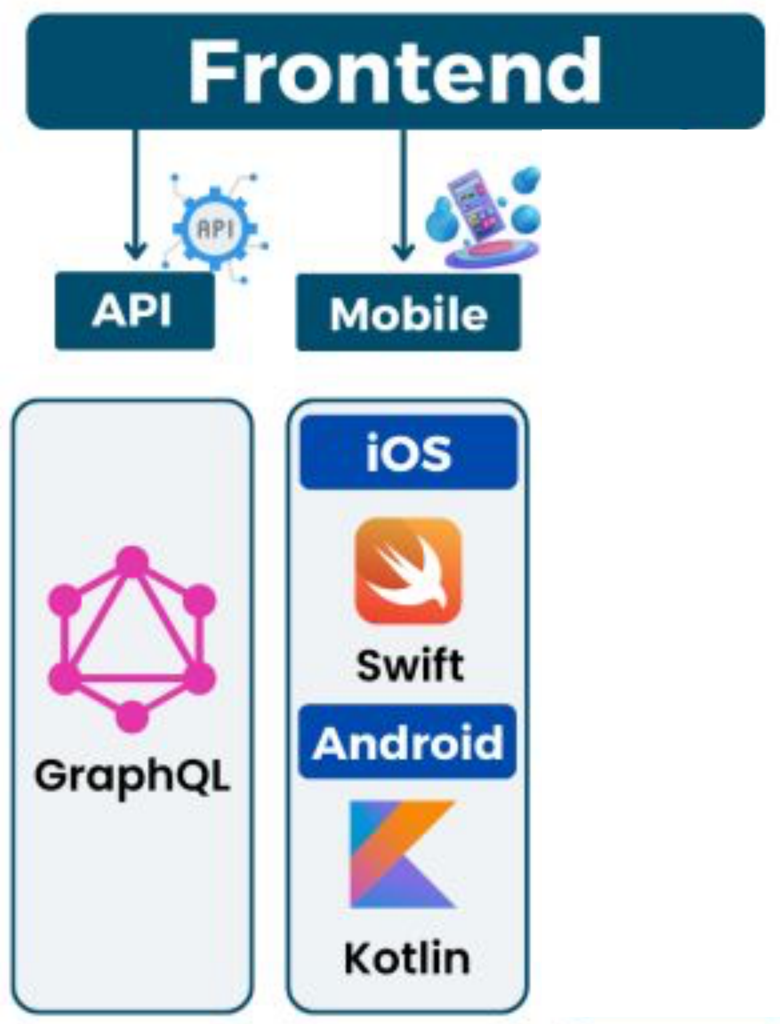
$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF} \; {\color{red}\uparrow\times 10}}$$

NETFLIX

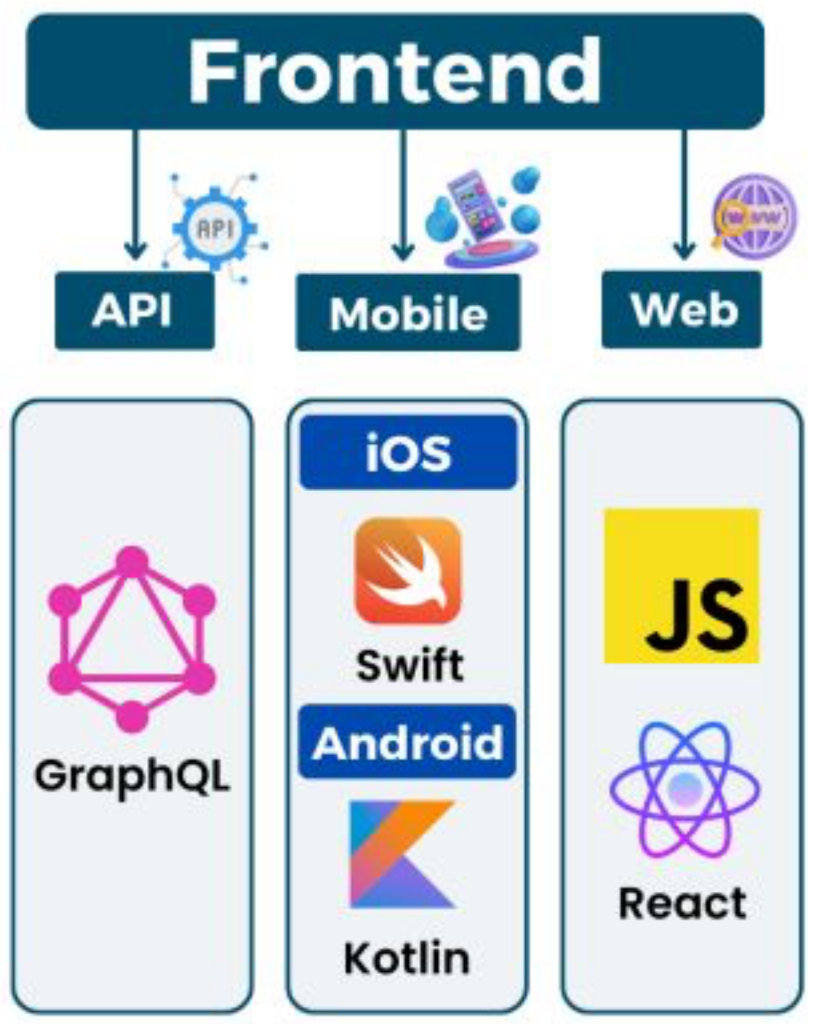
https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv



NETFLIX

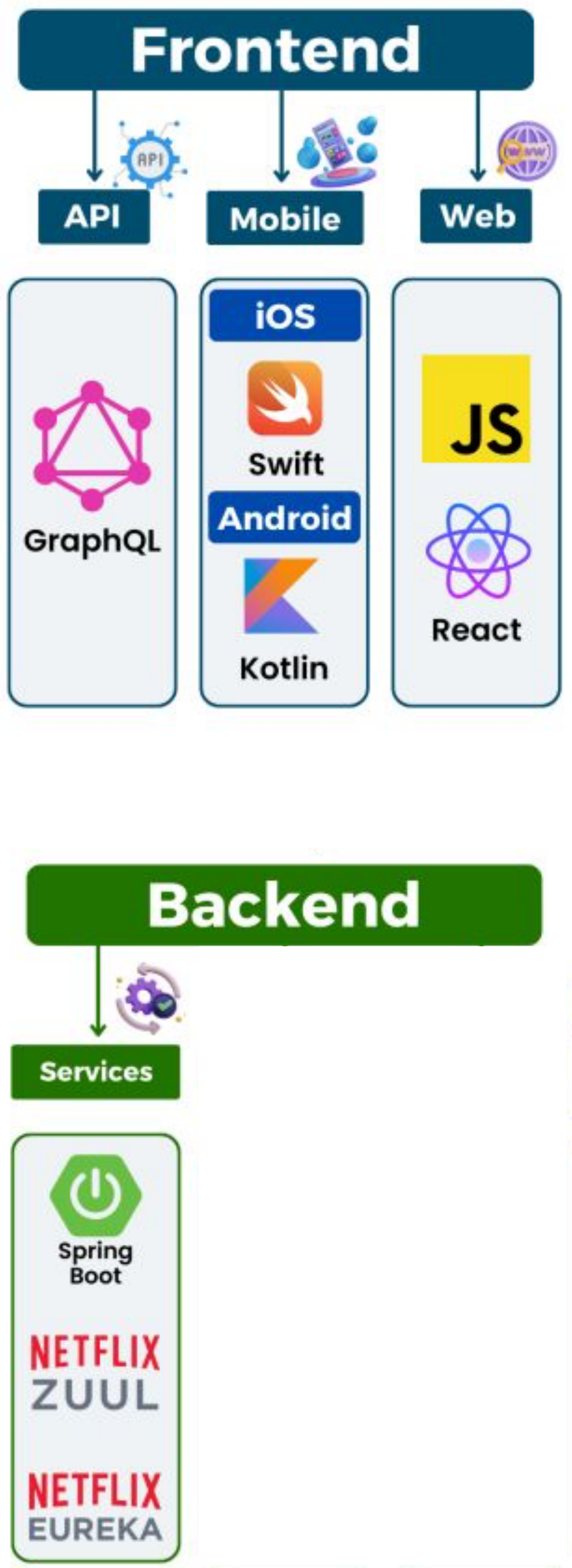


NETFLIX

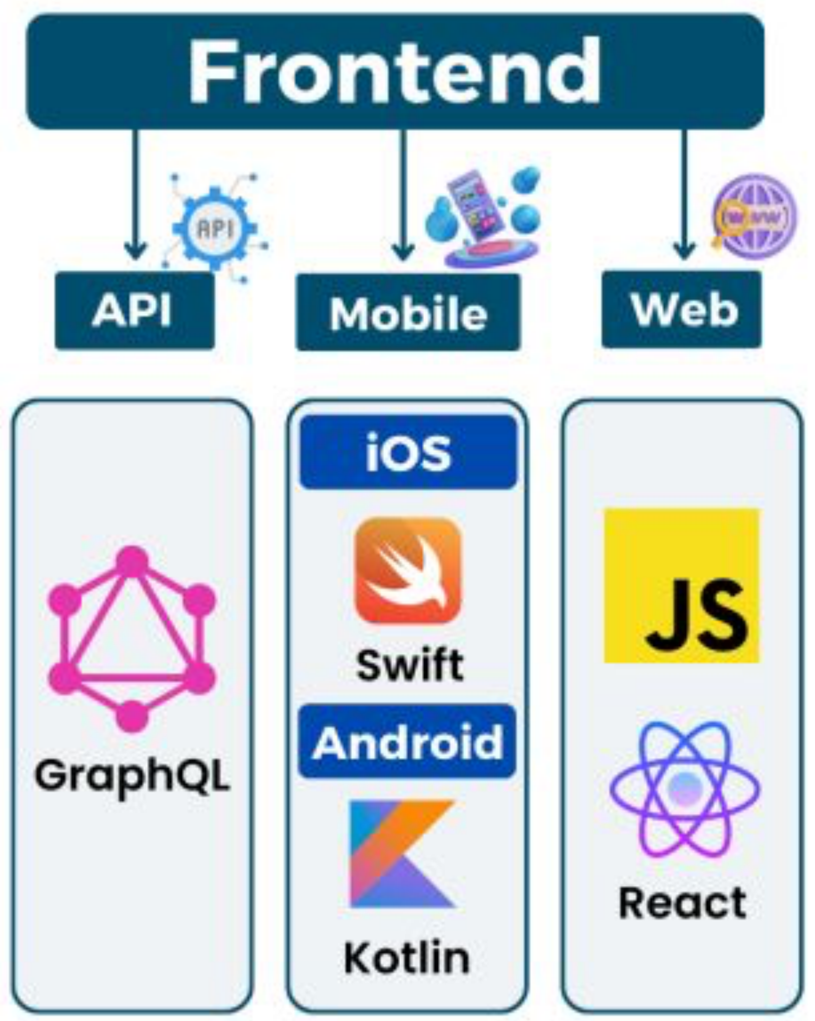


NETFLIX

https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv

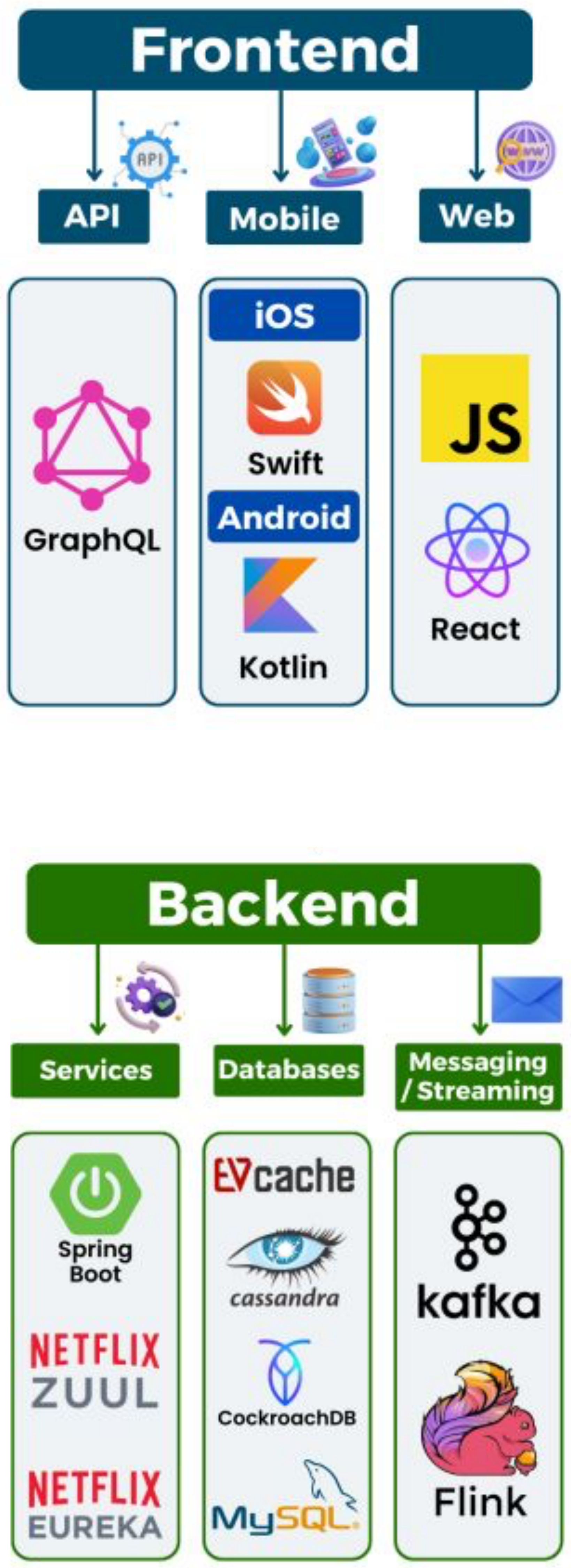

Frontend
API
Mobile
Web
GraphQL
iOS
Swift
Android
Kotlin
JS
React
Backend
Services
Spring Boot
NETFLIX ZUUL
NETFLIX EUREKA

NETFLIX



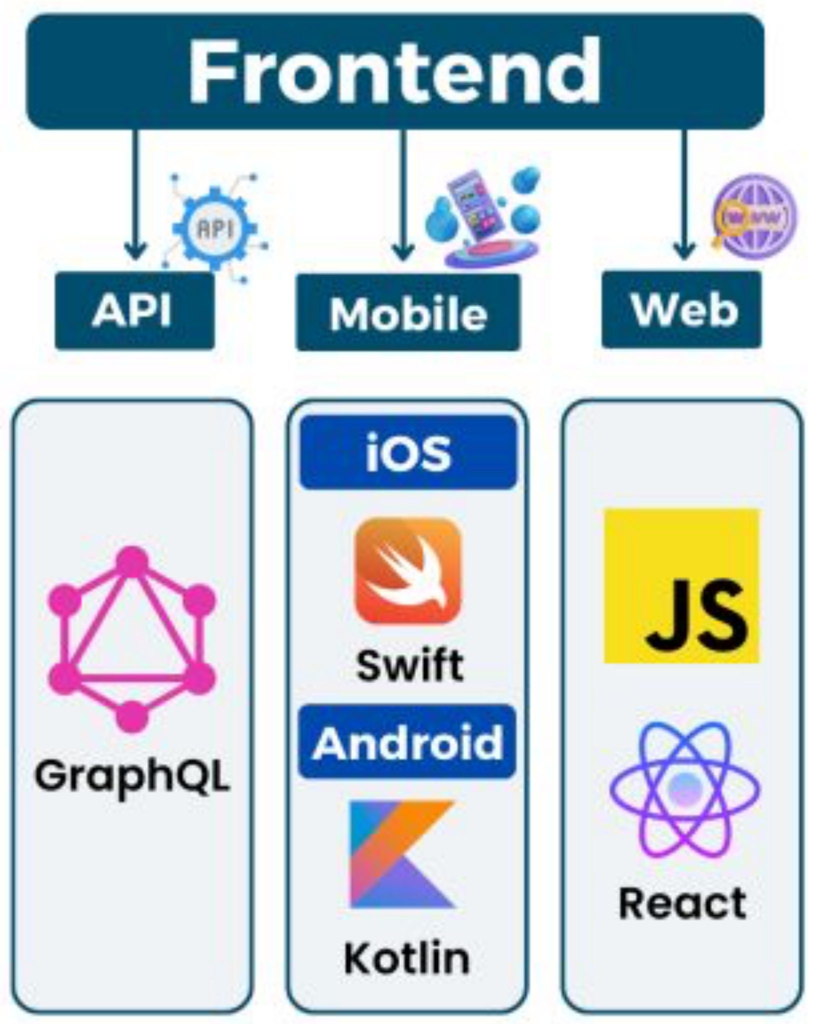
https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv

