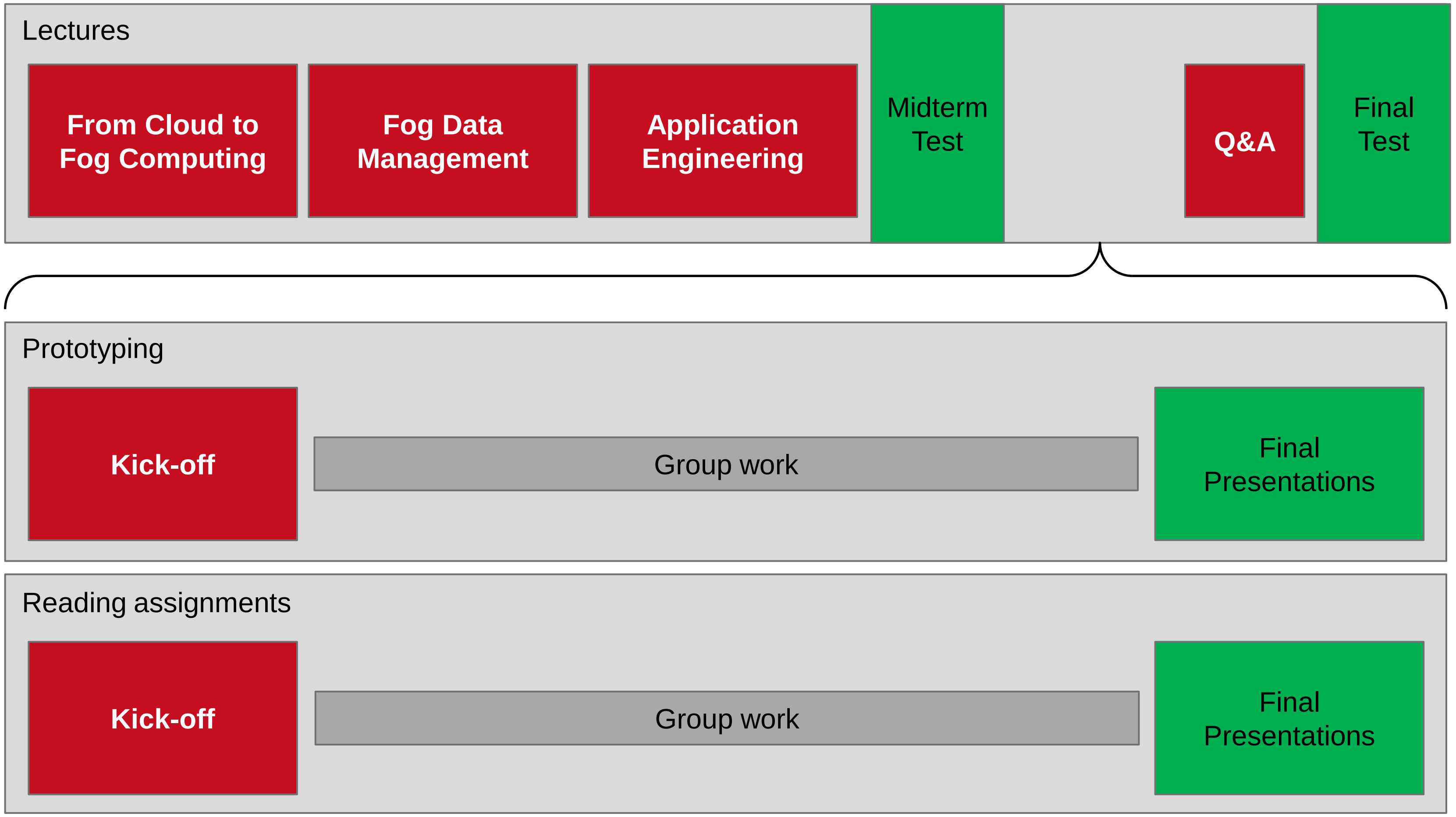
Fog Computing

Bermbach | Part 3: Application Engineering

Agenda


Lectures
From Cloud to Fog Computing
Fog Data Management
Application Engineering
Midterm Test
Q&A
Final Test
Prototyping
Kick-off
Group work
Final Presentations
Reading assignments
Kick-off
Group work
Final Presentations

