Microservices

José Oramas

[* Slides adapted from S. Latré (UAntwerp) and B. Volckaert (Ugent)]

IDLab
INTERNET & DATA LAB

imec

Agenda for today

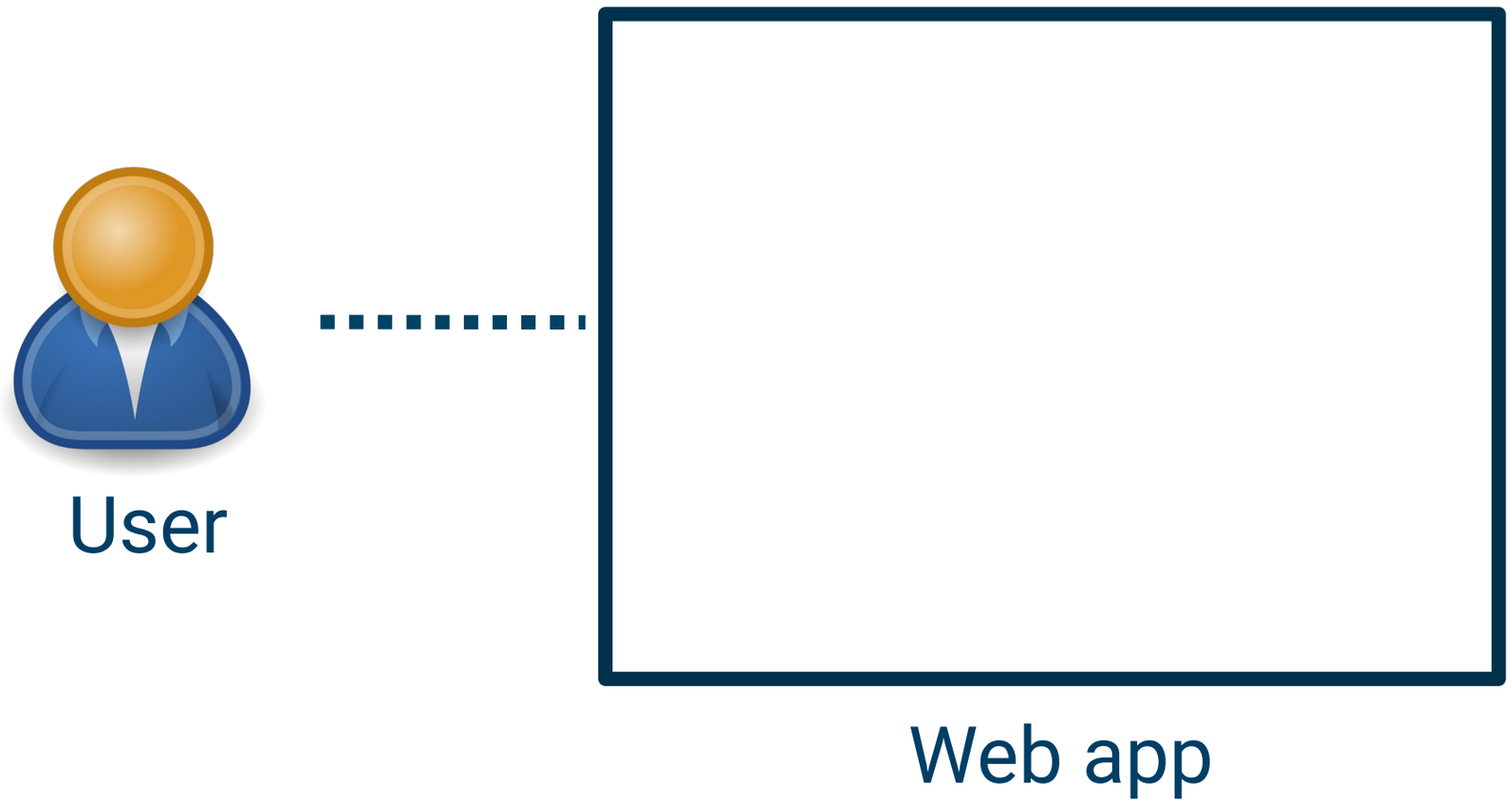
Microservices

- What is a microservice?
- Where do they stand next to SOAs and Web Services?

Relationships w.r.t. container technologies

- Motivations and characteristics
- LXC as a use case

Building an app: the traditional way



We need data

User

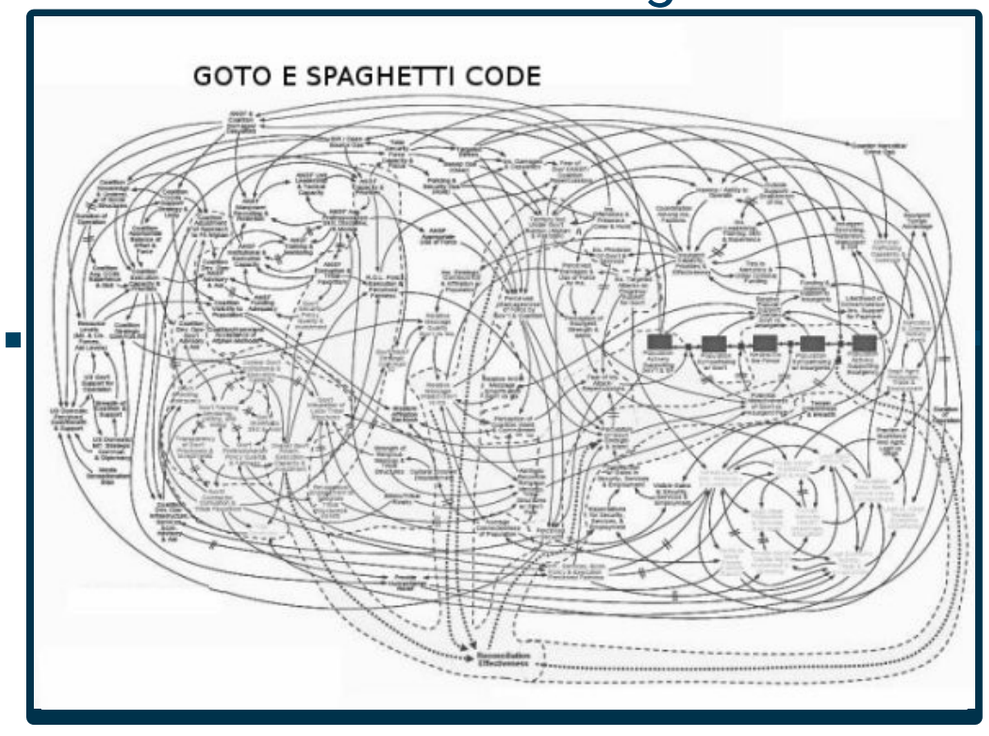
Web app

Databases
Services
File system
Data sources

All features in one app

User

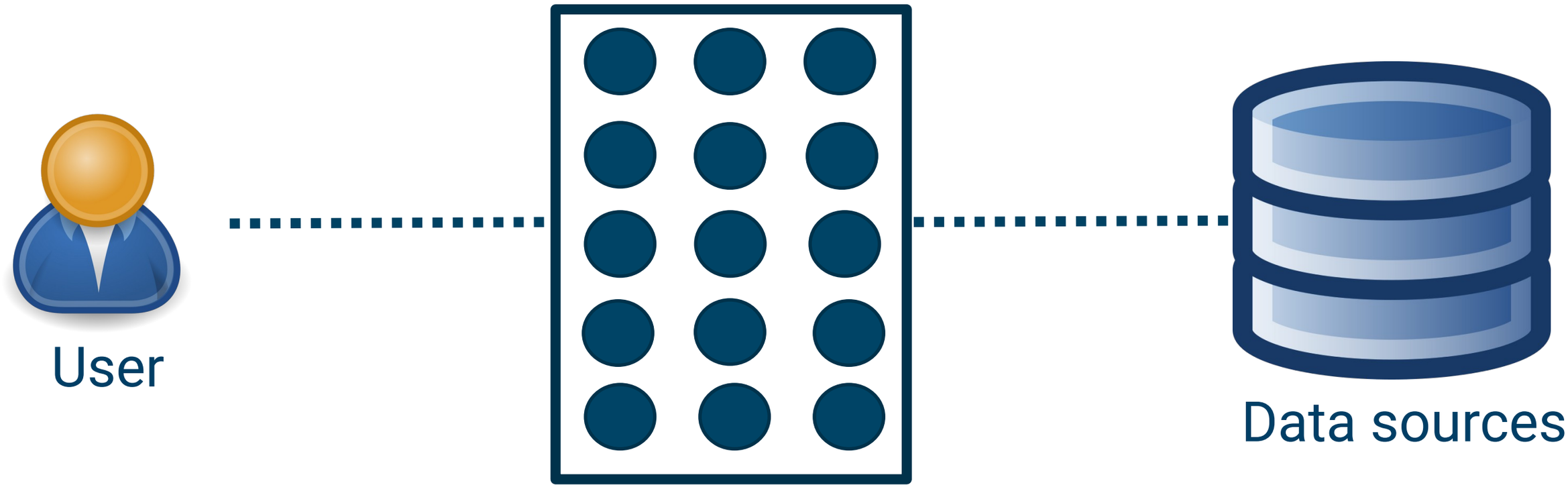
Data access logic
Functionality
Presentation logic



Shopping cart
Authentication
Payment
Input validation
Shipping information

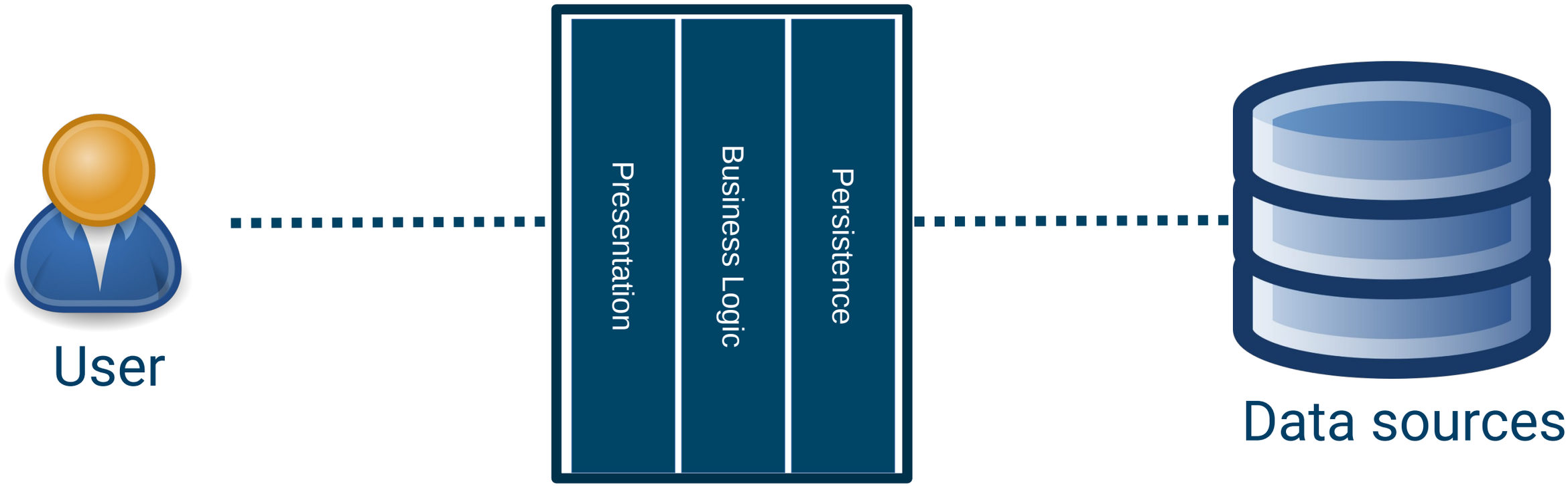
Databases
Services
File system
Data sources

Improving the internals



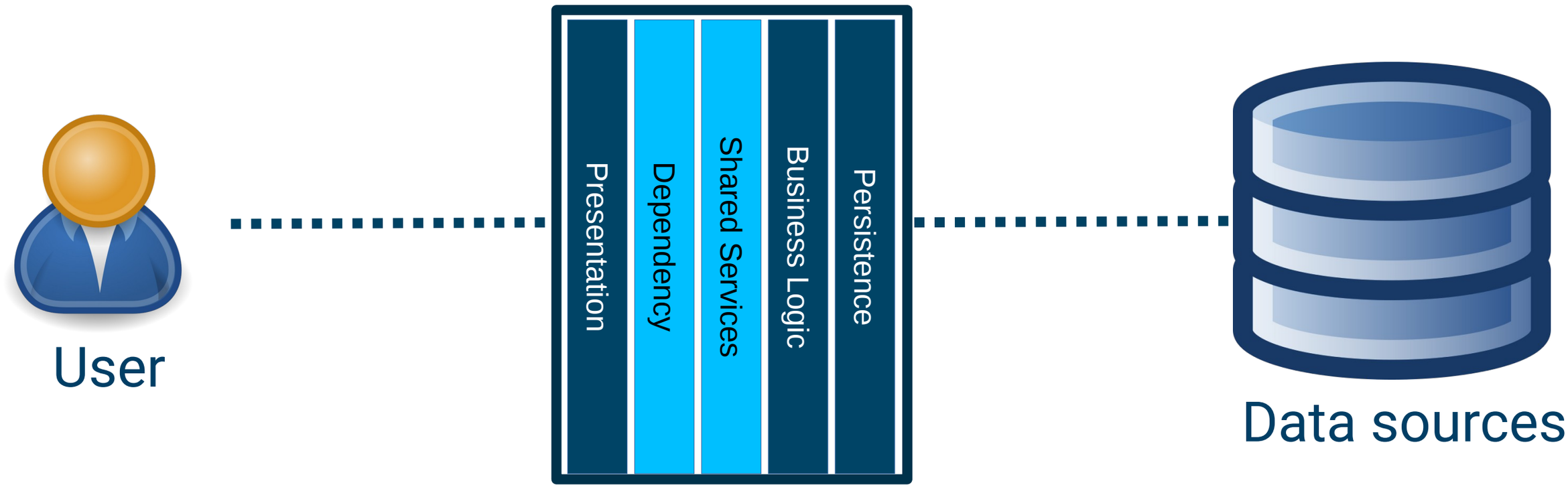
More organized design

Let’s make it layered



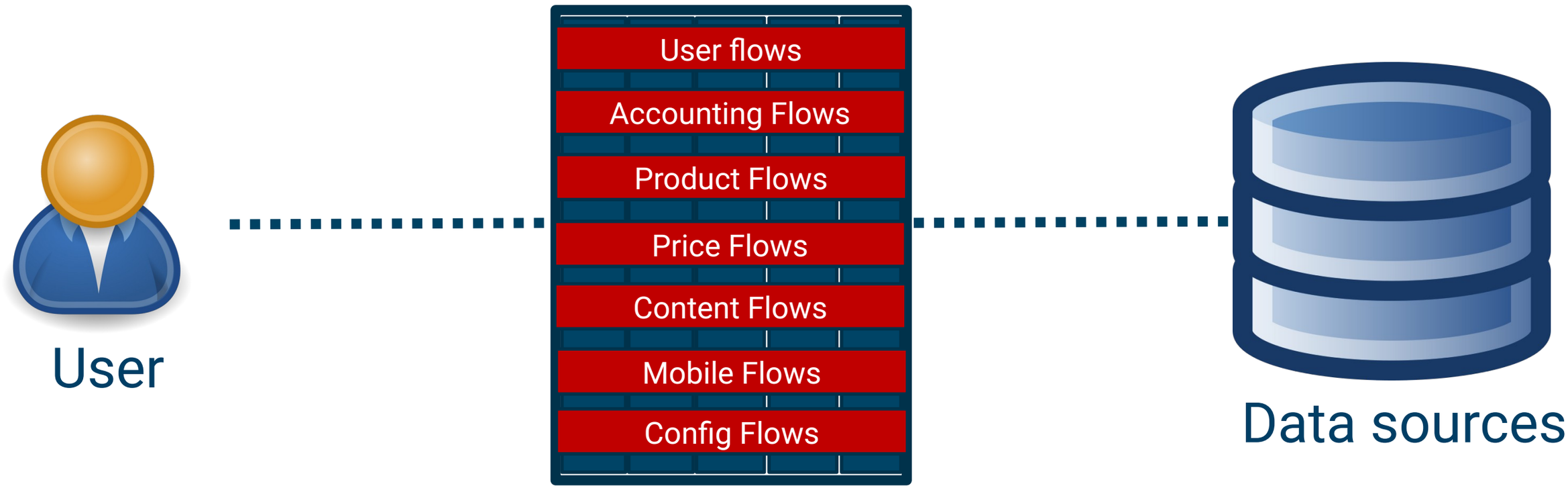
Grouping classes/components with common goals