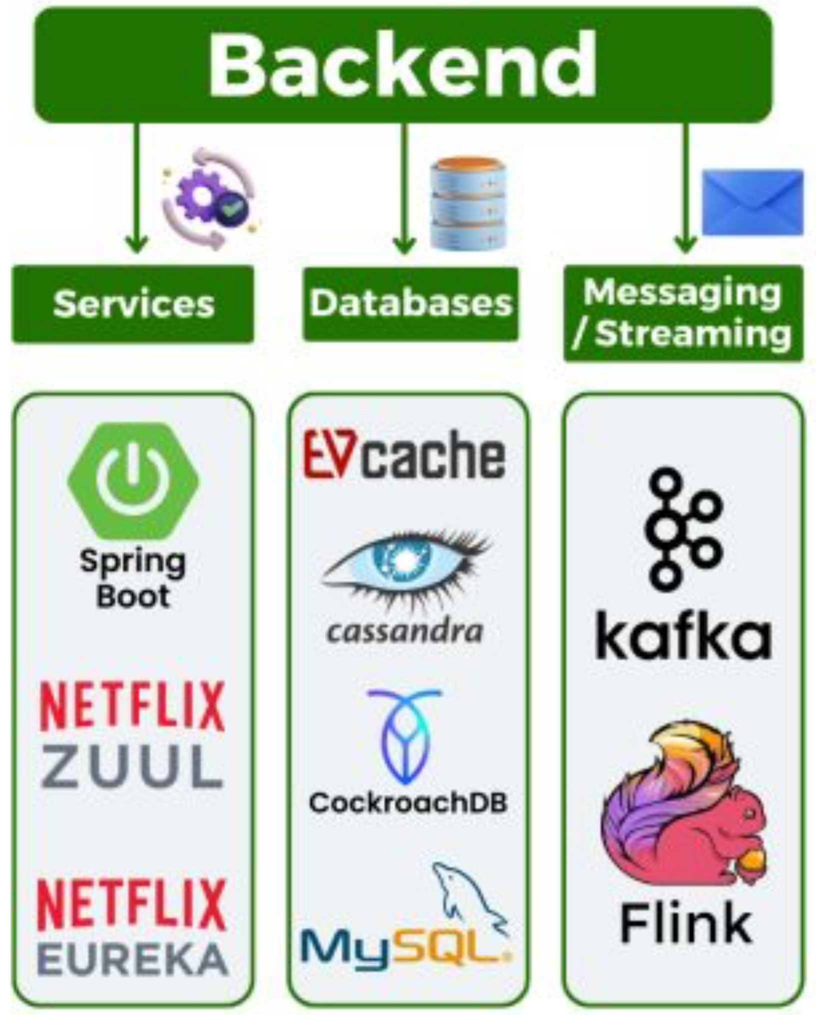
NETFLIX

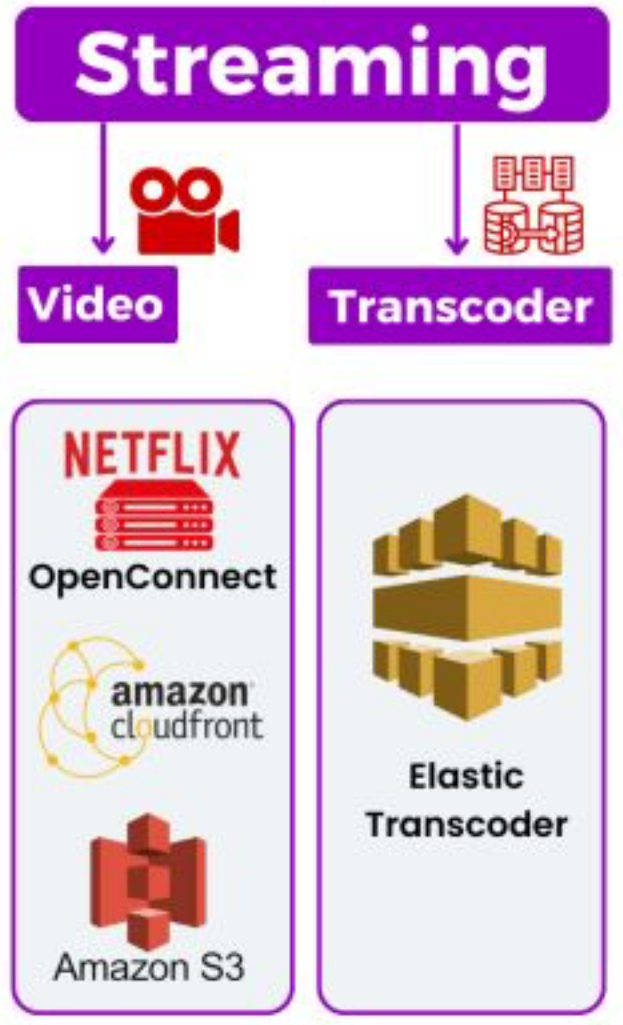


https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv

NETFLIX

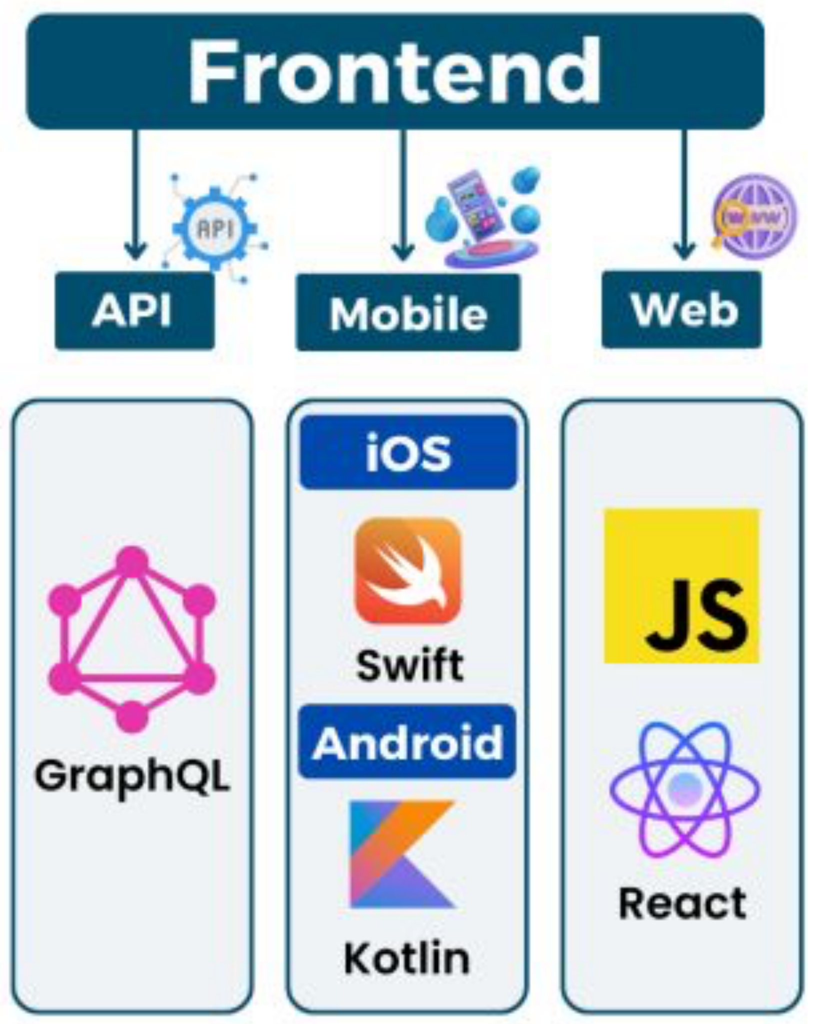


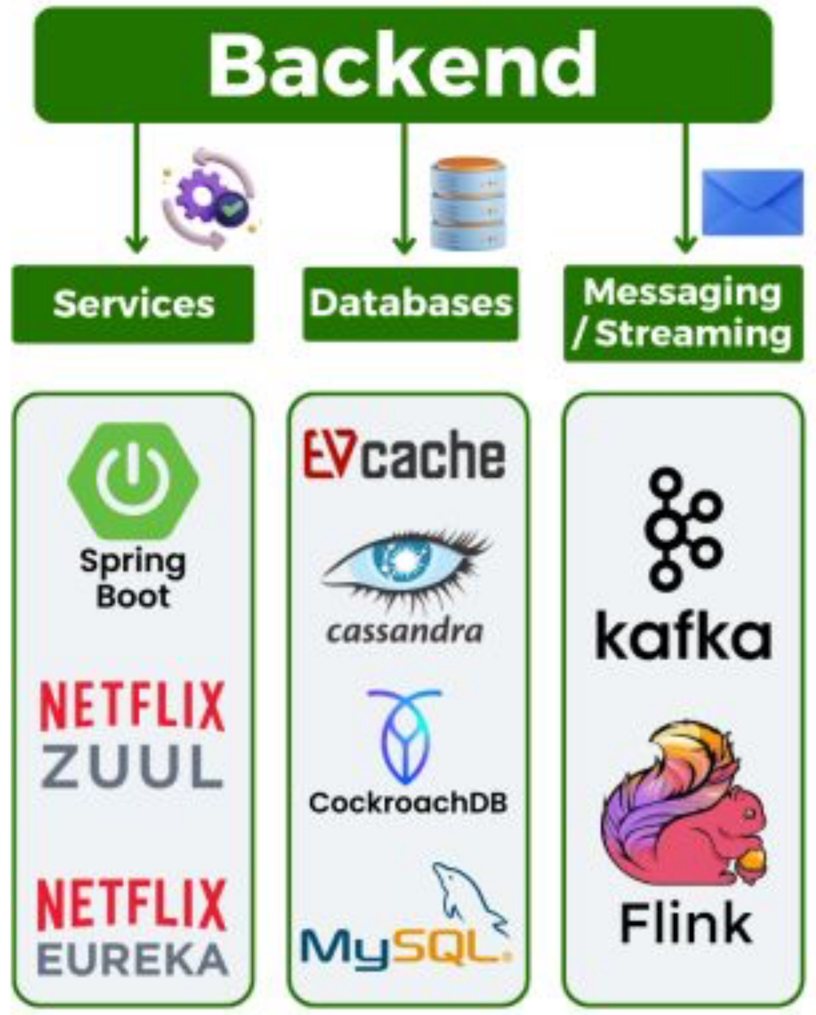


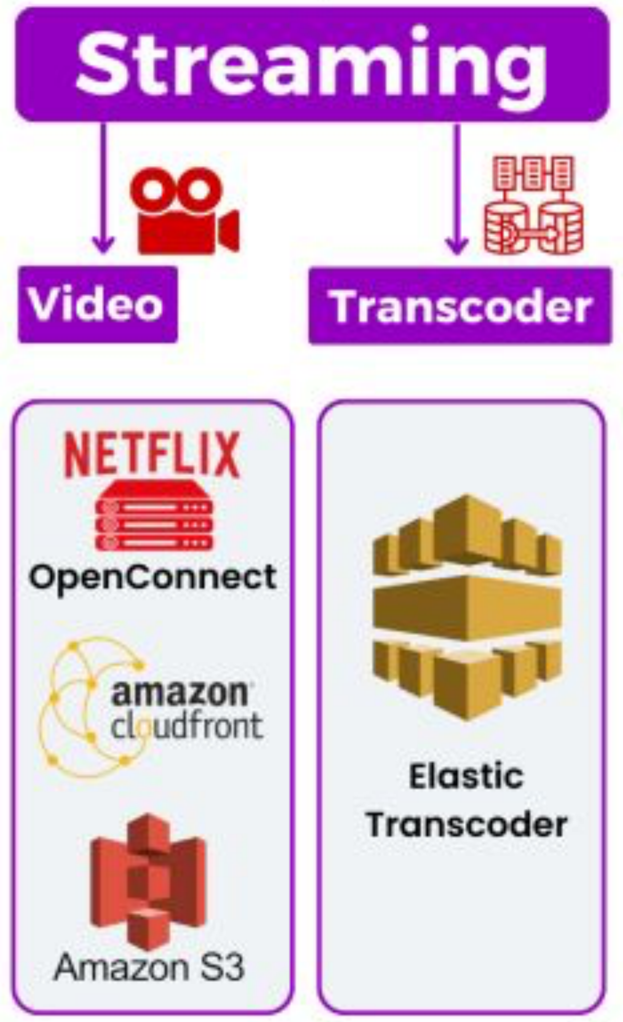


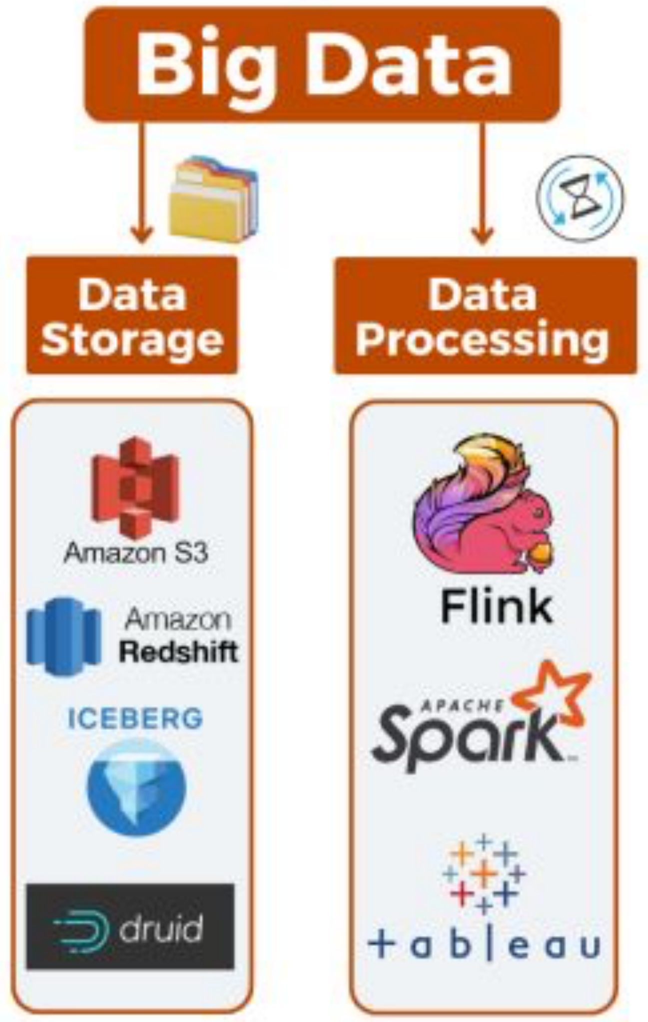
https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv

