SOA Deployment

30/07 - Web Service Dev & SOA

Adenda

What's On the menu? - Week 5

I: PSA (Public Service Announcement)

II: Deployment Decisions

III: Basic of Orchestration System


PSA
Public Service Announcement
30/07 - Web Service Dev & SOA

Public Service Announcement

A microservice is not a service in SOA. They have different ideologies (more on that post-midterm). It is possible to create a system that is both SOA & microservice-friendly. Just make sure **you are meeting** the course objectives of SOA during the presentations & exams.

Although cannot disclose exam questions to any student before the exam, you can ask me to create a mock exam for you (do bring something to note). Exam is for measuring the student's learning curve on the individual level. Knowing the exact exam questions before the exam will turn the exam into **a memory test.**

Have better exam/course structure idea? Welcome to any feedback for future course(s).

Vote Today - Submit as a part of the exercise (the vote detail refer to **Week 4** slide)

Comes As You are

- **9 hrs** with me before the presentation and exam. May not be present during the exam (not yet an internal lecturer & depending the availability of the invigilators).

  Make sure you know what you need to know before the exam.

- 6 hrs **in-class**: This week & next week will have only 1 question in the group exercise. Can use the remaining time to prepare and practice for the presentation & exam.

- 3 hrs of **the consultation**: **10AM-1PM** next week @ this room.

- During these times, need feedback for usable diagrams, writings, presentation slides, your mock exam? Hit me up.

II: Deployment Decisions

30/07 - Web Service Dev & SOA

Deployment Decisions

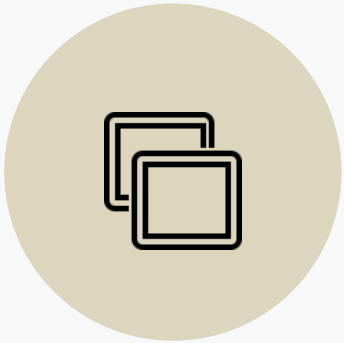
There are three deployment approaches. Each with different trade-offs.

Bare-Metal

Device-Dependent. Can do everything as long as the hardware is capable.

Container

OS-level virtualisation (e.g. docker). Lightest, most portable. Limited functionality. Require resources for virtualisation.



Virtual Machine

Hardware-level virtualisation. (Try to be) best of both worlds. More portable than Bare-Metal, More functionality, more resource-demanding and heavier than Container.

Deployment Decisions

Trade-Off: Security

Bare-Metal

High System Exposure: Once got in, may be able to **access the full system/the other components without** additional exploit/attack chain.

Container & Virtual Machine

Medium-Low System Exposure: Once escaped from the sandbox, **may need additional exploit/attack chain** to gain access to the full system/other components.

Deployment Decisions

Trade-Off: Performance

Bare-Metal

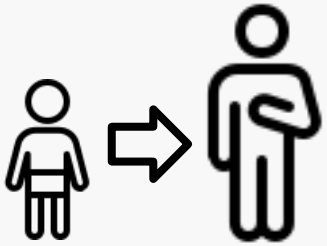
Raw Performance:
Depends mostly on physical factors (e.g. network cond., I/O speed, CPU/RAM capabilities, etc.).

Container & Virtual Machine

Overhead & Performance Loss:
Depends on software-related networking & virtualisation policies (e.g. routing rules, virtual disk I/O speed, etc.)

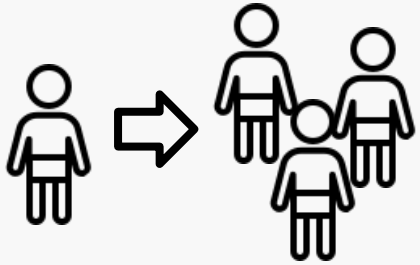
Deployment Decisions

Scalability



Vertical Scaling

Scaling Up: Increase resource to the existing component (e.g. More RAM, More GPU, More Storage).



Horizontal Scaling

Scaling Out: Increase the number of existing component.

Deployment Decisions

Trade-Off: Vertical Scaling

Bare-Metal

Rigid: Need to put in/take out physical components.

Container & Virtual Machine

Flexible: Just change the configuration.

Deployment Decisions

Trade-Off: Horizontal Scaling

Bare-Metal

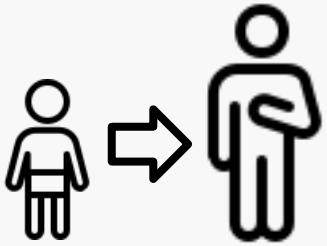
Costly: Adding another machine can be expensive.

Container & Virtual Machine

Costless: Adding/removing another container or VMs just few clicks/configs away.

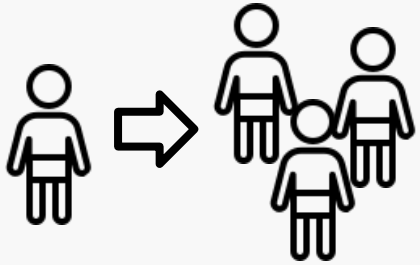
Deployment Decisions

Scalability: Considerations - Upper Limits of Scalability



Physical Limitation

Availability or add-on capability of hardware components (e.g. soldered circuits, sensors, portable devices).