We need more layers



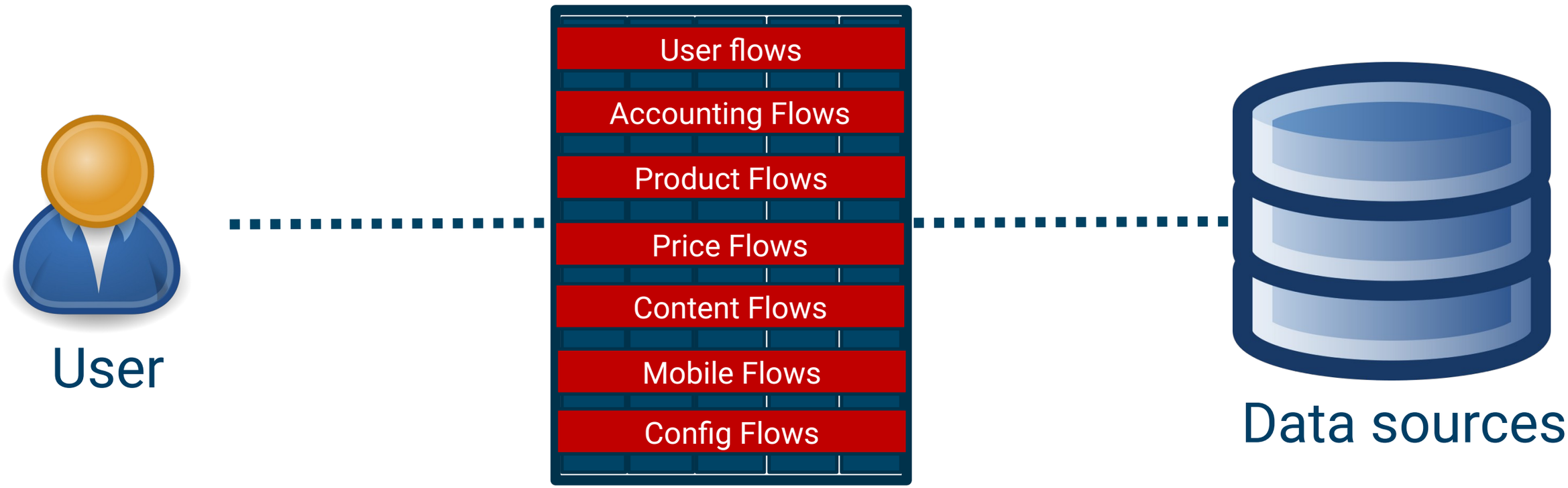
Adding layers to address specific requirements

New flows and features



Adding cross-layer specialized communication

Issues

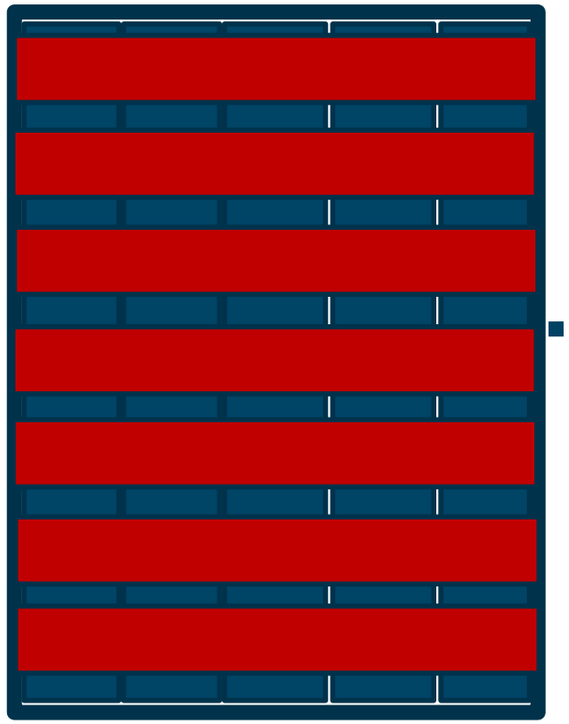


Maintainability

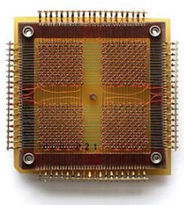
Agility / new feature coding time

Scaling: vertically

User



Data sources

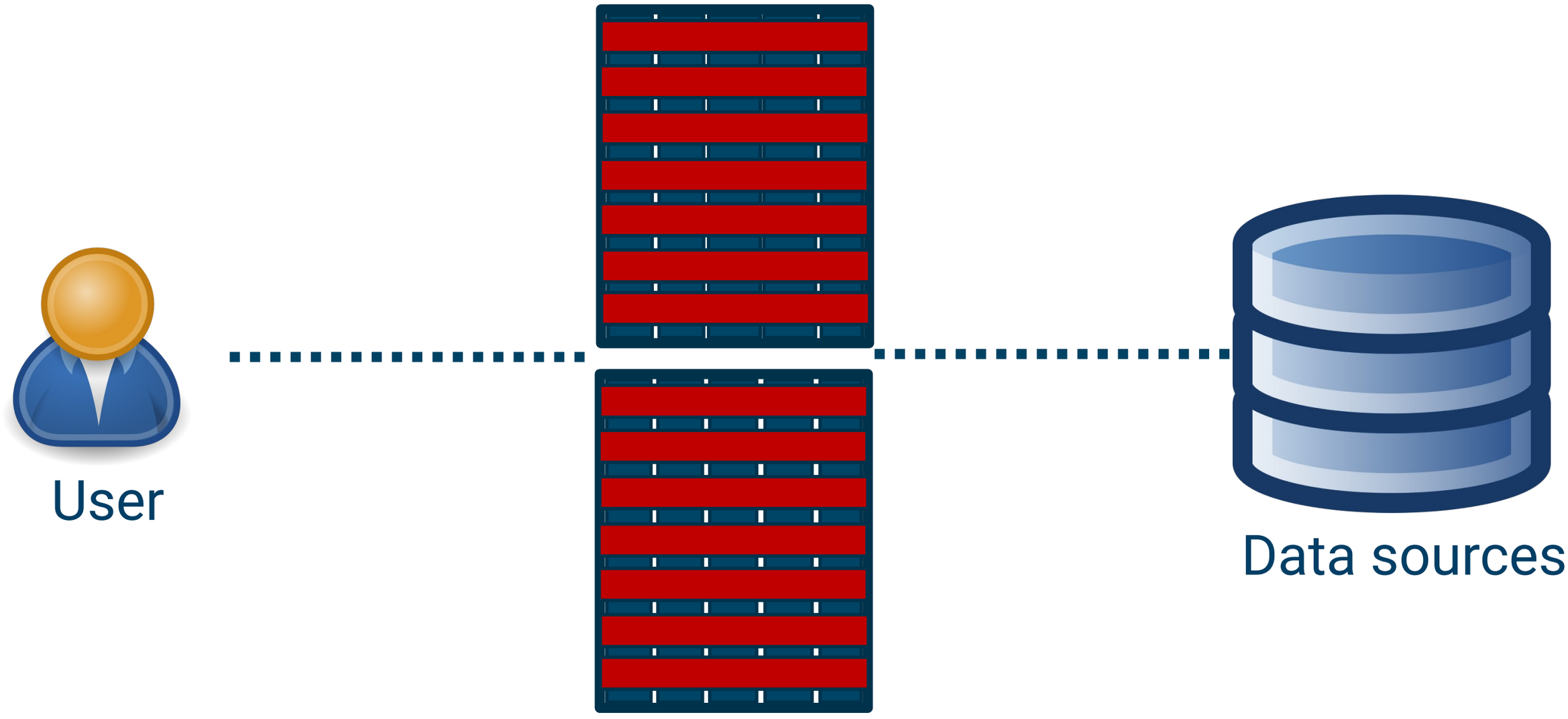


CPU

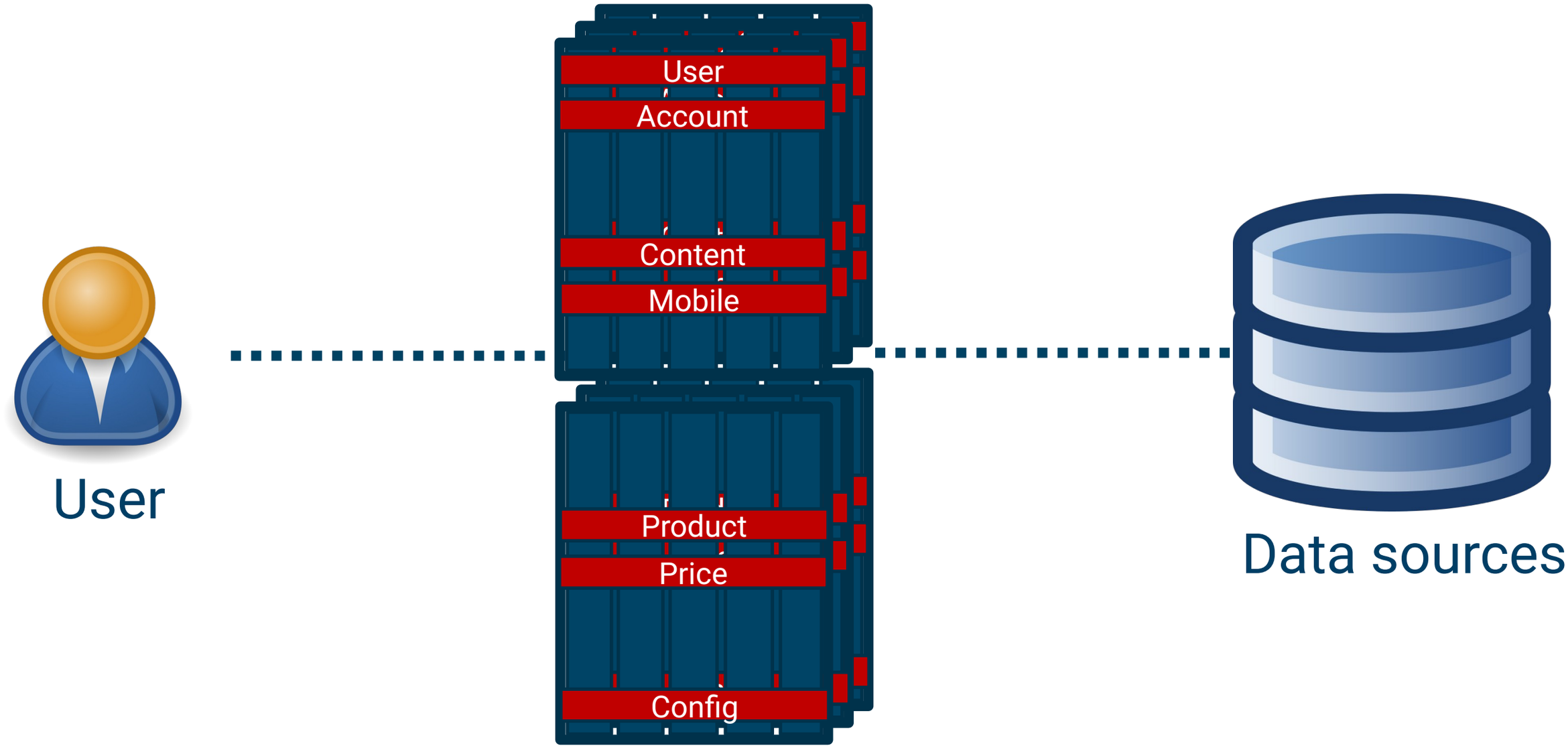
Memory

Storage

Scaling: horizontally



Scaling: split features



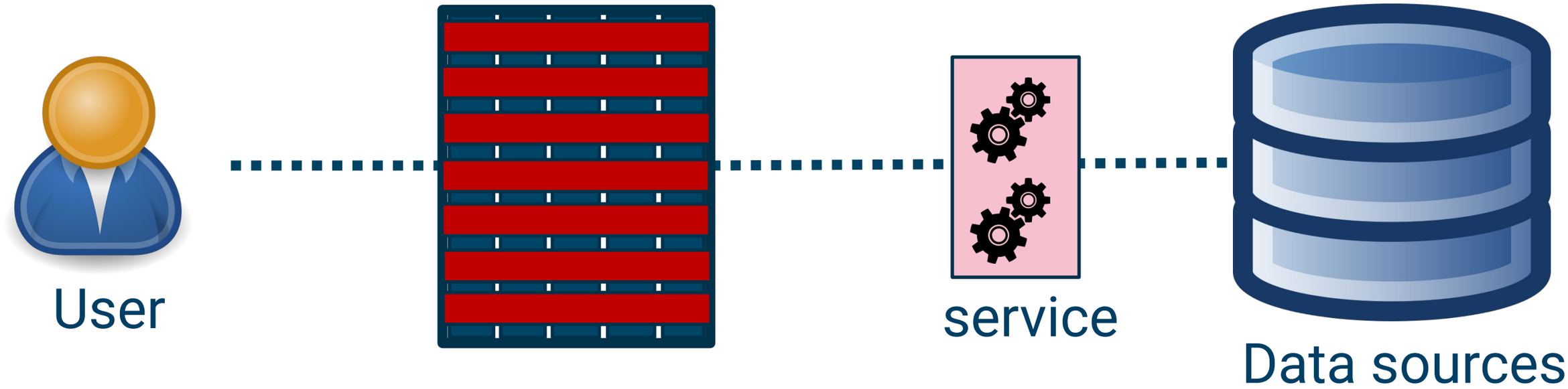
Still issues

User

- Scalability
- Load balancing
- Deployment
- Fault tolerance
- Dependencies
- Security

Data sources

Bring in the service concept

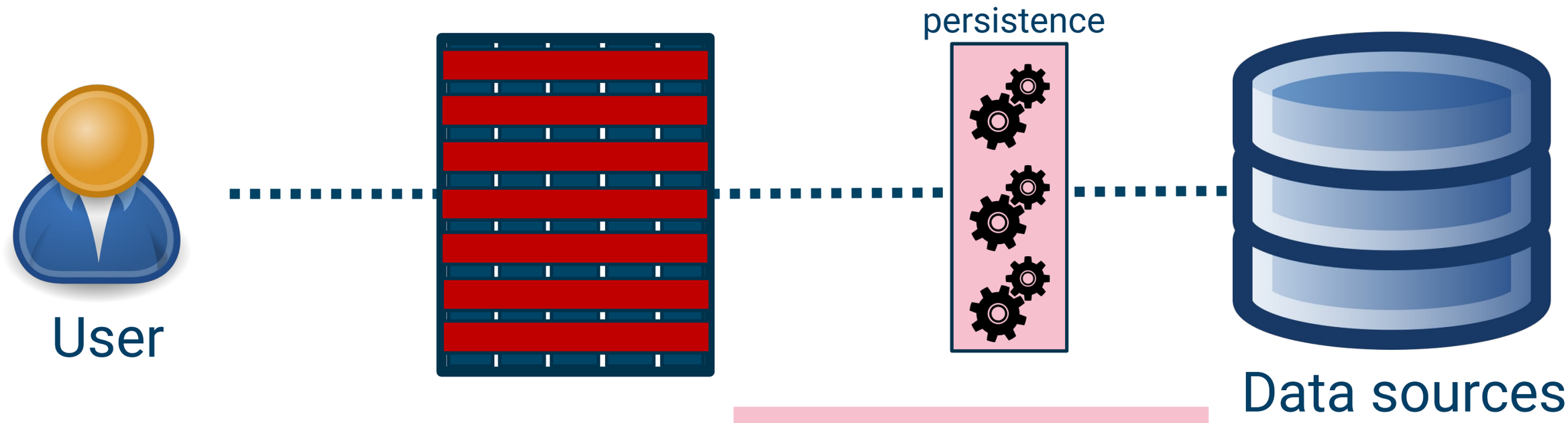


Service Oriented Architecture

Software architecture design pattern based on distinct pieces of software providing application functionality as services to other applications

