# Embedded Systems & IoT architecture

David Ngo

2021-02-25

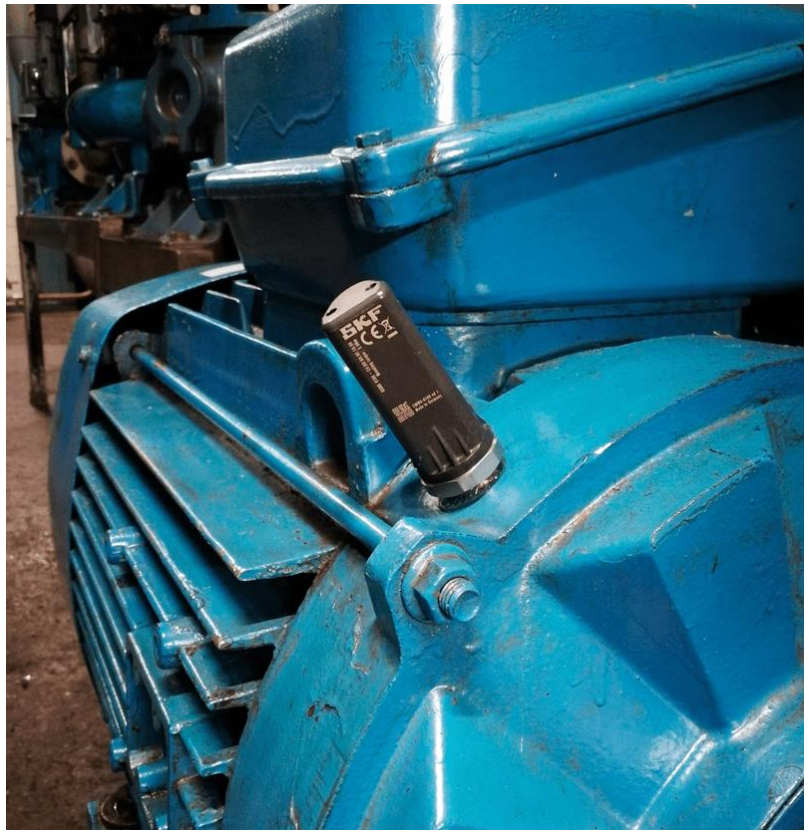
# About me

Working with embedded systems since 2008, focus on embedded software development

Mainly in medtech and industrial

Helping SKF to deliver Condition Monitoring products and be the market leader of rotating equipment performance.


SKF

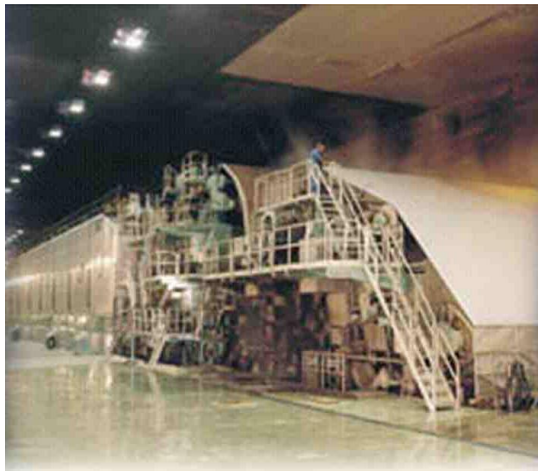

# SKF Condition Monitoring products

# SKF Industry presence

Railway vehicles

Mining

Power Generation
Steam Turbines



Paper Machines

Power Generation
Steam Turbines

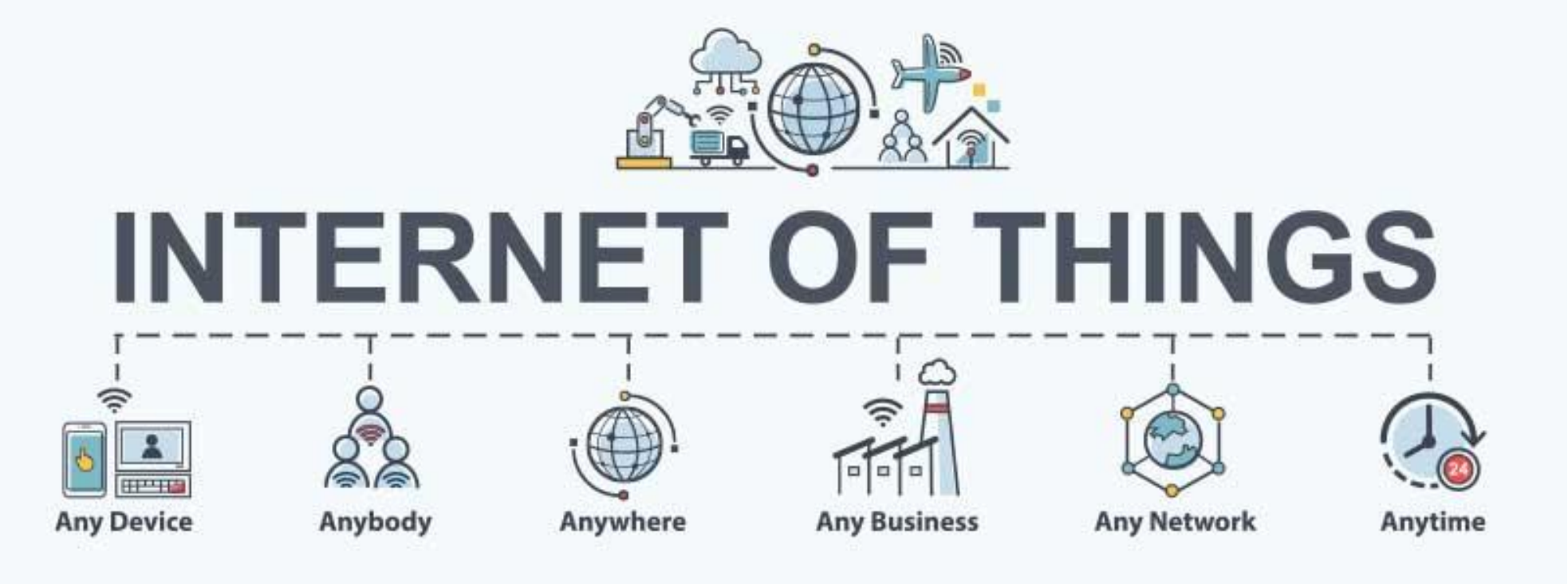
Cruising Ships

Wind

Printing

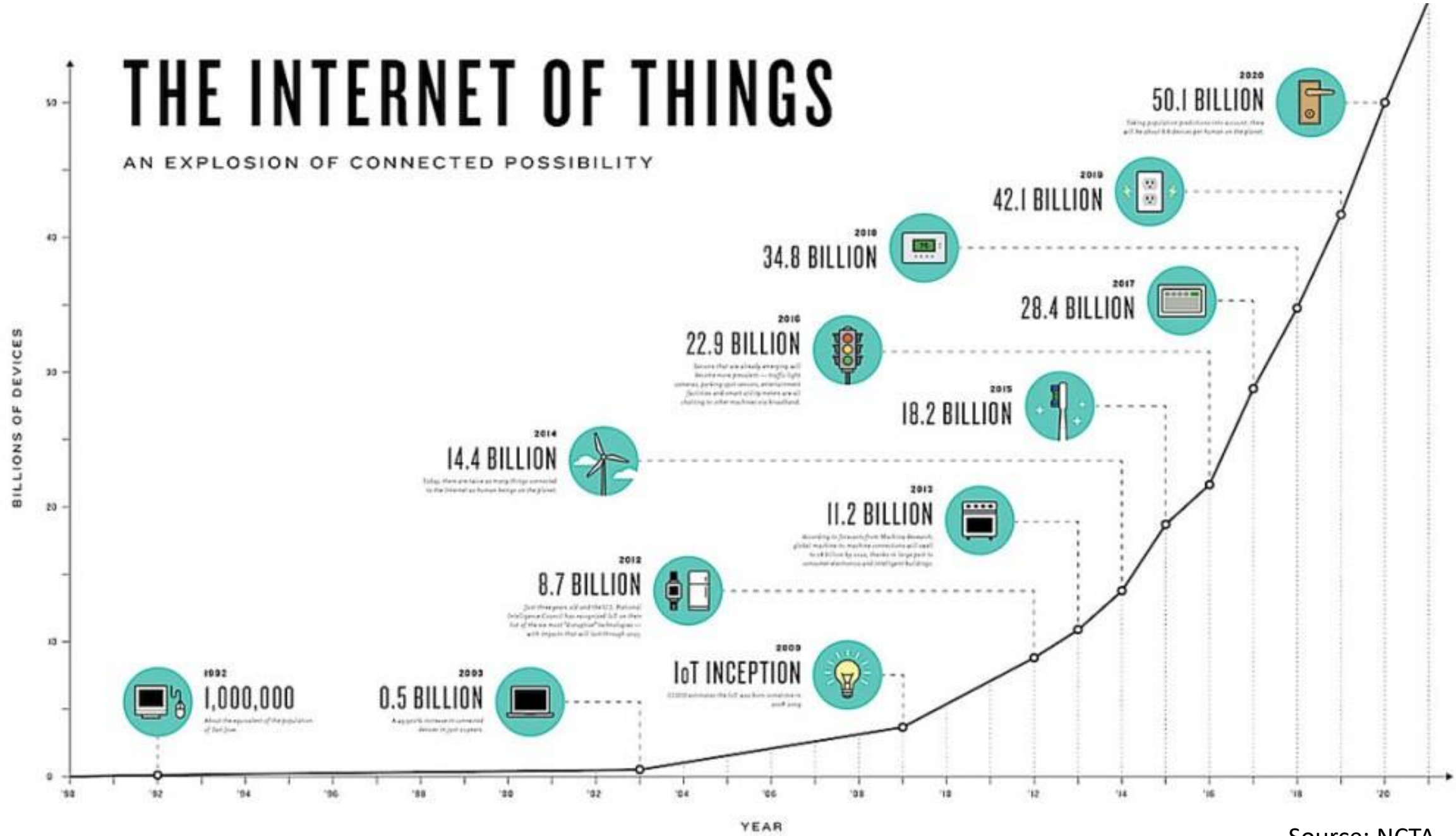
# Agenda

1. What is IoT
2. IoT system infrastructure
3. IoT devices and choice of Operating System
4. Communication protocols
5. IoT products examples




Source: https://ecsxtal.com

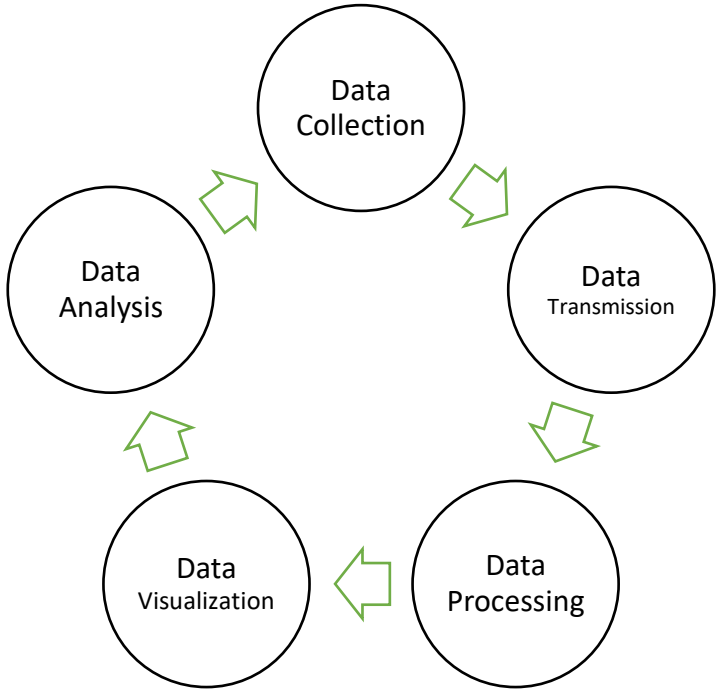
Source: NIST

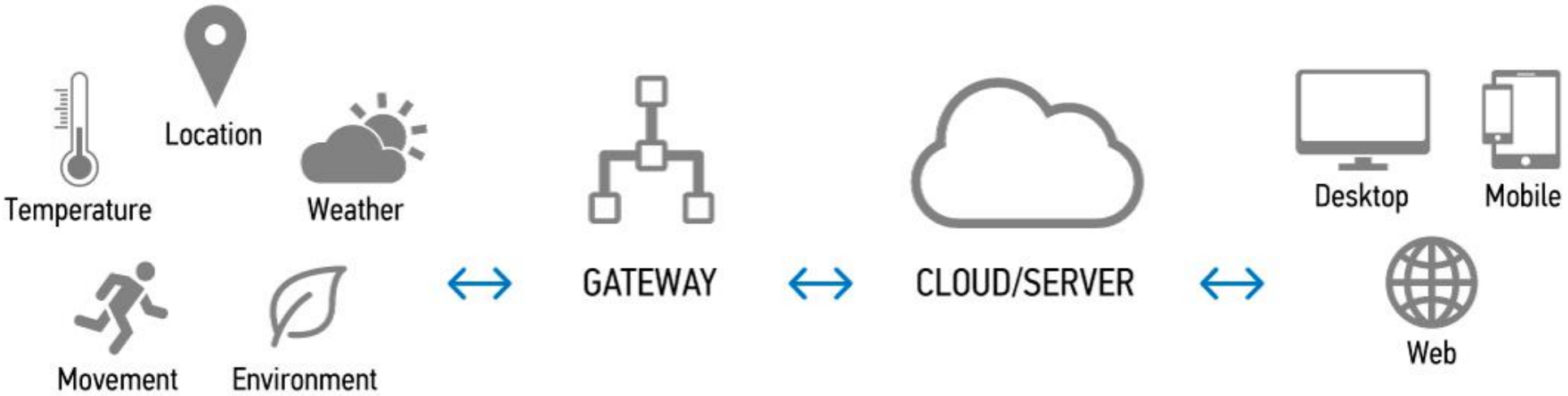

THE INTERNET OF THINGS
AN EXPLOSION OF CONNECTED POSSIBILITY
1992
1,000,000
2003
0.5 BILLION
2009
IoT INCEPTION
2012
8.7 BILLION
2013
11.2 BILLION
2014
14.4 BILLION
2015
18.2 BILLION
2016
22.9 BILLION
2018
34.8 BILLION
2019
42.1 BILLION
2020
50.1 BILLION
BILLIONS OF DEVICES
YEAR

Source: NCTA

# IoT System Infrastructure

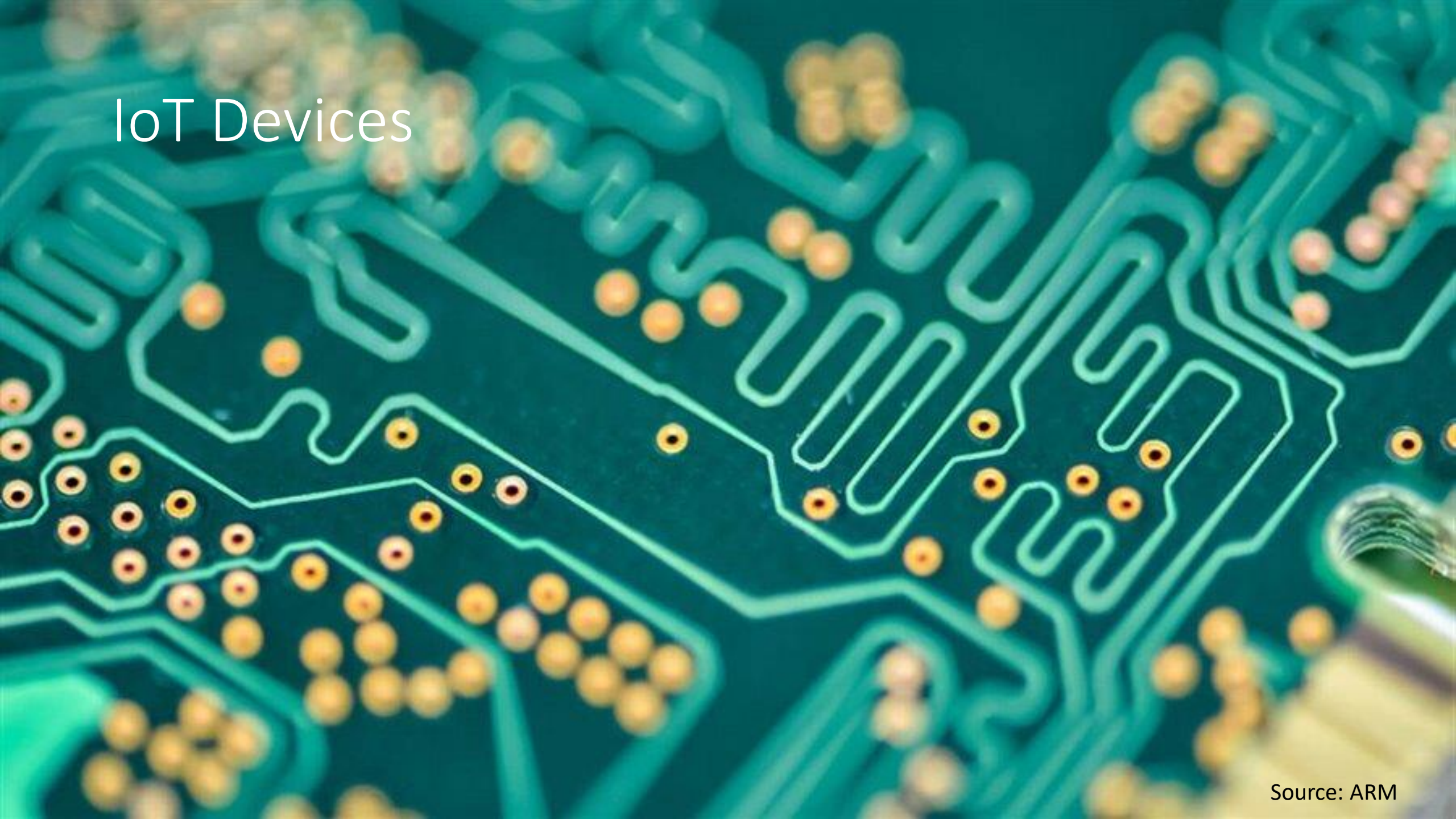
- IoT Devices
- IoT Gateways
- Cloud & Big Data
- End-User Applications