Design Pattern

Some design pattern requires a bottleneck or centralised component(s) (e.g. transaction validation - both in traditional banking & block-chain)

Deployment Decisions

2 Level of Self-Healing Capability:

Application-Level

Applicable for

- Any deployment approaches

System-Level

Applicable for

- Container (e.g. in k8s)
- Virtual Machine (e.g. Azure)

Deployment Decisions

2 Level of Self-Healing Capability:

Application-Level

Pros

- Context/domain -specific

Cons

- Require coding

System-Level

Pros

- Pre-created/embedded in the software

Cons

- Require config.

Deployment Decisions

Self-Healing Capability - System Level:

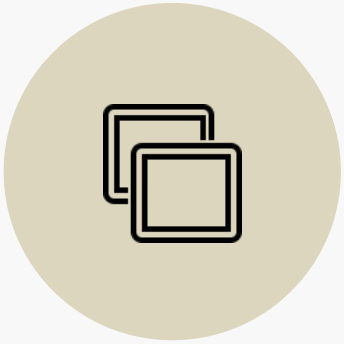
Container

In k8s:

- **Liveness:** Check a container's status. If fails, create a new container.
- **Readiness:** Check ability to serve a request. If fails (i.e. not ready), remove its address.

Deployment Decisions

Self-Healing Capability - System Level:



Virtual Machine

In Azure Virtual Machines:

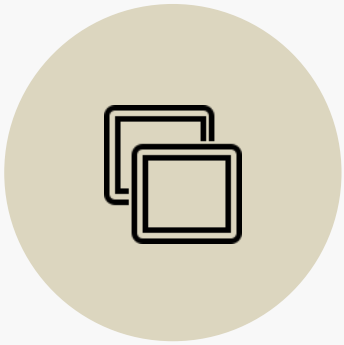
- **Hypervisor** health-check a virtual machine. If fails, reboot.
- **Fabric Controller** health-check a physical server which runs the virtual machines. If fails (i.e. not ready), reboot the server.

Note: Reboot the server may not heal (e.g. CrowdStrike incident).

Deployment Decisions

Self-Healing Capability - System Level:

Container



Virtual Machine

Note II: System level may check only the availability of components (i.e. ping, health-check). Not check quality of the *recovered/available* components.

Deployment Decisions

Self-Healing Capability - Application Level:

Application Level

Can be more in-depth to the application domain (but more coupling). Examples:

- **Performance:** Ping, Delay, Bit Rate.
- **Integrity:** Correctness, Avr Packet Loss.
- **Other:** Versioning, State, Bandwidth fluctuation.

Deployment Decisions

Limitations

Physical Constraints

- Limited cost, development time, time to deploy, computing power, storage, etc.

Competency-related

- Learning curve of maintainers.
- Inexperienced clients.

Deployment Decisions:

Running Example - Milk's Restaurant

Front-end

Bare-metal due to the resource limitation of tablets and for easy replacement.

Authenticator

Bare-metal for performance with DIY self-recovery mechanism.

Order Mgmt.

Container & Persistent Volume (i.e. Database) /w **Self-healing & Load Balancer**

(Examples on III)

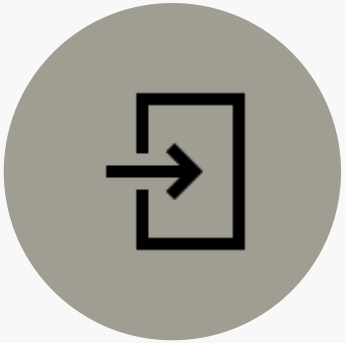
Queue Mgmt.

Kiosk-like Bare-metal /w its own database. Easy for Chefs to maintain **(Touchscreen & One power/reset button)**

Self-Recovery

Running Example - Authenticator: DIY Self-Recovery

Authenticator consists of 2 components:



Threads

Forwards an authentication request in a thread pipe, waiting for **a FB API response and/or timeout.**

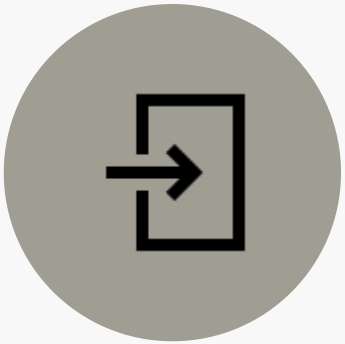
Supervisor

Allocate authentication requests into available threads, monitor requests timeout, and **flush any thread** pipe if it is **idle more than 5 minutes** (i.e. self-clean/recover).