Until we end up with monoliths again

Even services can be designed monolithic

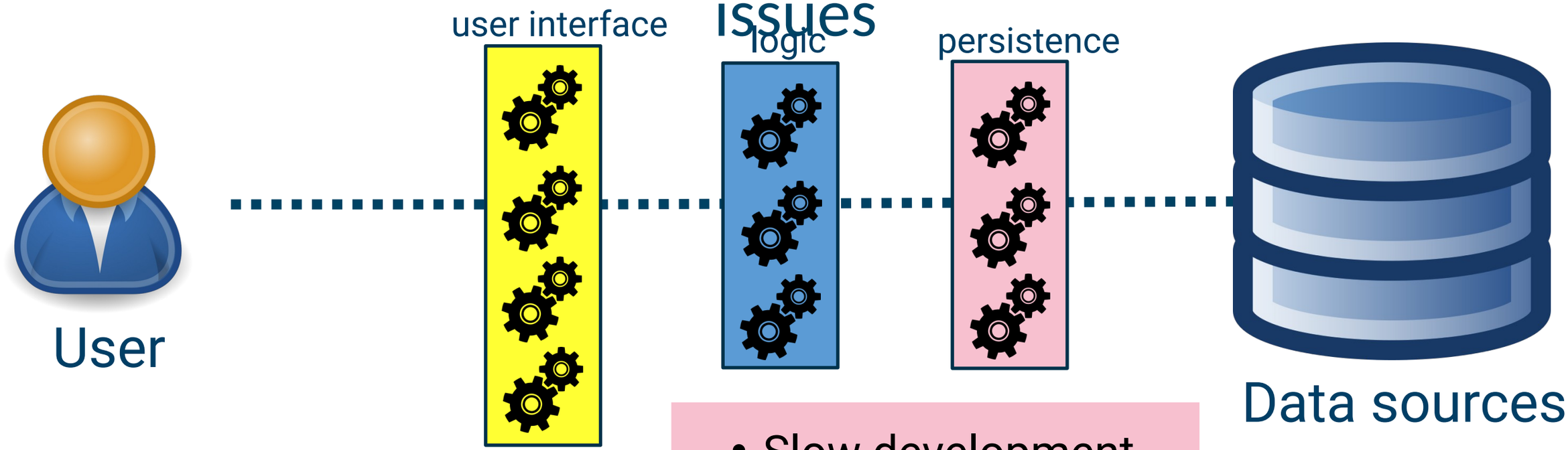


- Slow development
- Scalability issues
- Reusability issues

→ **Fat services**

Services as the new monolithic entities

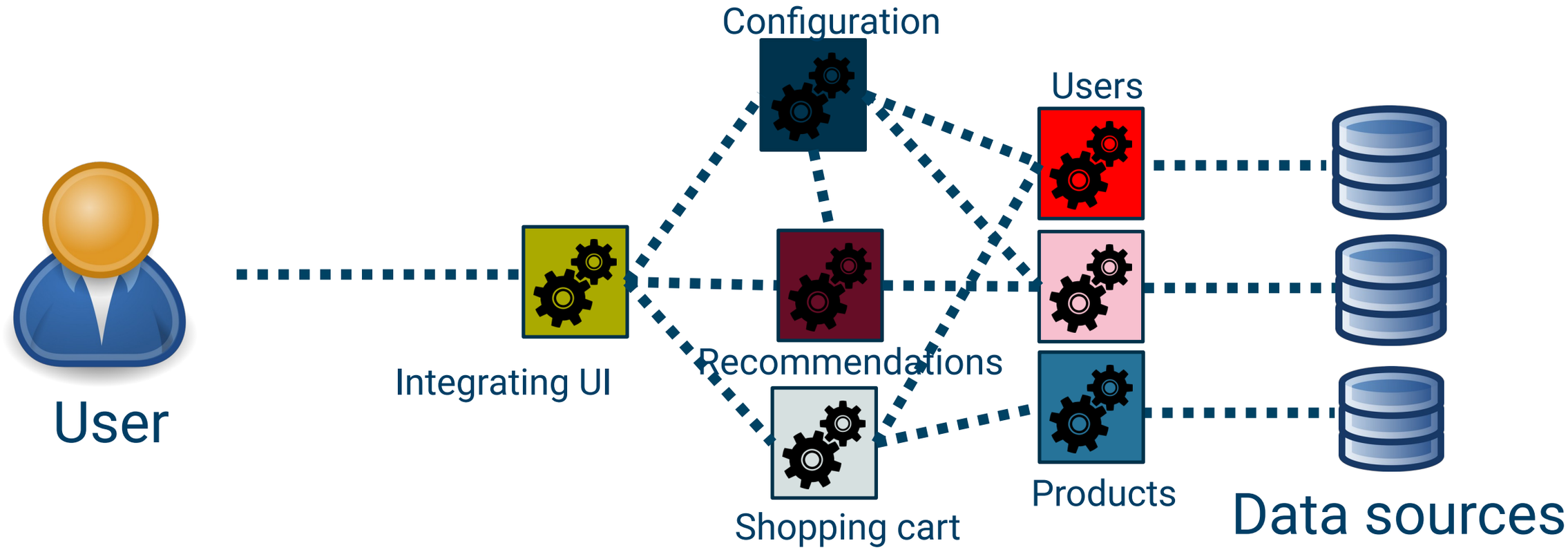
Full service based offering still does not solve all our issues



- Slow development
- Scalability issues
- Reusability issues

→ **Fat services**

New attempt: let's design with scalability in mind



Designing our application as loosely-coupled **microservices**

- Small functional granularity organized around capabilities
- Lightweight communication protocols
- Independently deployable and replaceable

Microservices

A way of designing software applications as suites of independently deployable services

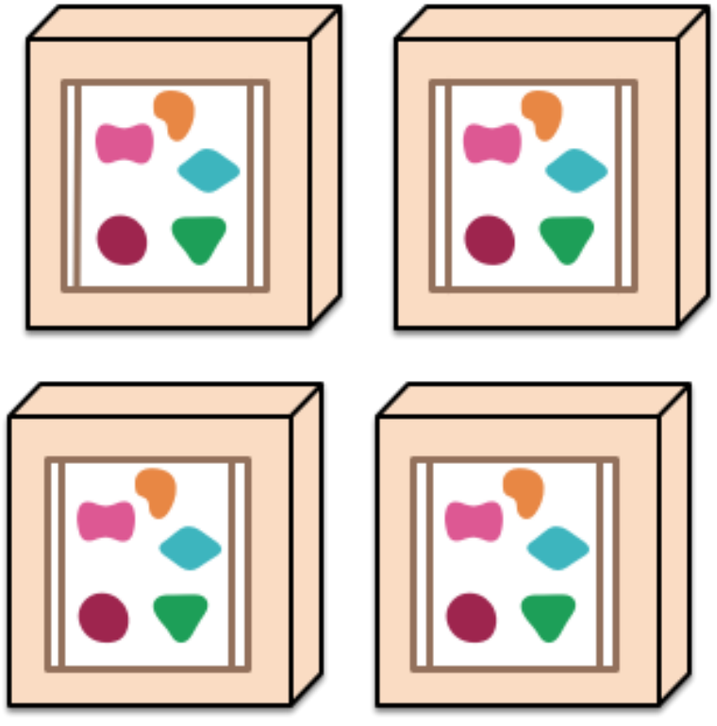
From monolithic application to suites of services

A monolithic application puts all its functionality into a single process...

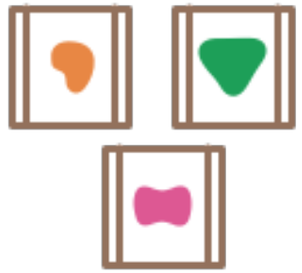
From monolithic application to suites of services

A monolithic application puts all its functionality into a single process...

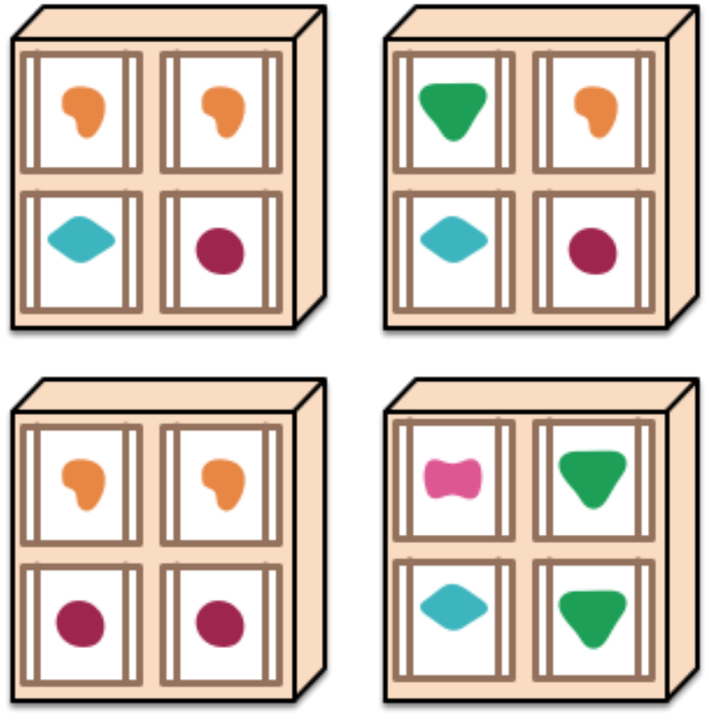
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Definition

Microservice architectural style

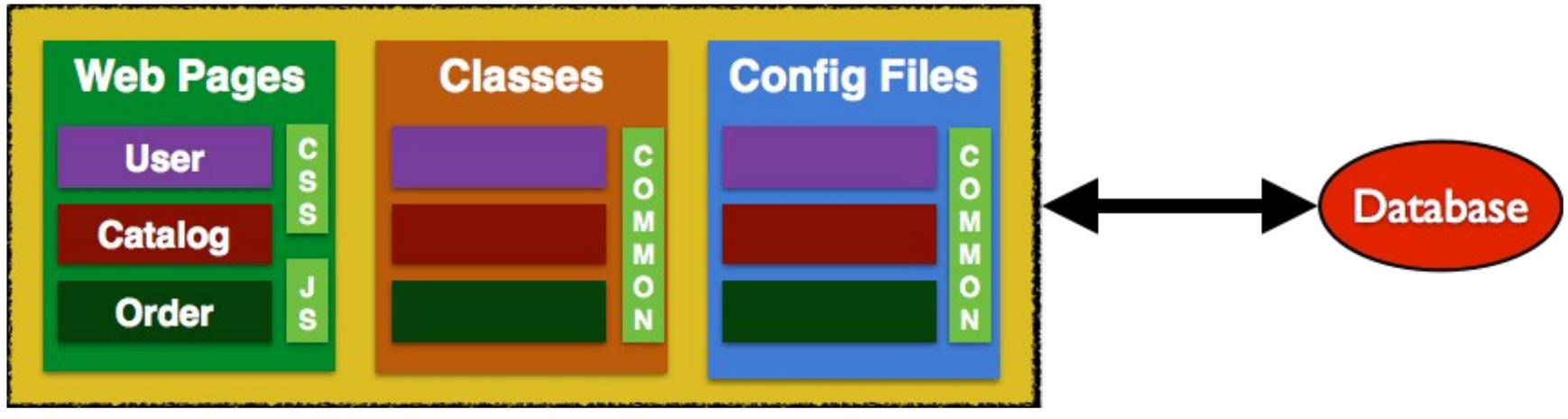
- Approach to developing a single application as a suite of small services
  - Each running in its own process
  - Communicating with lightweight mechanisms (e.g. REST or message queues)

Micro-Services

- Built around business capabilities
- Independently deployable by fully automated deployment machinery
- Bare minimum of centralized management
- Written in different programming languages
- Using different data storage technologies

Monolith example: Java EE shopping cart

Shopping cart monolithic WAR archive



- Web pages/CSS/JS for user, catalog and order component
- Classes for these three components + utility/common classes
- Configuration files for each component + for the application

Advantages

- well-known layered architecture
- IDE-friendly
- easy sharing, simplified testing, easy deployment

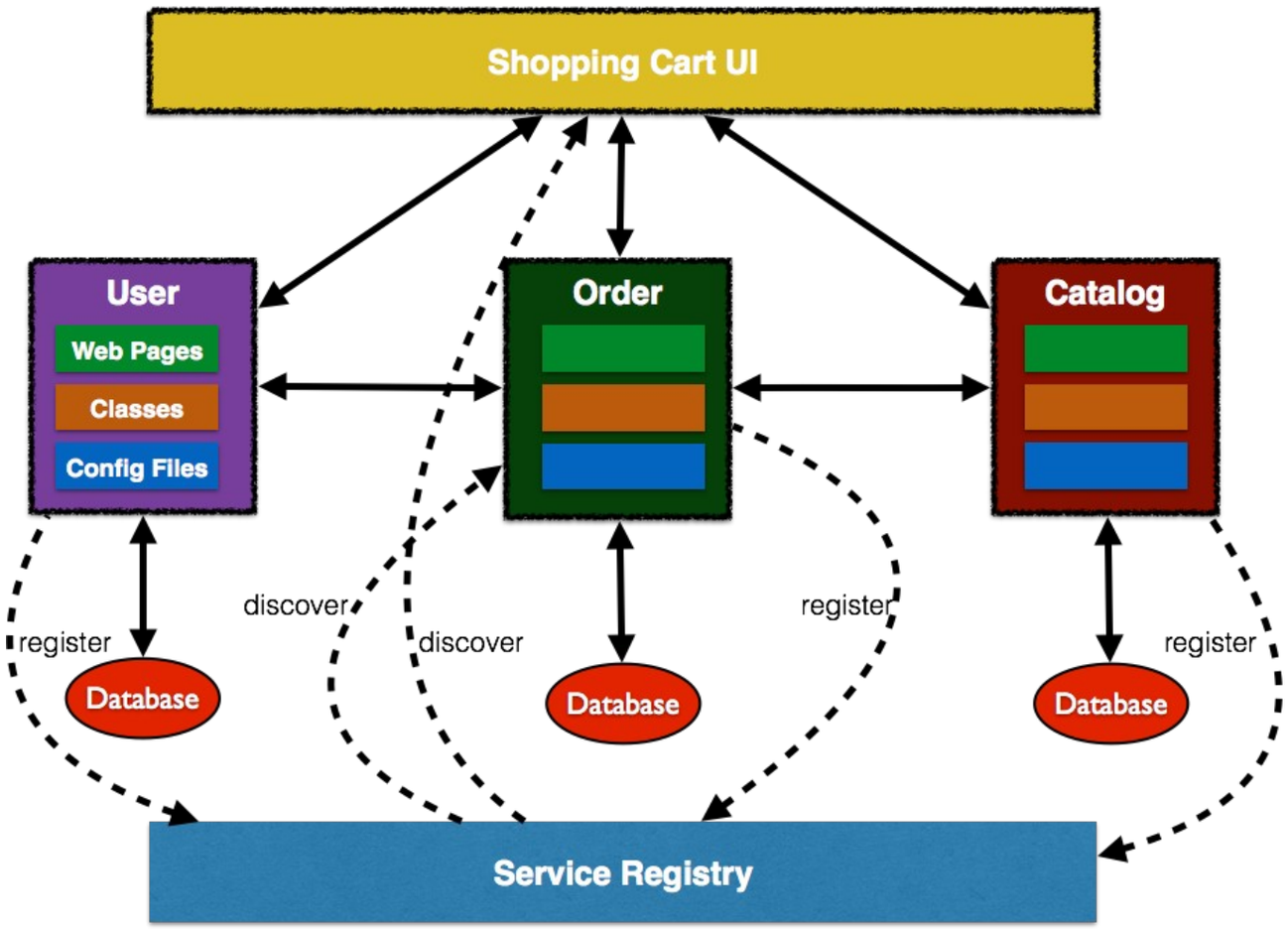
Disadvantages

- limited agility
- obstacle for continuous delivery
- “stuck” with a technology stack

Transforming the monolith

- Application functionality decomposed in individual processes
  - structure follows business logic
  - each has its own technology stack
    - Separate build / deployment process
  - no sharing of data stores
  - integrating UI defined as separate service
    - Common look and feel possible via standard CSS/JS

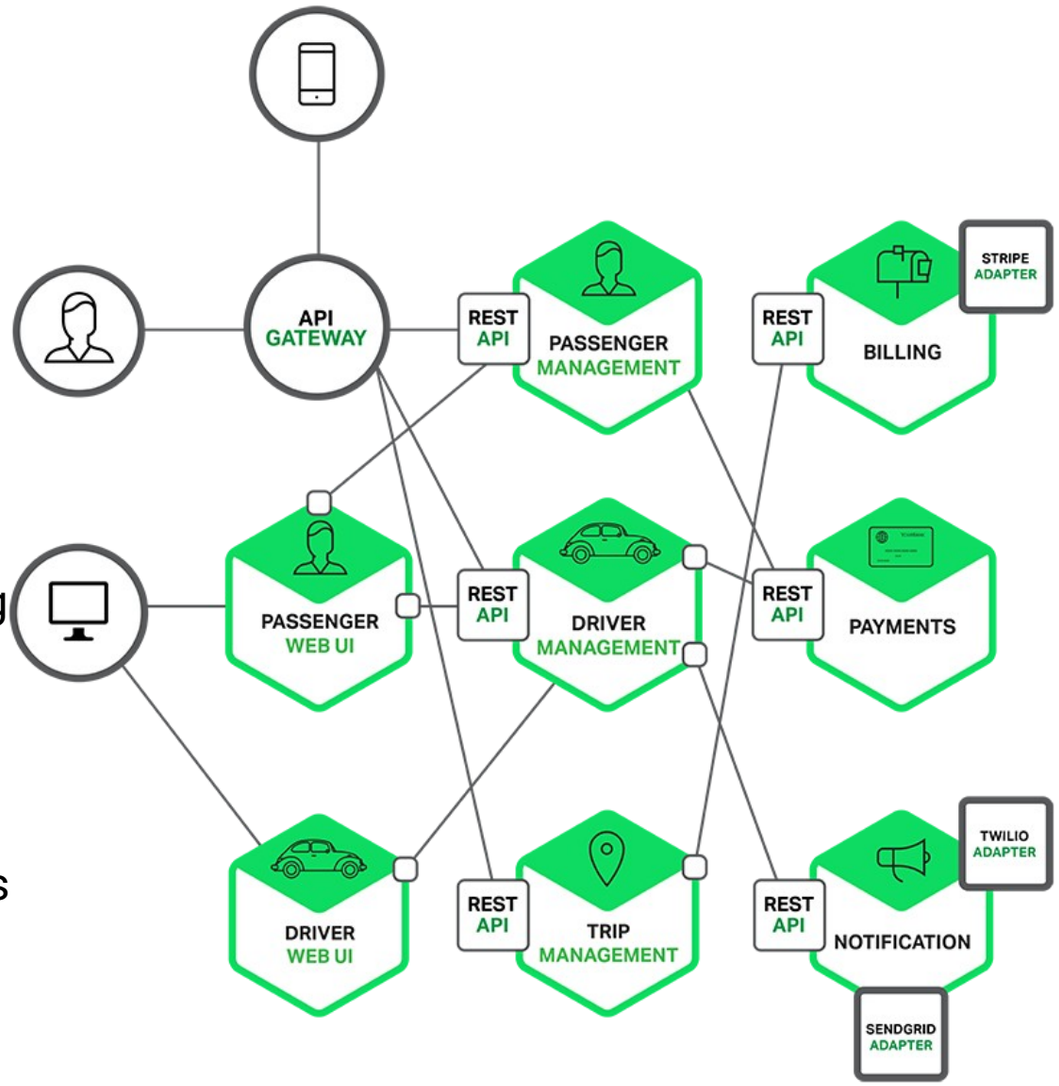
- Loose coupling via well-defined interfaces
  - Service registry (etcd, zookeeper)
  - REST or pub/sub interfaces