NETFLIX

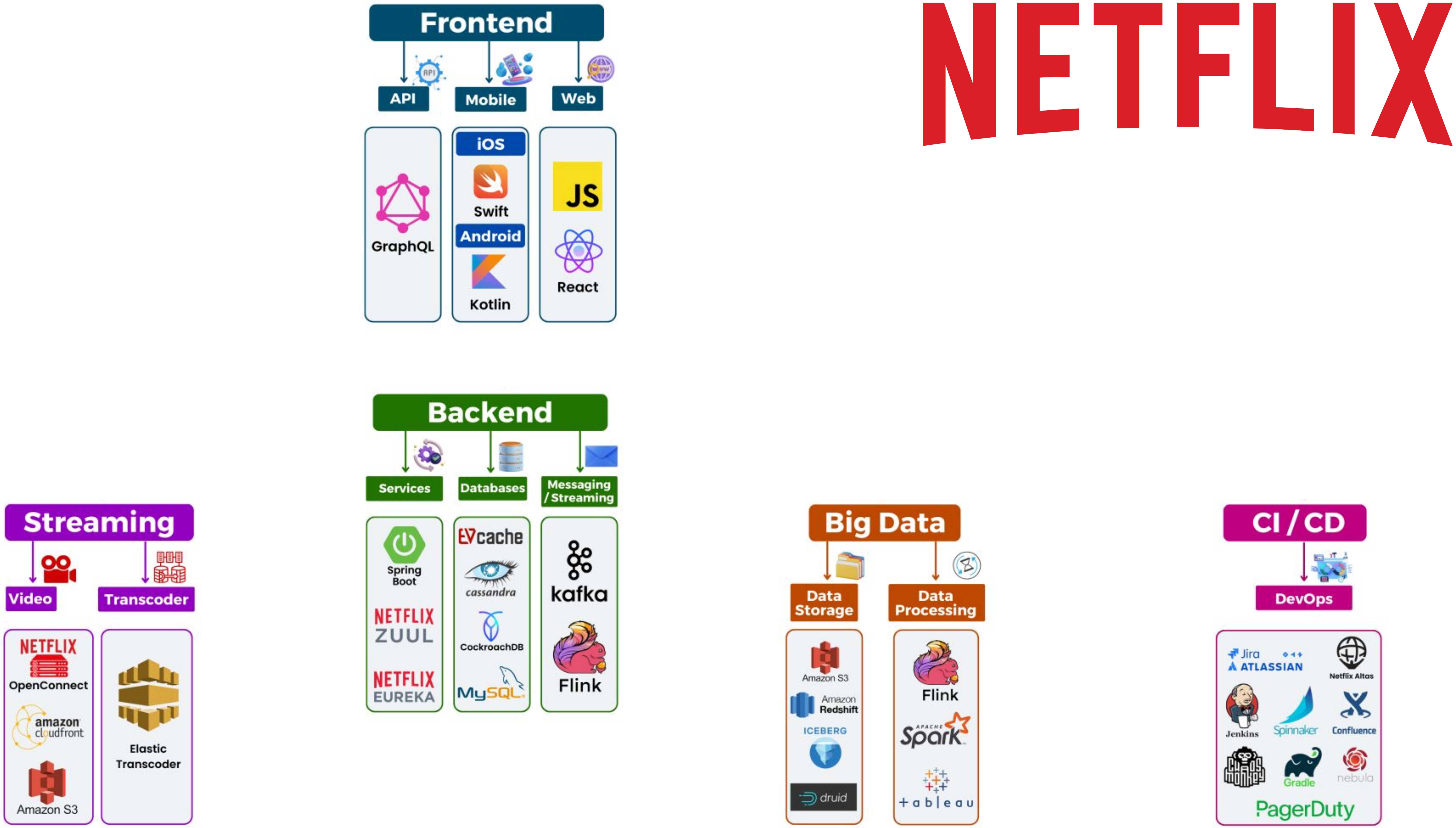


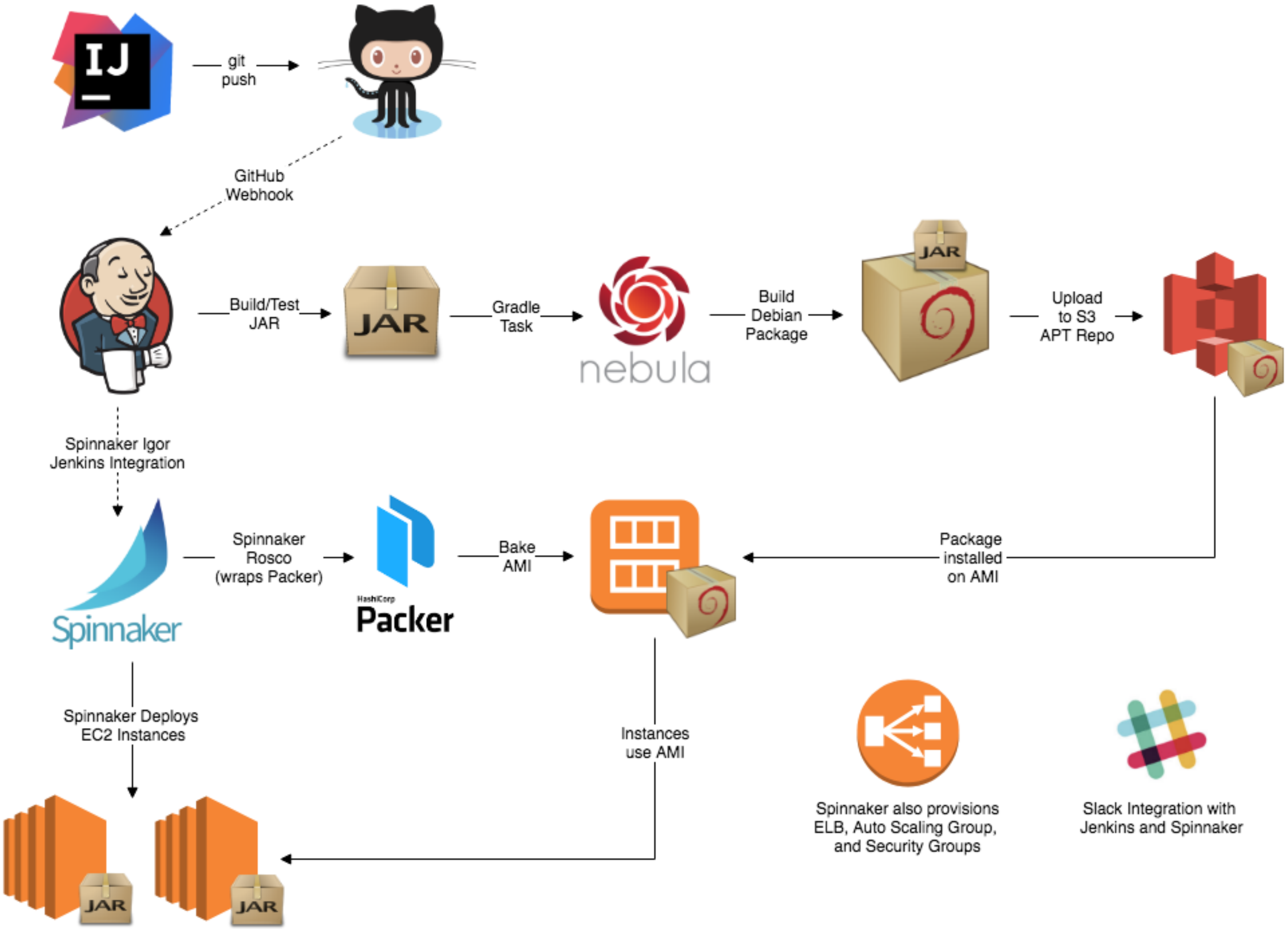




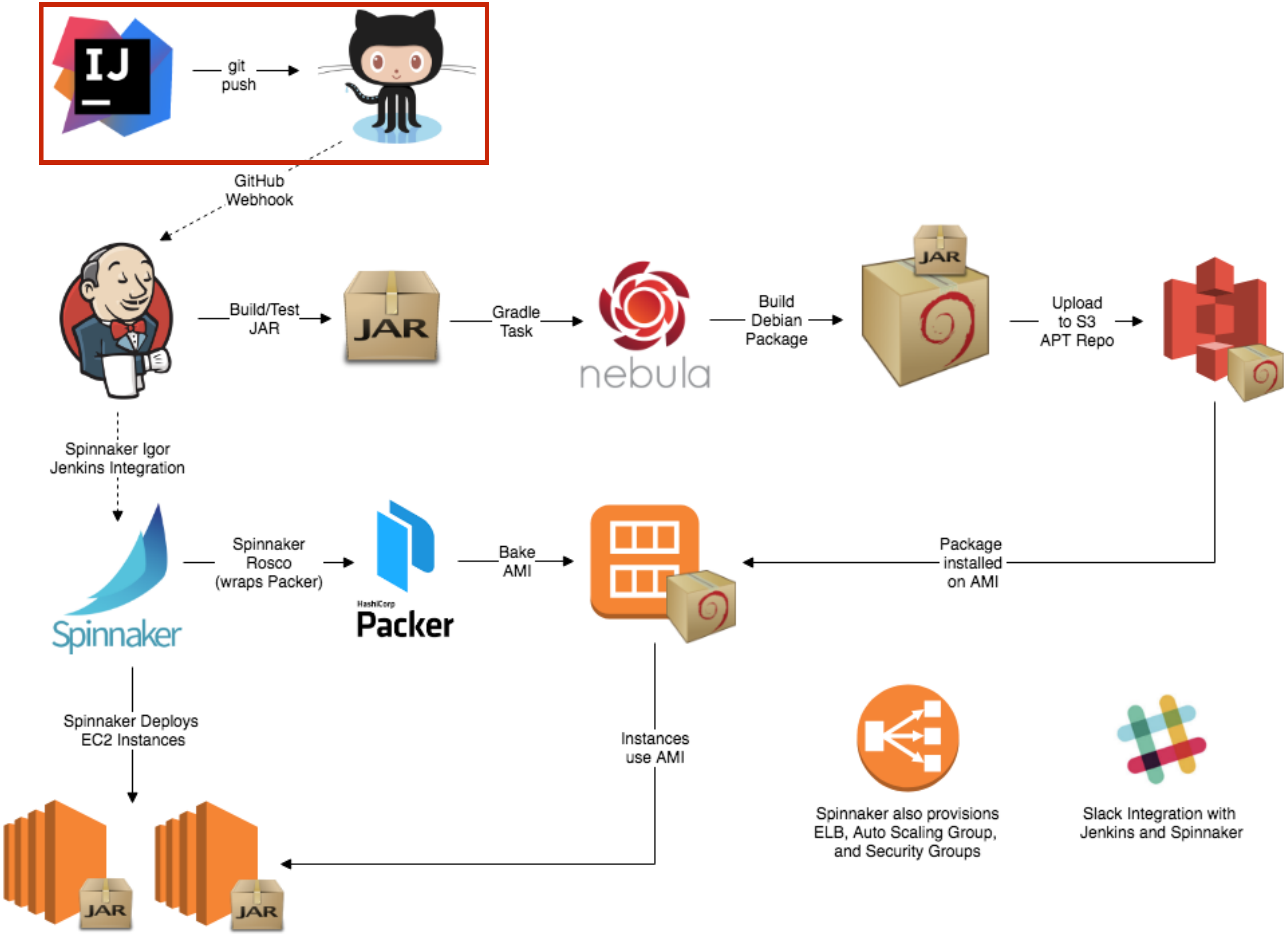


https://www.linkedin.com/posts/brijpandeyji_systemdesign-coding-softwareengineering-activity-7081232542984511489-pJFv

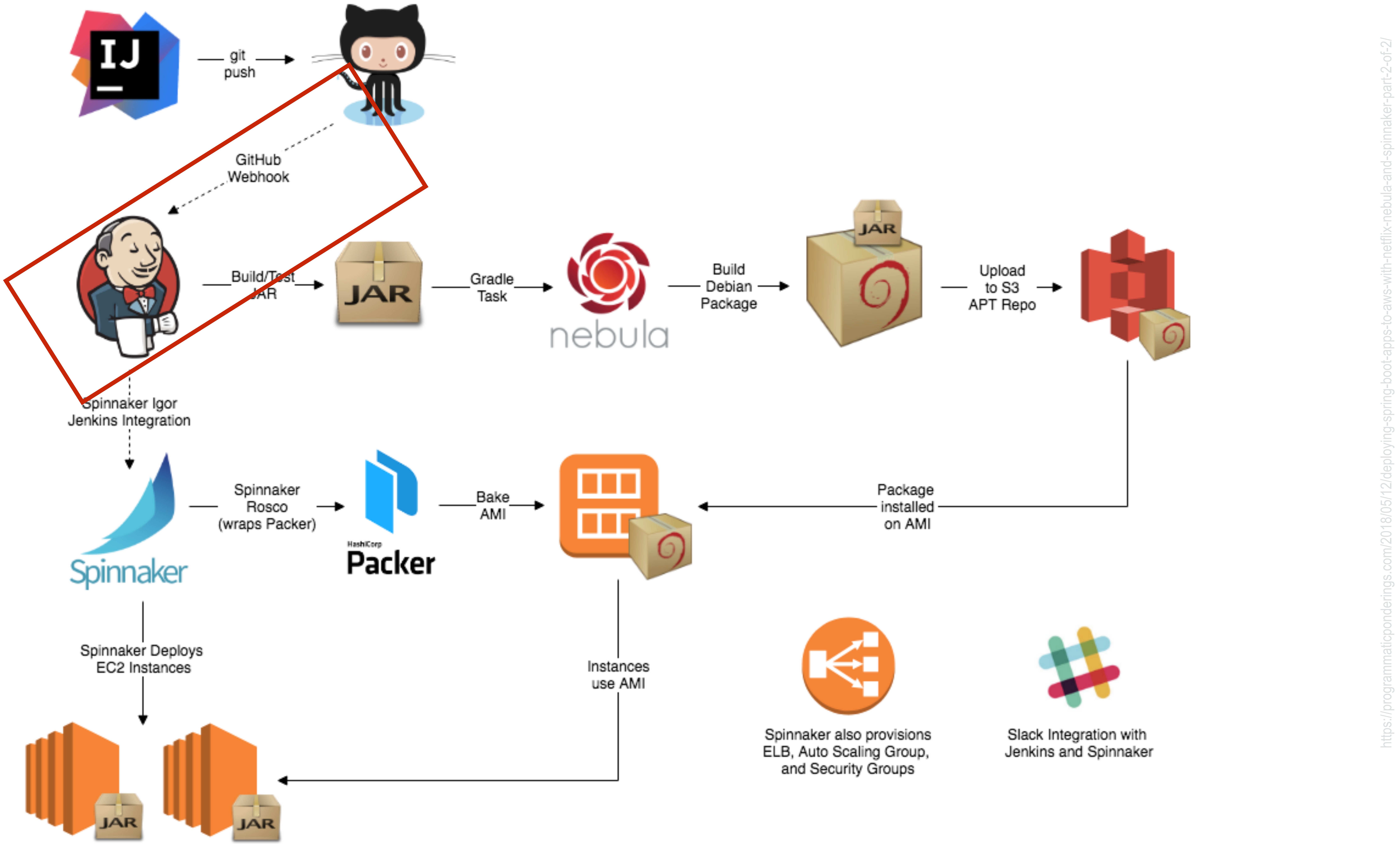

NETFLIX
Frontend
API
Mobile
Web
GraphQL
iOS
Swift
Android
Kotlin
JS
React
Backend
Services
Databases
Messaging / Streaming
Spring Boot
NETFLIX ZUUL
NETFLIX EUREKA
EVcache
cassandra
CockroachDB
MySQL
kafka
Flink
Streaming
Video
Transcoder
NETFLIX OpenConnect
amazon cloudfront
Amazon S3
Elastic Transcoder
Big Data
Data Storage
Data Processing
Amazon S3
Amazon Redshift
ICEBERG
druid
Flink
Spark
tableau
CI / CD
DevOps
Jira ATLASSIAN
Netflix Altas
Jenkins
Spinnaker
Confluence
Gradle
nebula
PagerDuty