Self-Recovery

Running Example - Authenticator: DIY Self-Recovery

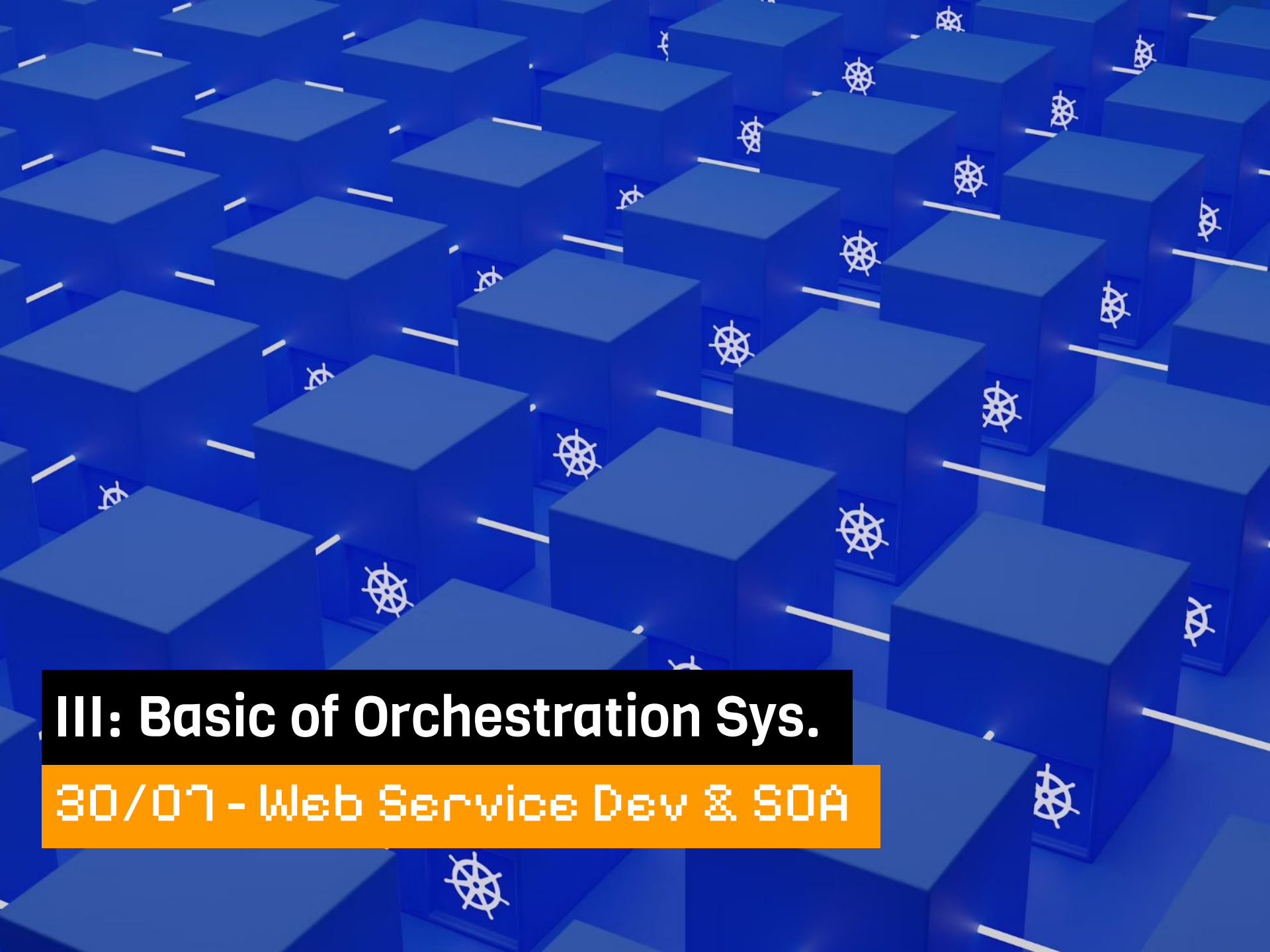
Authenticator consists of 2 components:



Threads

Supervisor

Observation: Authenticator can still down (e.g. Power outage)



III: Basic of Orchestration Sys.

30/07 - Web Service Dev & SOA

Prologue

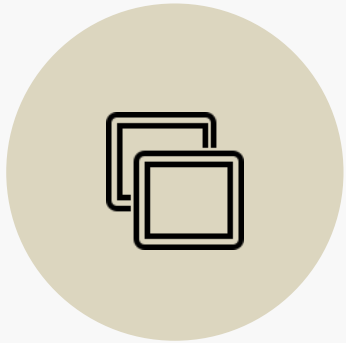
There are three deployment approaches:

Bare-Metal

Device-Dependent. Can do everything as long as the hardware is capable.

Container

OS-level virtualisation (e.g. docker). Lightest, most portable. Limited functionality. Require resources for virtualisation.



Virtual Machine

Hardware-level virtualisation. (Try to be) best of both worlds. More portable than Bare-Metal, More functionality, more resource-demanding and heavier than Container.

Prologue

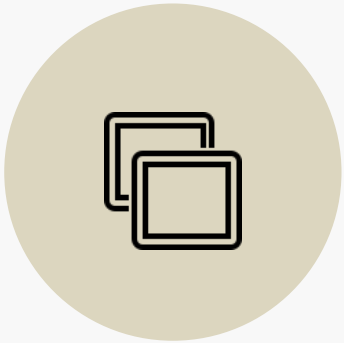
Problem: How to use components/services in different deployment approaches?

Bare-Metal

Device-Dependent. Can do everything as long as the hardware is capable.

Container

OS-level virtualisation (e.g. docker). Lightest, most portable. Limited functionality. Require resources for virtualisation.



Virtual Machine

Hardware-level virtualisation. (Try to be) best of both worlds. More portable than Bare-Metal, More functionality, more resource-demanding and heavier than Container.

Orchestration Sys.