Source: sam-solutions.com

# IoT Devices

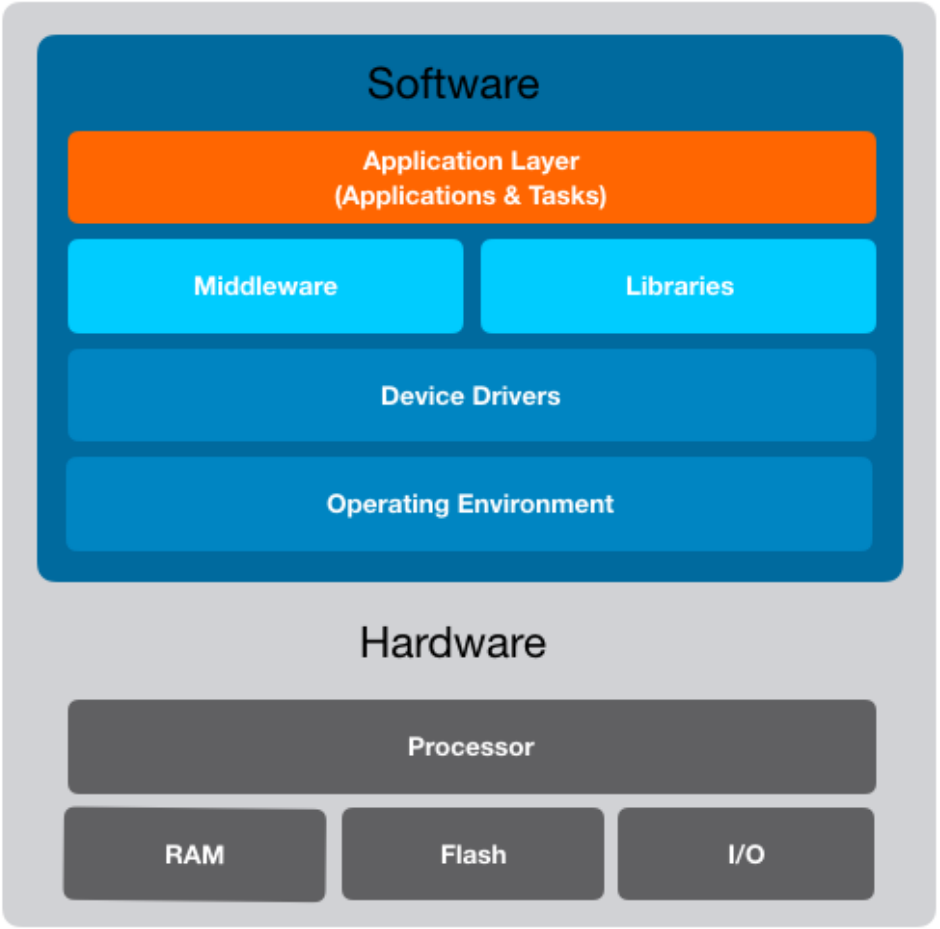


Source: ARM

# Embedded Systems

- SW & HW interconnected
- Constraints on HW
  - Power
  - Speed
  - Size
- HW cost vs SW cost
  - Volume
  - Need to get smart about software because that's where the costs are.

**Figure 1 – Generalized Layers of Software**

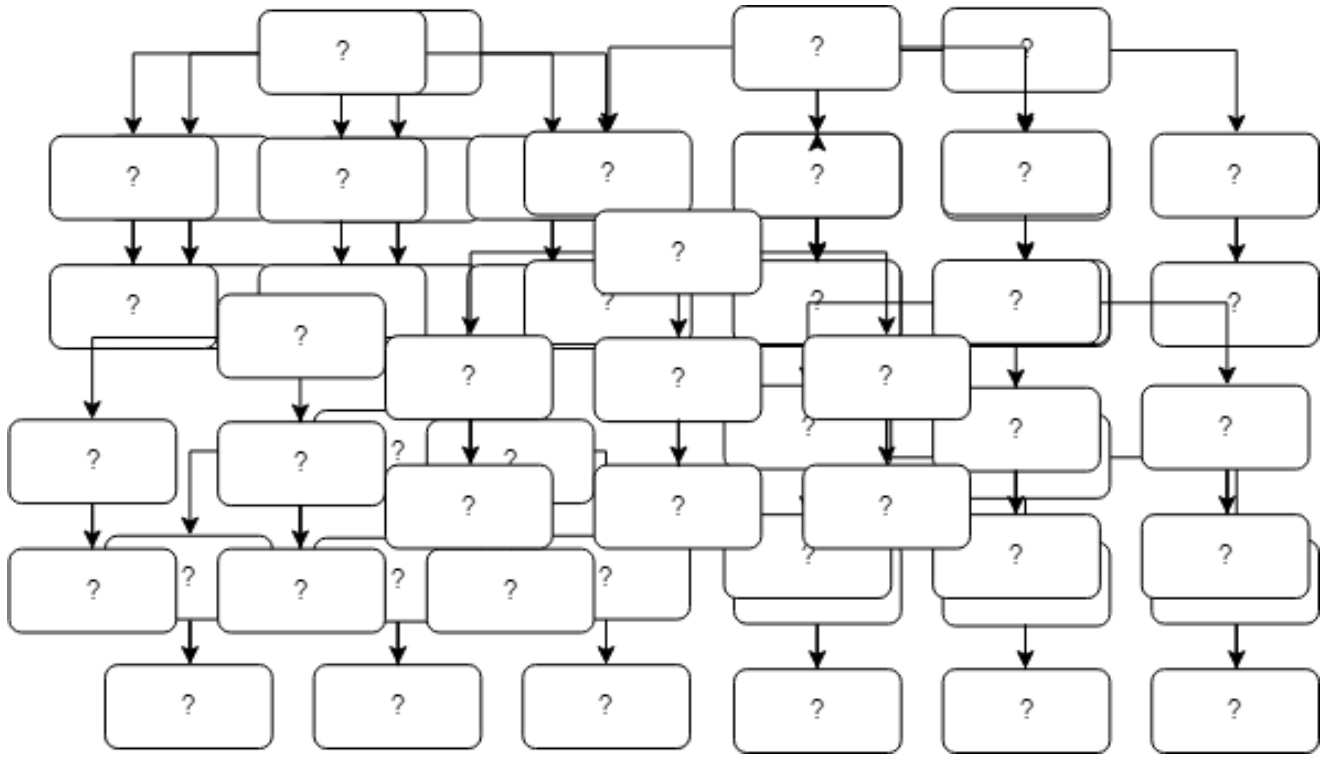



Source: Locolabs

# Architectural patterns

- Selecting the right architectural pattern will play a crucial role to the outcome of the project.
- Software architecture must be chosen wisely because once implemented it is not easy to change.

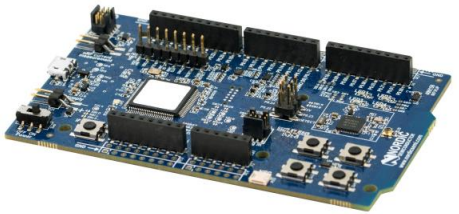
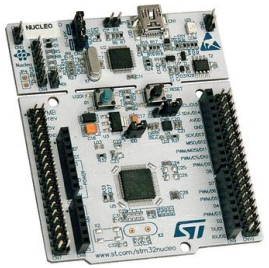
- Main forces
  - Cost
  - Time to Market
  - Quality

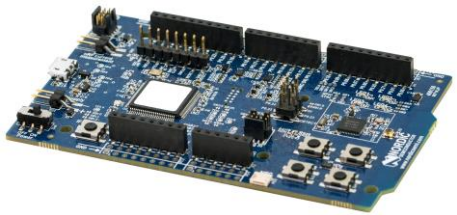
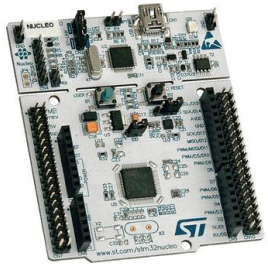


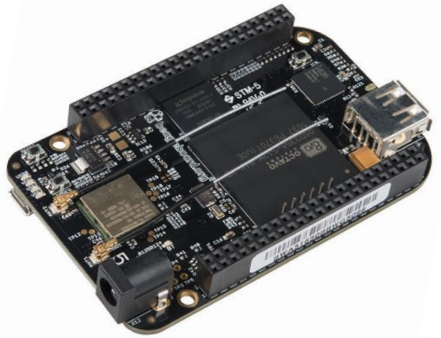
“Big ball of mud”

# Open-source OS landscape

- Bare metal, no OS
- Embedded OS / RTOS
  - FreeRTOS, Mbed OS, Zephyr, etc.
- Full-featured OS
  - Linux, etc.



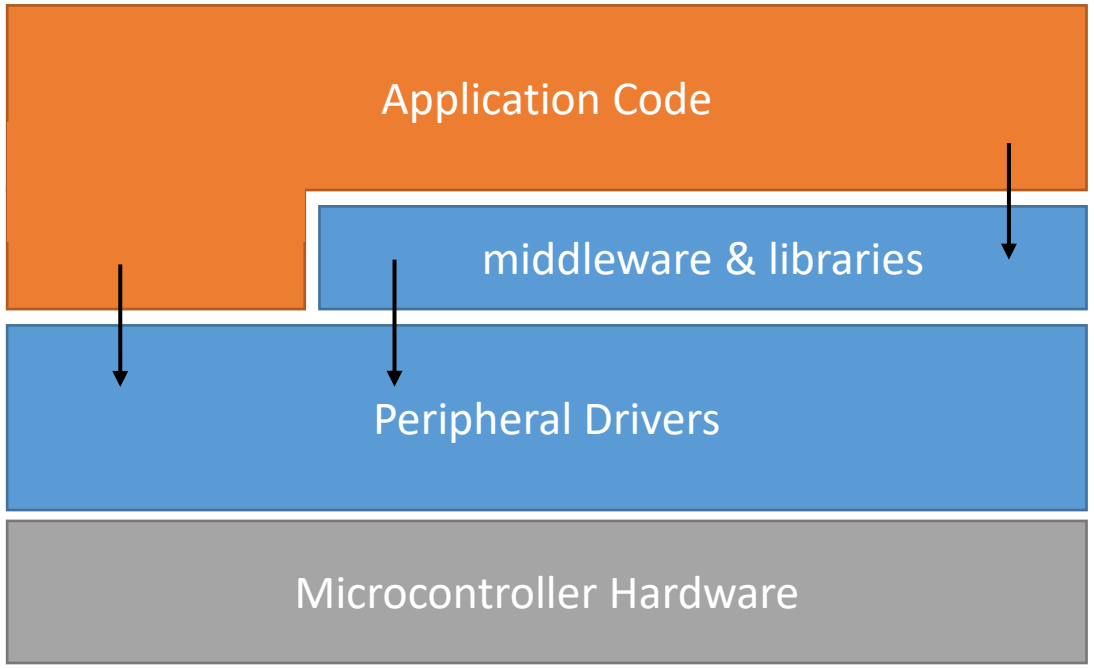

# OS used in embedded development

| Most Used | World | Americas | EMEA | APAC |
|---|---|---|---|---|
| Embedded Linux | 31% | 32% | 31% | **26%** |
| FreeRTOS | 27% | 25% | 24% | **37%** |
| Android | 14% | 12% | **10%** | **26%** |

Source: Embedded Markets Study, 2019

# Bare Metal

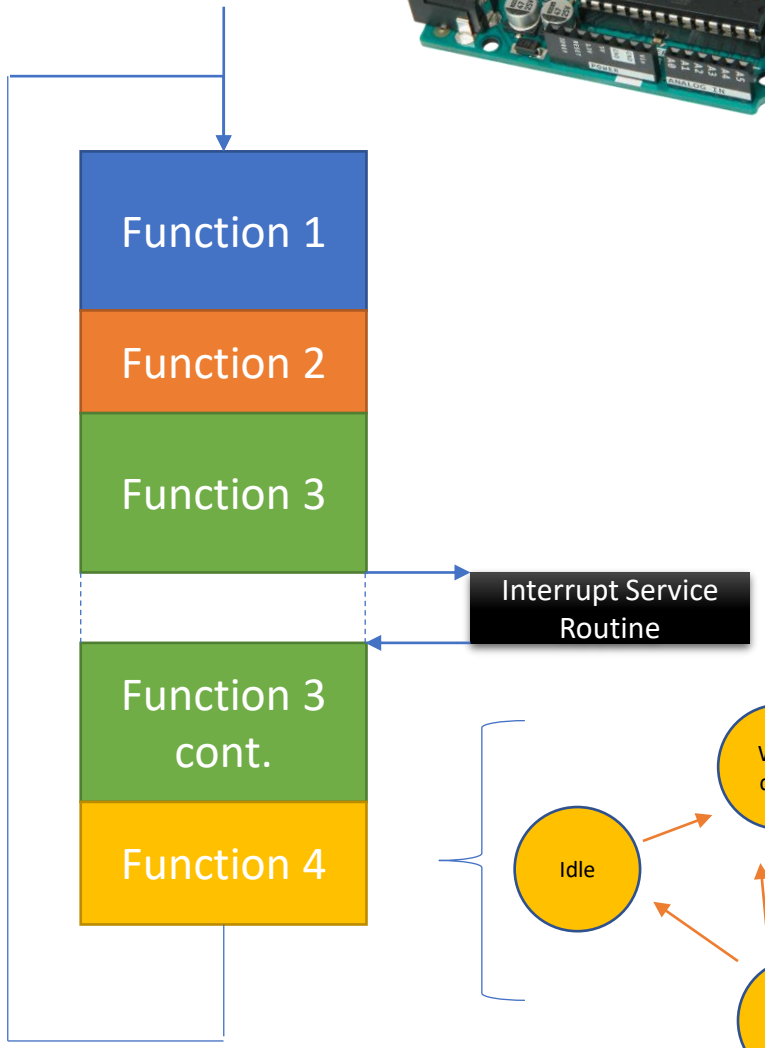

- Architecture style: Layer
  - Each layer is a group of modules that offer cohesive set of services.
  - Clear separation of concerns so that modules can be independently developed and maintained (portability, modifiability, reuse)
  - Layer n can use services of layer n-1 and not vice versa.
  - Bridging layers



# Bare Metal

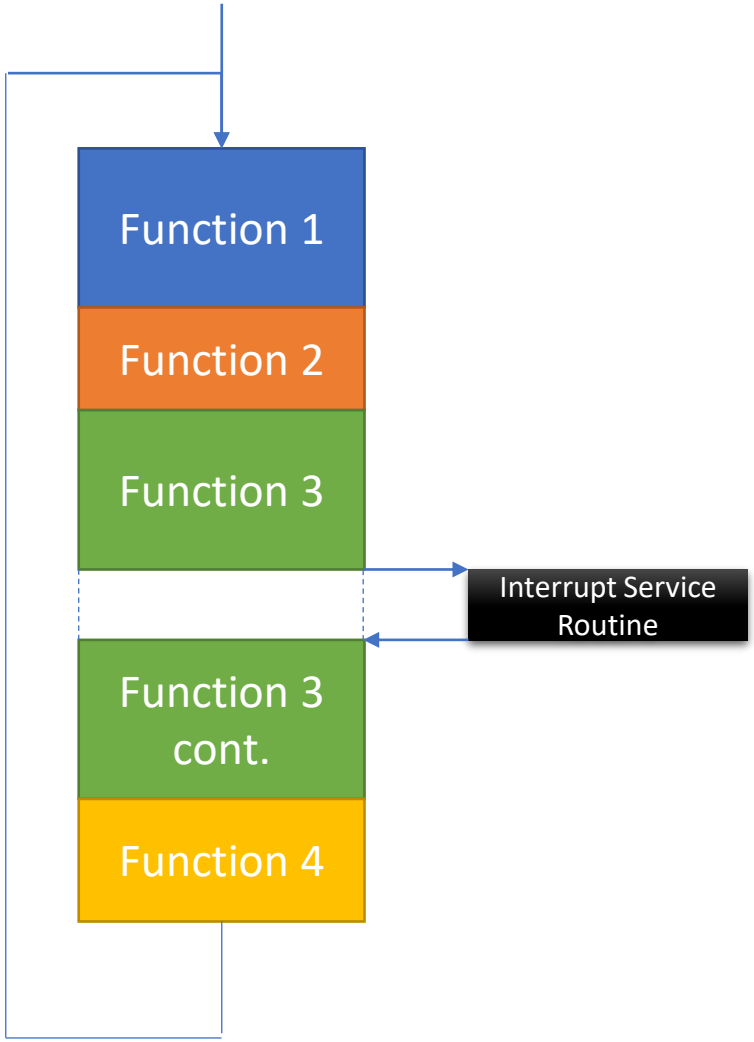
- Software Architecture
  - Super loop
  - Event driven
  - State machine
- Typical task order
  - Read inputs
  - Process information (using state machine)
  - Output results
  - Delay until next loop execution
- Task examples
  - ADC samples
  - Blink LED
  - Read/write data to flash
  - Serial communication

```
void main(void) {
    board_init();
    while (1) {
        adc();
        led();
        flash();
        serial();
    }
}
```