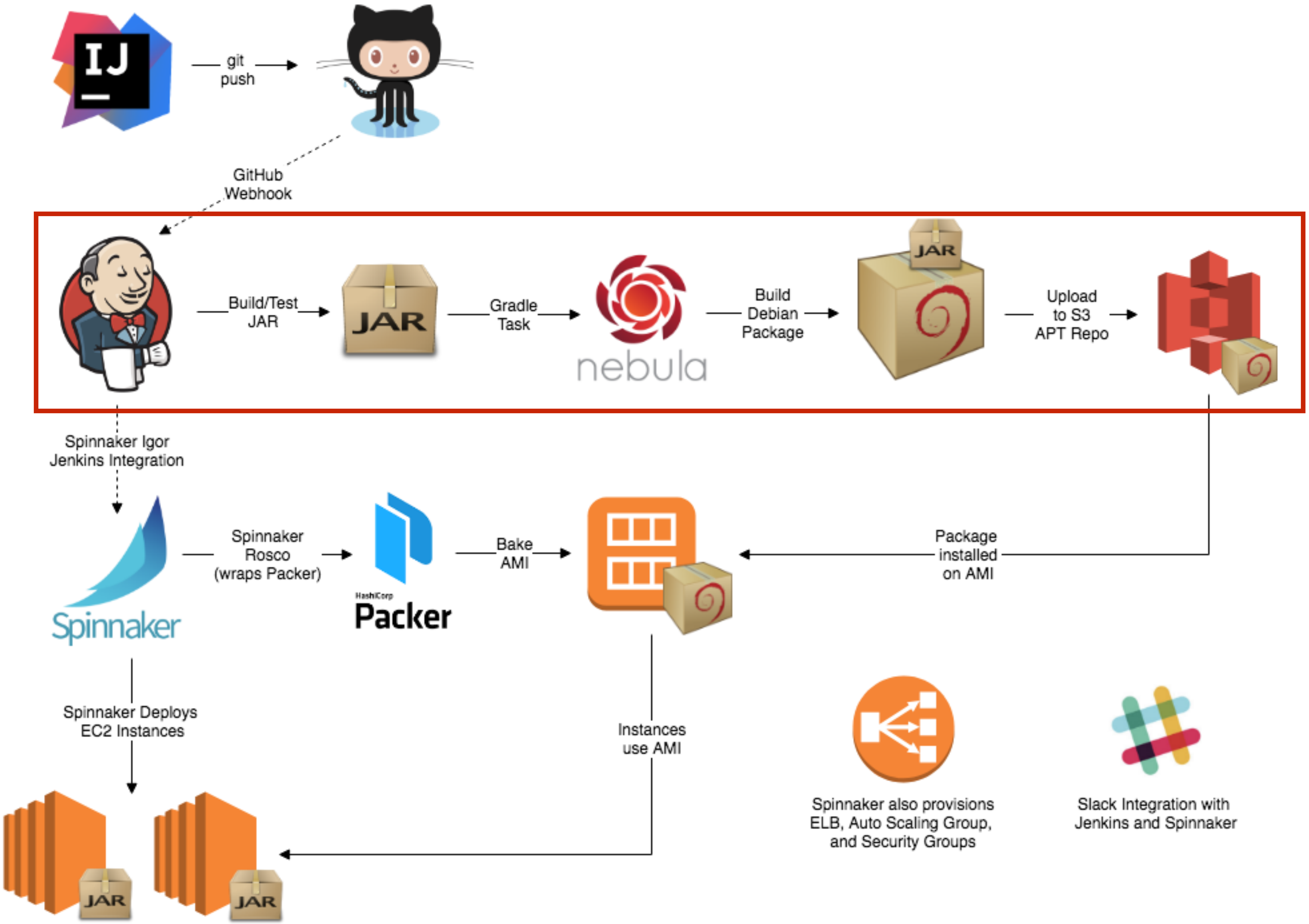
https://programmaticponderings.com/2018/05/12/deploying-spring-boot-apps-to-aws-with-netflix-nebula-and-spinnaker-part-2-of-2/

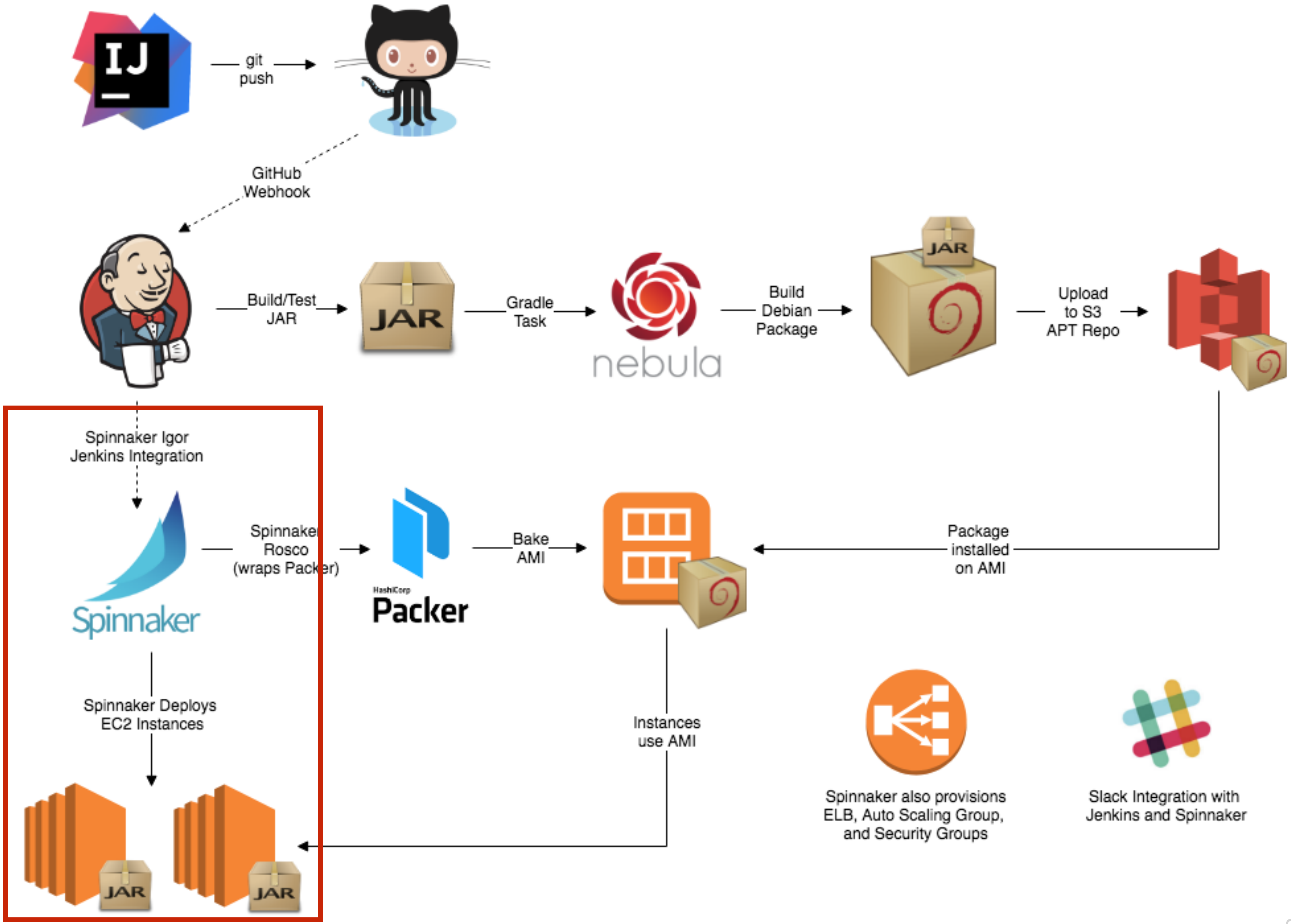


IJ
git push
GitHub Webhook
Build/Test JAR
JAR
Gradle Task
nebula
Build Debian Package
JAR
Upload to S3 APT Repo
Spinnaker Igor Jenkins Integration
Spinnaker
Spinnaker Rosco (wraps Packer)
HashiCorp
Packer
Bake AMI
Package installed on AMI
Spinnaker Deploys EC2 Instances
Instances use AMI
Spinnaker also provisions ELB, Auto Scaling Group, and Security Groups
Slack Integration with Jenkins and Spinnaker
JAR
JAR

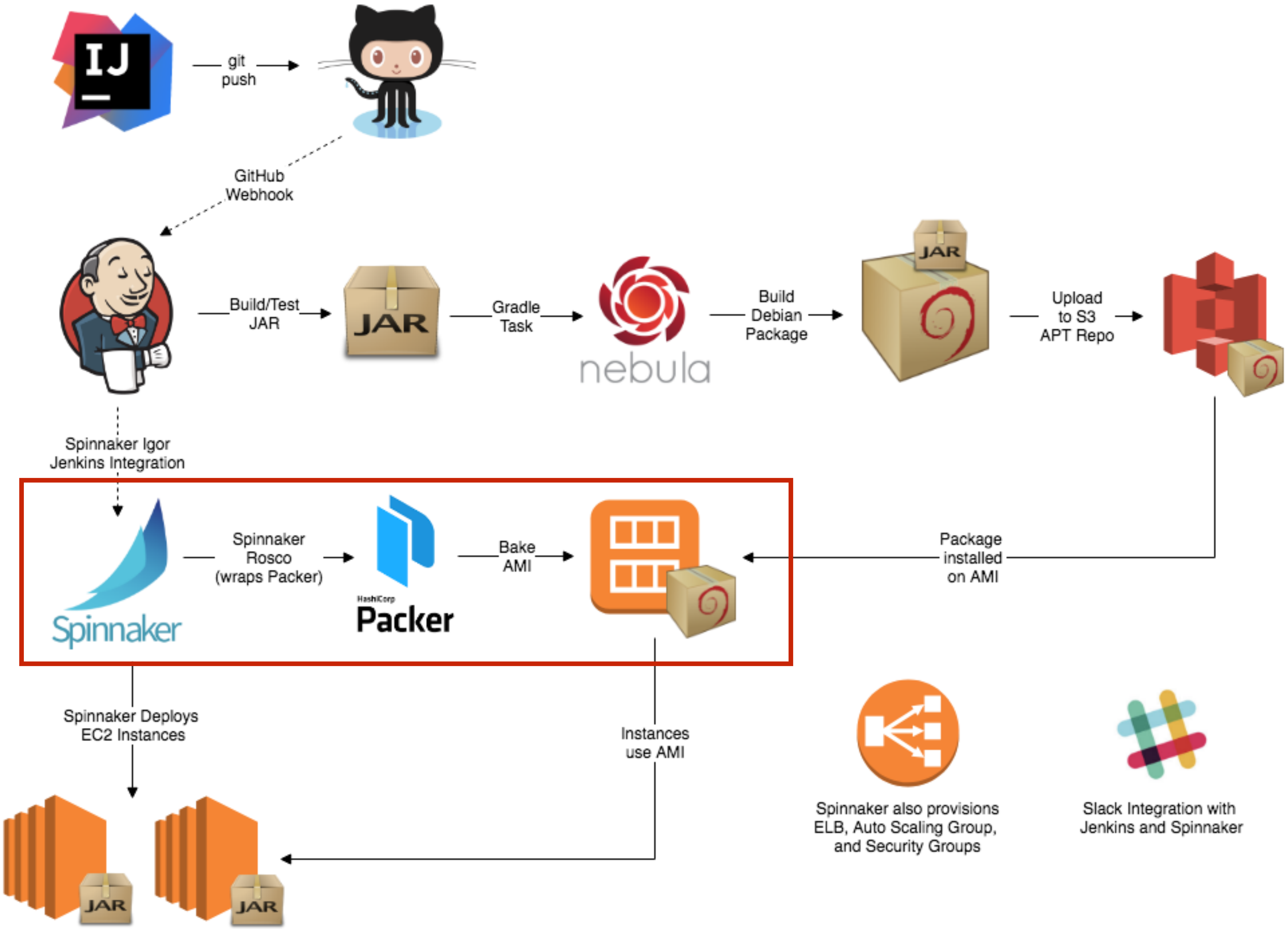


git push
GitHub Webhook
Build/Test JAR
JAR
Gradle Task
nebula
Build Debian Package
JAR
Upload to S3 APT Repo
Spinnaker Igor Jenkins Integration
Spinnaker Rosco (wraps Packer)
HashiCorp Packer
Bake AMI
Package installed on AMI
Spinnaker
Spinnaker Deploys EC2 Instances
Instances use AMI
Spinnaker also provisions ELB, Auto Scaling Group, and Security Groups
Slack Integration with Jenkins and Spinnaker
JAR
JAR
https://programmaticponderings.com/2018/05/12/deploying-spring-boot-apps-to-aws-with-netflix-nebula-and-spinnaker-part-2-of-2/

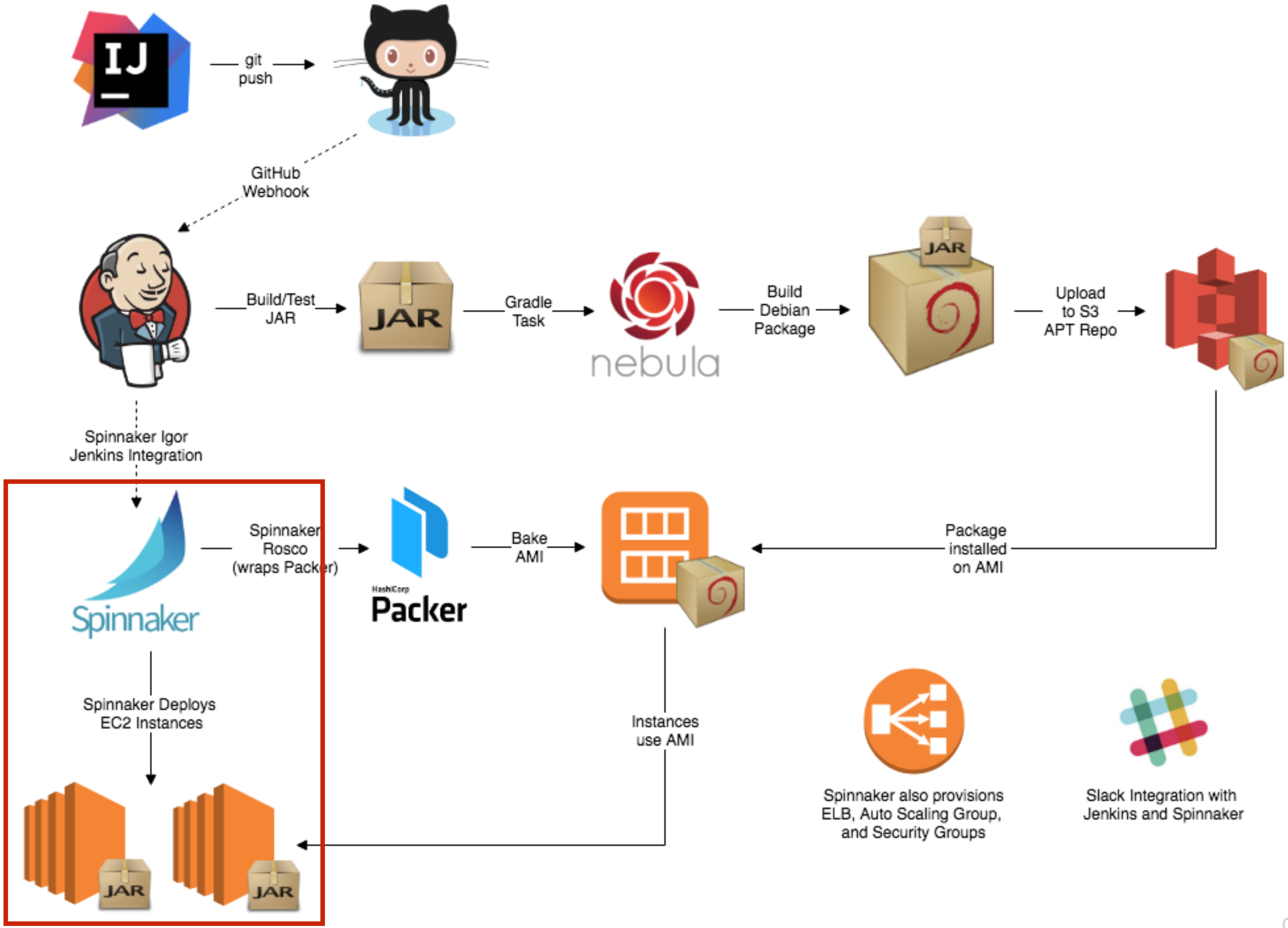

git push
GitHub Webhook
Build/Test JAR
JAR
Gradle Task
nebula
Build Debian Package
JAR
Upload to S3 APT Repo
Spinnaker Igor Jenkins Integration
Spinnaker
Spinnaker Rosco (wraps Packer)
HashiCorp Packer
Bake AMI
Package installed on AMI
Spinnaker Deploys EC2 Instances
Instances use AMI
Spinnaker also provisions ELB, Auto Scaling Group, and Security Groups
Slack Integration with Jenkins and Spinnaker
JAR
JAR