Componentization via services

- **Component**
- Unit of software that is independently replaceable or upgradeable
    - Libraries = linked into a program
    - Services = out-of-process component
- **So why choose services over libraries?**
    - Services are independently deployable
    - Changes in libraries = redeploying entire application
    - Explicit (remote) component interface when using services

- Services require more coarse-grained interfaces
    - remote calls more expensive than in-process calls

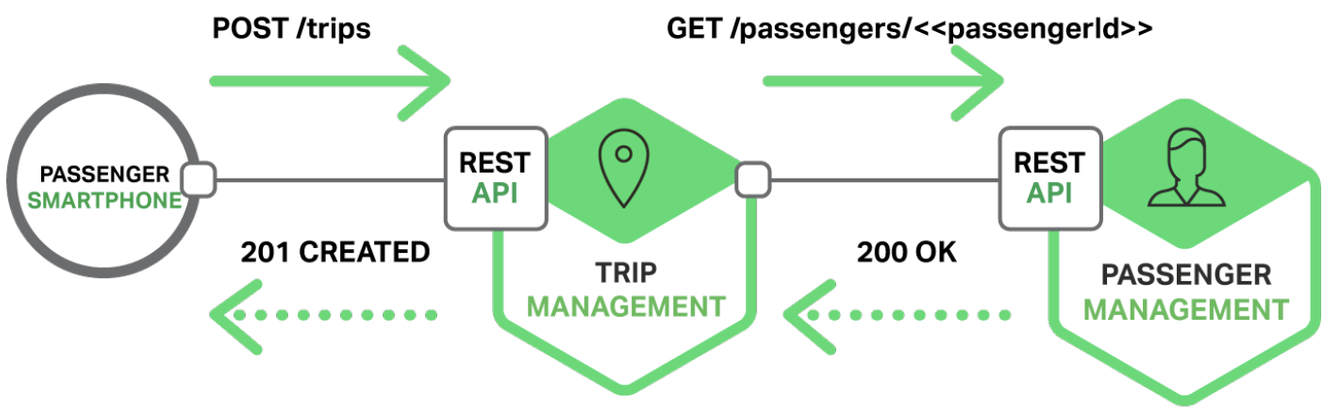


Smart services, dumb pipes

- **Enterprise Service Bus (ESB)**
  - Sophisticated message routing, choreography, transformation and business rule application

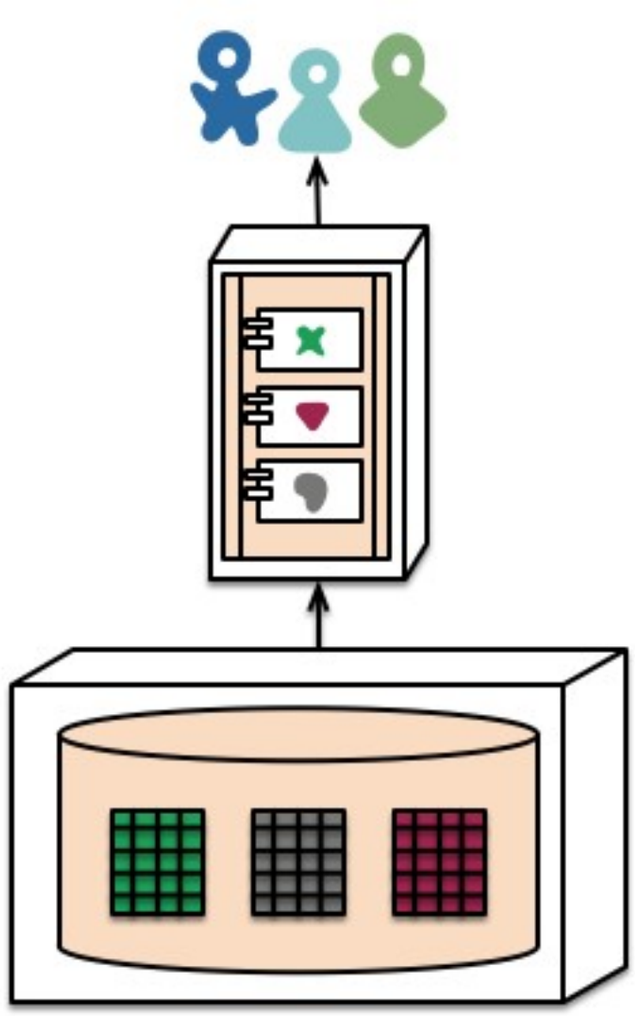
- **Microservices**
  - Act more as filters: receive request, apply logic, produce response

- Most commonly used protocols
  - REST (versus WS-Choreography, BPEL)
  - Reliable messaging over lightweight bus (Kafka, RabbitMQ, ZeroMQ, …)



Decentralized governance

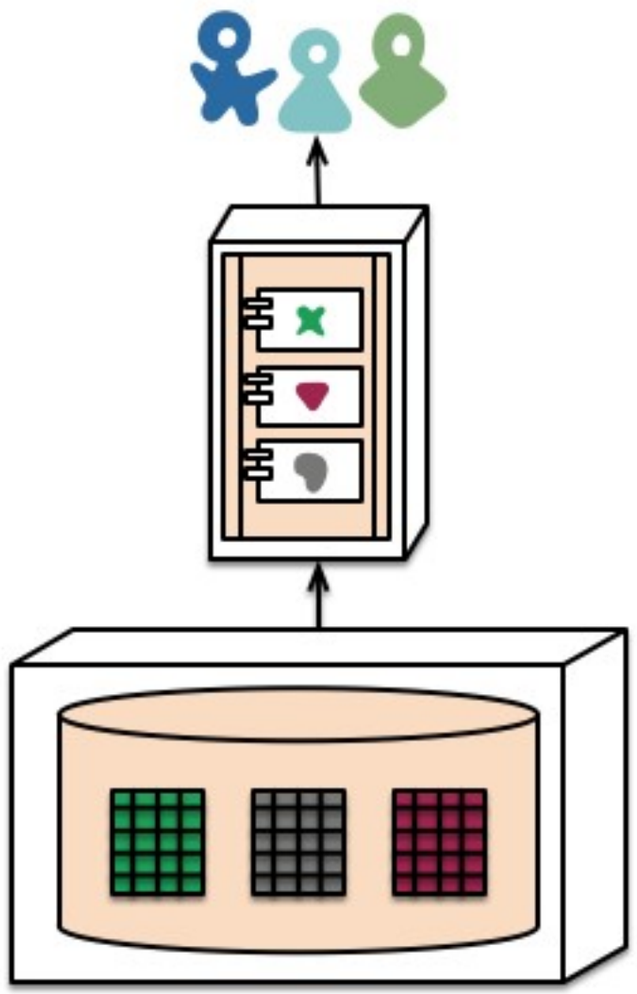
- Each service picks the right technology
- Service contracts define the interfaces
- Allows for different conceptual models, even of the same entity
  - e.g. “customer” is different for sales than it is for support department
- Polyglot persistence
- Transaction-less so cope with eventual consistency



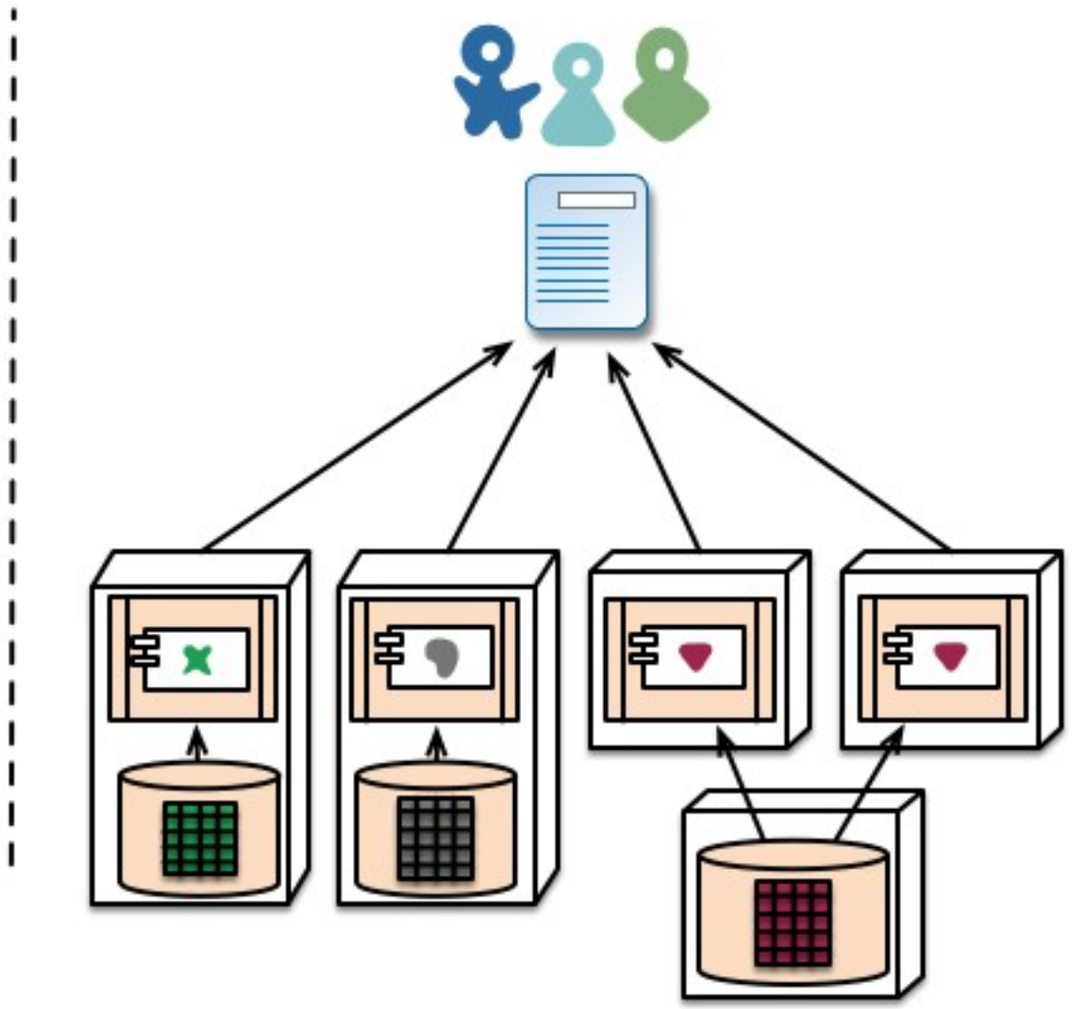
monolith - single database

Decentralized governance

- Each service picks the right technology
- Service contracts define the interfaces
- Allows for different conceptual models, even of the same entity
  - e.g. “customer” is different for sales than it is for support department
- Polyglot persistence
- Transaction-less so cope with eventual consistency



monolith - single database



microservices - application databases

Organized around business capabilities

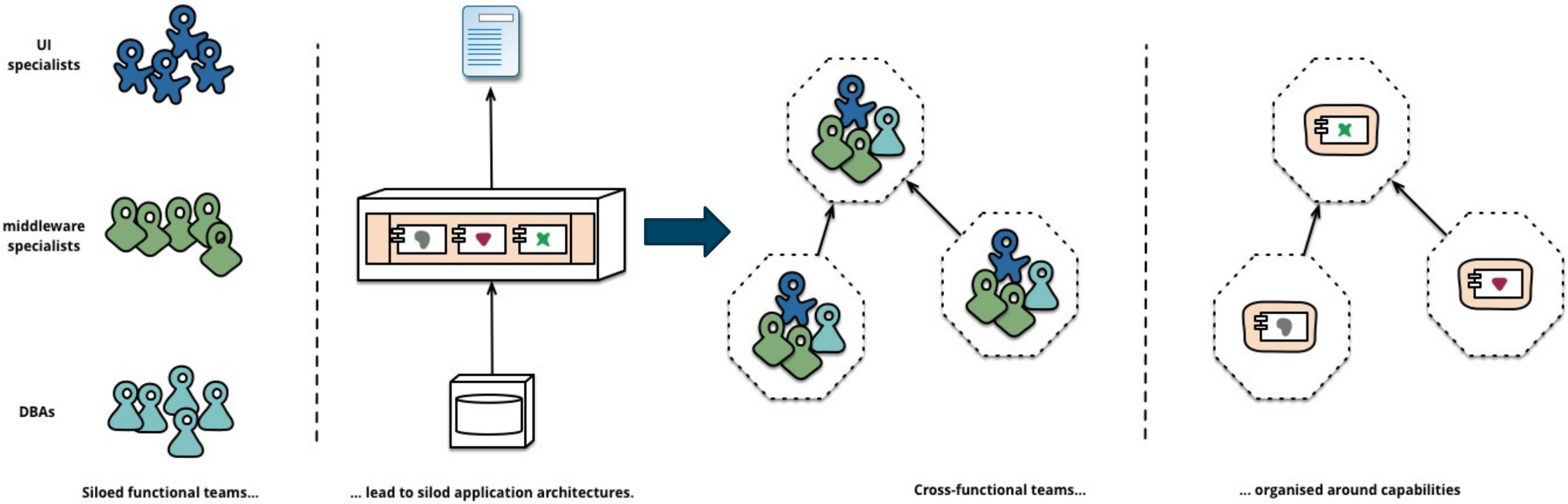
UI specialists

middleware specialists

DBAs

Siloed functional teams...

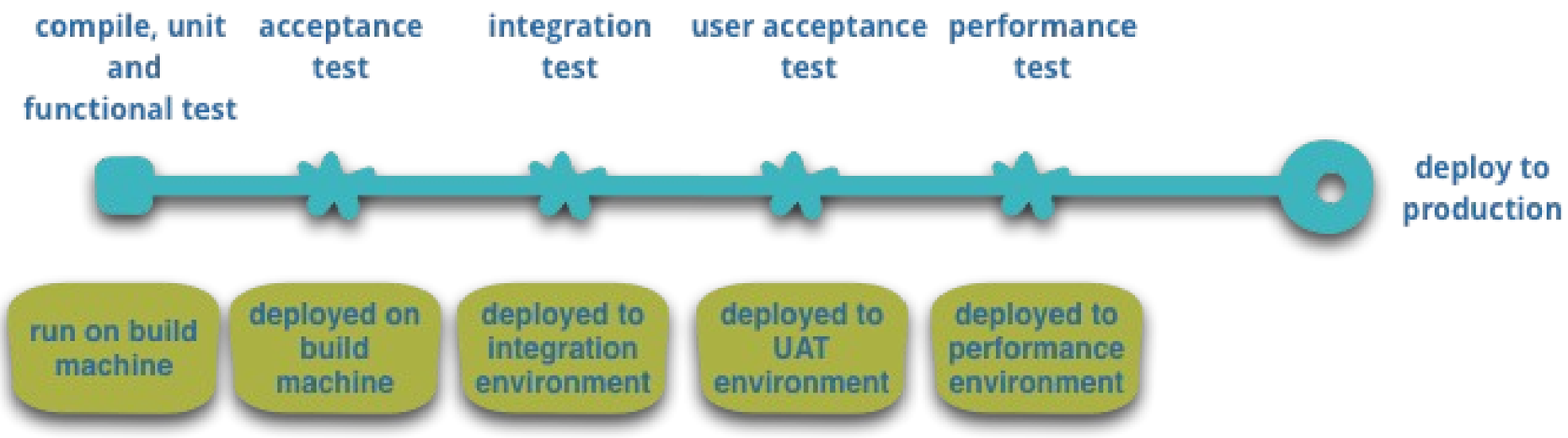
Organized around business capabilities



Continuous delivery automation

- Continuous integration
  - Integrating, building and testing code within the development environment
- **Continuous delivery**
  - Software <u>can</u> be released to production at any time
- Continuous deployment ~
  - Software is automatically pushed into production

- Automated tests ran at each stage of delivery.
- Deployment pipeline tools near-mandatory.



Products, not projects (aka DevOps)

“The traditional model is that you take your software to the wall that separates development and operations, throw it over and then forget about it.

Not at Amazon.

You build it, you run it.”