Application Engineering

How does the shift from cloud to fog computing affect applications? Surely, we cannot use the same application design and development processes…

Cloud vs. Fog Application Engineering

Application Design

- Architecture
- Communication
- Geo-awareness
- Fault-Tolerance
- Security

Development Process

- Rollout
- Testing

Cloud vs. Fog Application Engineering

Application Design

- Architecture
- Communication
- Geo-awareness
- Fault-Tolerance
- Security

Development Process

- Rollout
- Testing

Application Design

ARCHITECTURE

What is Architecture?

Architecture is constraint-based design

- Design without constraints probably is art

Constraints can be derived from a wide variety of sources

- Technical infrastructure (current landscape and expected developments)
- Business considerations (current landscape and expected developments)
- Time horizon (short-term vs. long-term requirements)
- Existing architecture
- Scalability
- Performance (based on performance requirements and definitions)
- Cost (development, deployment, maintenance)

Architectural Styles

Architectural Style: General principles informing the creation of an architecture
Architecture: Designing a solution to a problem according to given constraints

Architectural styles *inform* and *guide* the creation of architectures

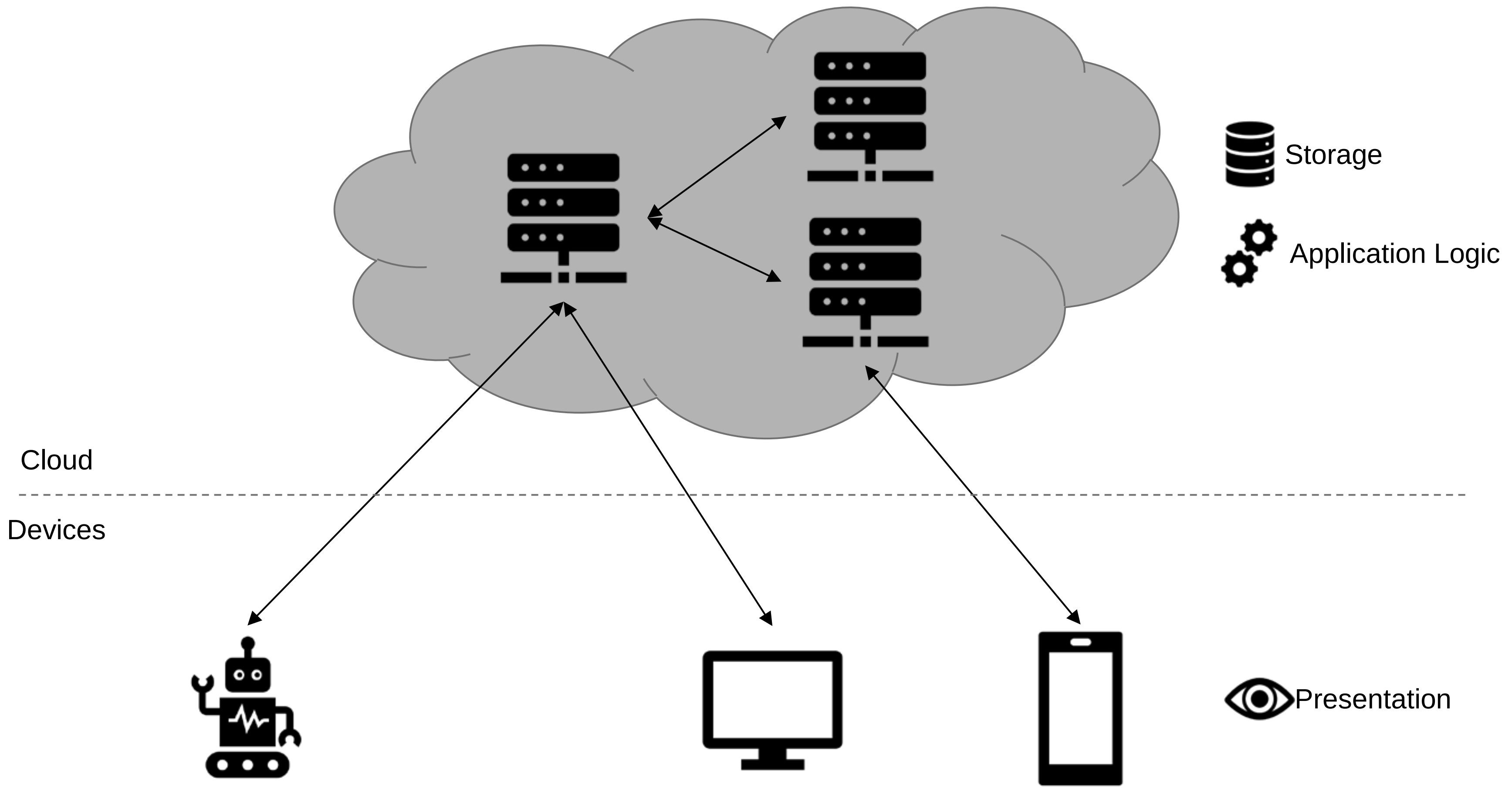
Architecture: Louvre

Architectural Style: Baroque

Architecture: Villa Savoye

Architectural Style: International Style

SoTA in Cloud Architecture: Microservices



Microservice Architecture Definitions I

“In short, the microservice architectural style is an approach to developing a single application as a suite of small [focused] services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.” [Martin Fowler]

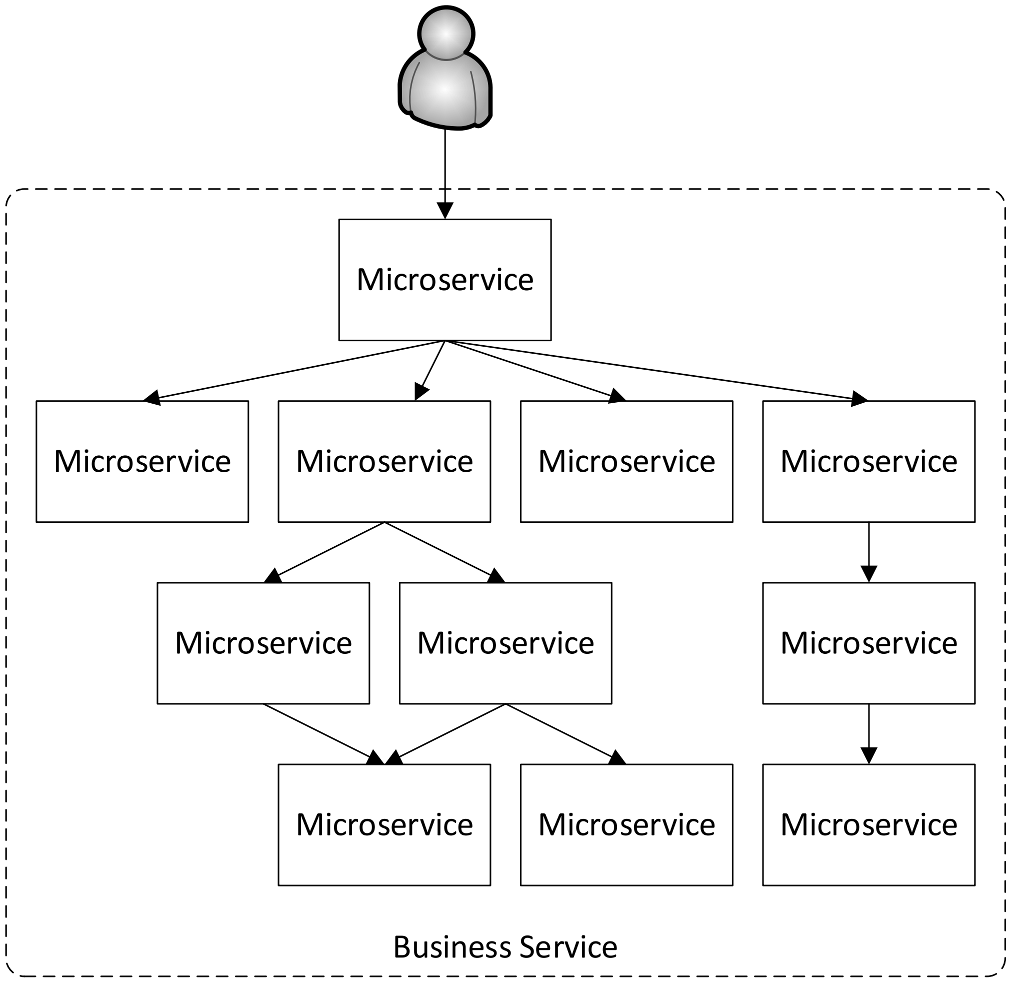
Microservice Architecture Definitions II