Introducing Orchestration System (e.g. k8s), key features are:

Routing between/to Services (e.g. DNS)

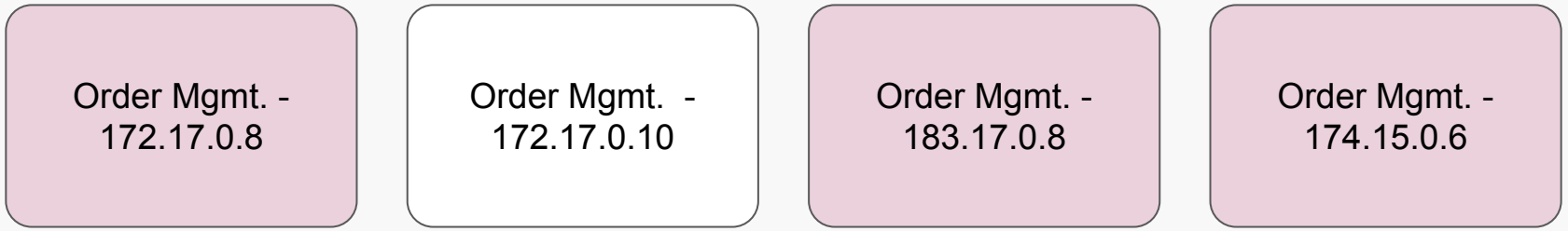
Health-Checking & Self-Recovery of Services

Load-Balancer

DNS

A running example - Order Mgmt.: Without DNS (1)

1. The second one is down.



Note: Assuming one container/pod, in k8s.

DNS

A running example - Order Mgmt.: Without DNS (2)

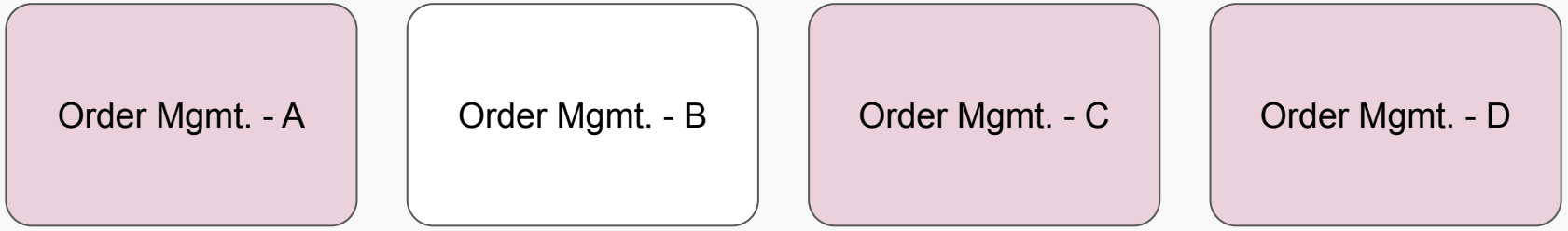
2. The second one is recovered (with different IP).

It can be difficult to “hard-code” IP address.

DNS

A running example - Order Mgmt.: With DNS

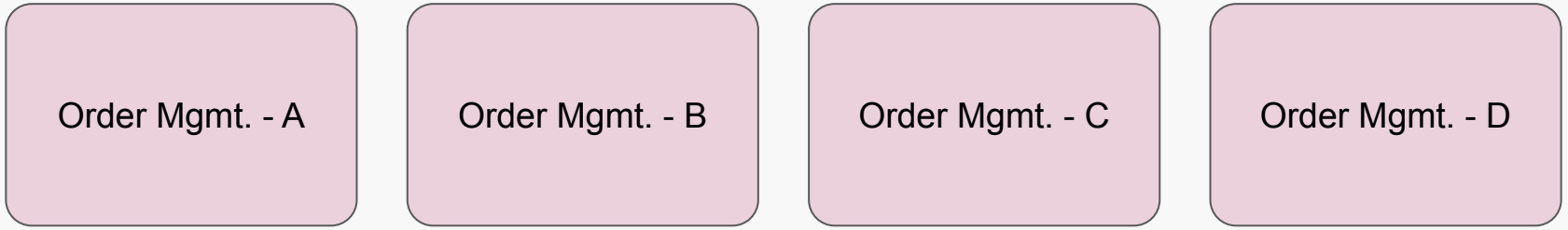
1. The second one is down.