Werner Vogels, CTO Amazon

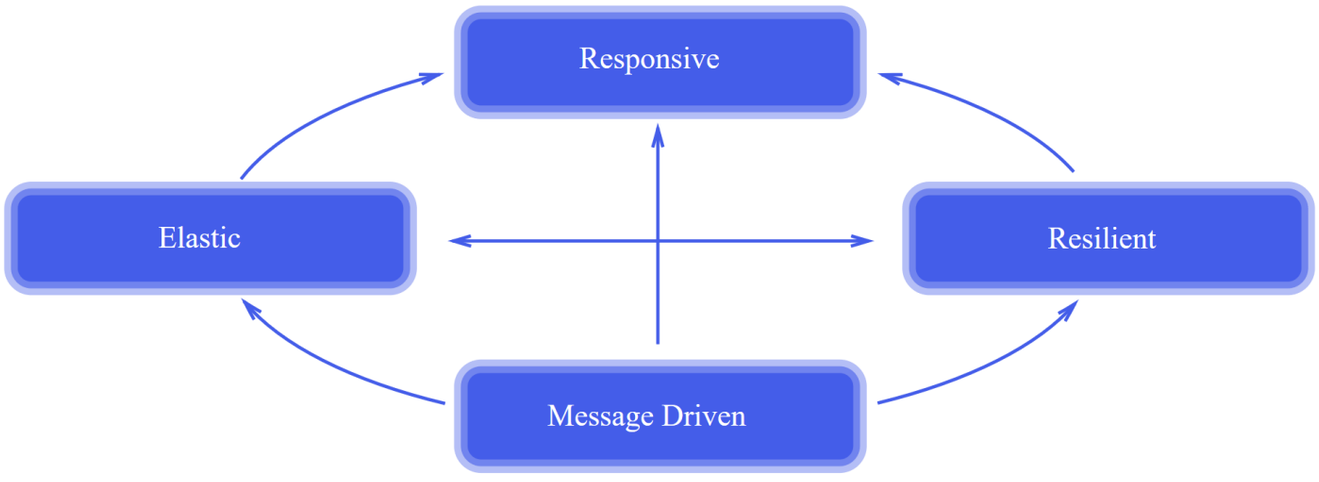
Break

[See you in 15 minutes] → 14h55

Reactive manifesto - www.reactivemanifesto.org

- Aims to condense knowledge on how to design highly scalable and reliable applications
  - Set of required architecture traits
  - Common vocabulary, technology-agnostic

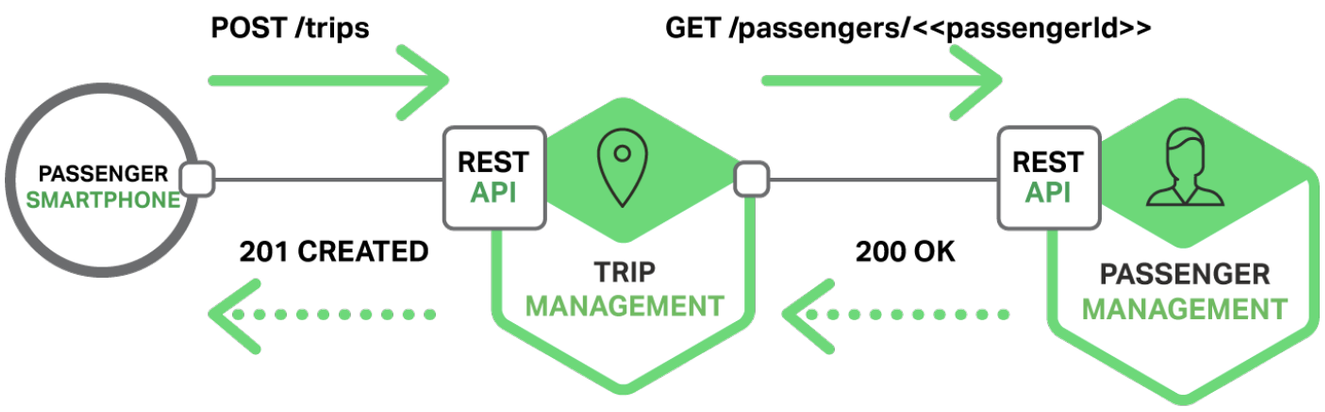
- Why? Because today's demands are simply not met by yesterday’s software architectures
  - Large applications a few years ago
    - >10 servers, response time: seconds, hours of offline maintenance, gigabytes of data
  - Today
    - Deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors, response time: milliseconds. Petabytes of data.



Architectural trait: **message driven**

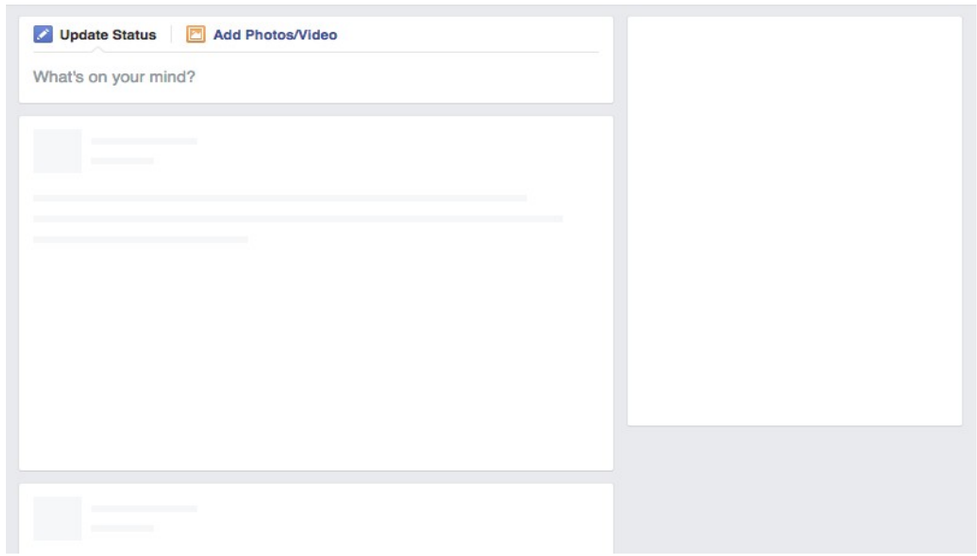
- Asynchronous message-passing between components
  - Addressable recipients await the arrival of messages and react to them
  - Establishes a boundary that enables
    - Loose coupling
    - Isolation
    - Location transparency

- Enables load management, elasticity and flow control by monitoring and shaping the message queues in the system



Architectural trait: **responsive**

- The system responds in a timely manner (aim: 0.1 seconds)
  - Client side **lazy loading**: first load important stuff and show it ASAP
  - Show progress
  - Individual slow performing service should not slow down others



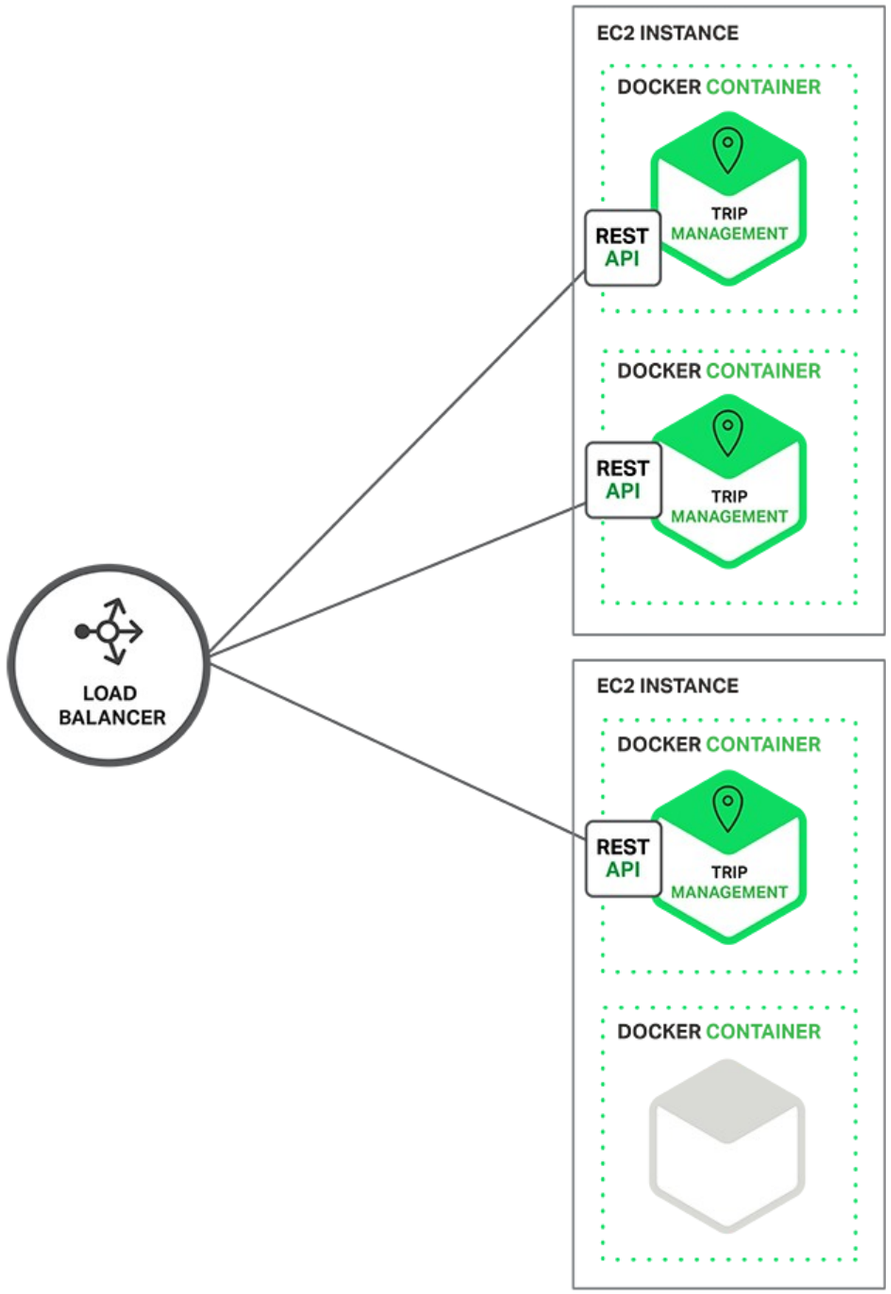
As far as users know, when the response time exceeds their expectation, the system is down

Architectural trait: **elastic**

- System stays responsive under varying workload
  - Changes in input rate lead to increased or decreased resource allocations
  - No contention points or central bottlenecks
  - Distribution of input amongst components

- An elastic system can allocate / deallocate resources for every individual component dynamically to match demand

- Predictive and reactive elastic scaling



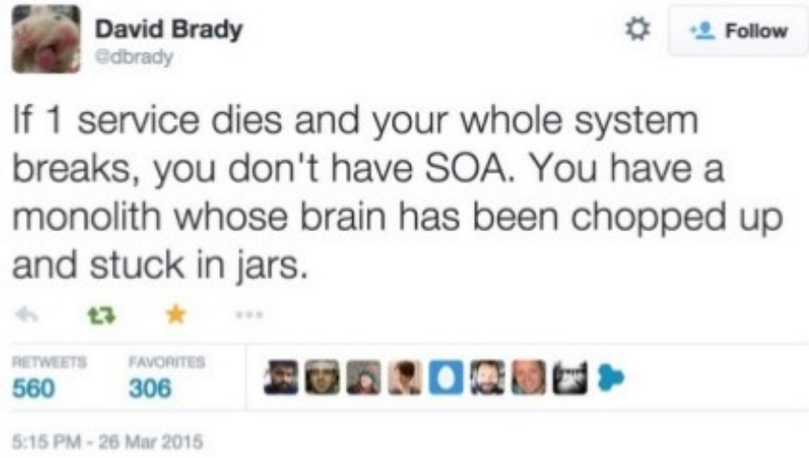
Architectural trait: **resilient**

- Any service call can fail

- Detect failures quickly by monitoring
- Service metrics (e.g. requests per second)
- Business metrics (e.g. orders per minute received)

  and automatically restore services when issues are detected

- Provide fallback services
- E.g. Netflix graceful degradation
  - If recommendation service is down revert to most popular movies instead of personalized picks



Architectural trait: **resilient**

No better example than...

[Taken from https://devops.com/wp-content/uploads/2014/03/simian-army.jpg]

Resilient to failure: Netflix's Simian Army

Chaos Monkey

- Introducing random failures in their production AWS services
- Team of engineers ready to intervene

Latency Monkey

- Introducing artificial delays

Janitor Monkey

- Seeks unused resources and disposes of them

Security Monkey

- Seek misconfigured services / security issues

The best defense against failures is to fail often, forcing your services to be built in a resilient way

Microservice frameworks

Microservice frameworks / dev tools

- **Dropwizard** (http://www.dropwizard.io)
  - Java framework for developing RESTful web services
- **VertX** (http://vertx.io/)
  - Toolkit for building reactive applications on JVM
- **Spring Boot** (https://projects.spring.io/spring-boot/)
  - Eases development of Spring applications
- **Restlet** (https://restlet.com/)
  - Microservice-oriented API creation / testing / execution
- **Spark** (http://sparkjava.com/)
  - Micro-framework for creating web applications in Java 8
- **Lagom** (http://www.lightbend.com/lagom)
  - Microservice framework
- **LimeDS** (http://www.limeds.be)
  - Rapid visual micro-service based application development
- And many more: WSO2, Jolie, Baratine, kumuluzEE, Jlupin, etc.

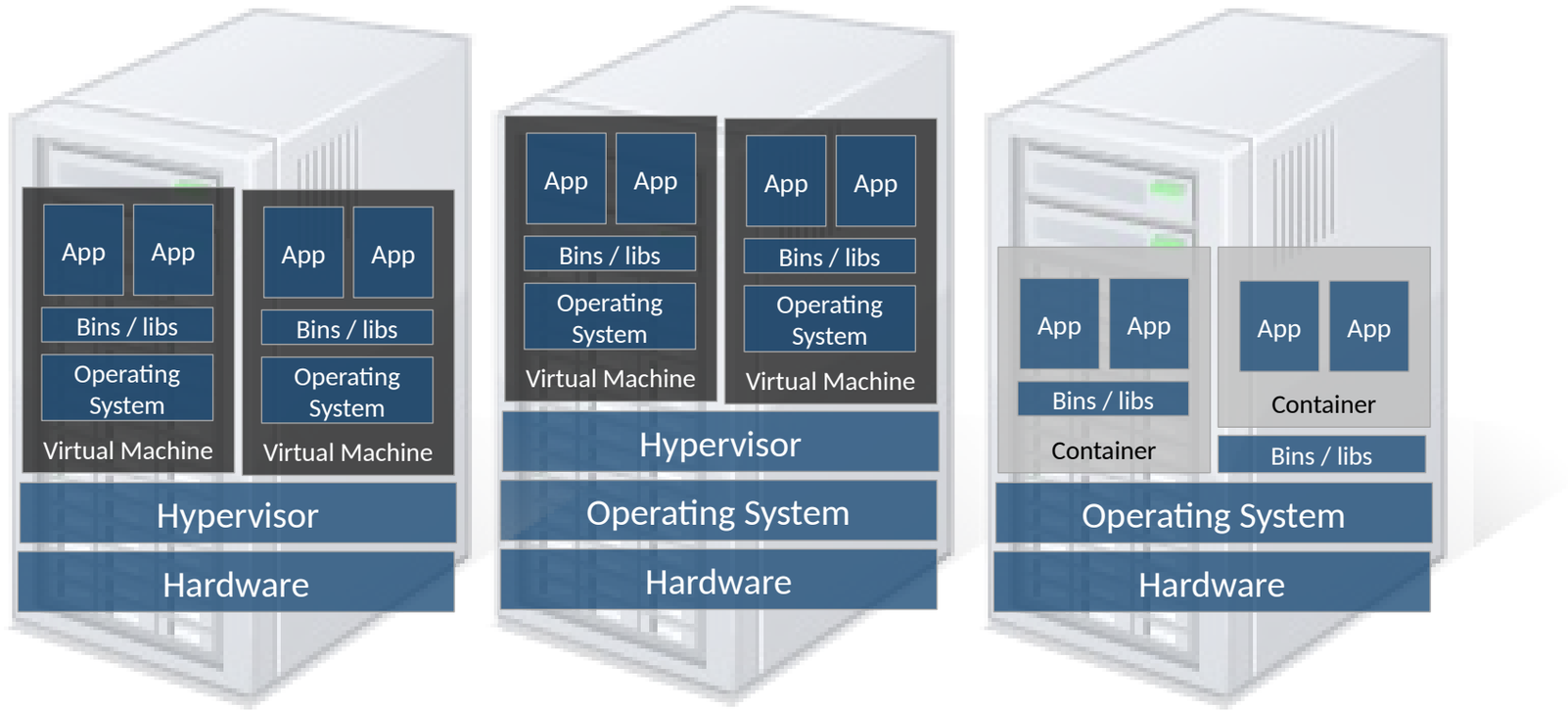
Interesting article comparing some of these:
https://cdelmas.github.io/2015/11/01/A-comparison-of-Microservices-Frameworks.html

Note:
- A lot of JVM-based microservice frameworks.
- JVM is no longer Java-language only
  it can also run Ruby, Python, Scala, Groovy, Clojure, JavaScript, etc.

So where’s the link with containers?