# Bare Metal

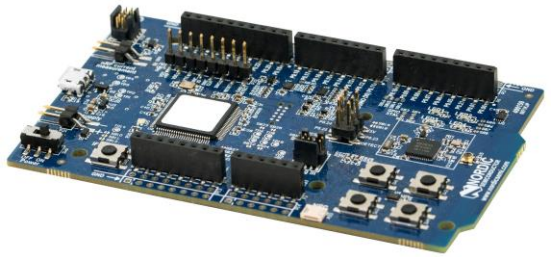

+ Low complexity
+ Everything is processed in the super loop
+ Easy to see the which functions being executed and when
+ Timing is fixed at every loop, except for short interrupts (incoming events)
+ Functions do not execute at the same time, thus no race condition
- No function can execute too long to ensure response time
- Difficult to add complex functionality
    - e.g. pdf generation, TLS communication, UI, complex time-consuming algorithms
    - Functions with long execution time could lead to heavy jitter and latency of other user inputs
    - Scheduling adjustment to fit new modules/libraries
    - Requires implementation with constraint resource



# Bare Metal

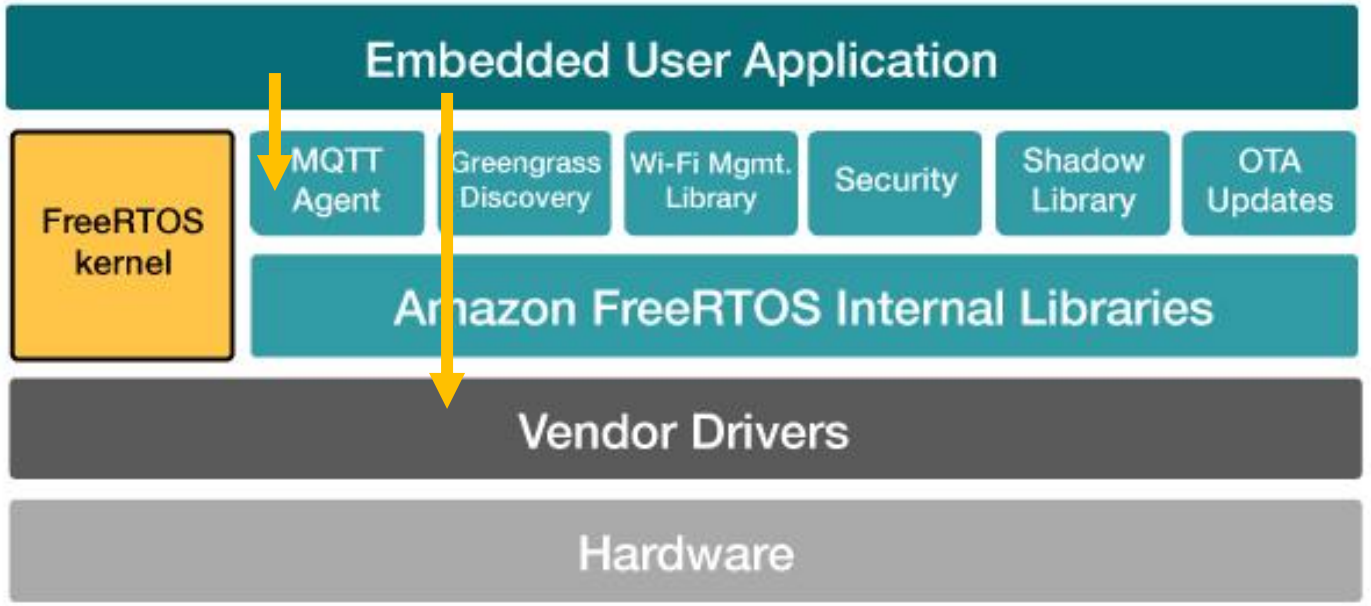
- When using bare metal?
  - Simple/ low complexity application
  - Real-time deadline is not a requirement
  - An application that does not require task/thread preemption
  - Lack of hardware resources to fulfill the need of an OS.
  - Don’t want to use third-party libraries and drivers

# Embedded OS



Layered pattern

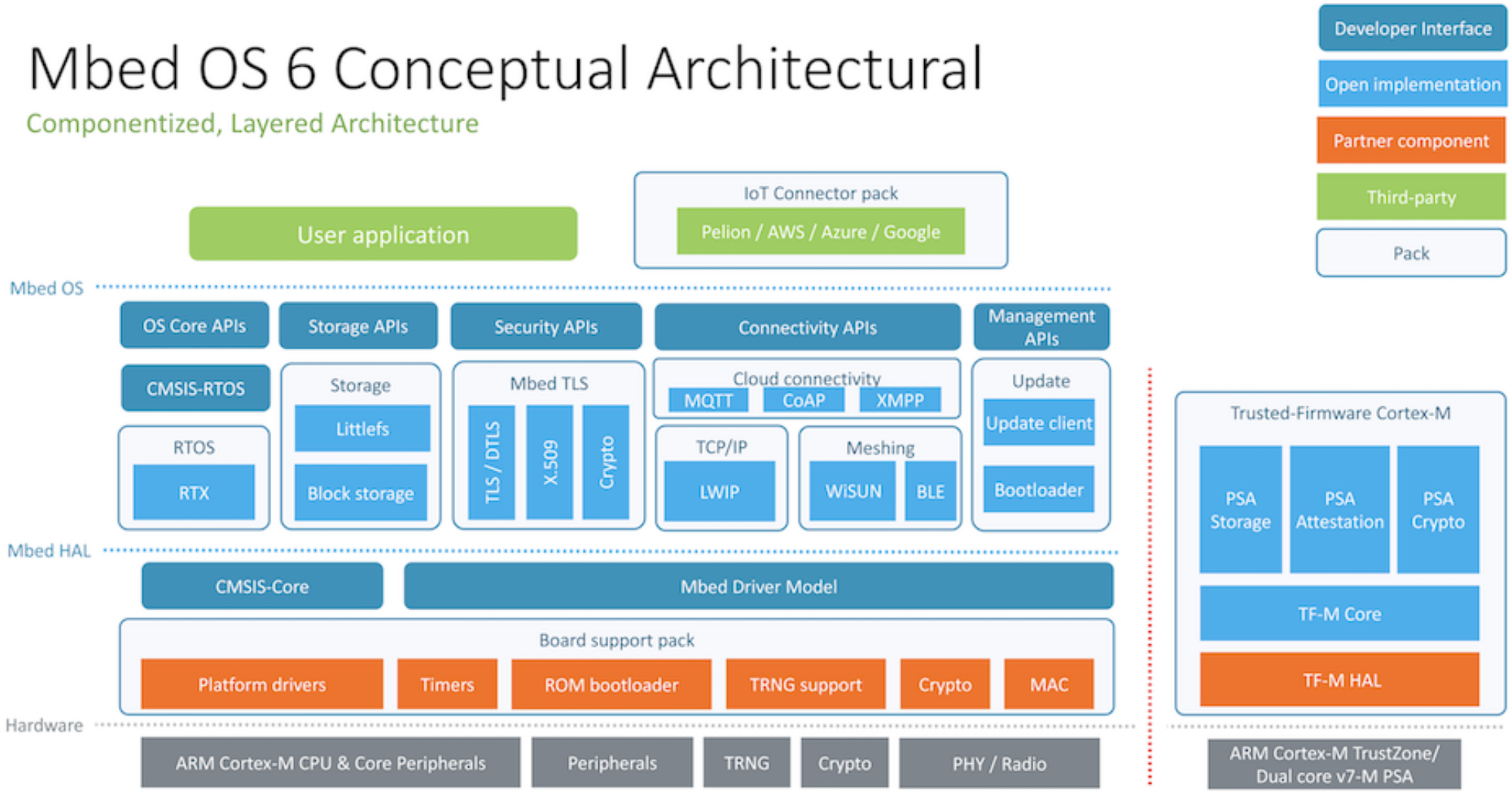
- Speed
  - Built-in modules/libraries for connectivity, security and updates
  - Integrated cloud services
  - Broad support
- Complexity
  - Support many hardware and tools
- Cost
  - Free and open source with community support




Source: FreeRTOS

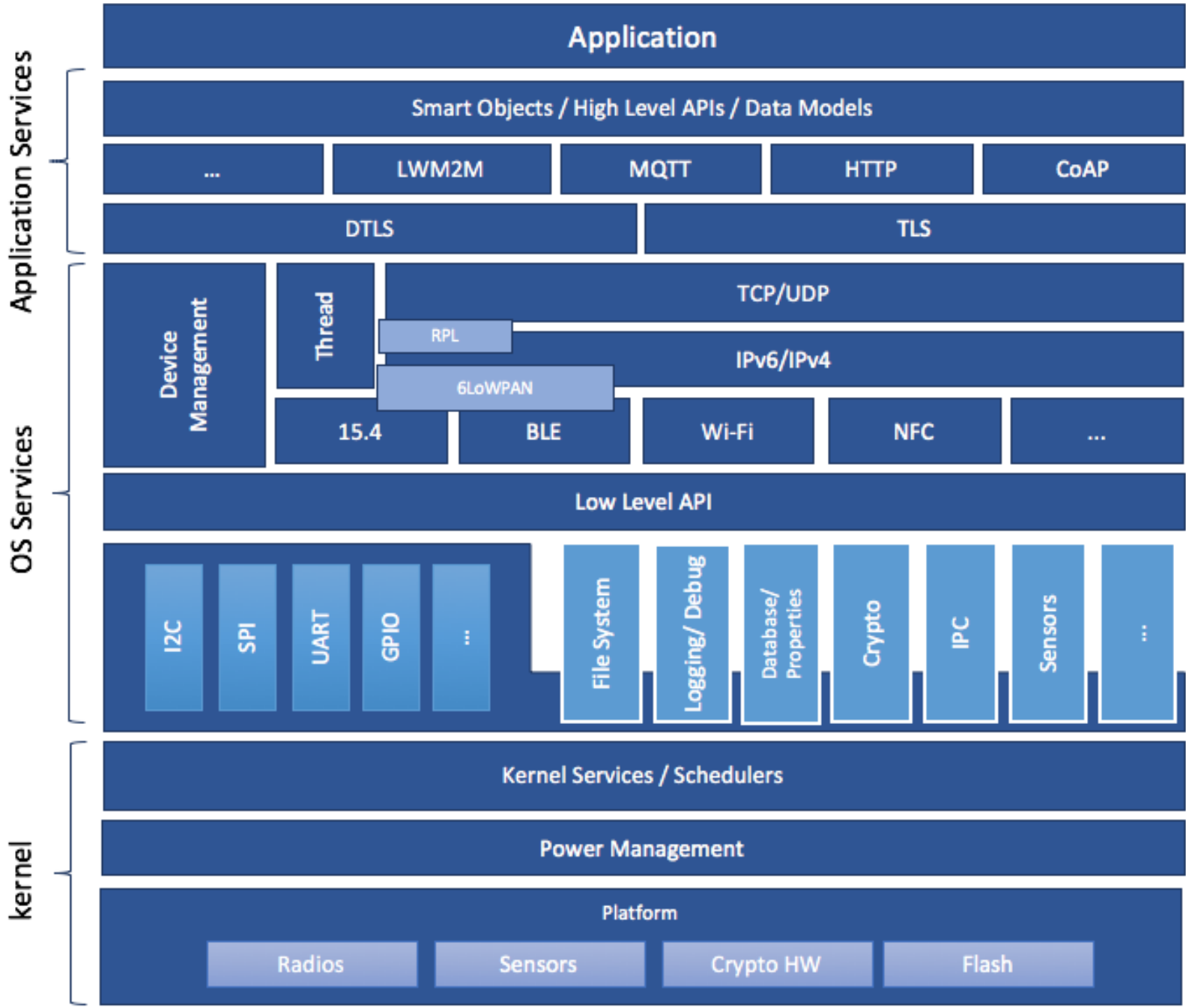
# Mbed OS 6 Conceptual Architectural

## Componentized, Layered Architecture




Source: Mbed

# Zephyr OS




Source: https://zephyrproject.org

# Embedded OS

- Concurrent tasks/threads handling with priority and scheduling
- Inter-process communication and synchronization
- Deterministic behavior and timely response to critical tasks.
- Modest range of middleware and IoT libraries such as USB, TLS, IPv4, IPv6, WiFi, CAN, MQTT, etc).
- Tiny memory footprint, 100-200 KB
- Fast boot time, 50ms – 100ms
- Long term maintenance and development costs decreased

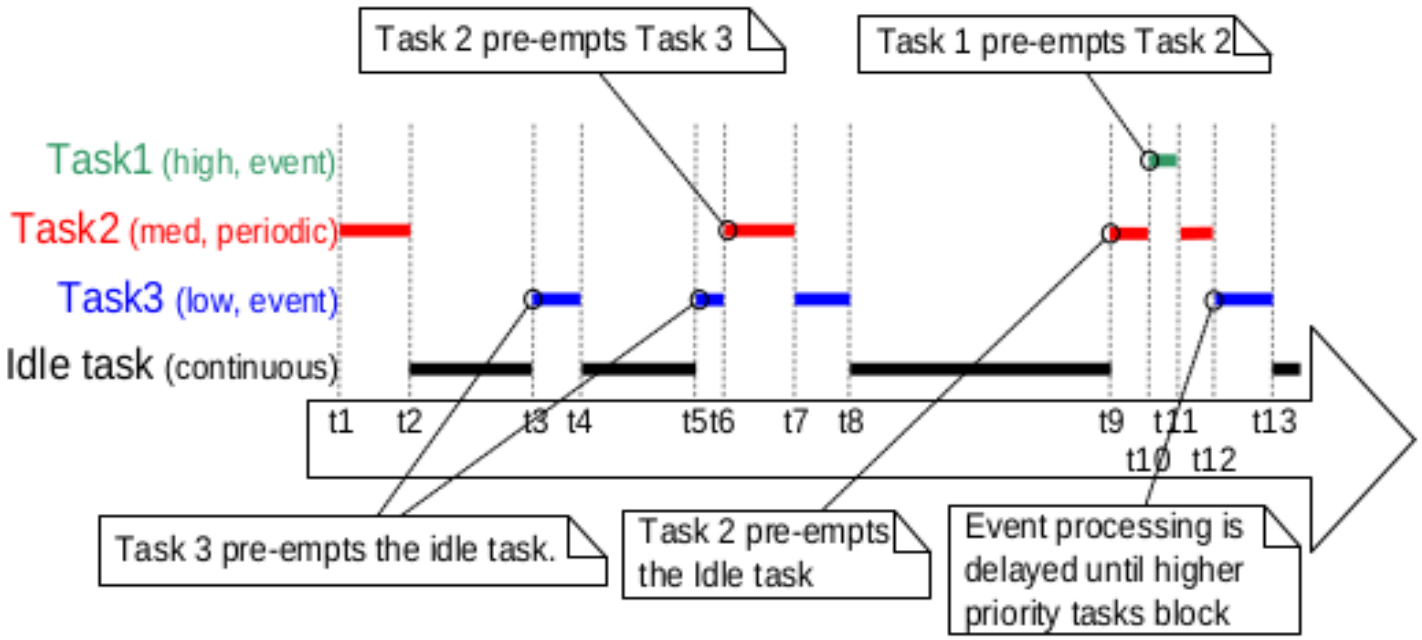
# Embedded OS

- Each task has its own private stack and priority.
- Each task is a loop itself.
- When a task to be executed depends on each scheduling mechanisms (round-robin, preemptive-based scheduling)
- No need to break-down long execution function to meet time limitation