DNS

A running example - Order Mgmt.: With DNS

2. The second one is recovered.



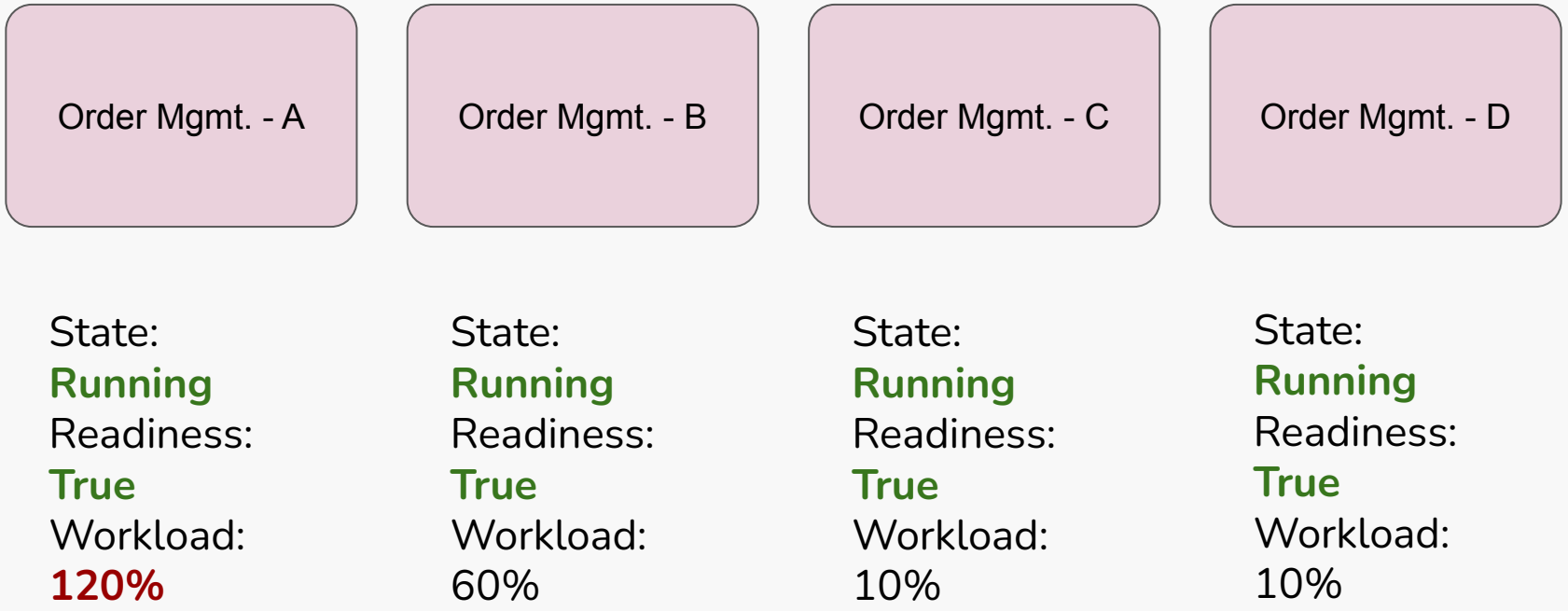
Call Them by Your (Host-)Name

For example:
Second One: B.order-mgmt.milkrestaurant.svc.cluster.local

Load Balancer

A running example - Order Mgmt.: Without Load Balancer

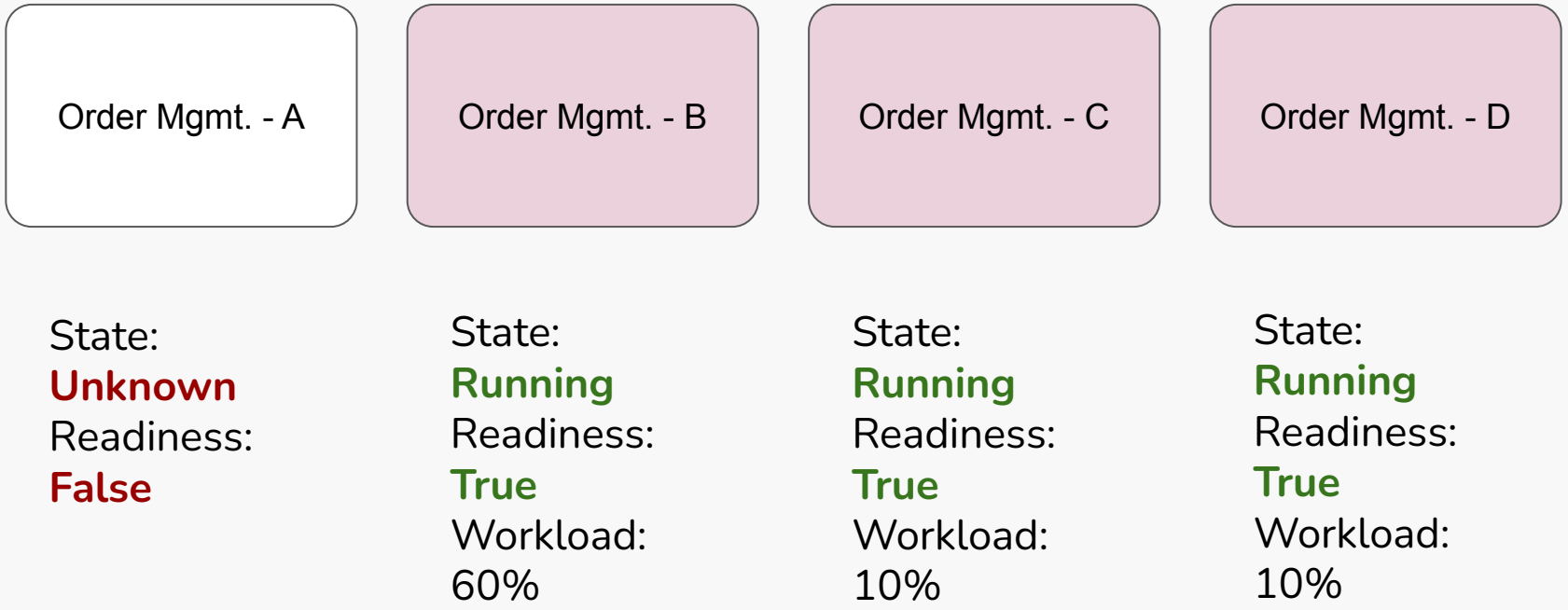
1. Workload can be unbalanced (e.g. the default address).