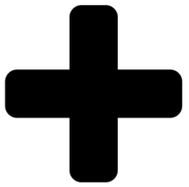
Containers as a microservice enabler

- **VMs**
  - Virtualization of hardware.
  - Flexible, robust and safe, but big performance hit
- **Containers**
  - Lightweight
  - Better use of resources (sharing host OS and potentially binaries/ libraries)



Type 1 Hypervisor | Type 2 Hypervisor | OS-level virtualization

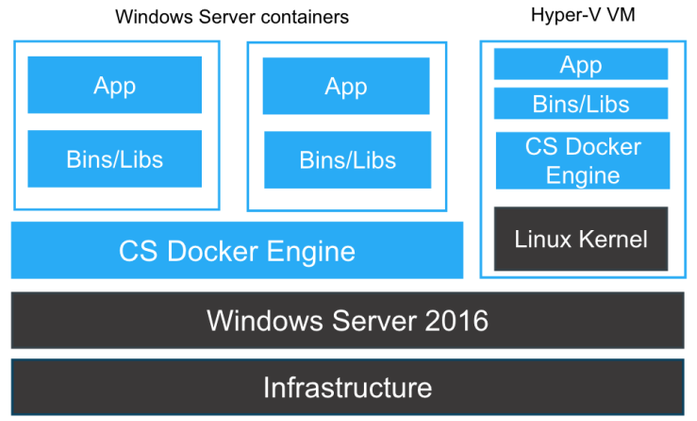
Pros and cons of containers



No performance hit due to emulation of instructions

Flexibility
- containerize a “system”
- containerize “application(s)”

Lightweight
- no entire OS in each container
- sharing of bins and libraries
- provisioned/instantiated in milliseconds to seconds
- minimal per-container penalty
- “Just Enough OS” on the server (e.g. CoreOS)



Cannot host a guest OS different from the host one*
Weaker isolation and thus security

*actually Docker on Windows also supports Hyper-V containers where Docker runs in a small Linux VM (Alpine Linux-based)

Container checklist

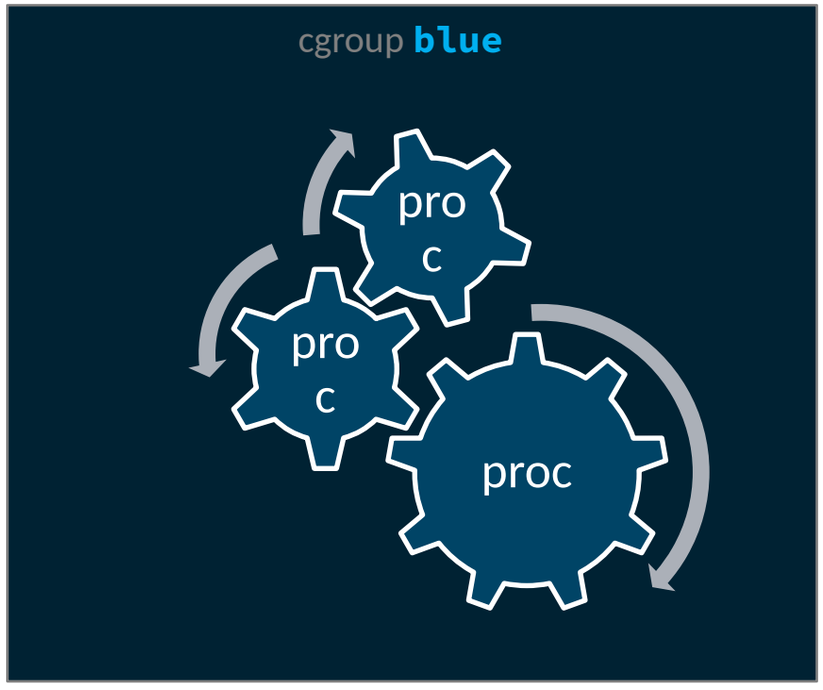
- Processes
- Throttling/limits
- Prioritization
- Resource isolation
- Root file system
- Security

Focus on **Linux Containers (LXC)** to explain the basics

Linux Control Groups (cgroups)

- **Problem**
- How do I throttle, prioritize, control and obtain metrics for a group of tasks (processes)?
- **Solution:** Linux control groups (cgroups)

- **Device Access**
- **Resource limiting**: memory, CPU, devices, block I/O, etc..
- **Prioritization**: who gets more of the CPU, memory
- **Accounting**: resource usage per group
- **Control**: freezing and checkpointing
- **Injection**: packet tagging

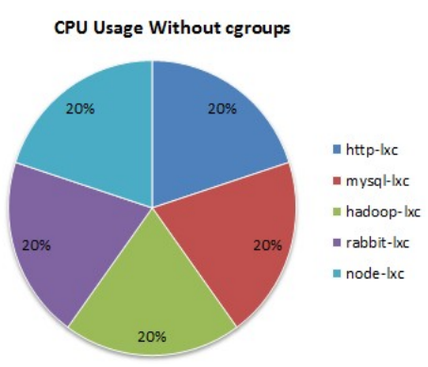


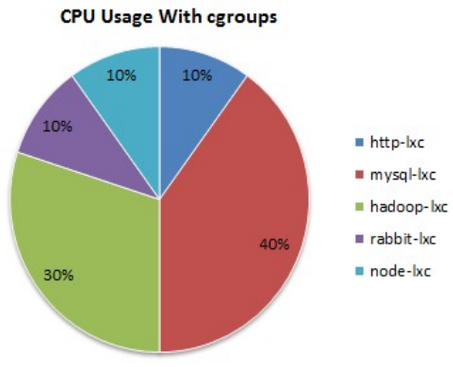
Linux cgroup subsystems

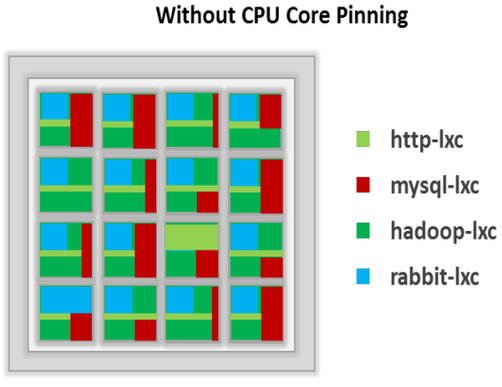
Subsystem	Tunable Parameters
blkio	- Weighted proportional block I/O access. Group wide or per device. - Per device hard limits on block I/O read/write specified as bytes per second or IOPS per second.
cpu	- Time period (microseconds per second) a group should have CPU access. - Group wide upper limit on CPU time per second. - Weighted proportional value of relative CPU time for a group.
cpuset	- CPUs (cores) the group can access. - Memory nodes the group can access and migrate ability. - Memory hardwall, pressure, spread, etc.
devices	- Define which devices and access type a group can use.
freezer	- Suspend/resume group tasks.
memory	- Max memory limits for the group (in bytes). - Memory swappiness, OOM control, hierarchy, etc..
hugetlb	- Limit HugeTLB size usage. - Per cgroup HugeTLB metrics.
net_cls	- Tag network packets with a class ID. - Use tc to prioritize tagged packets.
net_prio	- Weighted proportional priority on egress traffic (per interface).

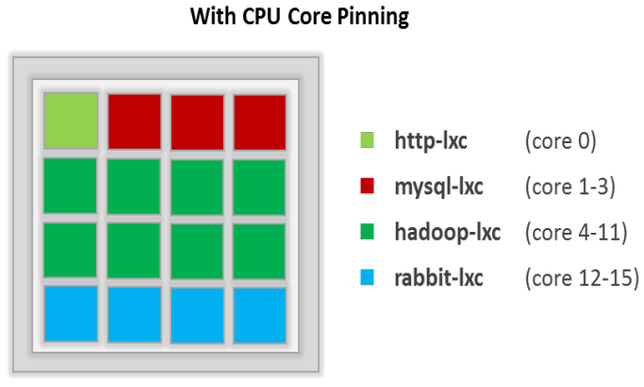
Linux cgroups: CPU control

- Use CPU shares (and other controls) to prioritize jobs / containers
- Carry out complex scheduling schemes
- Segment host resources
- Adhere to SLAs
- Pin containers / jobs to CPU cores
- Reduce core switching cost



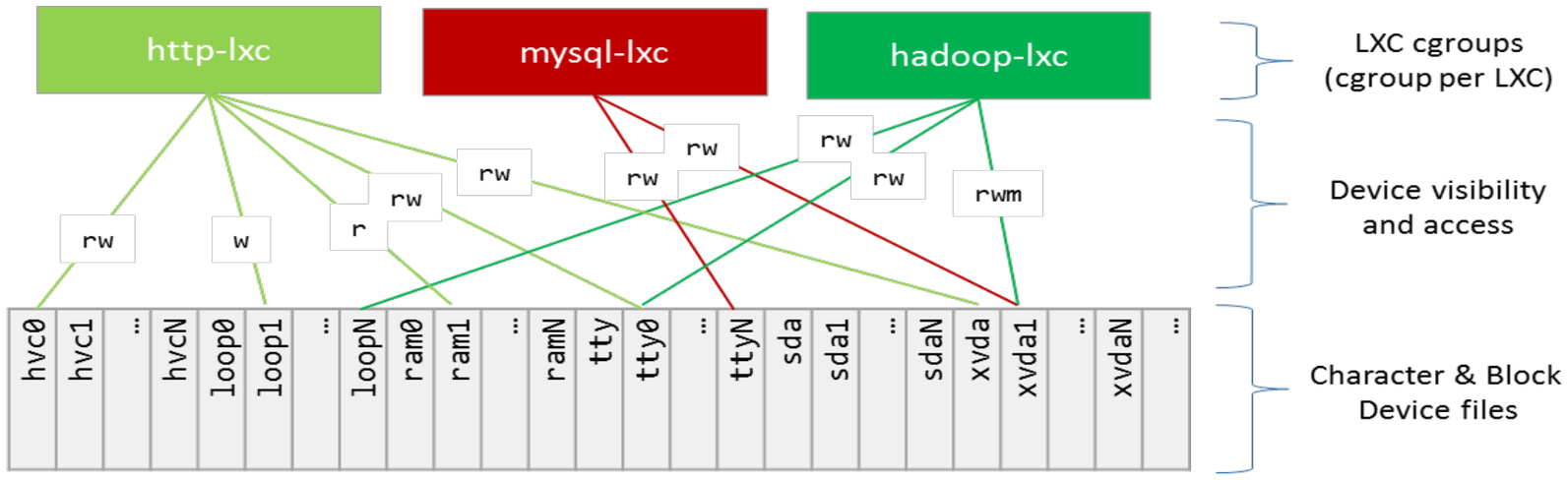




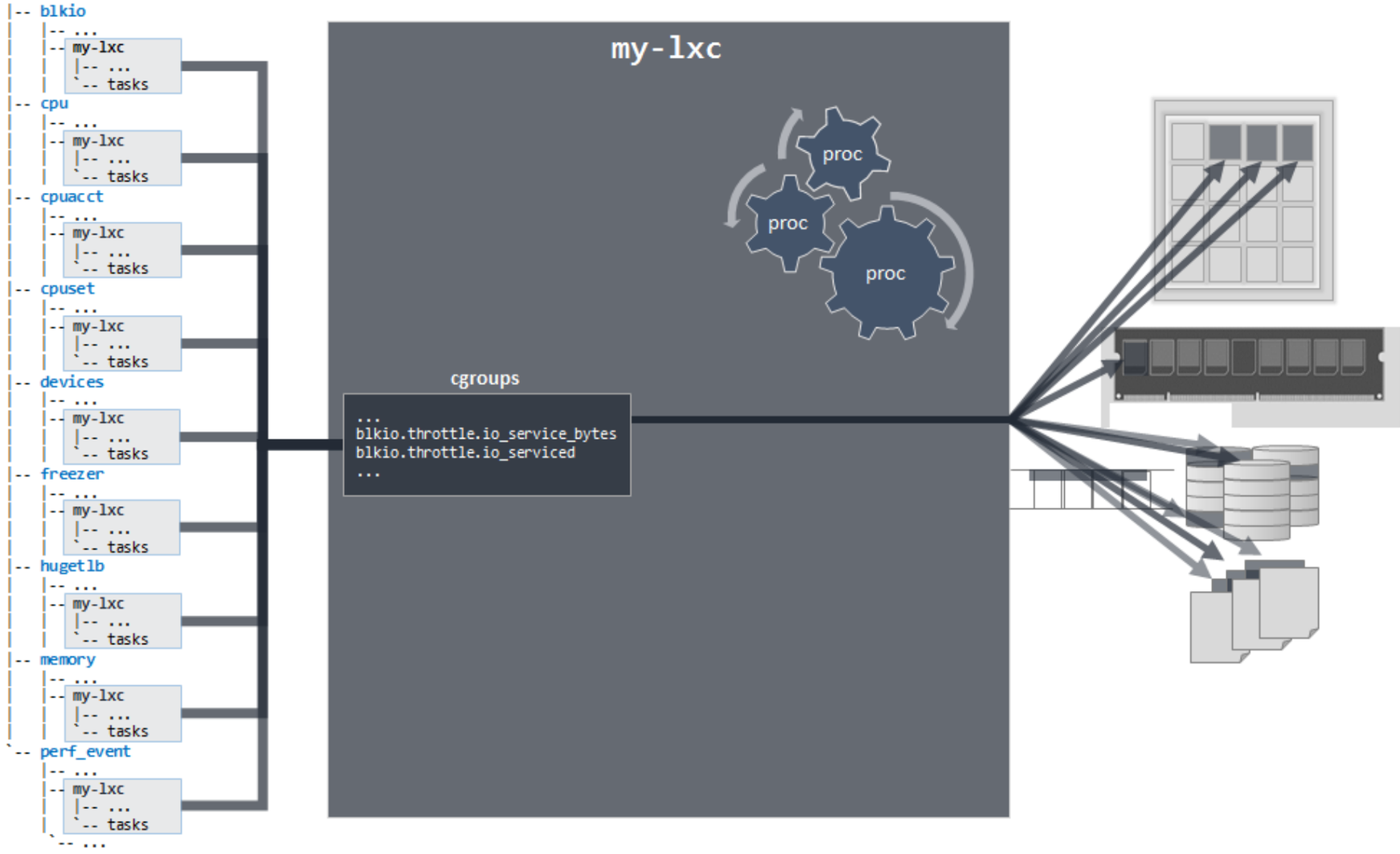


Linux cgroups: device access

- Limit device visibility; isolation
- Implement device access controls
- Secure sharing
- Segment device access
- Device whitelist / blacklist