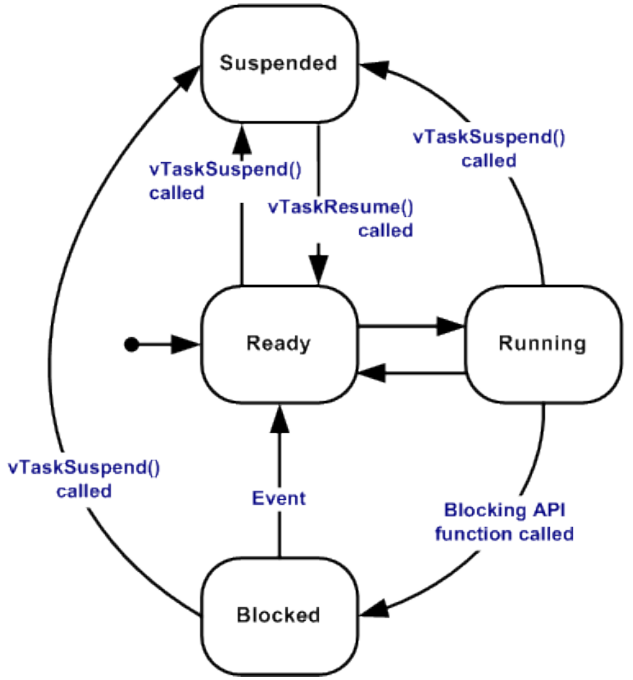
```
// bare metal
void main(void) {
    board_init();
    while (1) {
        func1();
        func2();
        func3();
   }
}
// func1() and
func2() cannot
execute too long
```

```
// Embedded OS
Task1(){
    while(1){
        func1(); //can execute as
long as it needs.
    }
}

Task2(){
    while(1){
        func2();//can execute as
long as it needs.
    }
}

void main(void) {
    createTask(&Task1,highPrio);
    createTask(&Task2,medPrio);
    createTask(&Task3,lowPrio);
    startScheduler();//never returns
}
```

# Embedded OS

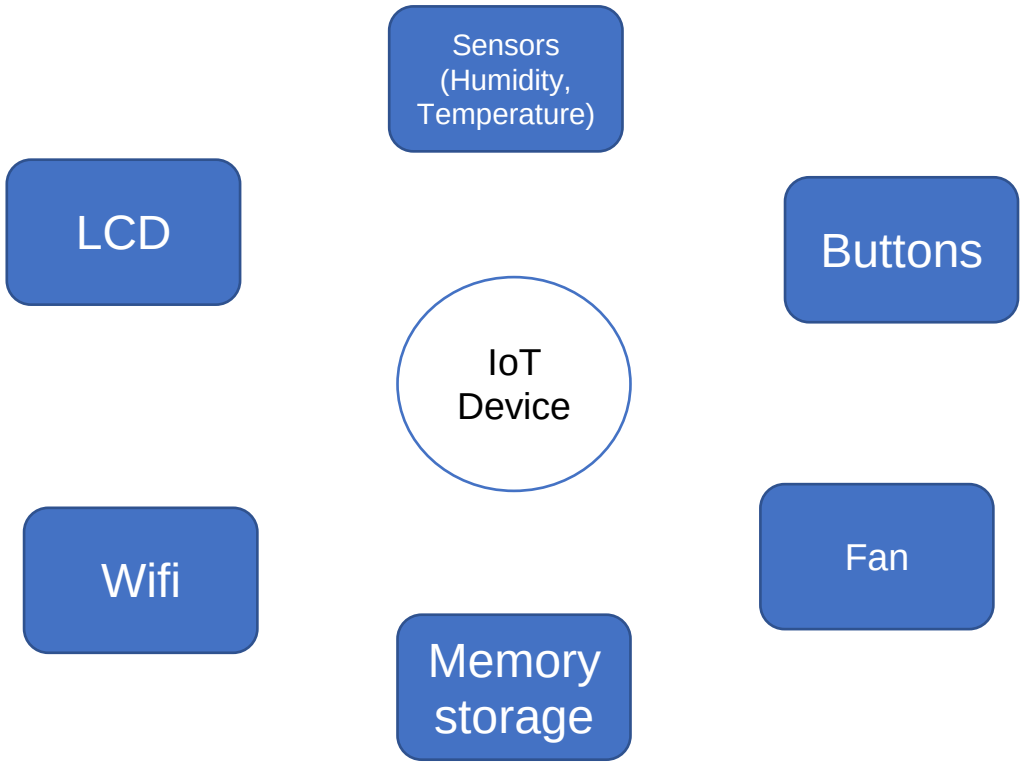



```
Task1(){
    while(1){
        func1();
    }
}

Task2(){
    while(1){
        func2();
    }
}

void main(void) {
    createTask(&Task1,highPrio);
    createTask(&Task2,medPrio);
    createTask(&Task3,lowPrio);
    startScheduler();//never returns
}
```



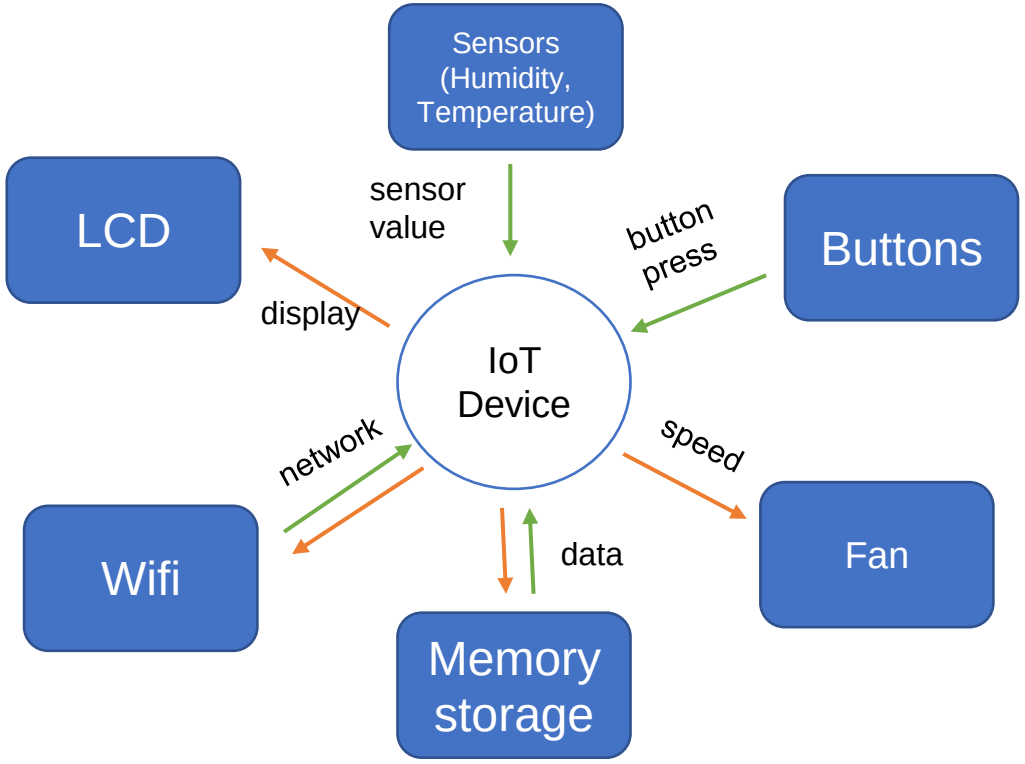
# Task/module decomposition

1. Identify major components
   - LCD
   - Buttons
   - Wifi
   - Memory storage
   - Sensors (Humidity, Temperature, Air quality)
   - Fan



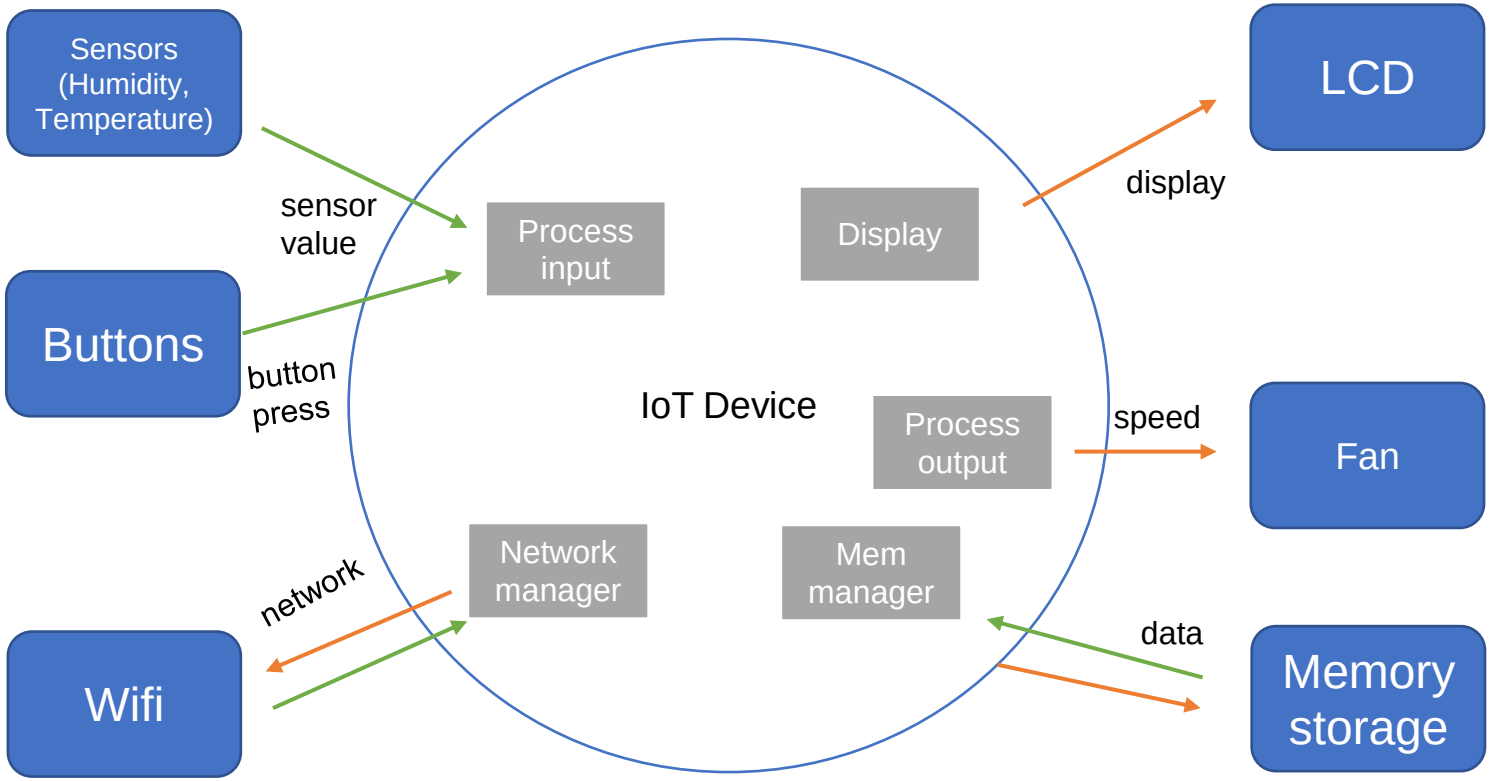

# Task/module decomposition

2. Label inputs/outputs



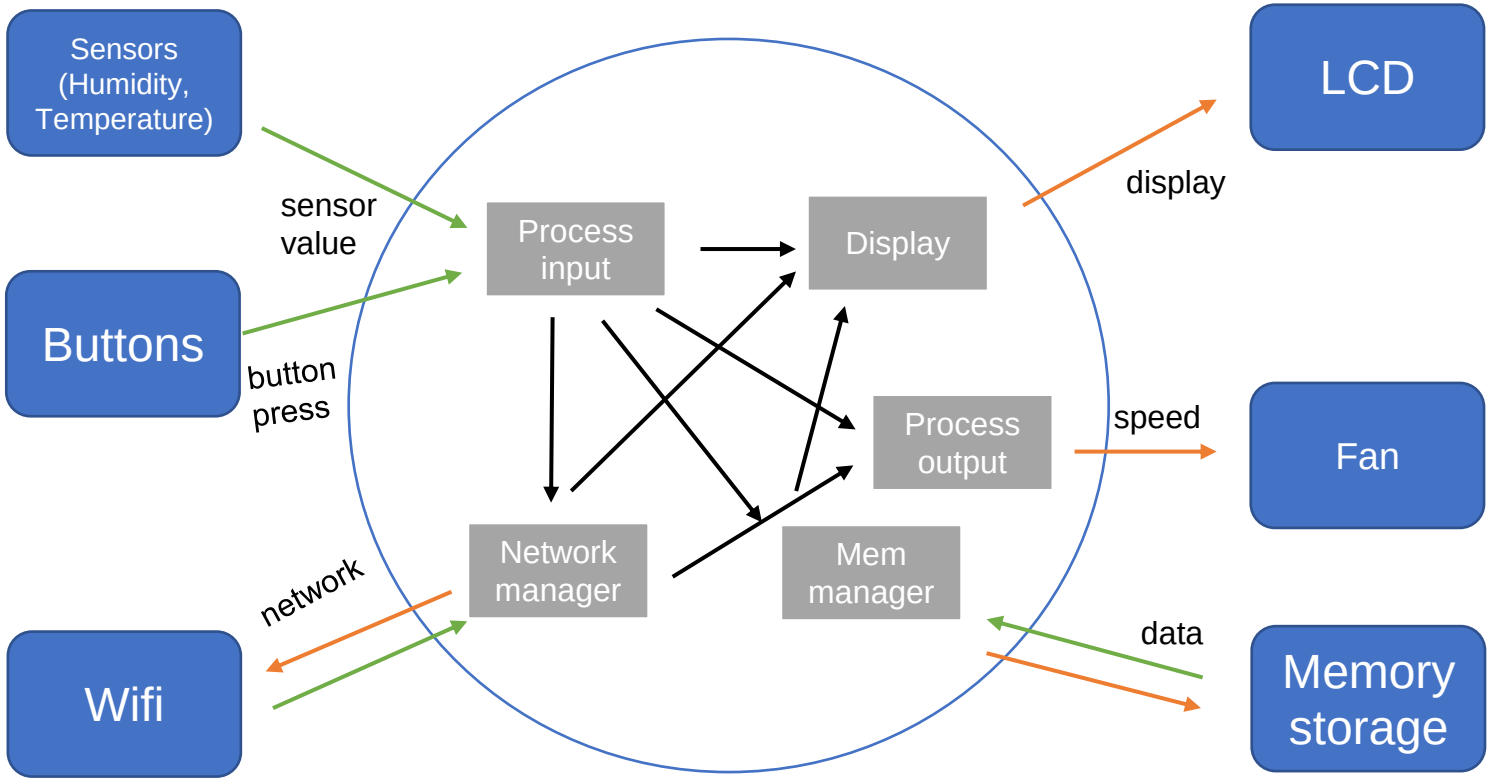
# Task/module decomposition

3. Identify 1st level tasks



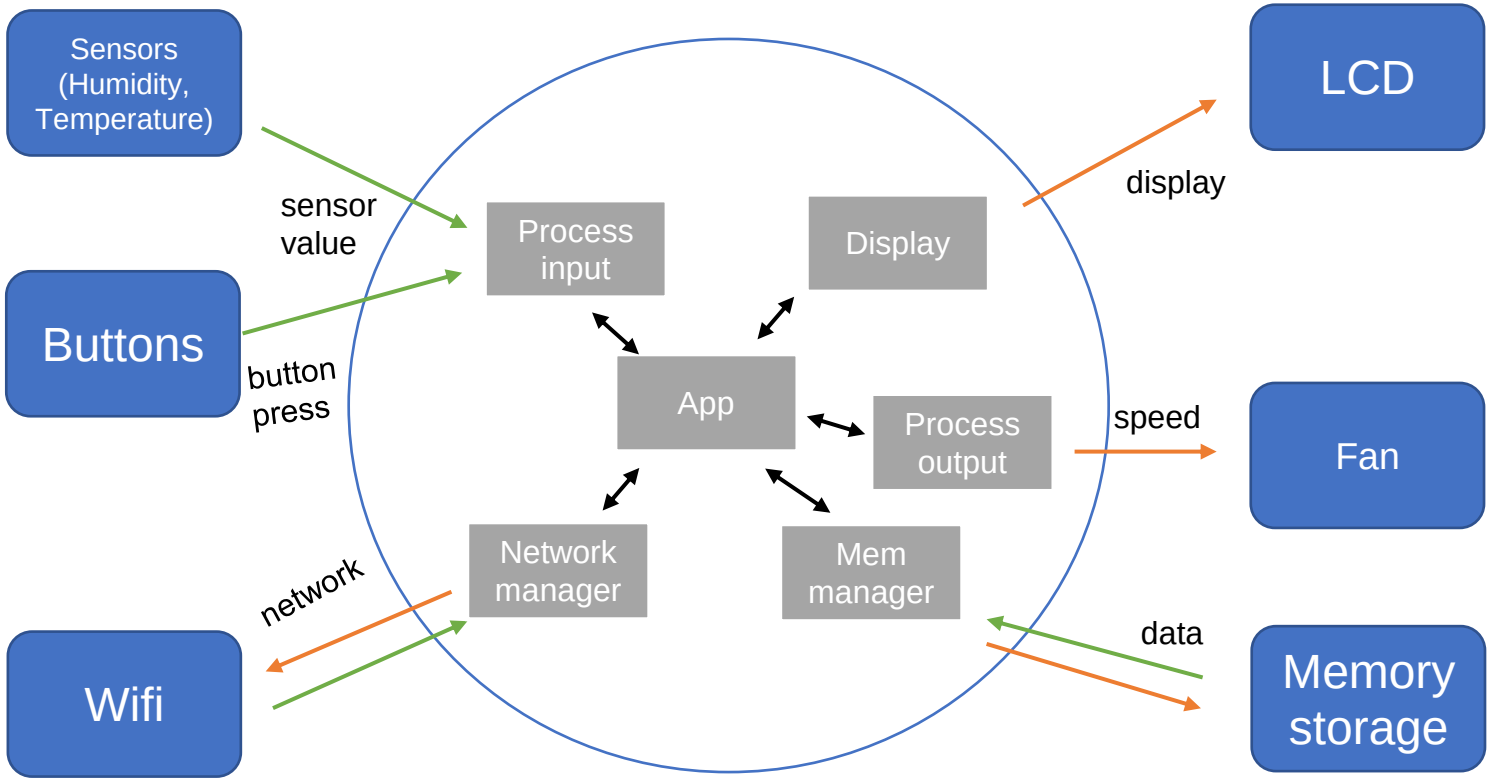
# Task/module decomposition

4. Determine dataflow and dependencies



# Task/module decomposition

5. Identify application tasks



# Embedded OS

+ Concurrent programming paradigm. Splitting the software on tasks makes it modular, improve program structure, maintainability and performance.