Load Balancer

A running example - Order Mgmt.: Without Load Balancer

2. This can unintentionally bombard the service, until it downs.



Load Balancer

A running example - Order Mgmt.: With Load Balancer

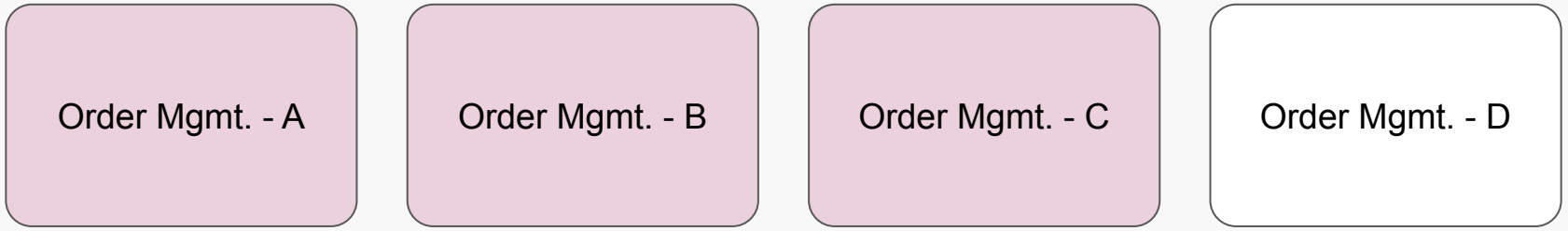
Perfectly Balanced. As They Should Be.

Order Mgmt. - A	Order Mgmt. - B	Order Mgmt. - C	Order Mgmt. - D
State: **Running** Readiness: **True** Workload: 50%	State: **Running** Readiness: **True** Workload: 50%	State: **Running** Readiness: **True** Workload: 50%	State: **Running** Readiness: **True** Workload: 50%

Self-Recovery

A running example - Order Mgmt.: Without Self-Recovery (1)

1. D is down (e.g. due to a database connection error).



State:
Running
Readiness:
True
Workload:
90%

State:
Running
Readiness:
True
Workload:
90%

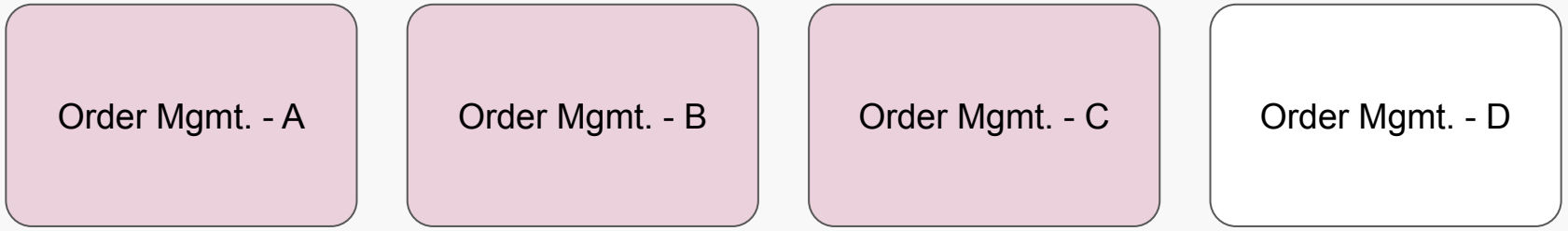
State:
Running
Readiness:
True
Workload:
90%

State:
Unknown
Readiness:
False
Workload*:
90%
(*before down)

Self-Recovery

A running example - Order Mgmt.: Without Self-Recovery (2)

2. The D's pending orders are evenly distributed to A, B, C.:
   Initial Workload (90%) + D's workload (90%/3) = 120%



State:
Running
Readiness:
True
Workload:
120%

State:
Running
Readiness:
True
Workload:
120%

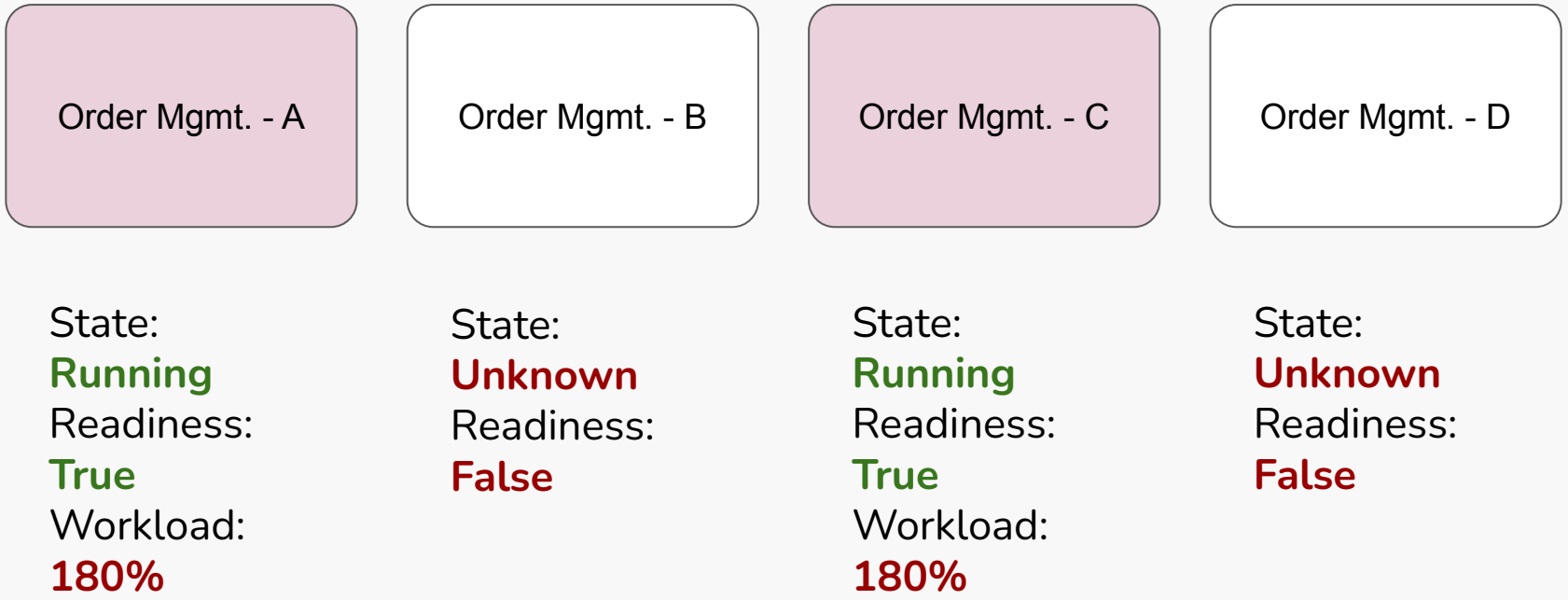
State:
Running
Readiness:
True
Workload:
120%