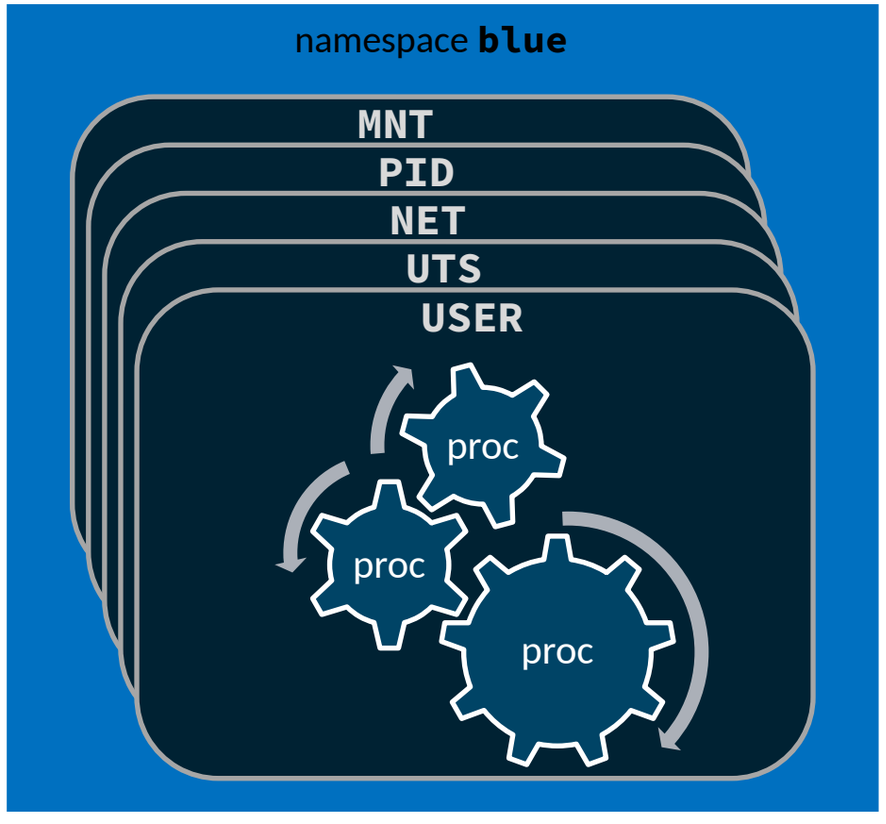
Container enabler: **cgroups**



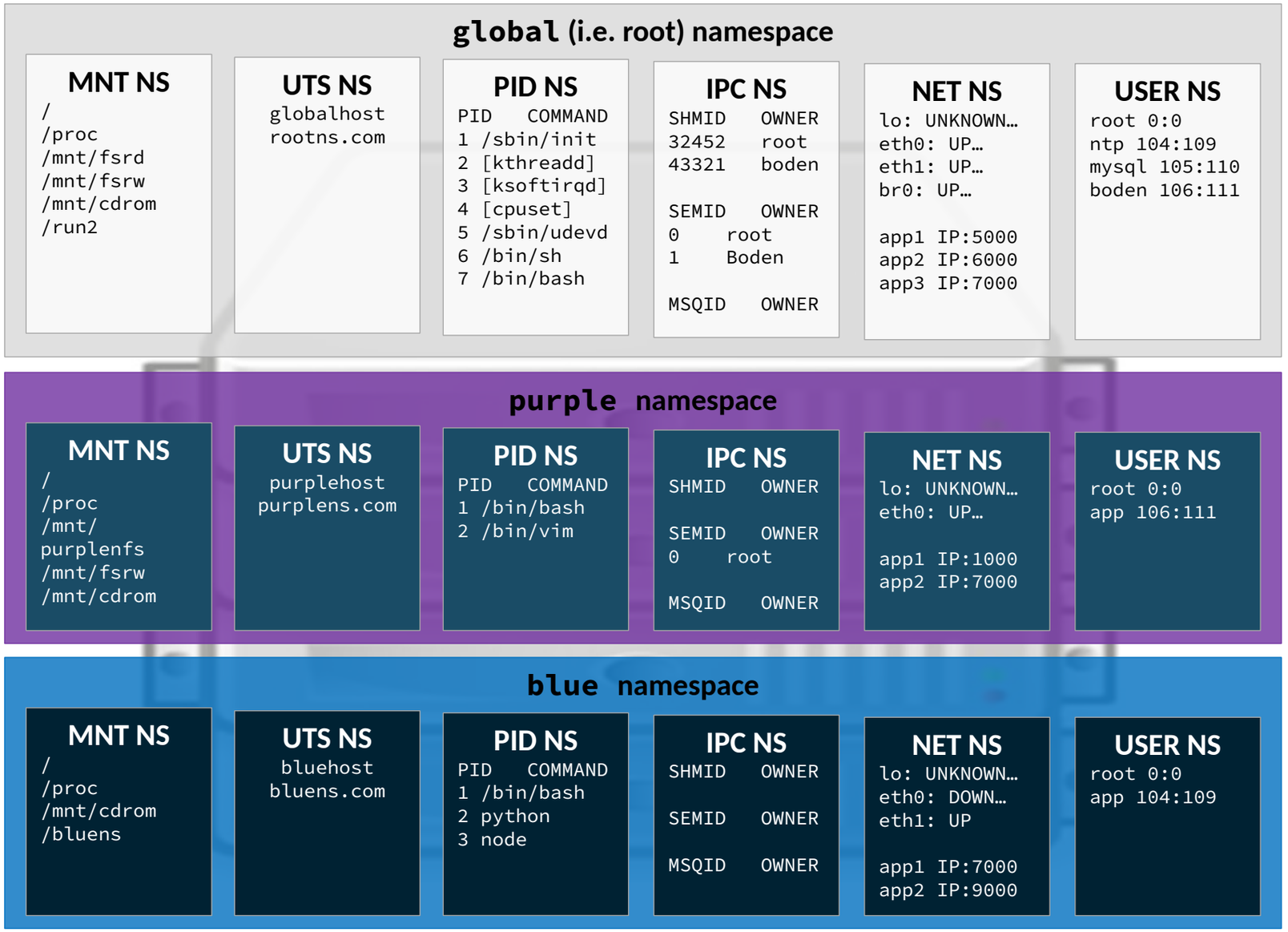
Linux namespaces

- **Problem**
- How do I provide an isolated view of global resources to a group of tasks (processes)?
- **Solution** namespaces
- Create abstraction of a system resource and make it appear as a separated instance to processes within a namespace

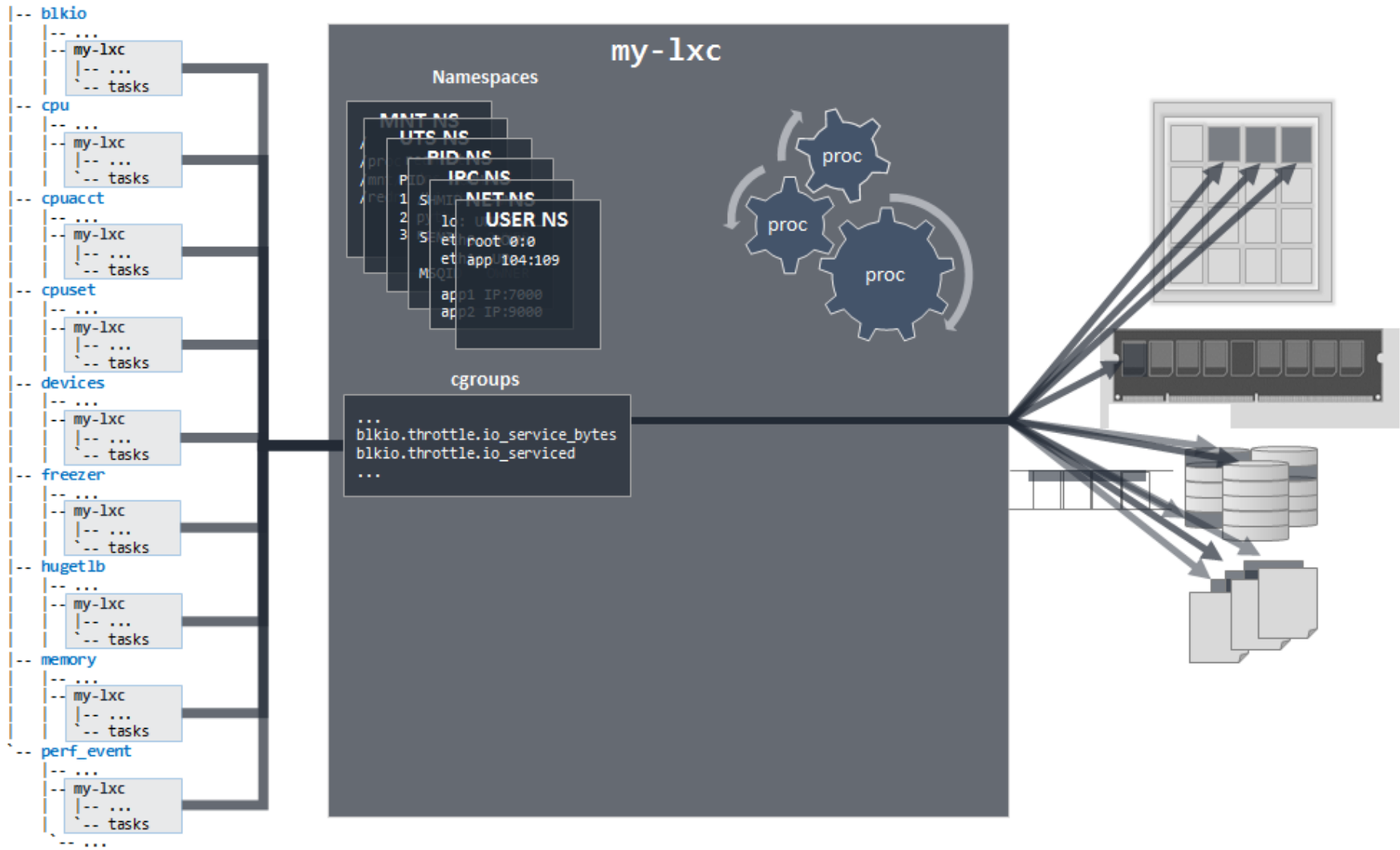
- **MNT;** mount points, files systems, etc.
- **PID;** processes
- **NET;** NICs, routing, etc.
- **IPC;** System V IPC
- **UTS;** host and domain name
- **USER;** UID and GID



Linux namespaces: conceptual overview

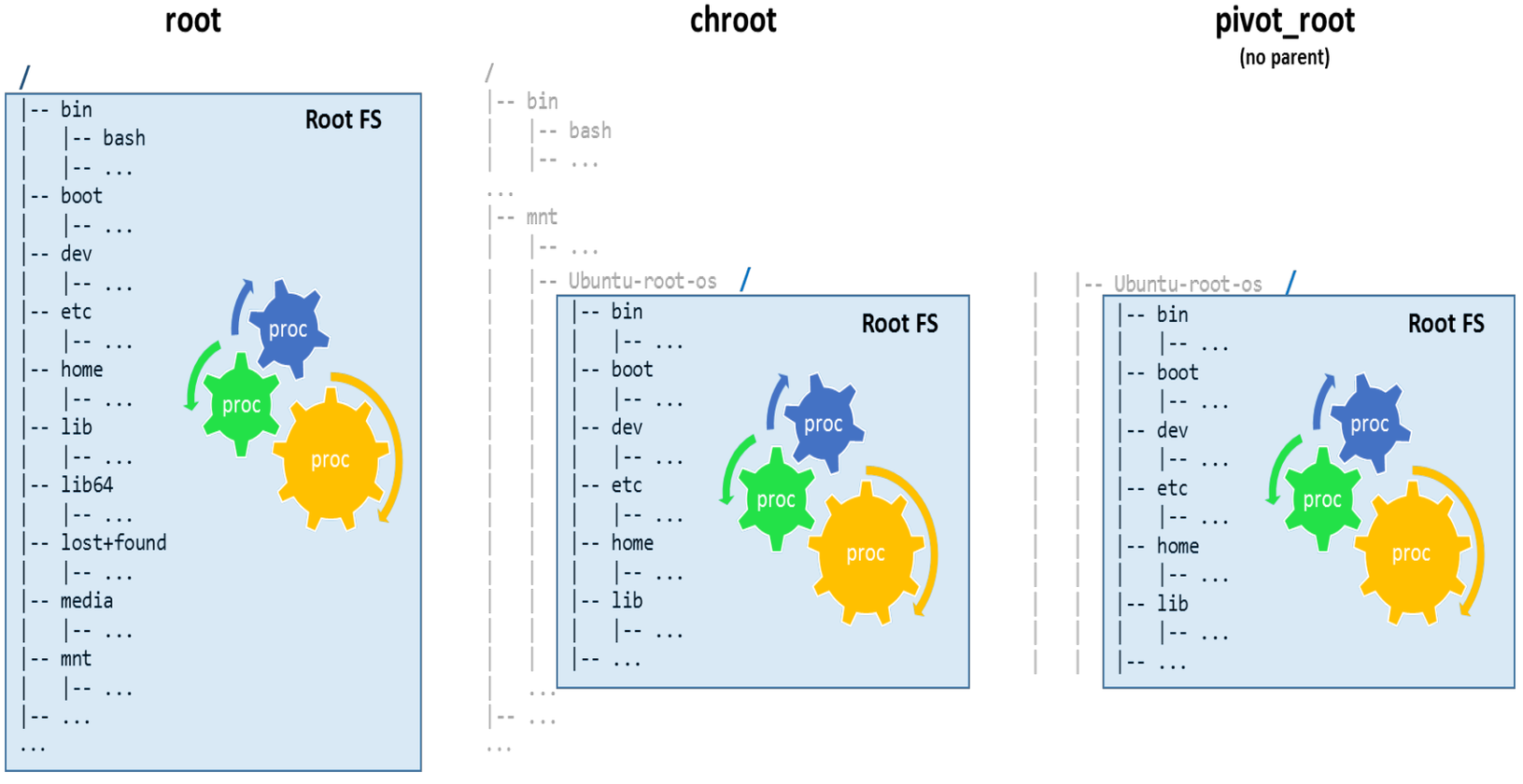


Container enabler: **namespaces**

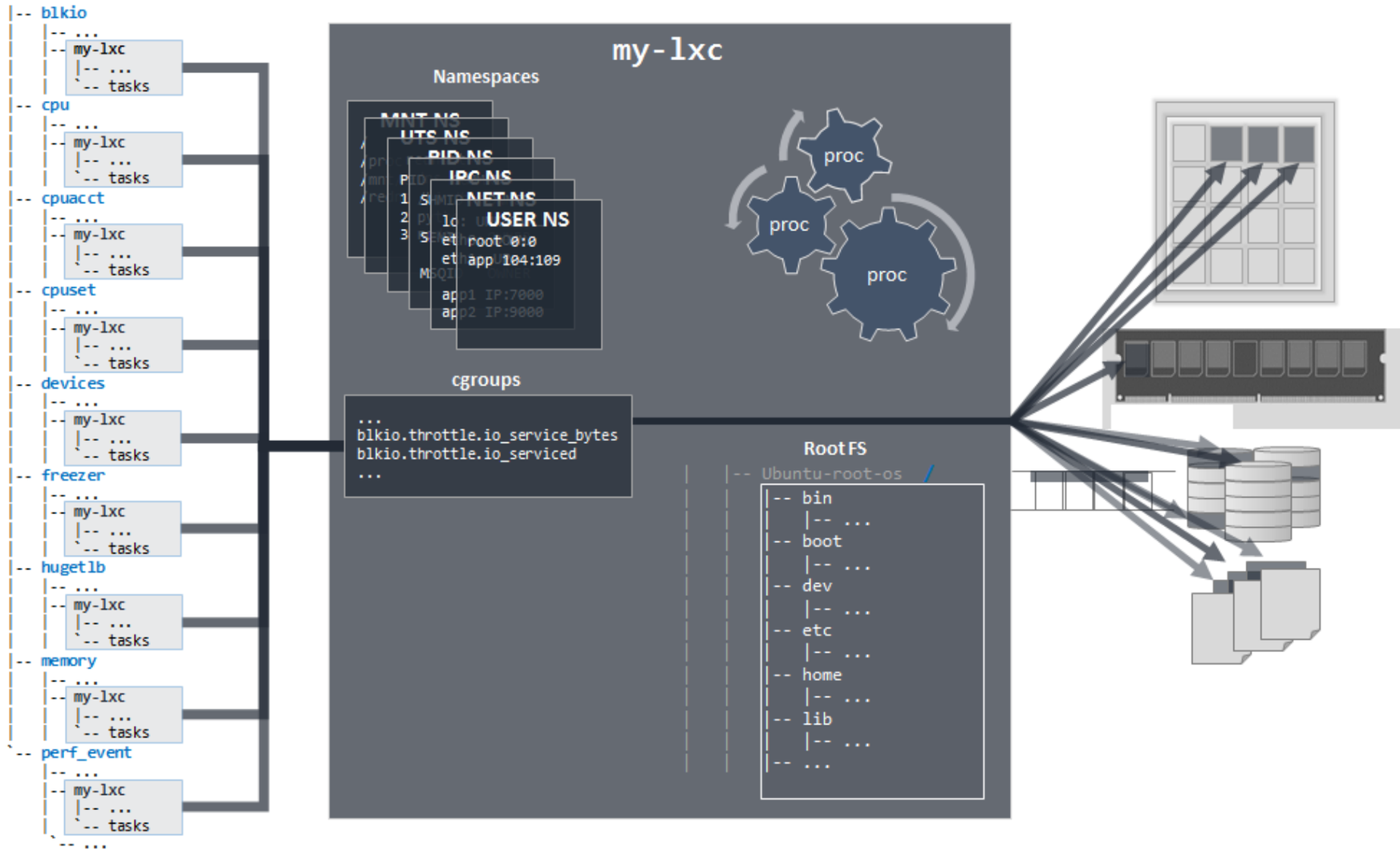


Changing root

- Problem
- Give each process the illusion that it is mounted from the root directory
- Solution
- **pivot_root** command duplicates entire root directory in MNT namespace