+ Performance: Guarantee responsiveness to external events in a timely manner

+ Scalability,  easier to add complex functionality

+ Portability: Move over to another HW vendor easily due to similar API

+ Portability and reusability, support various platforms. Migration cost decreased dramatically if moving to a different microprocessor.

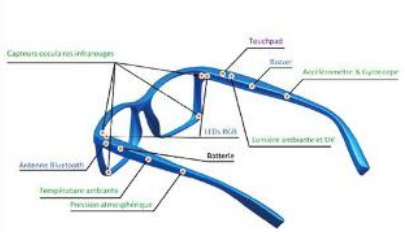
+ Security: Some level of security through libraries

+ Modifiability: Easy to add new features, easy to modify without much affecting current system

- Requires precious resources on low-end embedded systems

- Overhead for handling OS features such as scheduling, context switching

# Products Running Zephyr Today



Ellcie-Healthy Smart Connected Eyewear

Adero tracking devices

Rigado IoTGateway

ProGlove Scanning Gloves

GNARBOX 2.0 SSD

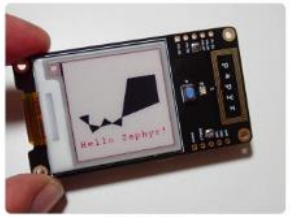
Antmicro Badge

Grush Gaming Toothbrush

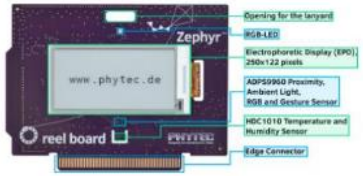
hereO Smartwatch

Blocks Modular Smartwatch

Intellinium Safety Shoes



Papyr



Reel Board

Source: The Linux Foundation

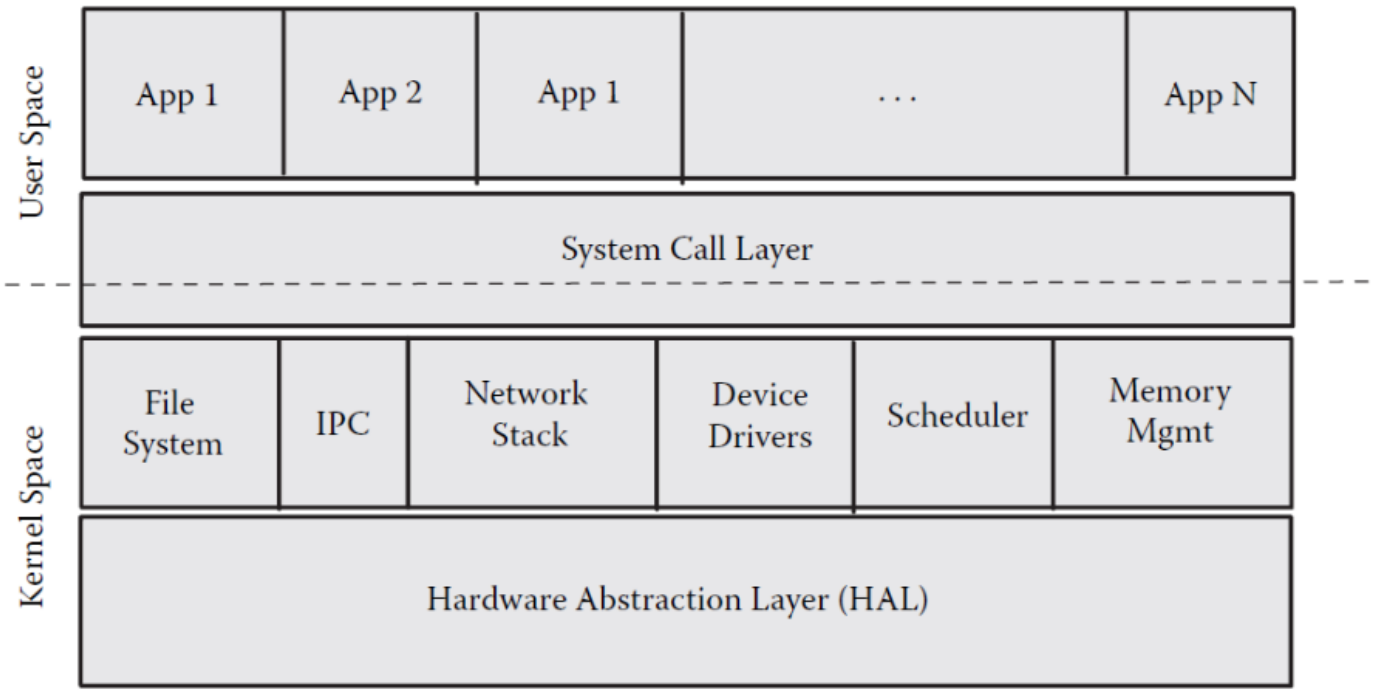
# Full-featured OS

- Open source
- Rich feature set
- Free or Low cost
- Hardware support

- HW Requirements
  - 32-bit and 64-bit processor architectures
  - 8 MB RAM
  - 4MB flash

# Full-featured OS

- Bootloader
- Kernel
- Filesystem
- Services
- Application / Programs

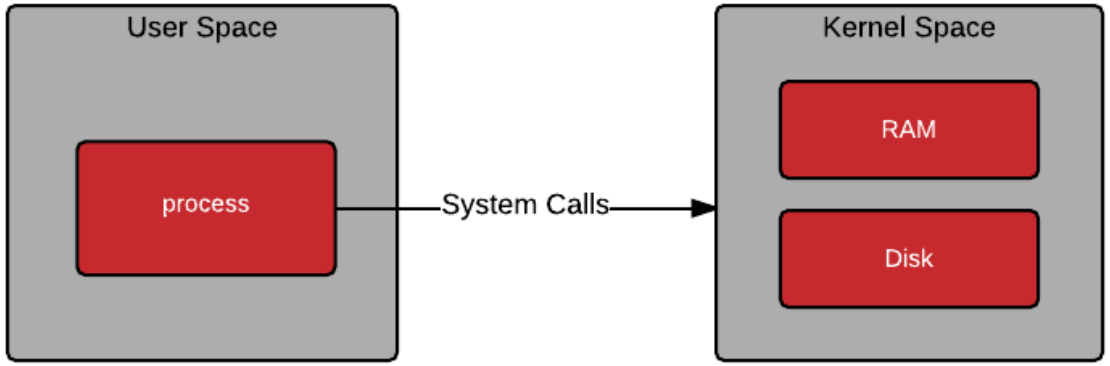



Source: http://ccrs.hanyang.ac.kr/webpage_limdj/embedded/LinuxArchitecture.pdf

# User space vs kernel space

**Memory divided into two distinct areas:**

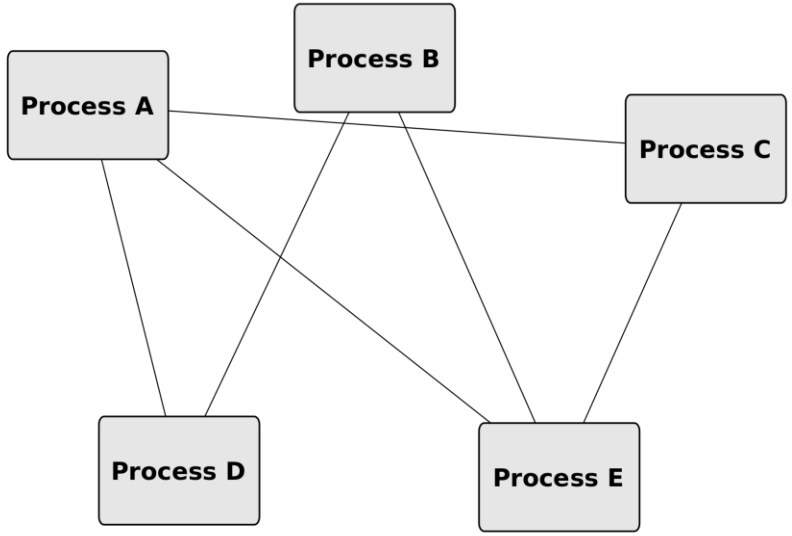
- **User space**
  - normal application software runs
- System calls
  - Examples: allocating memory (variables) or opening a file. Memory and files often store sensitive information owned by different users, so access must be requested from the kernel through system calls.
  - open(), read(), write(), close(), ioctl(), fork(), exit(), wait(), etc.
- **Kernel space**
  - where the code of the kernel is stored and executes under.
  - Strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers.



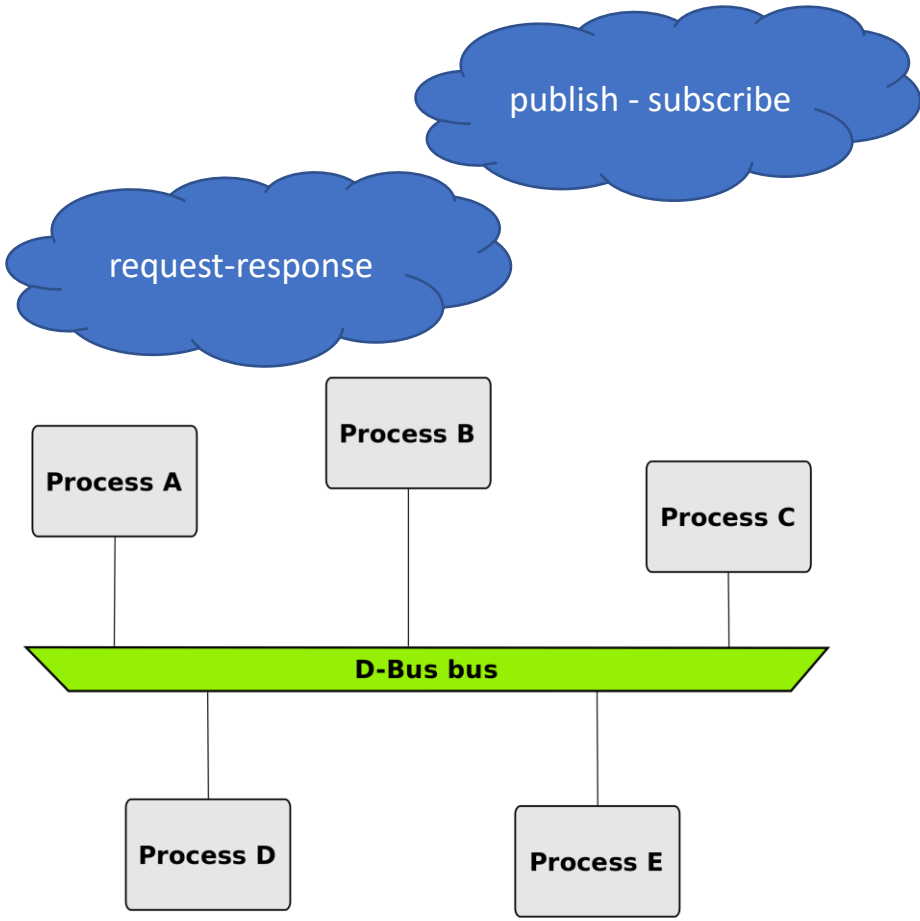

Source: RedHat

# Embedded Linux

- D-Bus: IPC mechanism
- a medium for local communication between processes running on the same host








Source: Wikipedia

# Full-featured OS

+ Availability of file-systems, network connectivity, and UI support.

+ Native security features, constantly scanned for vulnerability

+ Versatile, scalable and has support for practically every device driver and protocol.

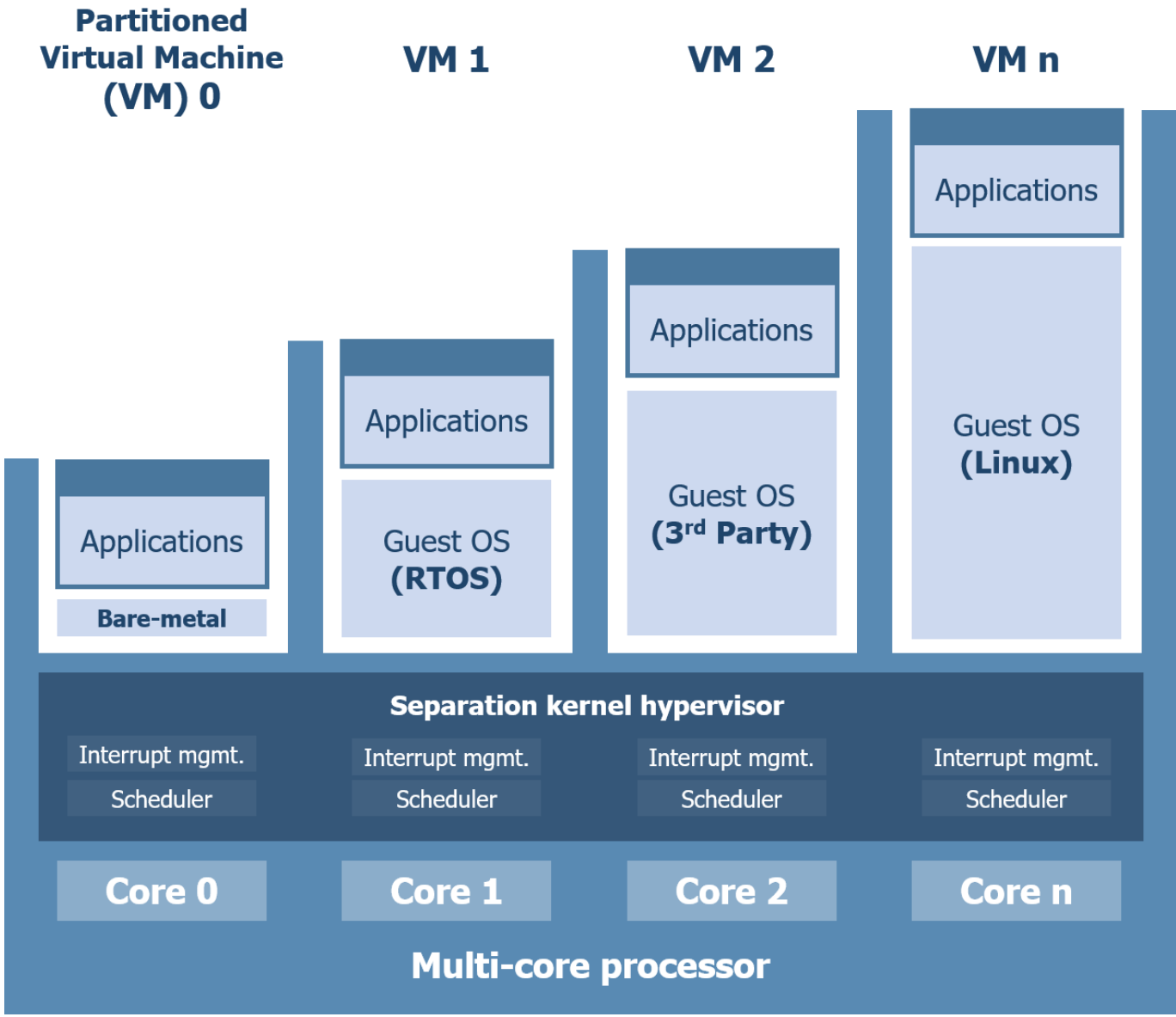
+ Modifiability, easy to add functionalities in the future, plenty of open-source applications available.