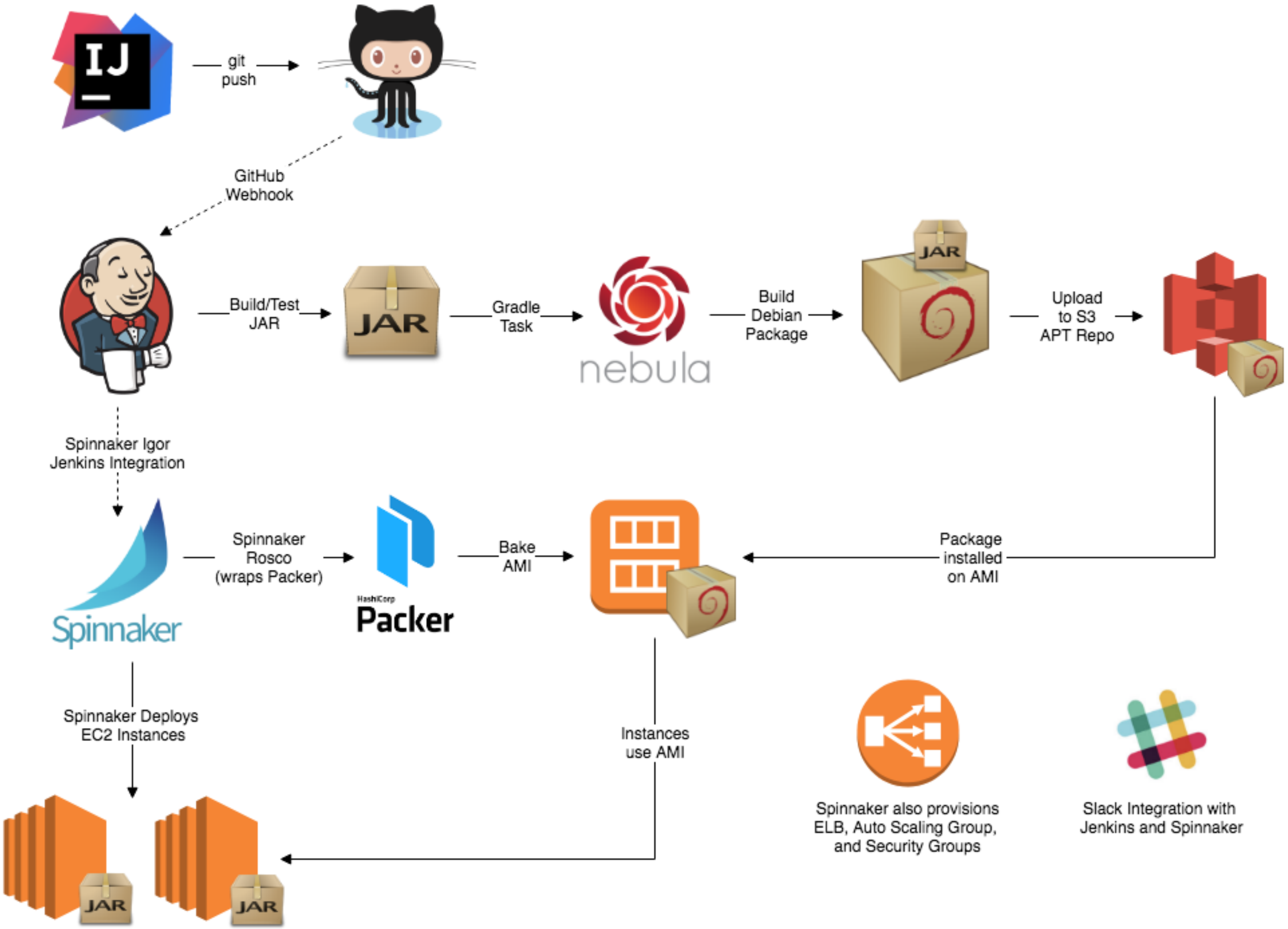


https://programmaticponderings.com/2018/05/12/deploying-spring-boot-apps-to-aws-with-netflix-nebula-and-spinnaker-part-2-of-2/

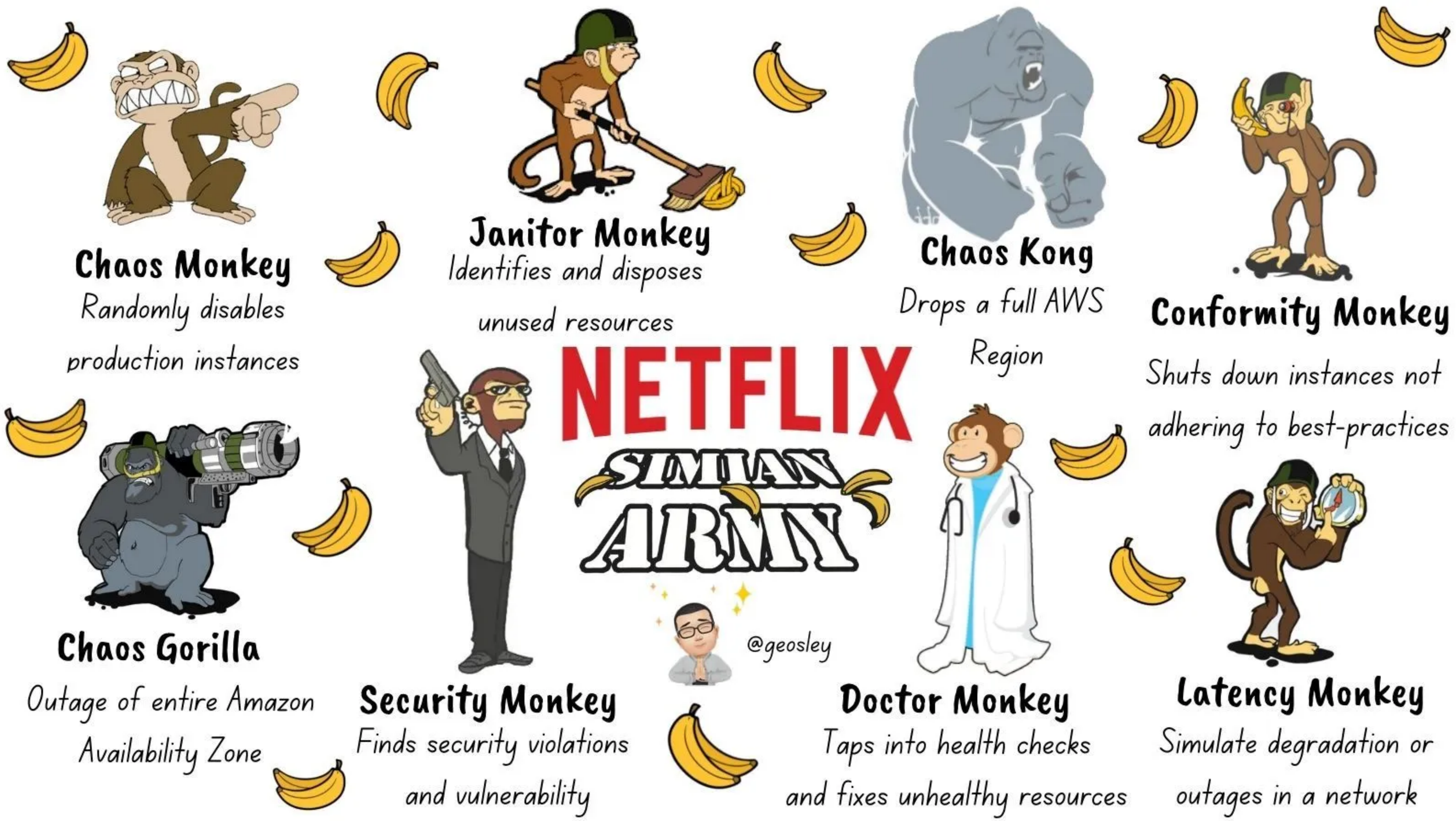


https://programmaticponderings.com/2018/05/12/deploying-spring-boot-apps-to-aws-with-netflix-nebula-and-spinnaker-part-2-of-2/


git push
GitHub Webhook
Build/Test JAR
JAR
Gradle Task
nebula
Build Debian Package
Upload to S3 APT Repo
Spinnaker Igor Jenkins Integration
Spinnaker Rosco (wraps Packer)
Spinnaker
HashiCorp
Packer
Bake AMI
Package installed on AMI
Spinnaker Deploys EC2 Instances
Instances use AMI
Spinnaker also provisions ELB, Auto Scaling Group, and Security Groups
Slack Integration with Jenkins and Spinnaker



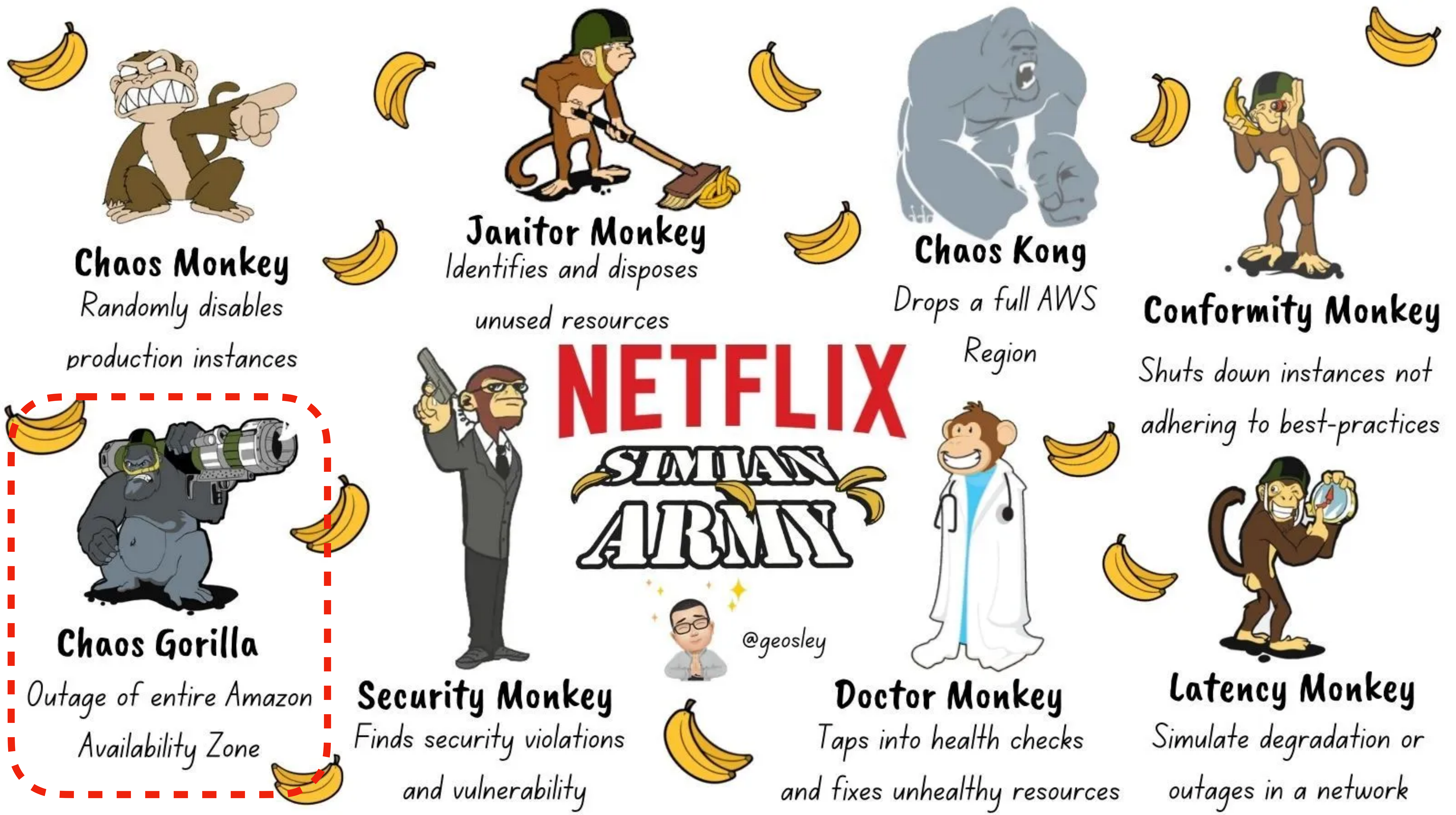
https://programmaticponderings.com/2018/05/12/deploying-spring-boot-apps-to-aws-with-netflix-nebula-and-spinnaker-part-2-of-2/



https://medium.com/@paulosejithu/unleashing-the-chaos-monkey-netflixs-ingenious-approach-to-building-resilient-systems-7e715b5f1b0f

https://medium.com/@paulosejithu/unleashing-the-chaos-monkey-netflixs-ingenious-approach-to-building-resilient-systems-7e715b5f1b0f

https://medium.com/@paulosejithu/unleashing-the-chaos-monkey-netflixs-ingenious-approach-to-building-resilient-systems-7e715b5f1b0f



https://medium.com/@paulosejithu/unleashing-the-chaos-monkey-netflix-ingenious-approach-to-building-resilient-systems-7e715b5f1b0f

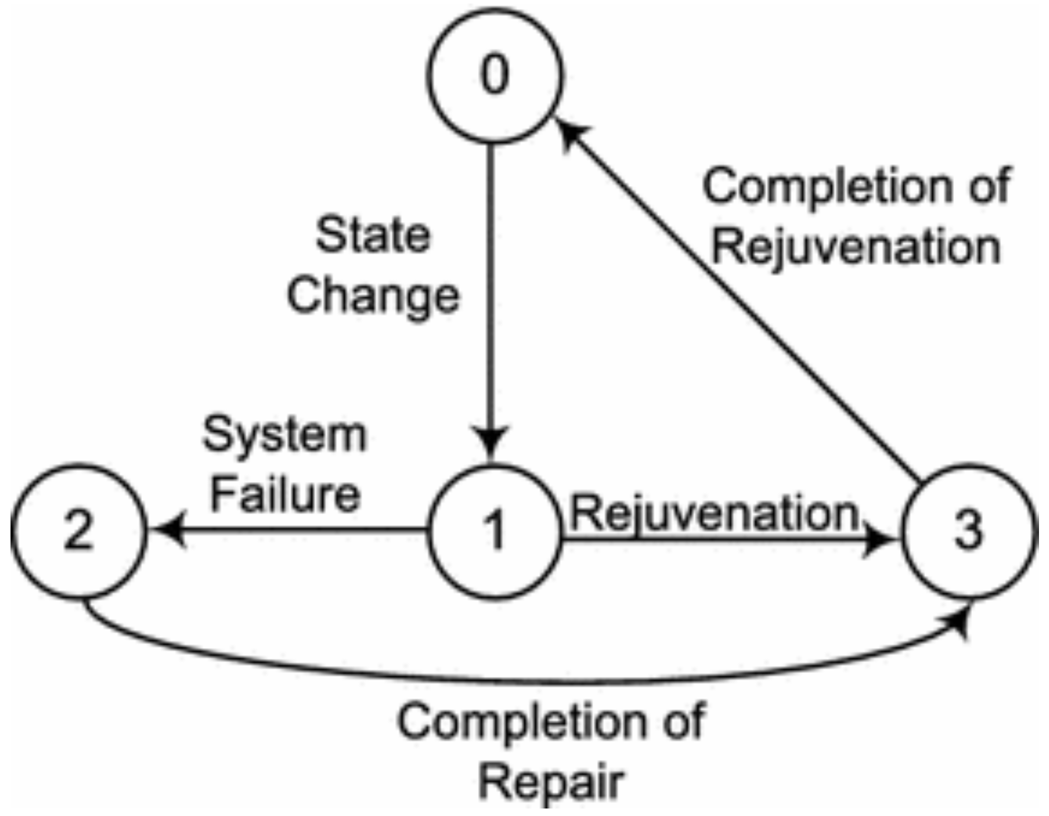
https://medium.com/@paulosejithu/unleashing-the-chaos-monkey-netflixs-ingenious-approach-to-building-resilient-systems-7e715b5f1b0f

Software rejuvenation

- Goal: clean up state to prevent accumulation of errors
  - *Insight: Reboot as a prophylactic*
  - *Does nothing about defects, but reduces probability of turning errors into failures*