Container enabler: **separate filesystem from root**

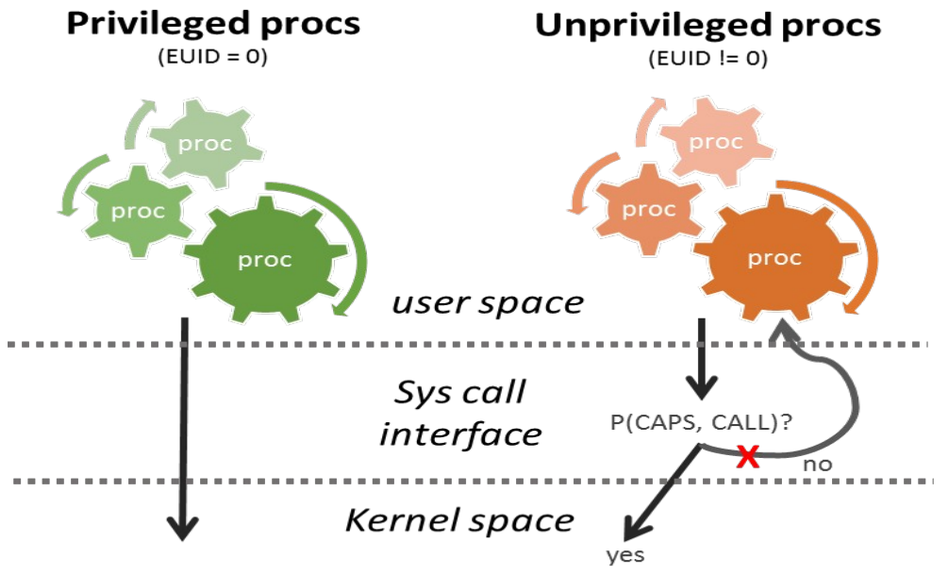


Secure containers

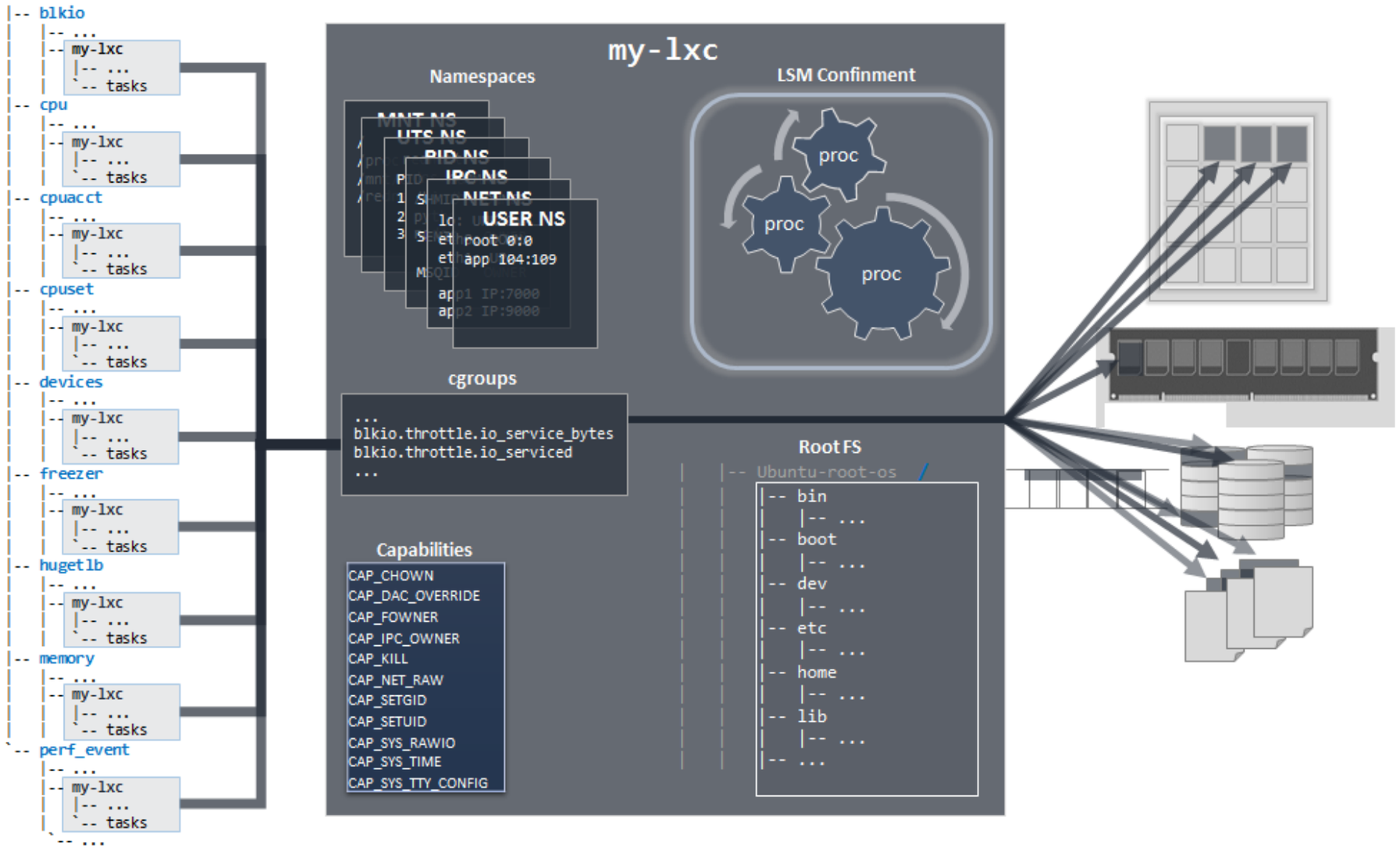
- **Problem**
- Hostile process can break out of container since entire system is not namespaced or containerized
- Some namespaces have leaks
  - If kernel has exploits, the container can exploit these too
- **Solution** – Linux Security Modules (LSM)
- Mandatory access control
- Define capabilities per process (system call access)

Capabilities

CAP_AUDIT_CONTROL
CAP_AUDIT_WRITE
CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH
CAP_FOWNER
CAP_FSETID
CAP_IPC_LOCK
CAP_IPC_OWNER
CAP_KILL
CAP_LEASE
CAP_LINUX_IMMUTABLE
CAP_MKNOD
CAP_NET_ADMIN
CAP_NET_BIND_SERVICE
CAP_NET_BROADCAST
CAP_NET_RAW
CAP_SETGID
CAP_SETPCAP
CAP_SETUID
CAP_SYS_ADMIN
CAP_SYS_BOOT
CAP_SYS_CHROOT
CAP_SYS_MODULE
CAP_SYS_NICE
CAP_SYS_PACCT
CAP_SYS_PTRACE
CAP_SYS_RAWIO
CAP_SYS_RESOURCE
CAP_SYS_TIME
CAP_SYS_TTY_CONFIG

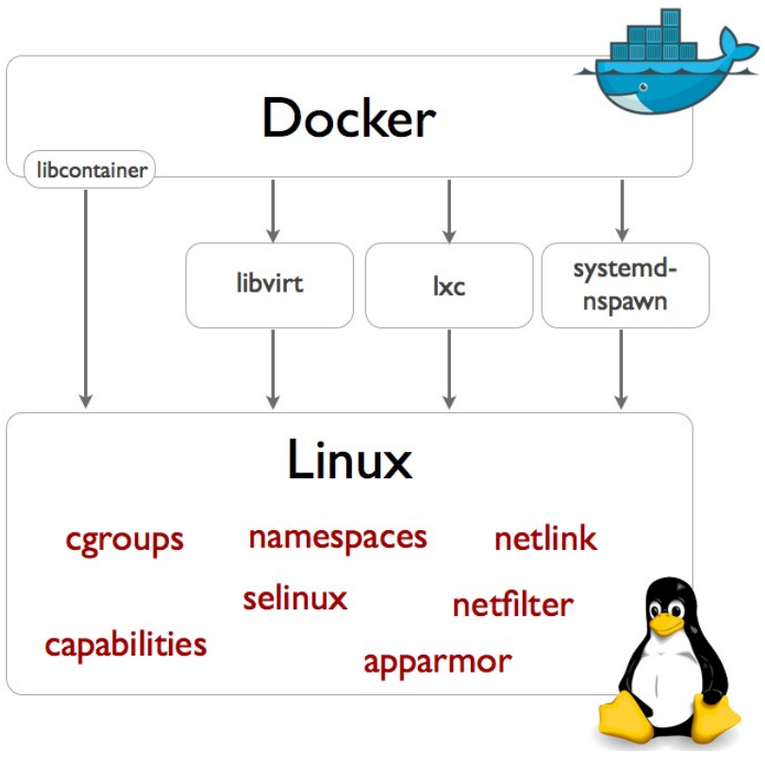


Container enabler: **Linux security modules**



Related Technologies

Containers



Lightweight OS

CoreOS

PROJECT ATOMIC

Core

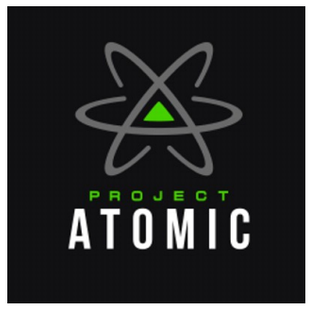
Nano Server

Management

kubernetes

Lightweight OS

- Lightweight OSs being developed focused on container-usage
  - CoreOS (focused on server containers)
  - Red Hat Project Atomic (focused on server containers)
  - Ubuntu Core / Snappy (focused on IoT)
  - Microsoft Nano Server (focused on server containers)

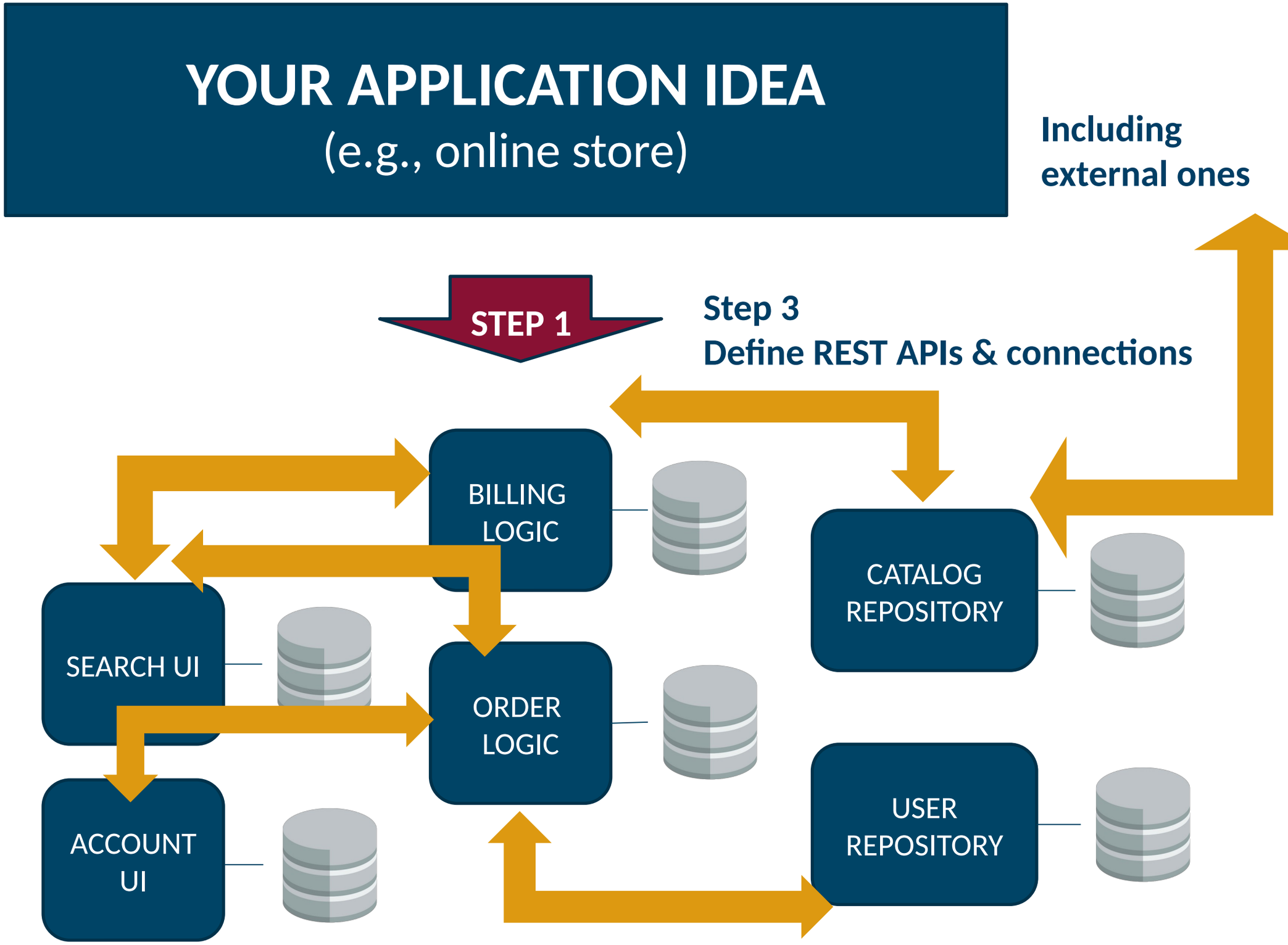


Overview of container management systems

- Amazon EC2 container service (https://aws.amazon.com/ecs/)
  - Docker over managed cluster of Amazon EC2 instances
- Azure Container Services (https://azure.microsoft.com/en-us/services/container-service/)
  - Manage containers on Azure
- CoreOS Fleet (https://coreos.com/fleet/)
  - Container management for CoreOS clusters
- Docker Swarm (https://www.docker.com/products/docker-swarm)
  - Dockers native cluster management system
- Google Container Engine (https://cloud.google.com/container-engine/)
  - Kubernetes-based Docker container management on the Google Cloud platform
- Kubernetes (http://kubernetes.io/)
  - Open source, automated container deployment, scaling and management
- Mesosphere Marathon (https://mesosphere.github.io/marathon/)
  - Container orchestration platform for Mesos
- Nomad (https://www.nomadproject.io/)
  - Distributed, high-availability, datacenter-aware scheduler

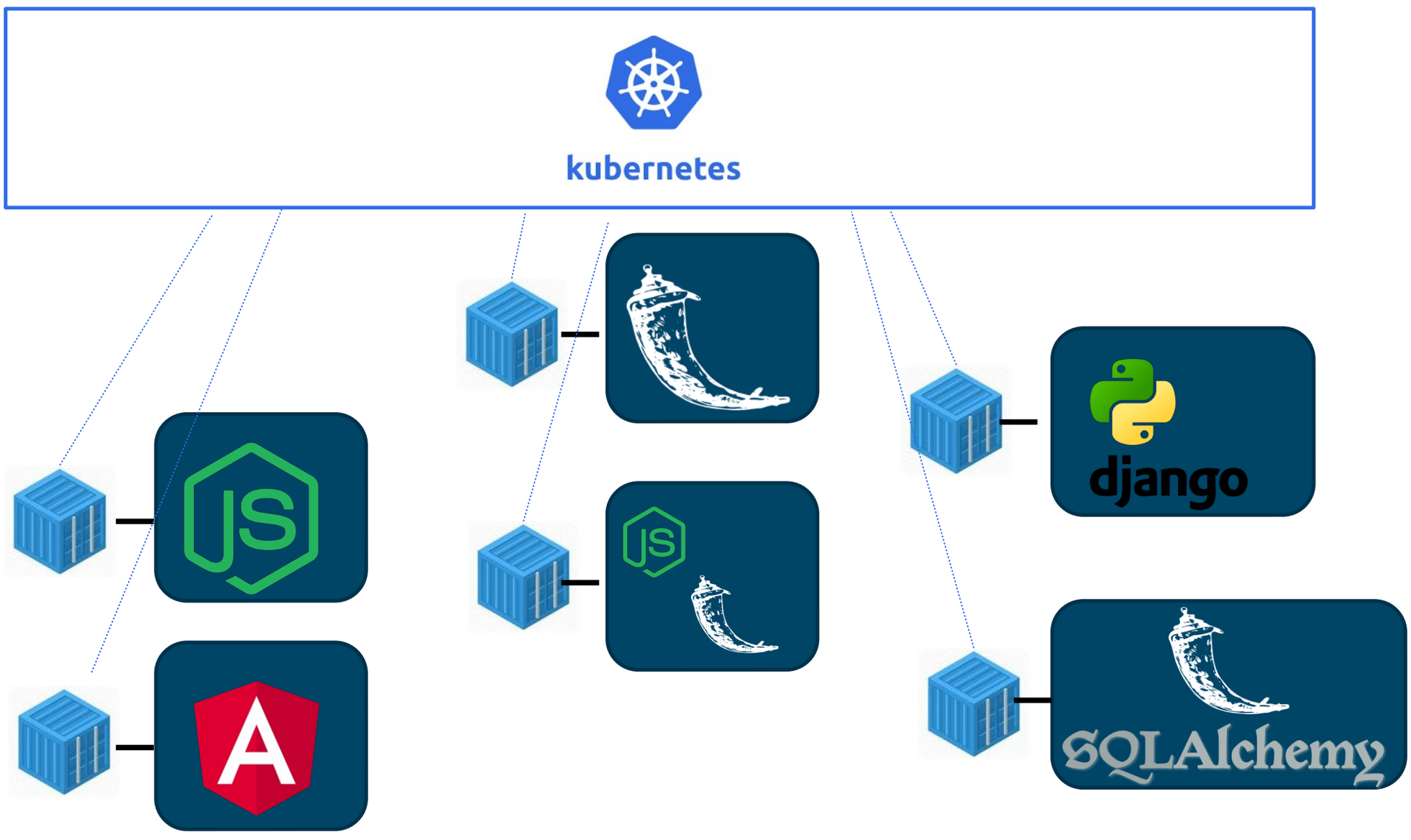
Bringing it all together….
How do you build a future proof distributed architecture?

Architectural view



Deployment view

Step 1: Choose which technology & platform is most suited for each microservice
Step 2: Package each service in a container
Step 3: Let Kubernetes manage the containers
+ setup scaling rules
Step 4: Deploy them on your cloud or infrastructure



To Summarize…

Microservices

- A single application as a suite of small services
- Smart nodes → light communication

Relationships w.r.t. container technologies

- Deployment unit for microservices
- Light and flexible

Pay Attention to…

Microservices

- Main characteristics and design principles
- Positioning wrt. to other service-related concepts

Relationships w.r.t. container technologies

- Main characteristics
- Positioning wrt. VMs
- How they enable microservices

Questions?

Further Reading

- **The Reactive Manifesto**
  https://www.reactivemanifesto.org/

- **Microservices**
  http://www.w3.org/TR/ws-gloss/#webservice

- **Netflix's Symian Army**
  https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116

Microservices

José Oramas

IDLab
INTERNET & DATA LAB

imec