- Needs significant CPU resources, typically in range of few MB for both RAM and ROM footprint
- Soft real-time
- Booting time in several seconds
- High HW cost. Not suitable for 8 or 16-bit MCUs and low power devices.
- Steep learning curve, complex build systems, long compile time, etc.
- Open-source licensing obligation

# Choosing the right one

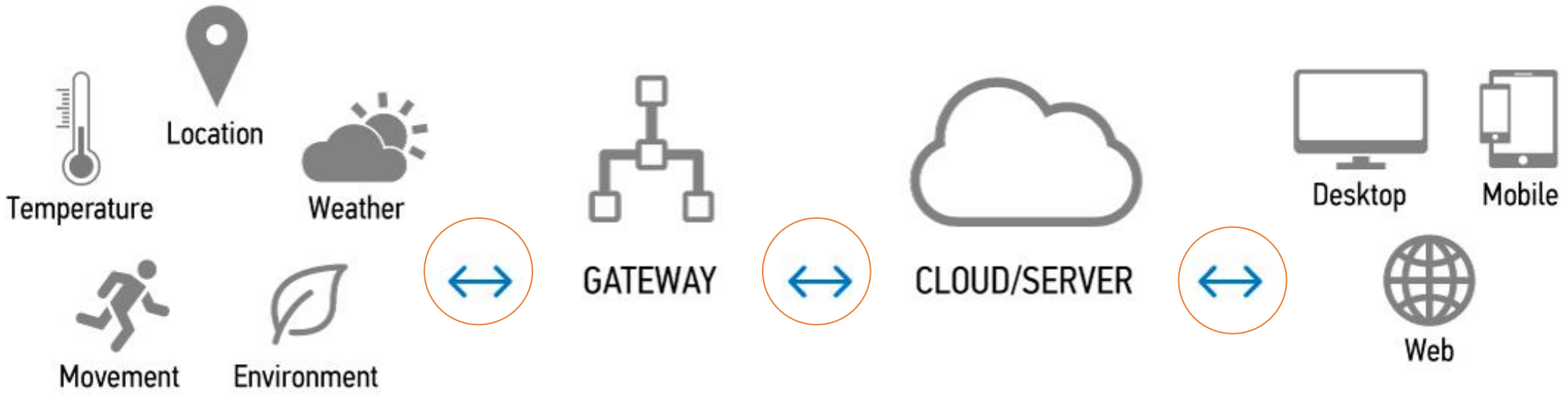
- What is the target HW? What is the available resources on MCU?
- What is the required functionality and performance?
- Does it need real-time requirements?
- How much code do you need to develop? How much standard library it has?
- How complex are the algorithms involved?
- How hard will it be to maintain that code later on?
- Which one has been used in other products? Which one the team is familiar with?
- Languages - Does it support Java, Python, C, C++
- Do you need safety-certified embedded OS?
- Maturity of OS, good technical support available

# Future




Source: https://www.pertech.co.il/lynxsecure

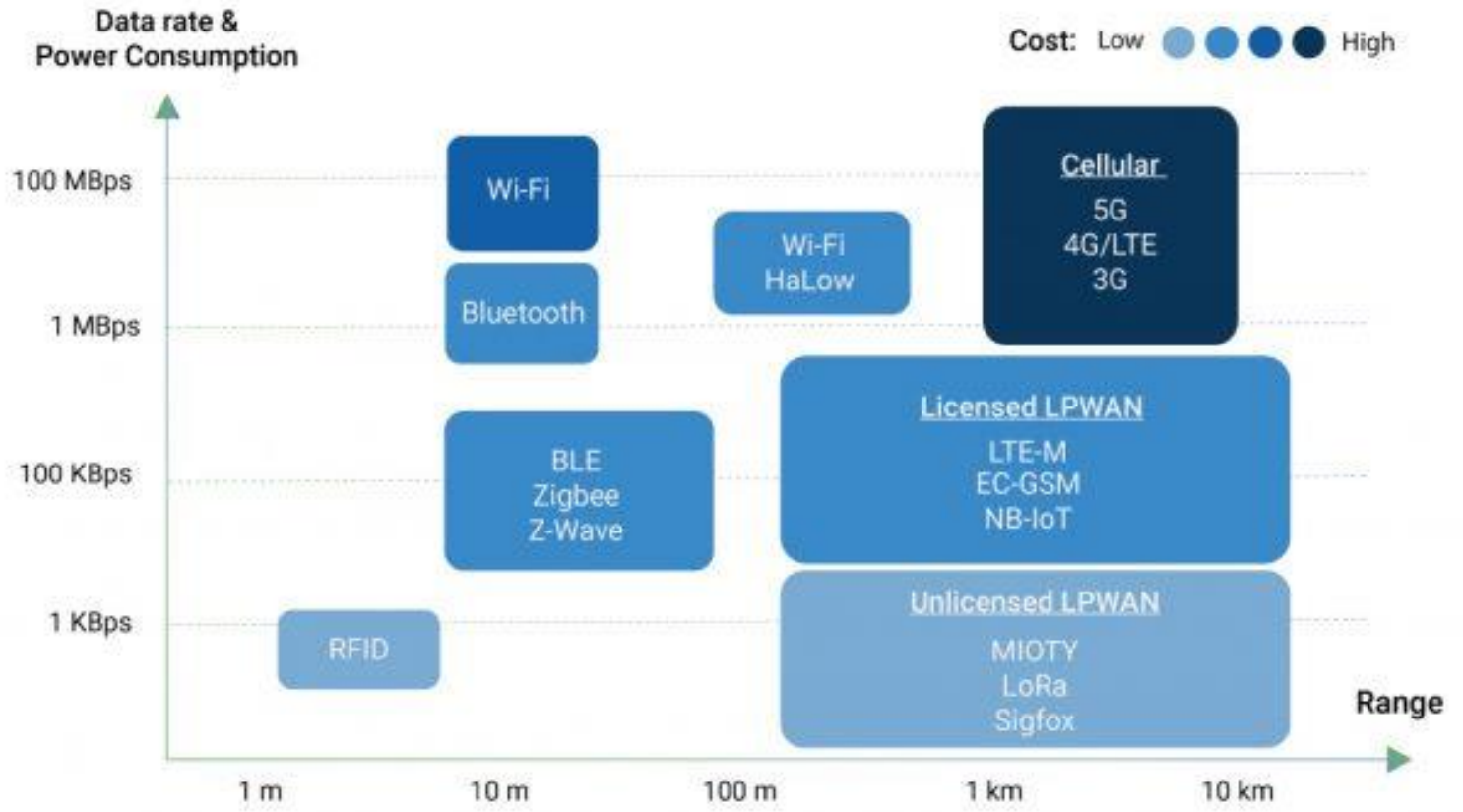
# Communication protocols



# Communication protocols

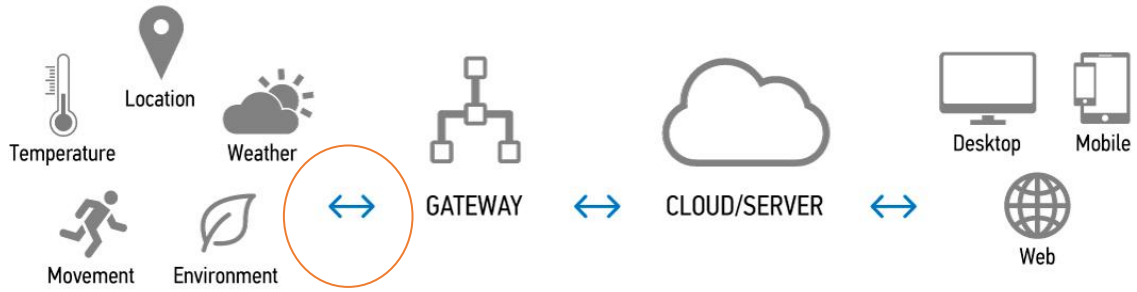
Main design constraints

- Power consumption
- Range
- Cost



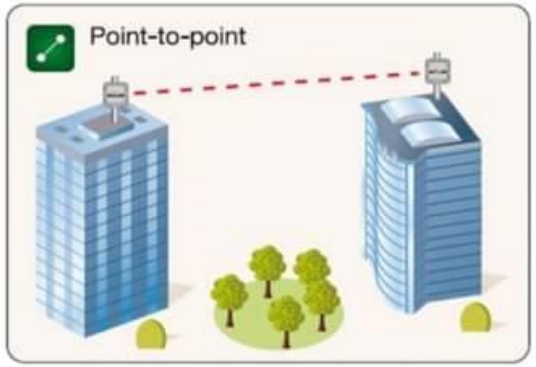

Source https://industrytoday.com/best-uses-of-wireless-iot-communication-technology

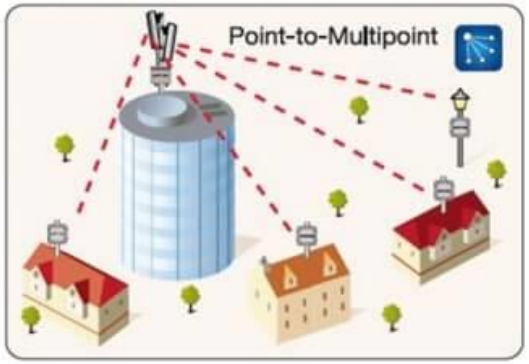
# Communication protocols

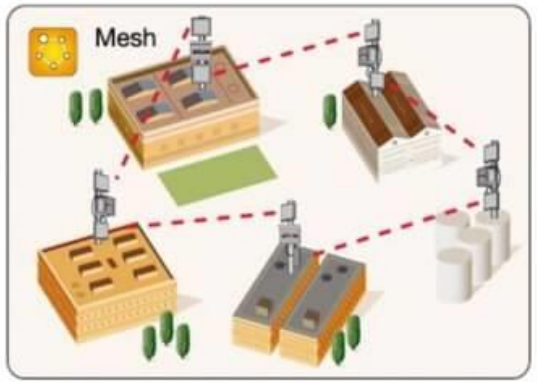


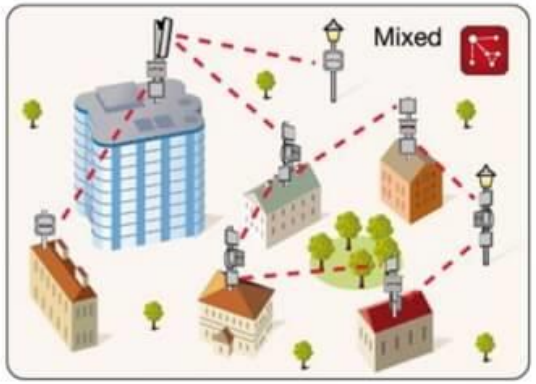

- Bluetooth Low Energy
- LoRaWAN
- Wifi
- Many mesh network
  - Zigbee
  - Z-Wave
  - Bluetooth mesh
  - Proprietary mesh

Source: https://www.fluidmesh.com/wireless-mesh-networks



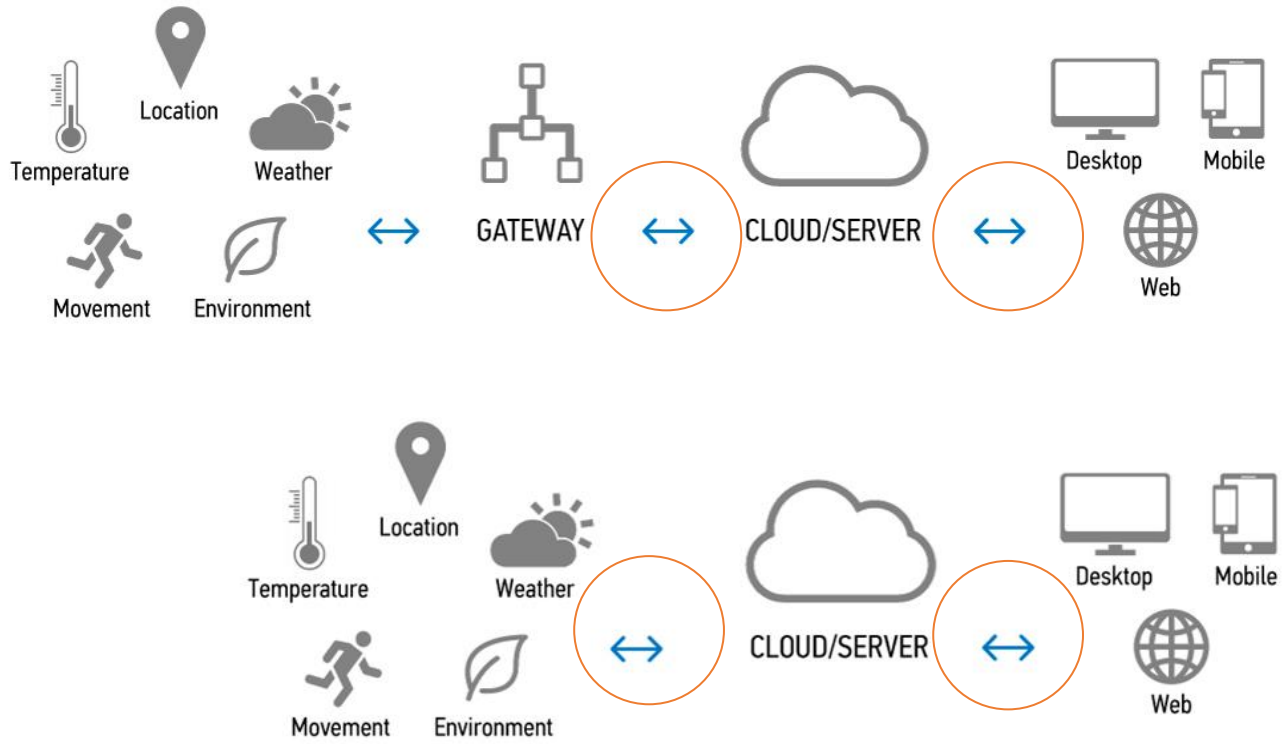



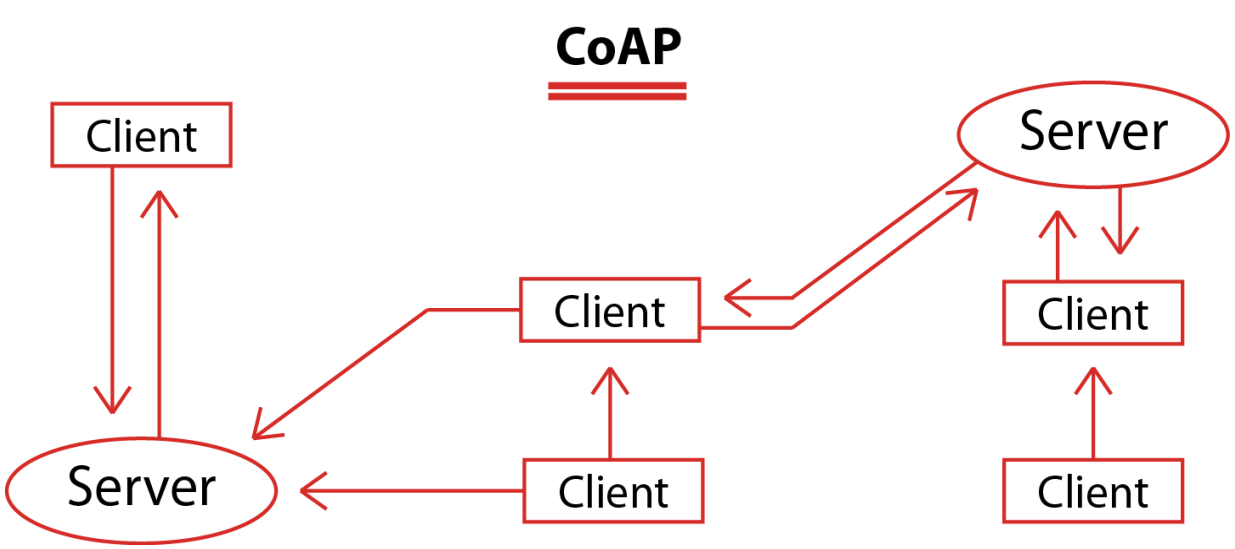
# Communication protocols

- Low power
  - NB-IoT (5G)
  - LTE-M (5G)
- Plugged in
  - Wifi
  - Ethernet
  - 3G/4G/LTE

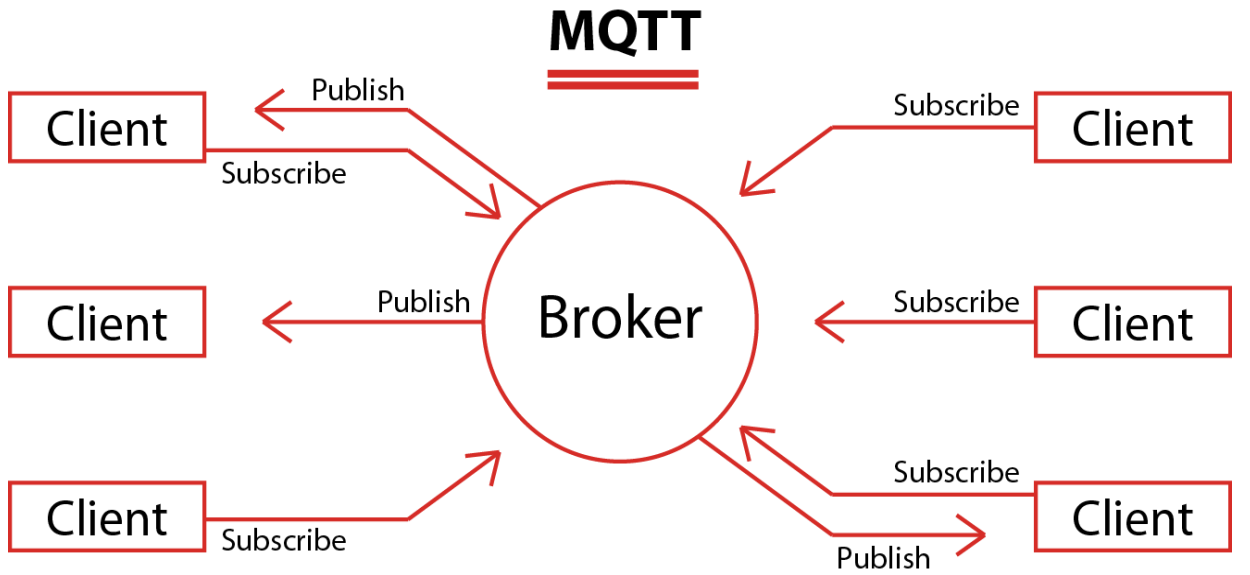



???
---
TCP/IP

HTTPS
---
TCP/IP




Client – Server architectural style




Pub/Sub architectural style

Constrained Application Protocol (CoAP), is a **client-server protocol (one-to-one)**.