State:
Unknown
Readiness:
False

Self-Recovery

A running example - Order Mgmt.: Without Self-Recovery (3)

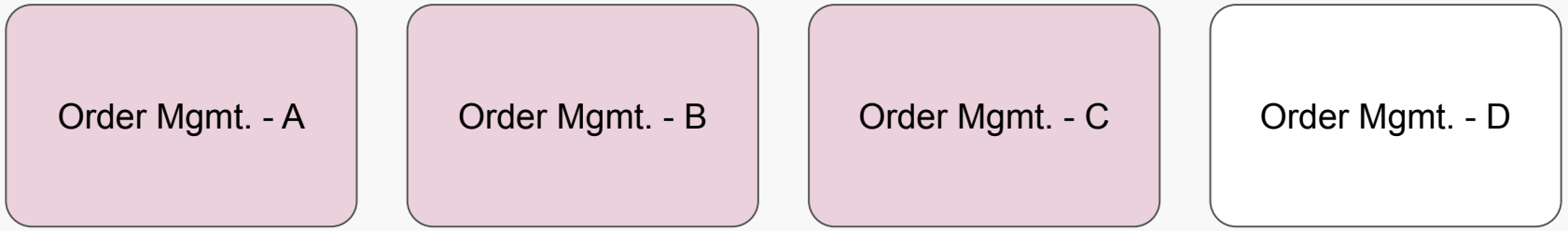
3. B is down due to the prolonged and increased workload.
4. The B's workload evenly distributed to A, C. Possible cascading failure.



Self-Recovery

A running example - Order Mgmt.: With Self-Recovery (1)

1. D is down.
2. K8s health-check of each service.



State:
Running
Readiness:
True
Workload:
90%

State:
Running
Readiness:
True
Workload:
90%

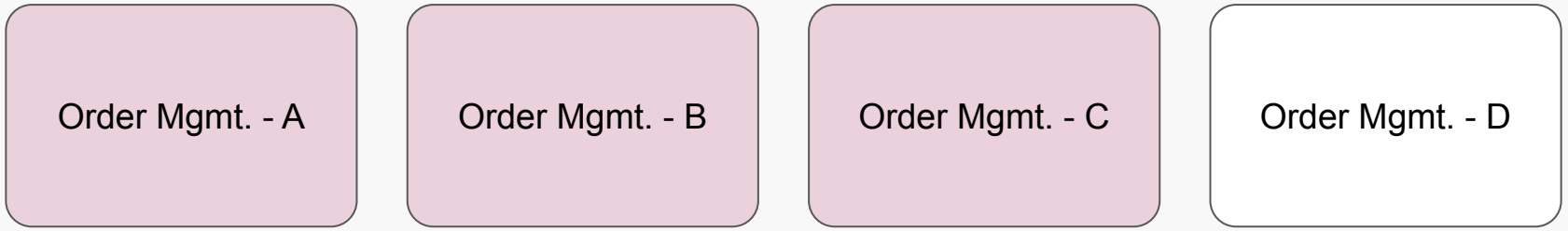
State:
Running
Readiness:
True
Workload:
90%

State:
Unknown
Readiness:
False
Workload*:
90%
(*before down)

Self-Recovery

A running example - Order Mgmt.: With Self-Recovery (2)