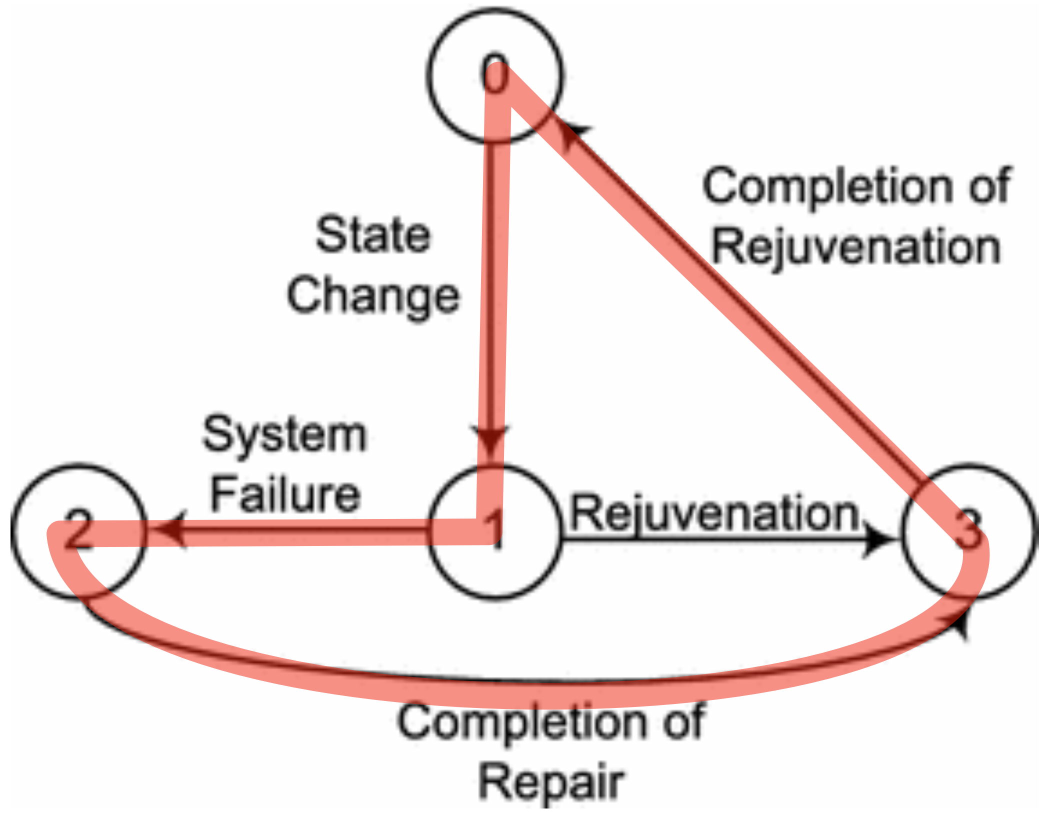
Software rejuvenation

- Goal: clean up state to prevent accumulation of errors
  - *Insight: Reboot as a prophylactic*
  - *Does nothing about defects, but reduces probability of turning errors into failures*
- Turns unplanned downtime into planned downtime
  - *Dynamic version of "preventive maintenance"*
  - *Release leaked resources, wipe out data corruption, ...*



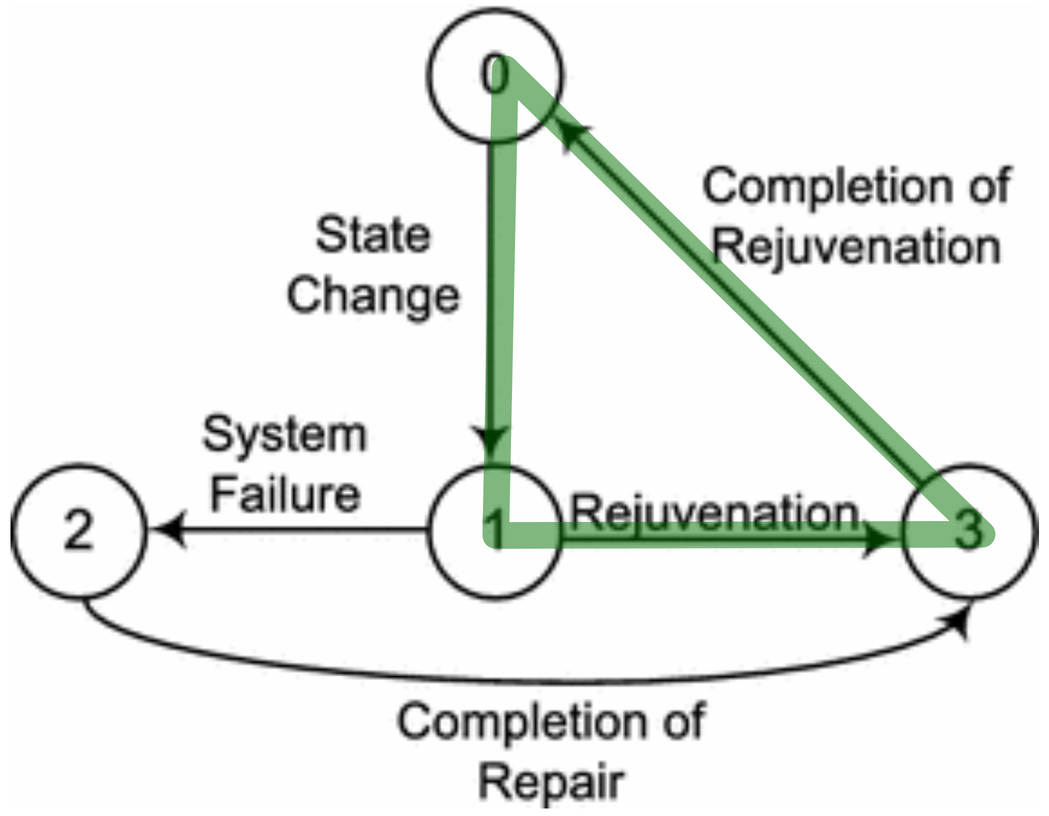
Software rejuvenation

- Goal: clean up state to prevent accumulation of errors
  - *Insight: Reboot as a prophylactic*
  - *Does nothing about defects, but reduces probability of turning errors into failures*
- Turns unplanned downtime into planned downtime
  - *Dynamic version of "preventive maintenance"*
  - *Release leaked resources, wipe out data corruption, ...*



Software rejuvenation

- Goal: clean up state to prevent accumulation of errors
  - *Insight: Reboot as a prophylactic*
  - *Does nothing about defects, but reduces probability of turning errors into failures*
- Turns unplanned downtime into planned downtime
  - *Dynamic version of "preventive maintenance"*
  - *Release leaked resources, wipe out data corruption, ...*



How to reduce unavailability by 10× ?

$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF}\ \uparrow\times 10}$$

How to reduce unavailability by 10× ?

$$\text{Unavailability} \cong \frac{\text{MTTR} \downarrow \div 10}{\text{MTTF}}$$

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
    - Automation
    - Prediction/anticipation
    - Trade-offs between FPs and FNs

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
  - Automation
  - Prediction/anticipation
  - Trade-offs between FPs and FNs

Detection/Prediction says...

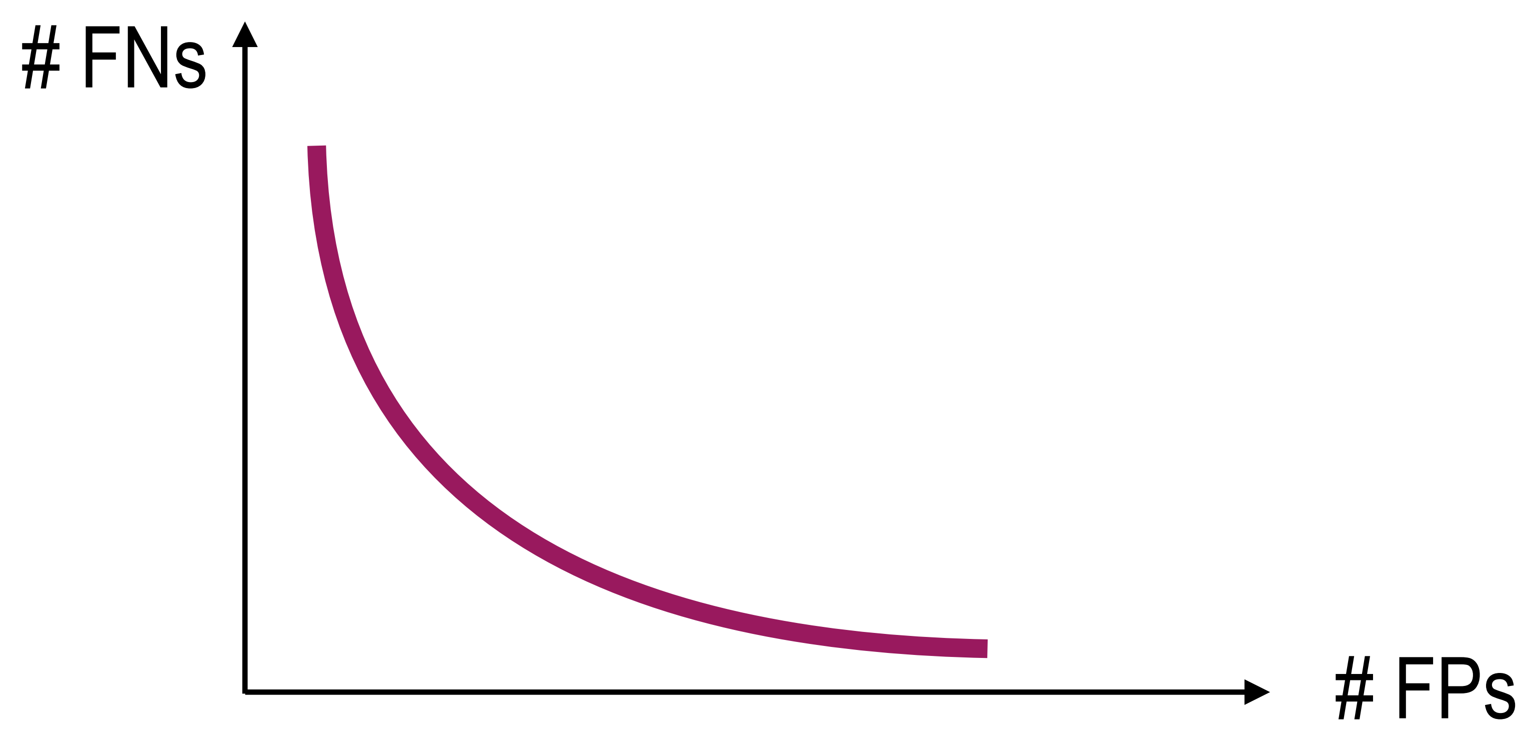
Truth is...		Failure	No Failure
	No Failure	FP	TN
	Failure	TP	FN

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
  - Automation
  - Prediction/anticipation
  - Trade-offs between FPs and FNs

Detection/Prediction says...

Truth is...	Failure	No Failure
No Failure	FP	TN
Failure	TP	FN

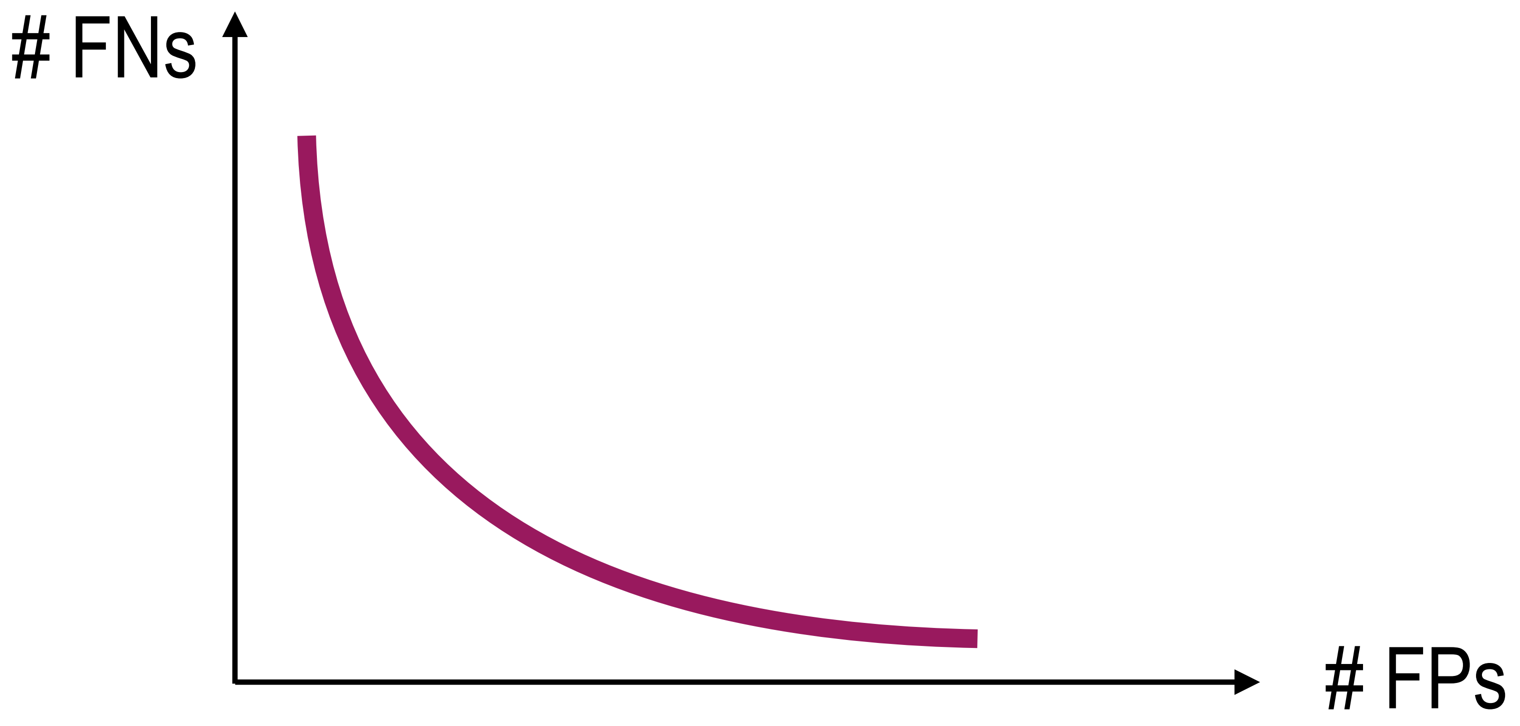


Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
    - Automation
    - Prediction/anticipation
    - Trade-offs between FPs and FNs
- How to reduce $T_{diagnose}$?
    - Lots of instrumentation, ML, ...
    - Also a function of what recovery mechanism have available

Detection/Prediction says...

Truth is...	Failure	No Failure
No Failure	FP	TN
Failure	TP	FN

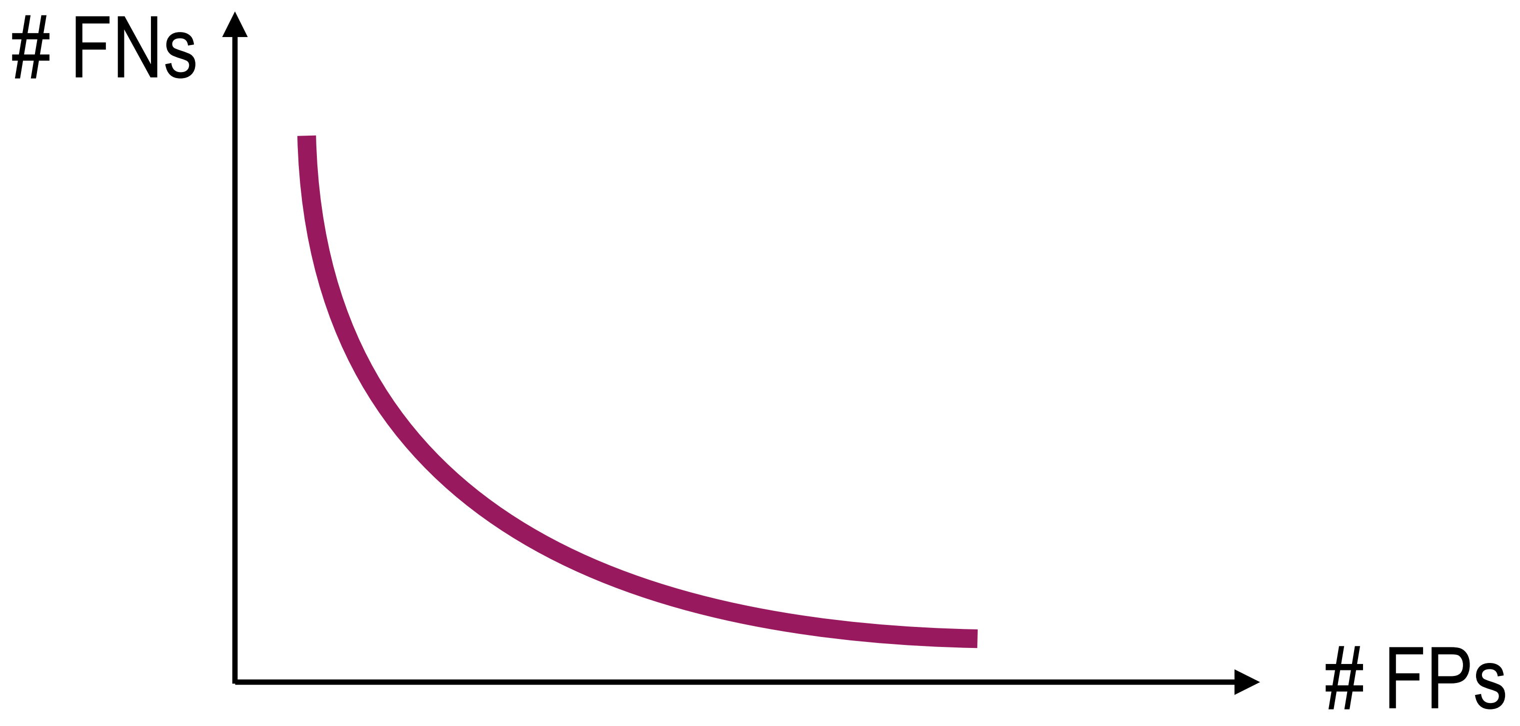


Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
  - Automation
  - Prediction/anticipation
  - Trade-offs between FPs and FNs
- How to reduce $T_{diagnose}$?
  - Lots of instrumentation, ML, ...
  - Also a function of what recovery mechanism have available
- How to reduce T_{repair}?
  - Mostly app-specific
  - Reboot is universal

Detection/Prediction says...