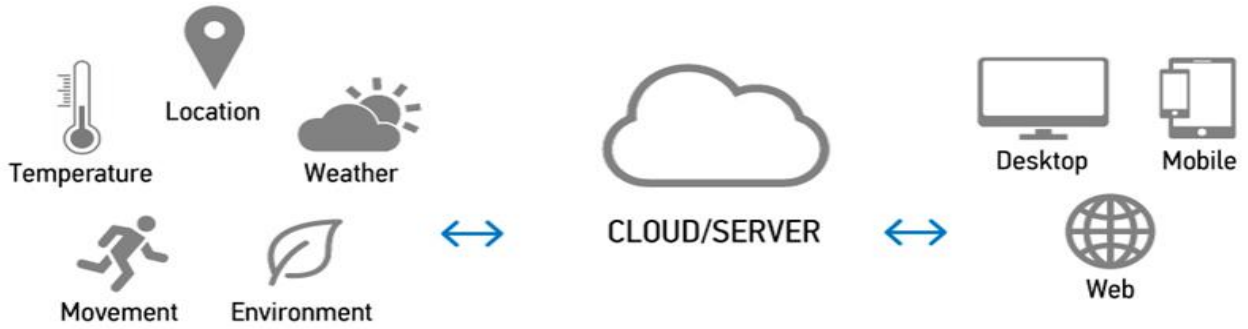
- Designed for interoperability with the HTTP and RESTful web through proxy.
- Request / response model

Message Queue Telemetry Transport (MQTT), is a **publish-subscribe protocol** that facilitates **many-to-many communication** mediated by brokers.

- Designed for bandwidth-efficient and high latency network
- Separation between producers and consumer of data.

Source: pickdata.net

# MQTT

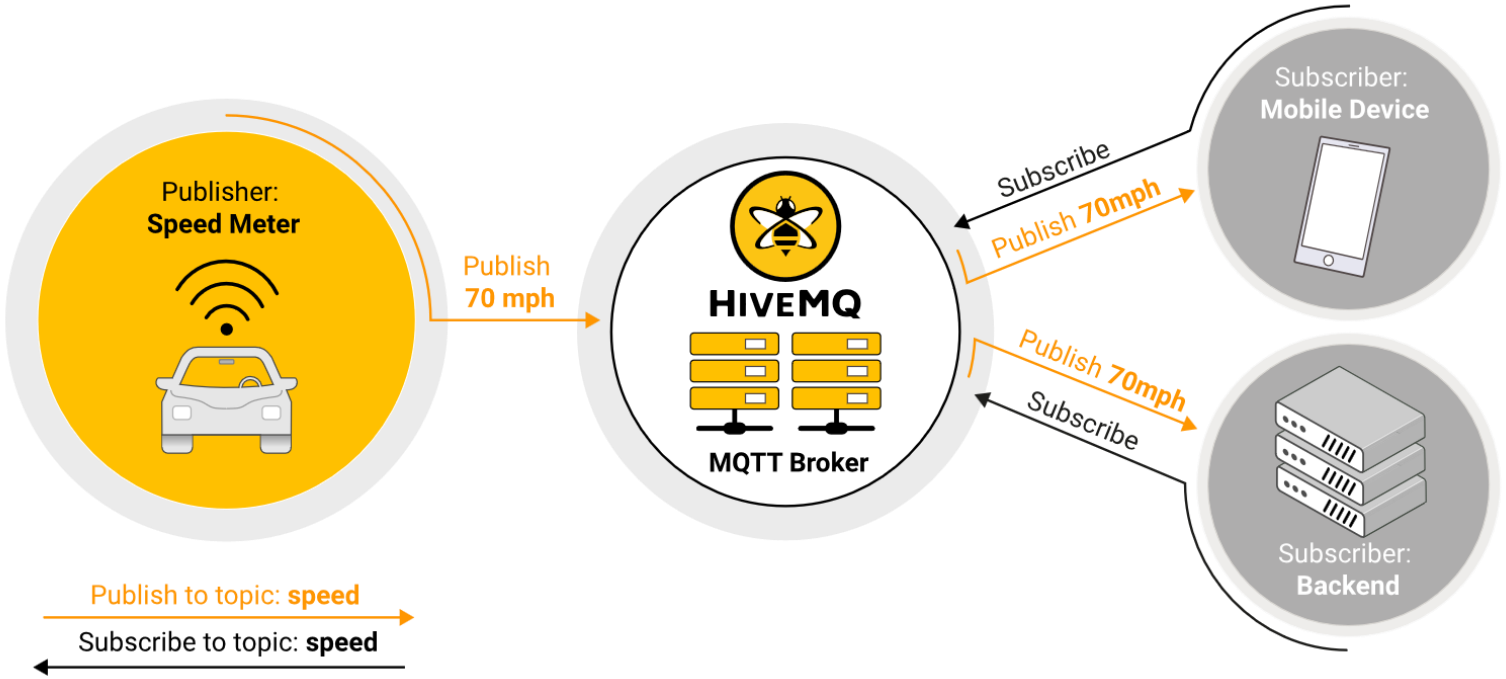



**Space decoupling:** Publisher and subscriber do not need to know each other (for example, no exchange of IP address and port).

**Time decoupling:** Publisher and subscriber do not need to run at the same time.

**Synchronization decoupling:** Operations on both components do not need to be interrupted during publishing or receiving.

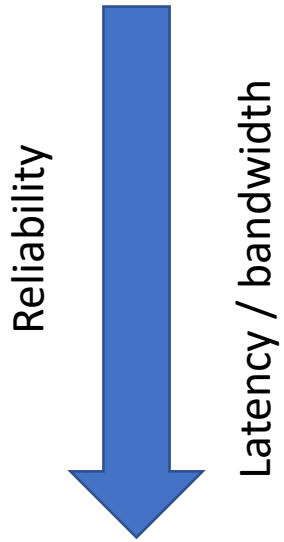
**Scalability**: Easy to scale up system in number of publishers and subscribers.




Source: HiveMQ

# MQTT QoS

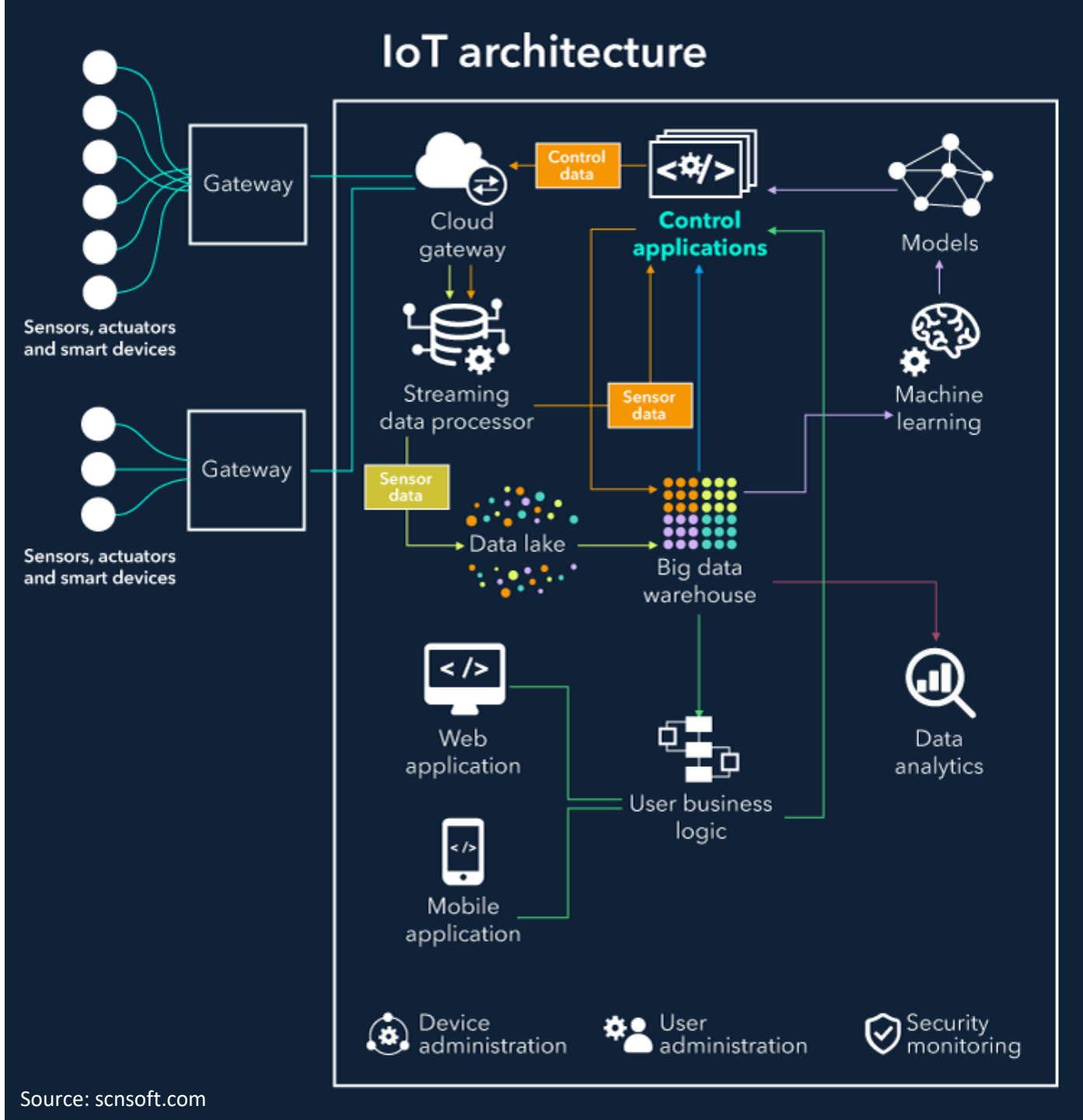
- How hard the broker/client will try to ensure message is received.
- 0: Message delivered at most once, with no confirmation (fire-and-forget)
- 1: Message delivered at least once, with confirmation required.
- 2: Message delivered exactly once by using a 4-step handshake.



# MQTT useful features

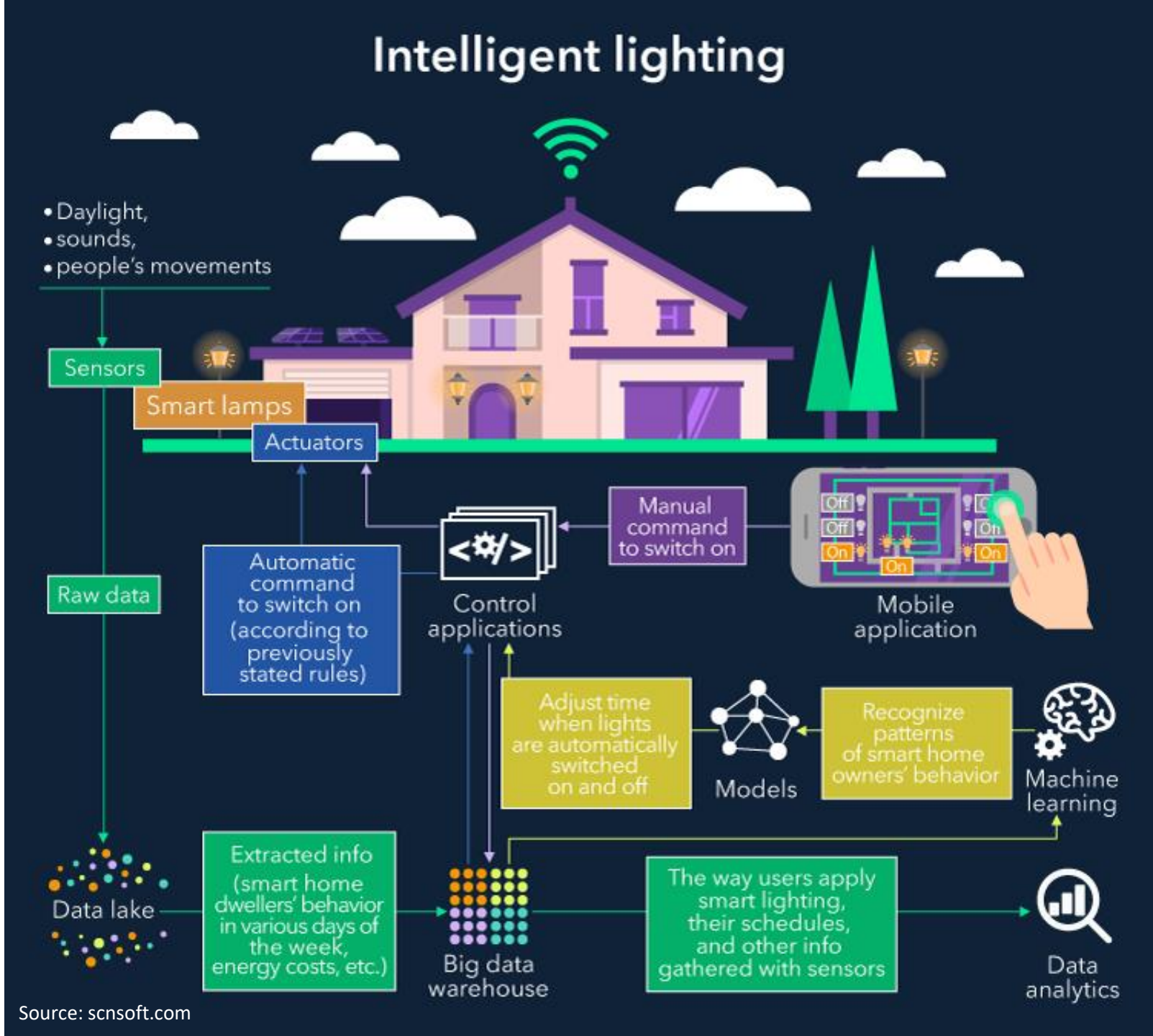
- Persistent session
  - Avoid having to re-subscribing every time the connection is interrupted
- Retained message, ‘last known good’
  - If a topic is only updated infrequently, then without a retained message, a newly subscribed client may have to wait a long time to receive an update.
- Last Will and Testament
  - ‘Last Will’ message that will be delivered to other clients when a client disconnects unexpectedly.
- Keepalive
  - Ensure the connection between client and broker is still open