3. K8s discovers that D is down.



State:
Running
Readiness:
True
Workload:
90%

State:
Running
Readiness:
True
Workload:
90%

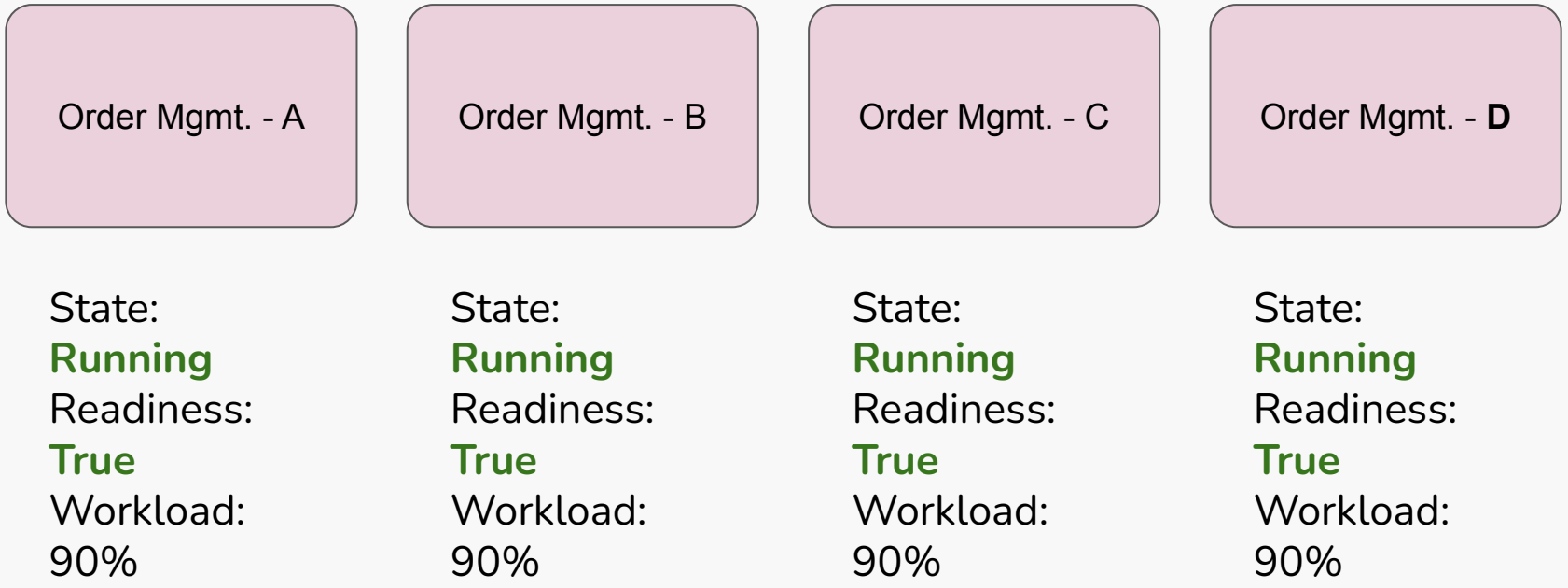
State:
Running
Readiness:
True
Workload:
90%

State:
Unknown
Readiness:
False
Workload*:
90%
(*before down)

Self-Recovery

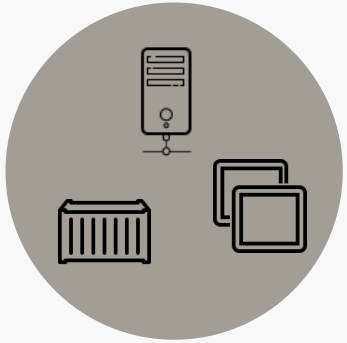
A running example - Order Mgmt.: With Self-Recovery (3)

4. K8s recovers D.
5. The workload returns to normal.



Orchestration Sys.

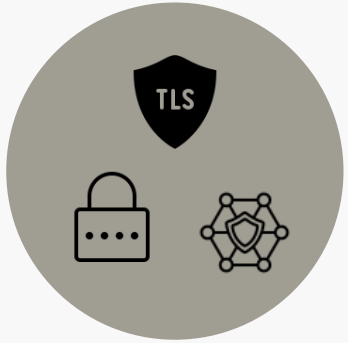
Advantage:



Compatible with any deployment approach.

Got nice features by default

(Like those mentioned previously).

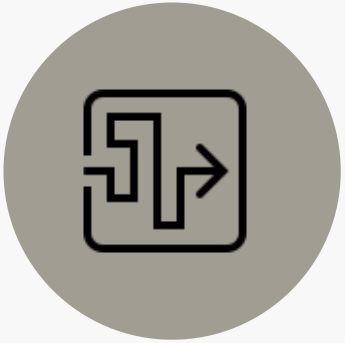


Enhanced Security

TLS, OAuth, Role-based Access Control-capable.

Orchestration Sys.

Disadvantage:



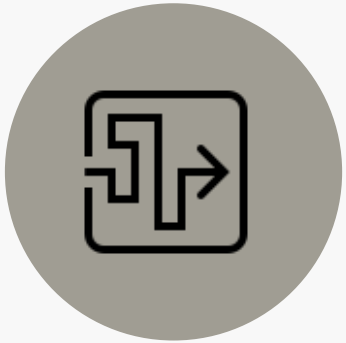
Complex to config and maintain

Difficult to start

(Steep learning curve & Can be costly to transition to).

Require deployment plan

K8s Storage Types



Persistent Volume

Independent to a service; if a container/service restarts, the data stays (e.g. Database).

Ephemeral Volume

Dependent to a service; if a container/service restarts, the data is gone (e.g. Caches, In-memory states)

K8s Units

From Smallest:

- **Container:** A self-contained app-based virtualisation. Can be used to serve a specific business function (e.g. a service in SOA)
- **Pod: One pod** consists of **one or multiple** containers.
- **Node: One node** consists of **one or multiple** pods**.**
- **Cluster: One cluster** consists of **one or multiple** nodes.

K8s Common Vocab.

- **Ingress:** A gatekeeper; Manage external access to Pod.
- **Service:** A method for exposing a network application that is running Pod. **Not same as a service in SOA.**
- **Gateway API:** An extension of k8s's API kinds for advanced traffic routing. **Not same as API Gateway (will include in Post-Midterm).**

Group Exercise & Vote

30/07 - Web Service Dev & SOA