Truth is...	Failure	No Failure
No Failure	FP	TN
Failure	TP	FN



Leased resources

- Goal: avoid resource leakage without fancy resource tracking

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*
- Requests carry TTL => automatically purged when TTL runs out

Reboot-based Recovery

Reboot-based Recovery

https://steamcommunity.com/sharedfiles/filedetails/?id=214241765
https://www.sciencedirect.com/topics/nursing-and-health-professions/scalpel
https://aknextphase.com/crossing-the-job-search-finish-line/better-things-are-coming/

Reboot-based Recovery

https://steamcommunity.com/sharedfiles/filedetails/?id=214241765
https://www.sciencedirect.com/topics/nursing-and-health-professions/scalpel
https://aknextphase.com/crossing-the-job-search-finish-line/better-things-are-coming/

Step 1: Modularize system into fine-grained components

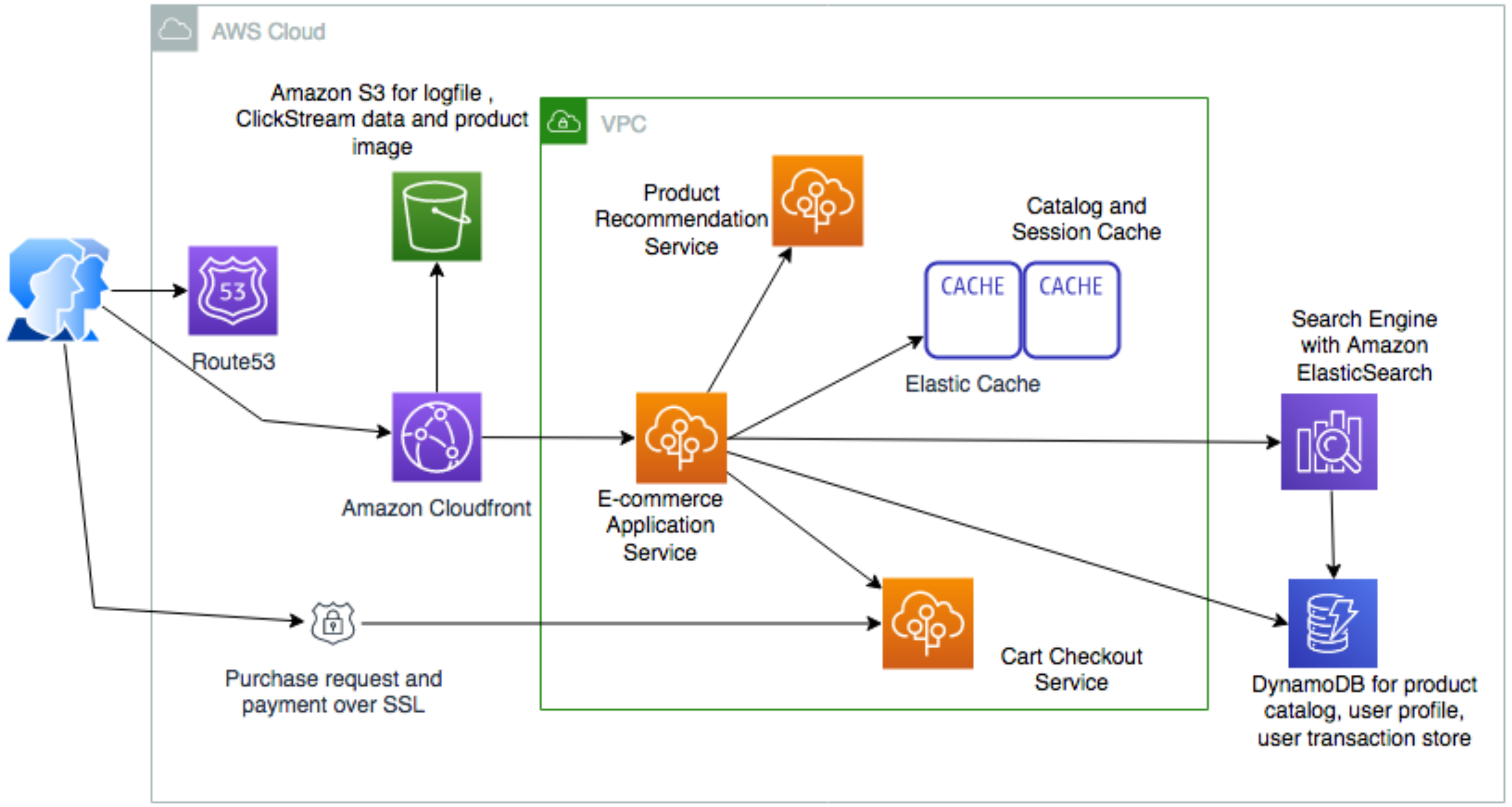
- Components with individual loci of control
  - *Well defined interfaces*
  - *Small in terms of program logic and startup time*

https://twitter.com/Werner/status/741673514567143424/photo/1

https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

Step 1: Modularize system into fine-grained components

- Components with individual loci of control
  - *Well defined interfaces*
  - *Small in terms of program logic and startup time*

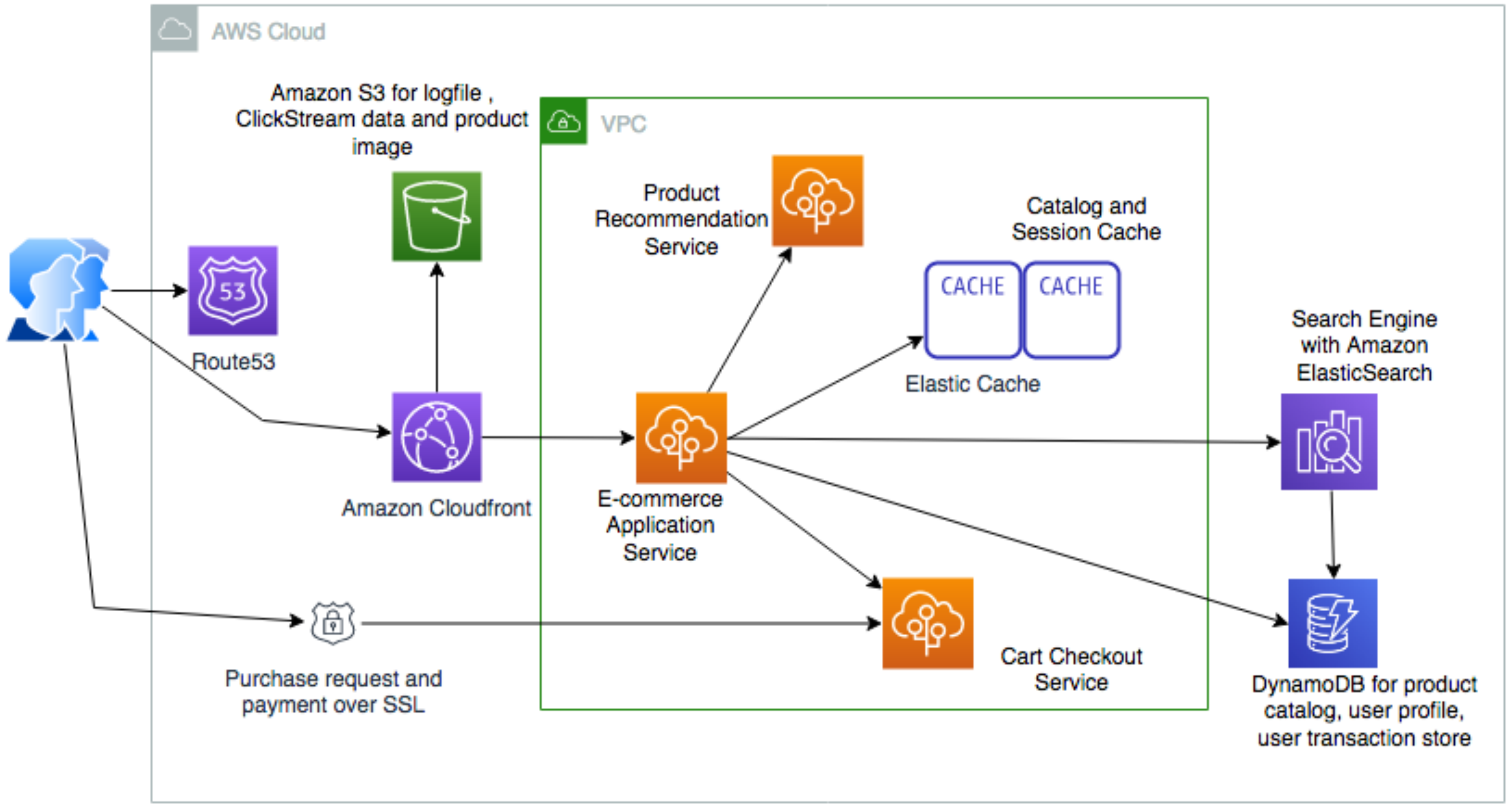


https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

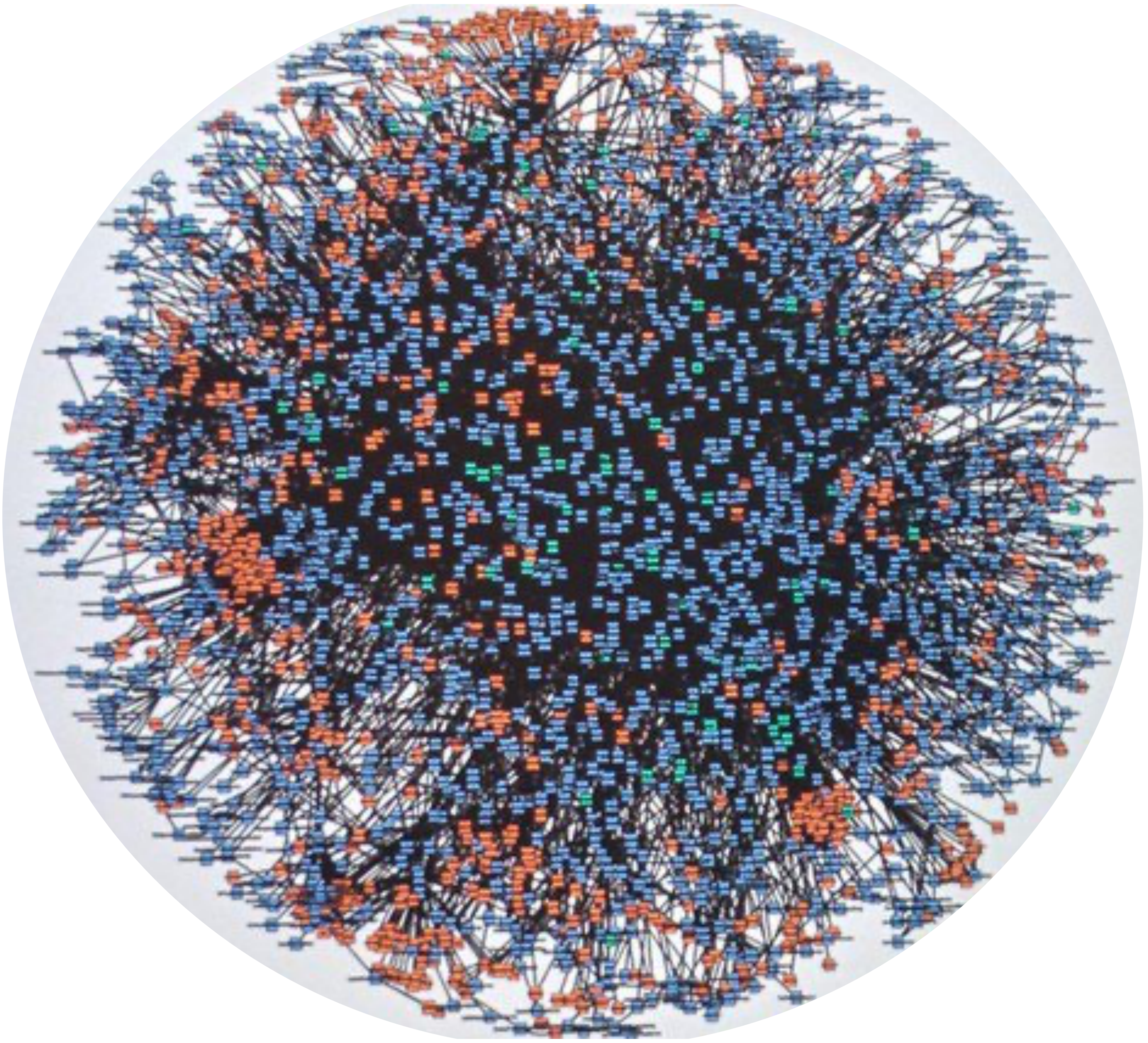
https://twitter.com/Werner/status/741673514567143424/photo/1

Step 1: Modularize system into fine-grained components

- Components with individual loci of control
  - *Well defined interfaces*
  - *Small in terms of program logic and startup time*



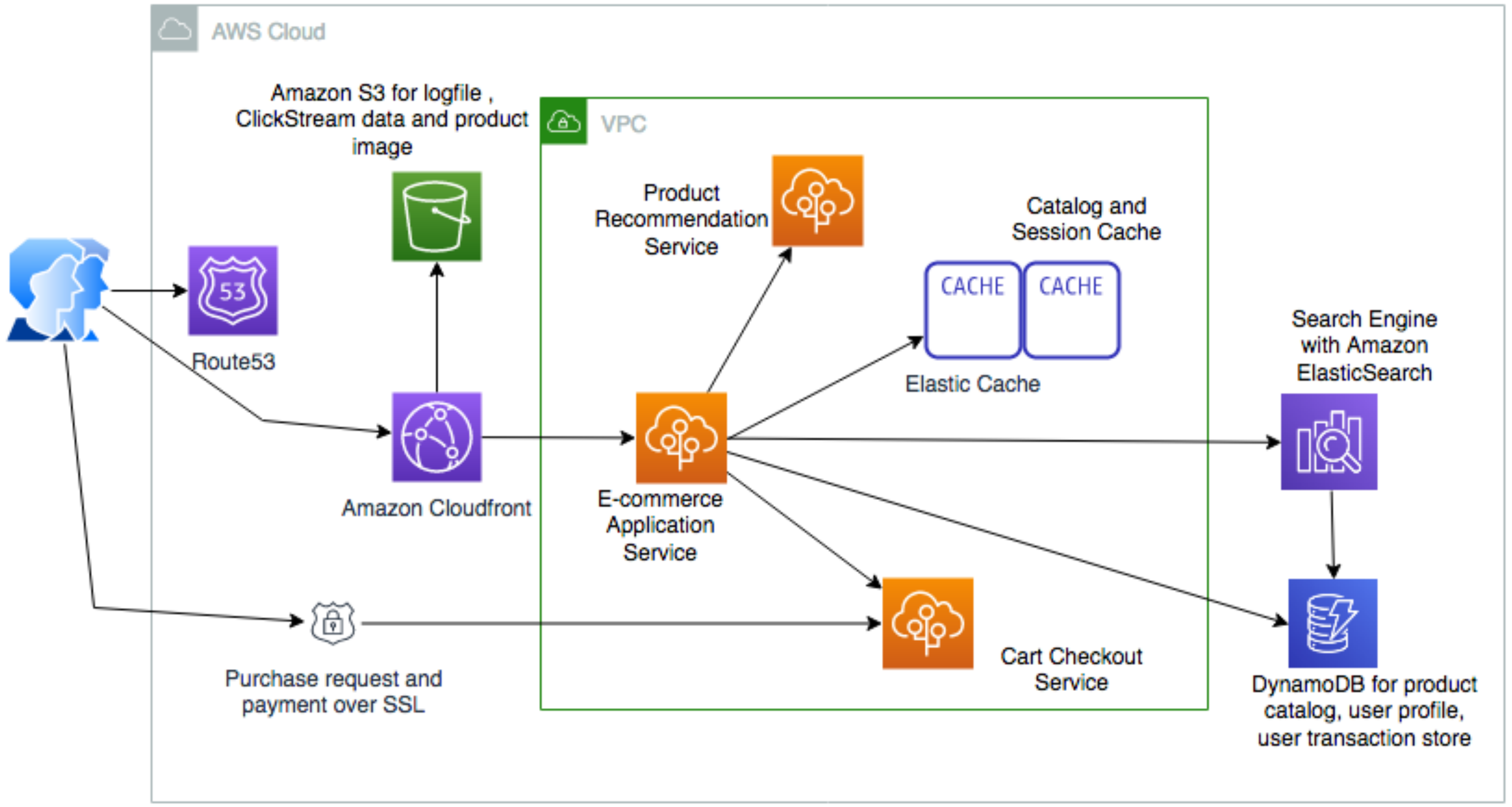
https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture



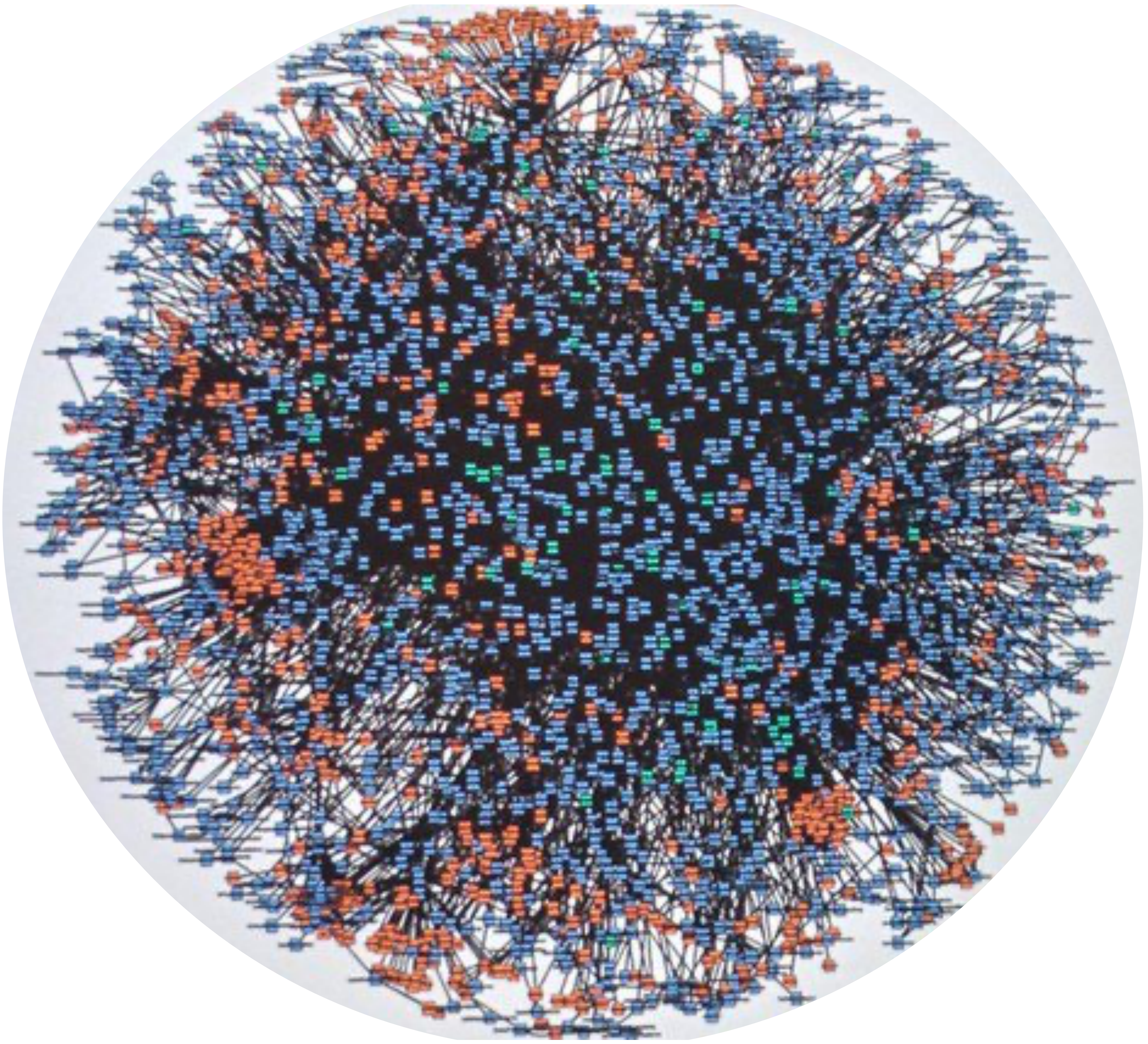
https://twitter.com/Werner/status/741673514567143424/photo/1

Step 1: Modularize system into fine-grained components

- Components with individual loci of control
  - *Well defined interfaces*
  - *Small in terms of program logic and startup time*
- $T_{reboot} = T_{restart} + T_{initialization}$



https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture



https://twitter.com/Werner/status/741673514567143424/photo/1

Leased resources

- Goal: avoid resource leakage without fancy resource tracking

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*
- Requests carry TTL => automatically purged when TTL runs out

Problems with microrebooting

1. Component reboot can induce state corruption/inconsistency that persists across microrebooting
2. A component I depend on (i.e., need to call) is microrebooting when I need it
3. How to avoid resource leakage after arbitrary microrebooting?
4. How does a component reintegrate after microrebooting?

...

State segregation

- Goal: prevent microreboot from inducing corruption or state inconsistency

State segregation

- Goal: prevent microreboot from inducing corruption or state inconsistency
- Keep all state that should survive a reboot in dedicated state stores
  - *stores located outside the application ...*
  - *... behind strongly-enforced high-level APIs (e.g., DBs, KV stores)*

State segregation

- Goal: prevent microreboot from inducing corruption or state inconsistency
- Keep all state that should survive a reboot in dedicated state stores
  - *stores located outside the application ...*
  - *... behind strongly-enforced high-level APIs (e.g., DBs, KV stores)*
- Segment the state by lifetime
  - *apply modularization idea to all state: session state vs. persistent state*

State segregation

- Goal: prevent microreboot from inducing corruption or state inconsistency
- Keep all state that should survive a reboot in dedicated state stores
  - *stores located outside the application ...*
  - *... behind strongly-enforced high-level APIs (e.g., DBs, KV stores)*
- Segment the state by lifetime
  - *apply modularization idea to all state: session state vs. persistent state*
- Separate data recovery from app recovery => do each one better

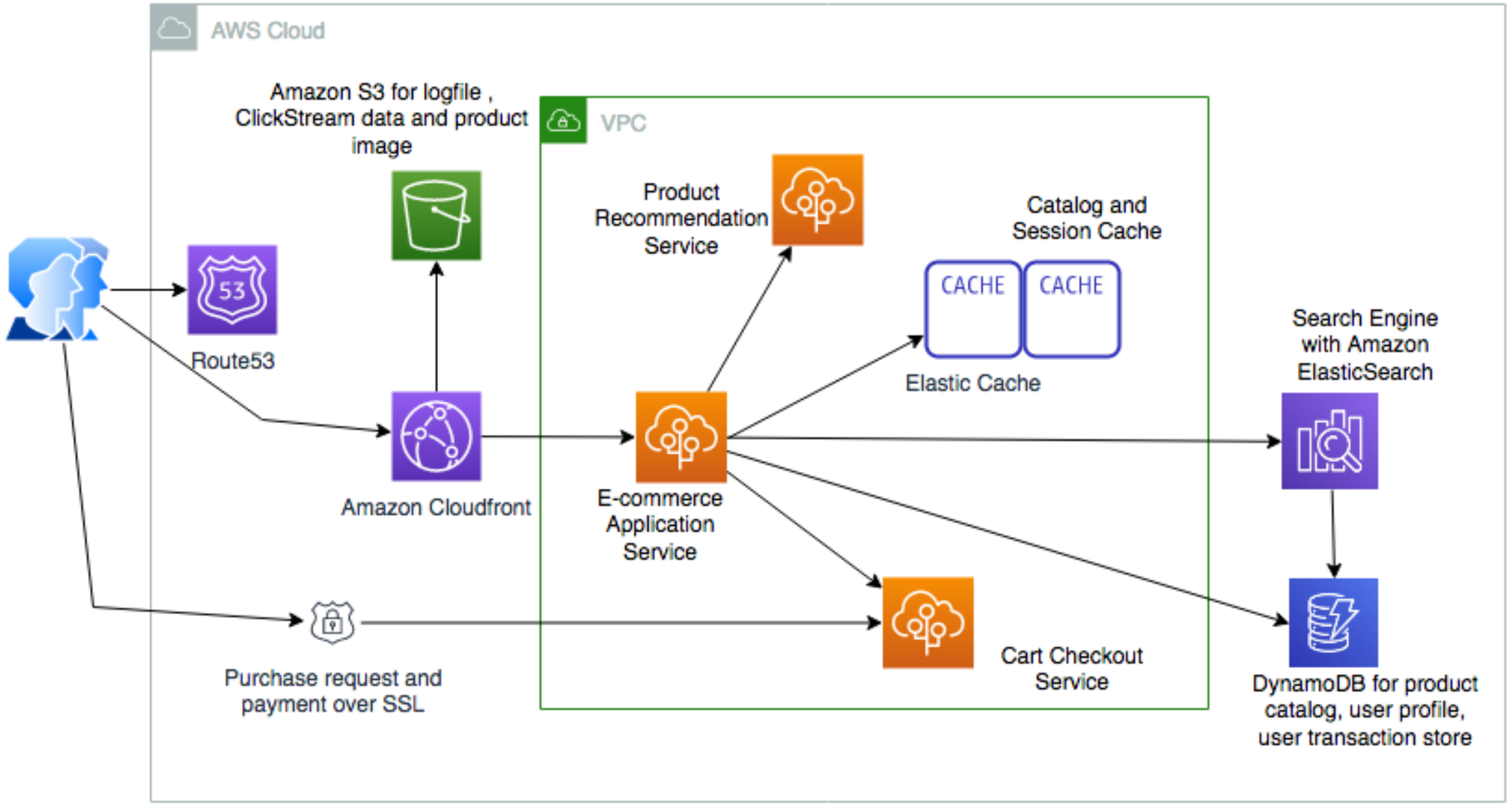
Problems with microrebooting

1. Component reboot can induce state corruption/inconsistency that persists across microrebooting
2. A component I depend on (i.e., need to call) is microrebooting when I need it
3. How to avoid resource leakage after arbitrary microrebooting?
4. How does a component reintegrate after microrebooting?

...

Functional decoupling

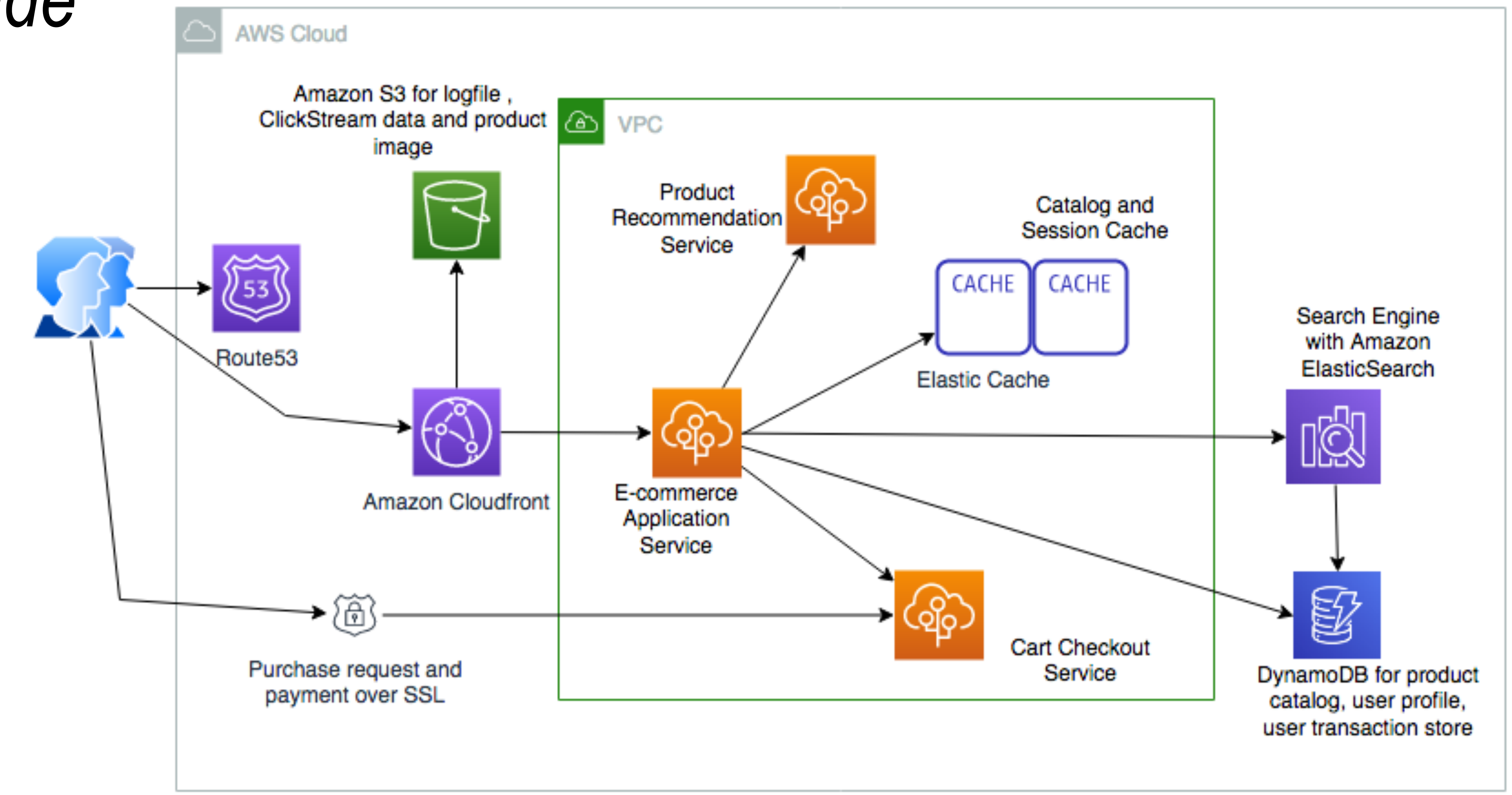
- Goal
  - *reduced disruption of system during restart*



https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

Functional decoupling

- Goal
  - *reduced disruption of system during restart*
- No direct references (e.g., no pointers) across component boundaries
  - *Store cross-component references outside component*
    - Naming indirection through runtime
    - Marshall names into state store



https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

Problems with microrebooting

1. Component reboot can induce state corruption/inconsistency that persists across microrebooting
2. A component I depend on (i.e., need to call) is microrebooting when I need it
3. How to avoid resource leakage after arbitrary microrebooting?
4. How does a component reintegrate after microrebooting?

...

Leased resources

- Goal: avoid resource leakage without fancy resource tracking

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*

Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*
- Requests carry TTL => automatically purged when TTL runs out

Problems with microrebooting

1. Component reboot can induce state corruption/inconsistency that persists across microrebooting
2. A component I depend on (i.e., need to call) is microrebooting when I need it
3. How to avoid resource leakage after arbitrary microrebooting?
4. How does a component reintegrate after microrebooting?

...

Retryable interactions

- Goal
  - *seamless reintegration of microrebooted component by recovering in-flight requests transparently*

Retryable interactions

- Goal
  - *seamless reintegration of microrebooted component by recovering in-flight requests transparently*
- Interact via timed RPCs or equivalent
  - *if no response, caller can gracefully recover*
  - *timeouts help turn non-Byzantine failures into fail-stop events*
  - *RPC to a microrebooting module throws RetryAfter(t) exception*

Retryable interactions

- Goal
  - *seamless reintegration of microrebooted component by recovering in-flight requests transparently*
- Interact via timed RPCs or equivalent
  - *if no response, caller can gracefully recover*
  - *timeouts help turn non-Byzantine failures into fail-stop events*
  - *RPC to a microrebooting module throws RetryAfter(t) exception*
- Action depends on whether RPC is idempotent or not

Problems with microrebooting

1. Component reboot can induce state corruption/inconsistency that persists across microrebooting
2. A component I depend on (i.e., need to call) is microrebooting when I need it
3. How does a component reintegrate after microrebooting?
4. How to avoid resource leakage after arbitrary microrebooting?

...