# Cloud & backend



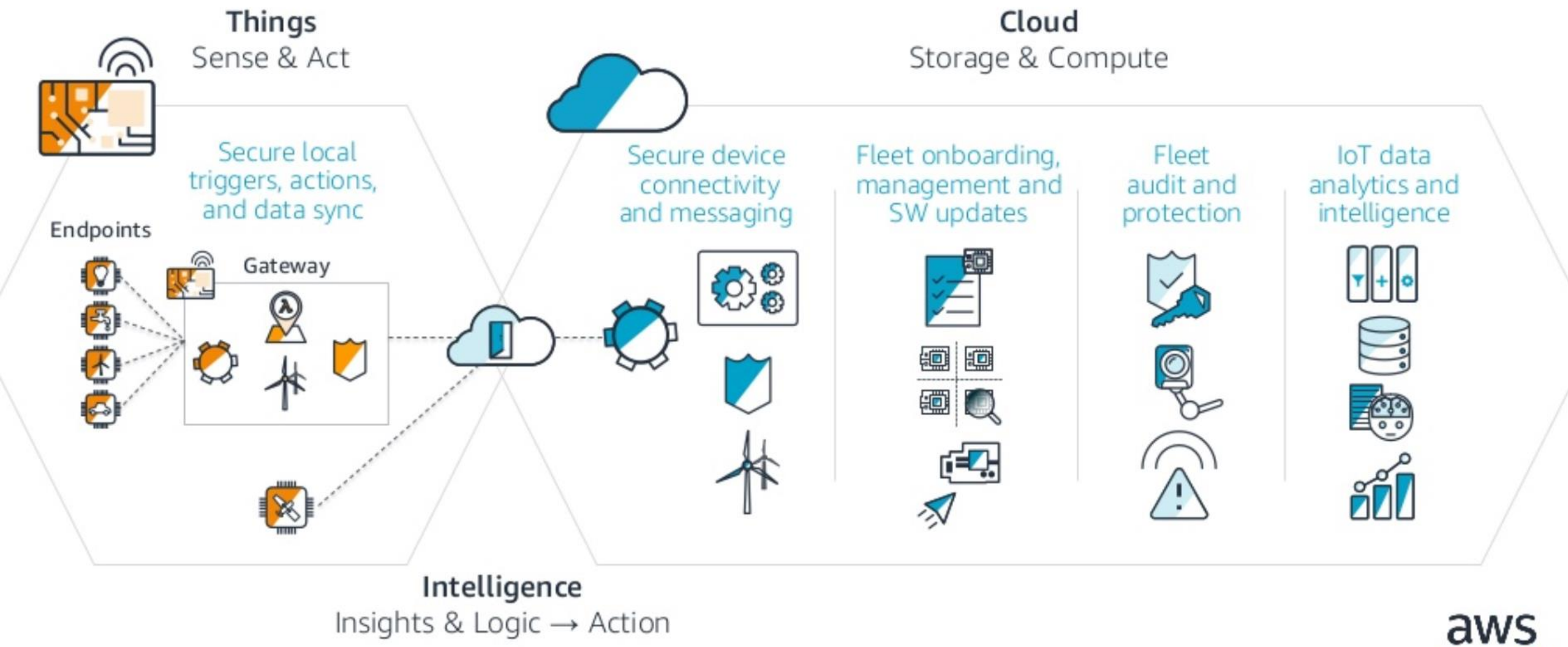

Source: scnsoft.com

# Cloud & backend



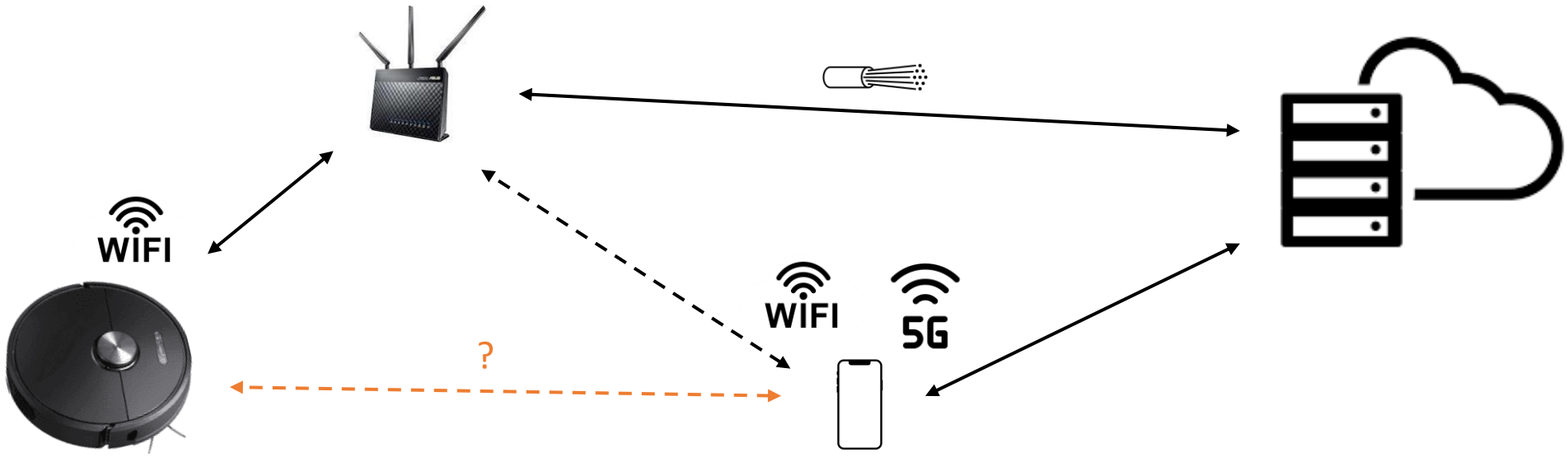

Source: scnsoft.com

# AWS IoT Architecture




aws

# Examples of IoT product



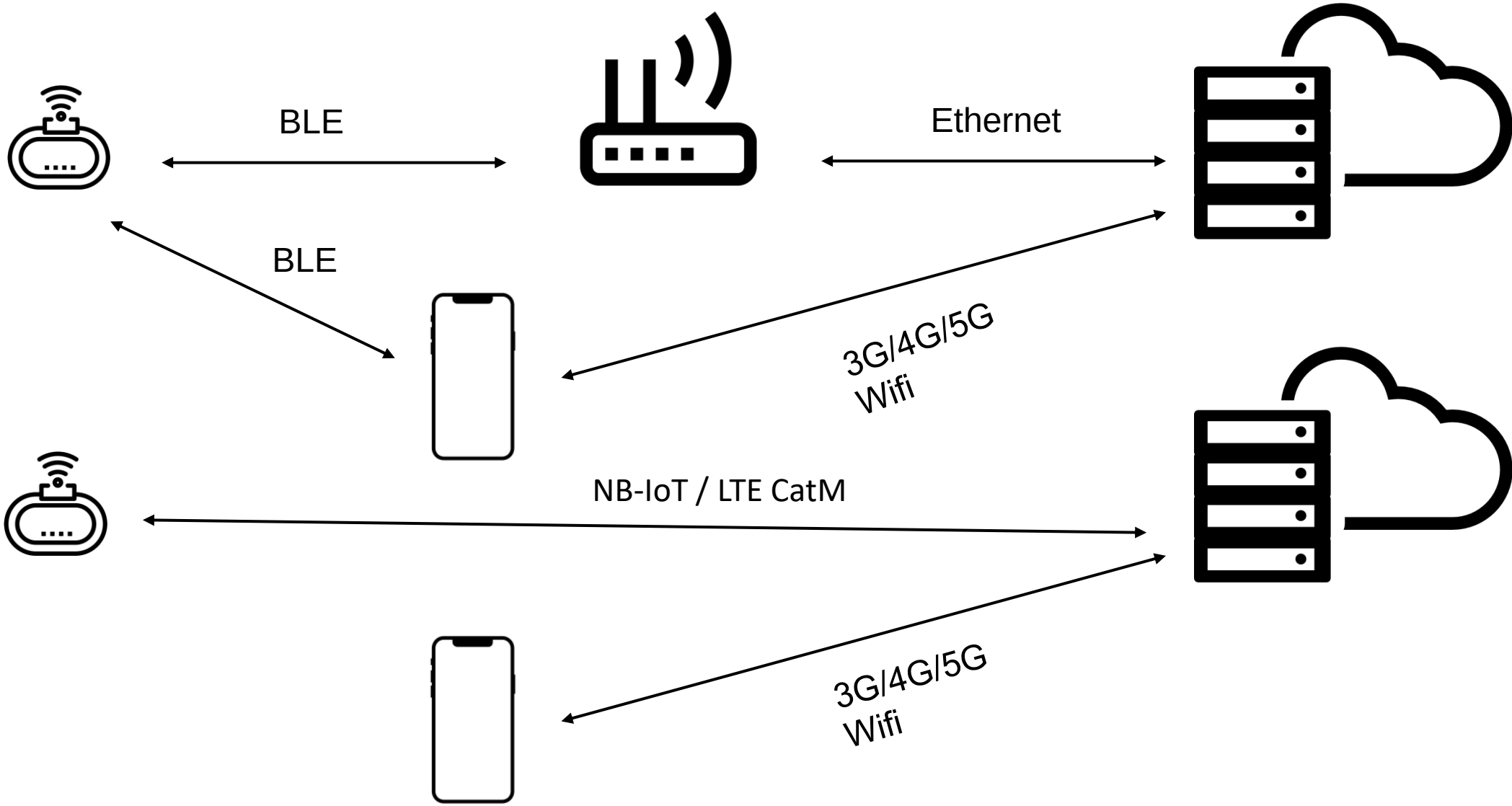

Robot vacuum cleaner

# Case 1: selection of HW/SW

- Small, wearable embedded device that perform measurement
- Collected data to be send to cloud for processing and storage
- Coin cell battery powered, minimum 1 year
- Low cost

- What is sample rate and duration required for measurement? How often?
- Which wireless network infrastructure can be used? Wifi? Bluetooth? Lora? 5G?
- How much data to be transferred?
- Which MCU has low power mode? Which communication method use low power?
- Can we use single MCU for both measurement and wireless communication?

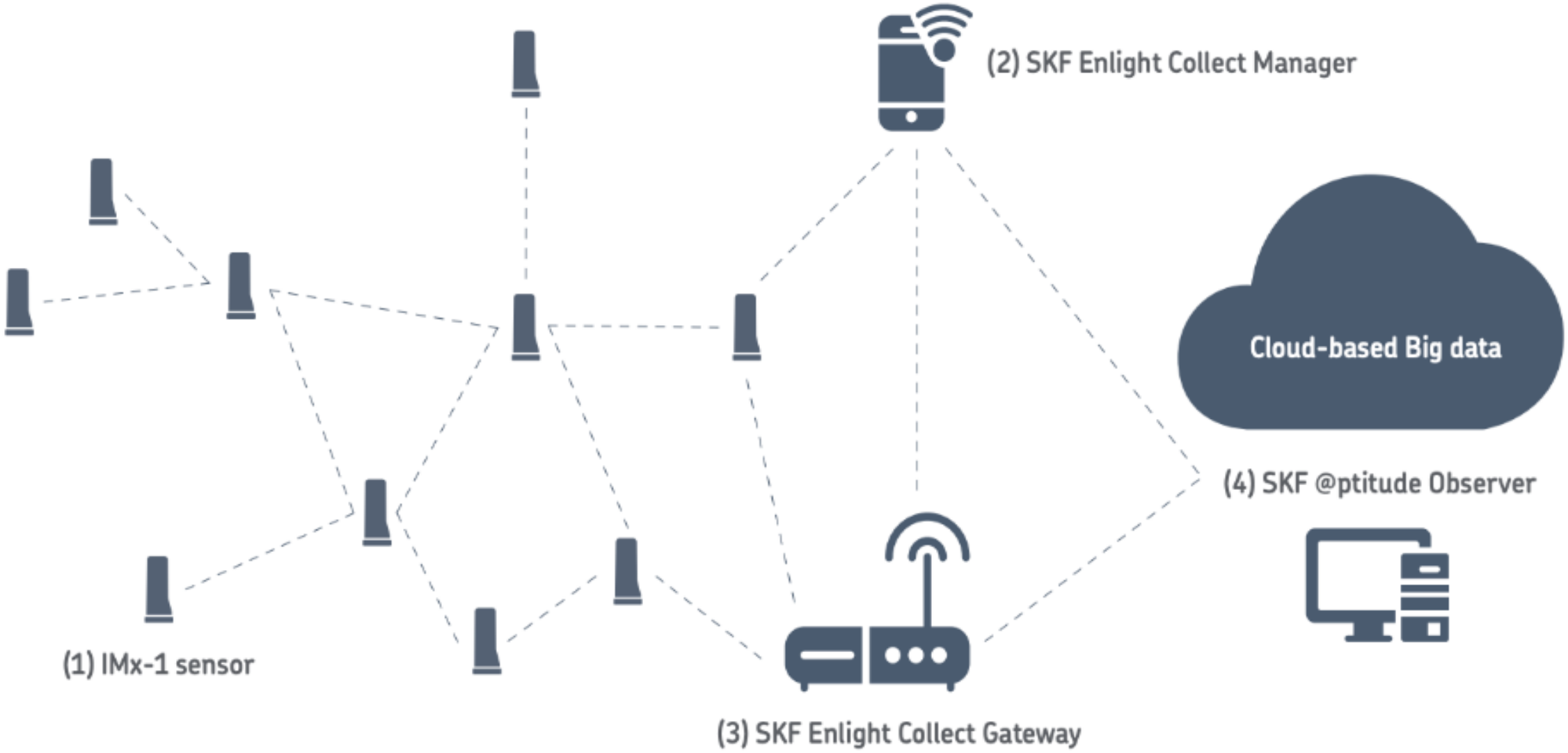
# Case 1: selection of HW/SW

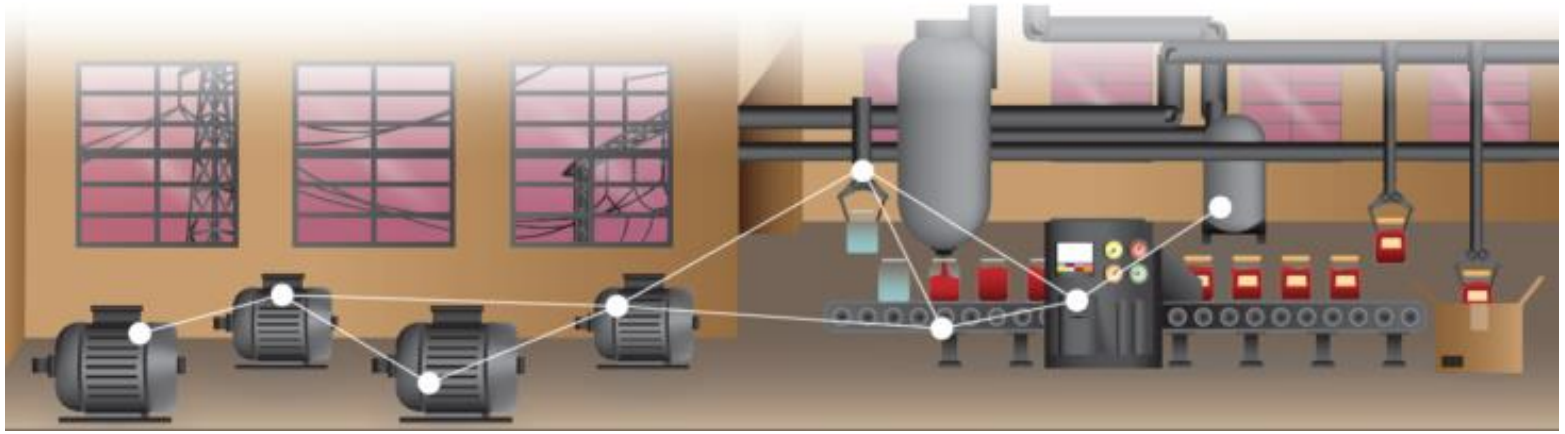


# Case 2: selection of HW/SW

- Industrial embedded device that perform measurement
- Data send to cloud for processing and storage
- ~100 devices per location
- Small battery powered, 3 year
- Low cost
- Hash environment

- What is sample rate and duration required for measurement? How often?
- Which wireless network infrastructure can be used? Wifi? Bluetooth? Lora? 5G?
- How much data to be transferred?
- Which MCU has low power mode? Which communication method use low power?
- Can we use single MCU for both measurement and wireless communication?


(2) SKF Enlight Collect Manager
Cloud-based Big data
(4) SKF @ptitude Observer
(1) IMx-1 sensor
(3) SKF Enlight Collect Gateway

Source: LumenRadio

# Case 3: selection of HW/SW

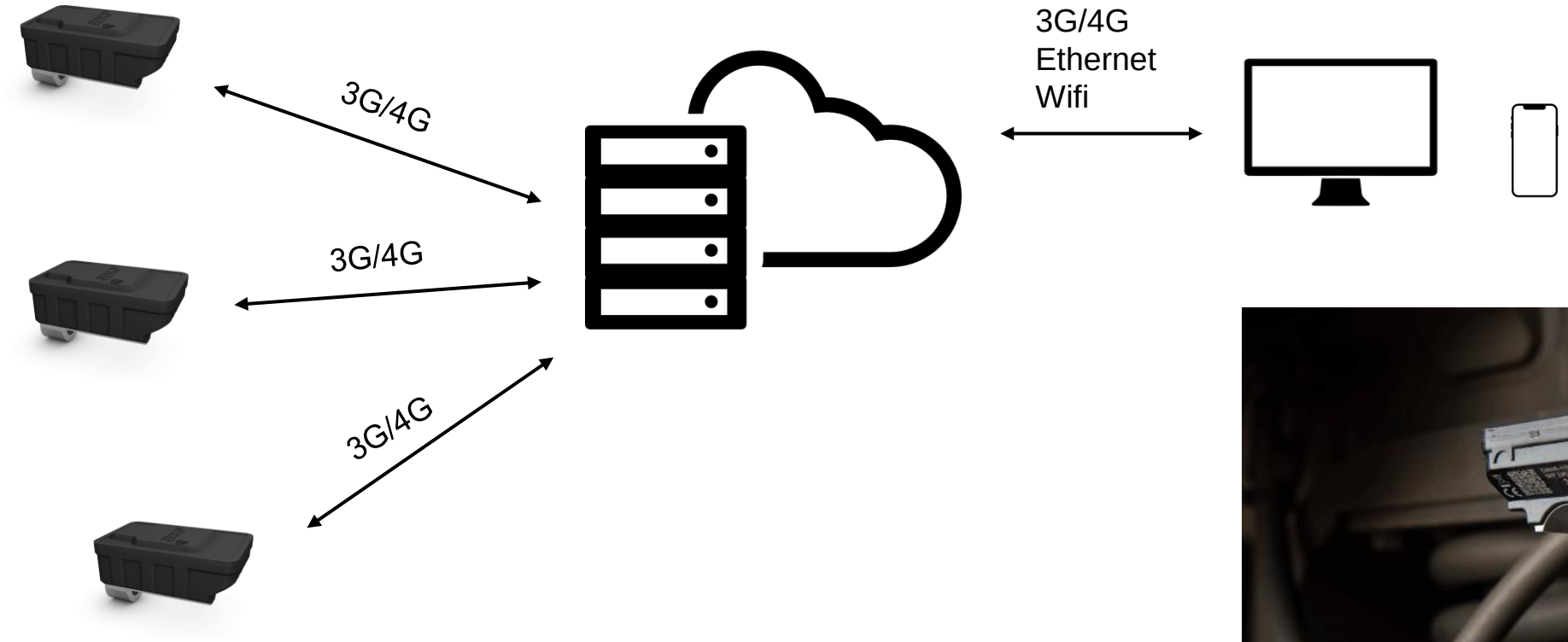
- Industrial embedded device that perform measurements on running vehicles
- Collected data to be send to cloud for processing and storage
- Battery powered
- Wireless
- Easy installation

- What is sample rate and duration required for measurement? How often?

- Which wireless network infrastructure can be used? Wifi? Bluetooth? Lora? 5G?

- How much data to be transferred?

- Which MCU has low power mode? Which communication method use low power?


3G/4G
3G/4G
3G/4G
3G/4G
Ethernet
Wifi


Thank you