“A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices.“

“Loosely coupled, service-oriented architecture with bounded contexts”

[Adrian Cockroft]

Microservice Architecture



Each user request is satisfied by some sequence of services

Most services are not externally available

Each service communicates with other services through service interfaces

Service depth?

Source: Bass, L. and Weber, I. and Zhu, L. (2015)

Example: Amazon design rules

All teams will henceforth expose their data and functionality through service interfaces.

Teams must communicate with each other through these interfaces.

There will be no other form of inter-process communication allowed:

- no direct linking, no direct reads of another team's data store,
- no shared-memory model, no back-doors whatsoever.
- The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they[services] use.

All service interfaces, without exception, must be designed from the ground up to be externalizable.

Source: Bass, L. and Weber, I. and Zhu, L. (2015)

Characteristics of Microservices [Fowler]

1. Componentization via Services
   - A component is a unit of software that is independently replaceable and upgradeable.
   - Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services.
2. Organized around Business Capabilities
   - Cross-functional teams
3. Products not Projects
   - A development team takes full responsibility for the software in production (“you build it, you run it”).

http://martinfowler.com/articles/microservices.html

Characteristics of Microservices [Fowler]

4. Smart endpoints and dumb pipes
   - Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their own domain logic and act more as filters in the classical Unix sense – receiving a request, applying logic as appropriate and producing a response.
   - These are choreographed using simple REST-"ish" protocols rather than complex protocols or orchestration by a central tool.
5. Decentralized Governance
   - Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing.
6. Decentralized Data Management
   - Application Databases

http://martinfowler.com/articles/microservices.html

Characteristics of Microservices [Fowler]

7. Infrastructure Automation
   - Automated, continuous deployment and testing processes
8. Design for failure
   - A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.
   - Example: Netflix's Simian Army
9. Evolutionary Design
   - The key property of a component is the notion of independent replacement and upgradeability.
   - Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process.

http://martinfowler.com/articles/microservices.html

Microservice Design Principles I

Microservice design principles combine well-known modular software design guidelines (e.g. Unix design maxims) and best-practice experience in building SOA at scale (e.g. Netflix, Amazon Web Services, SoundCloud)

A microservice architecture is domain-specific, i.e. bounded contexts of microservices must be chosen depending on data models, reflecting the scope of business capabilities appropriately; see also [1]

Microservices are aligned to (individual) product ownership responsibilities

- Tooling and frameworks for development: should depend on individual team capabilities and preferences; implement own tooling, if appropriate
- Runtime configuration: owners test and manage release configurations for deployed artifacts themselves
- Observability: custom monitoring and debugging

Adopted from: Nadareishvili et al. (2016) – Microservice Architecture

[1] Eric Evans (2003) – Domain Driven Design: Tackling Complexity in the Heart of Software

Microservice Design Principles II

Microservice architectures adopt (shared) platforms in their development, deployment and runtime environment

- Infrastructure: Rely on managed, on-demand provisioning of virtualized hardware resources, e.g. Cloud-IaaS, container-based virtualization
- Use tools for automation for code versioning, testing and deployment
- Data stores: individual instances (or shards) per microservice, but shared best practices and technology
- Service orchestration: interoperability through the adoption of common discovery and loadbalancing tools
- Shared security and identity management
- Company policies, e.g., logging to central auditing platform

Based on: Nadareishvili et al. (2016) – Microservice Architecture

Fog Architecture: Microservices?

Pros

- Communication via interface calls
- No shared memory
- Design for failure
- Variable / unstable environment

=> Loose coupling

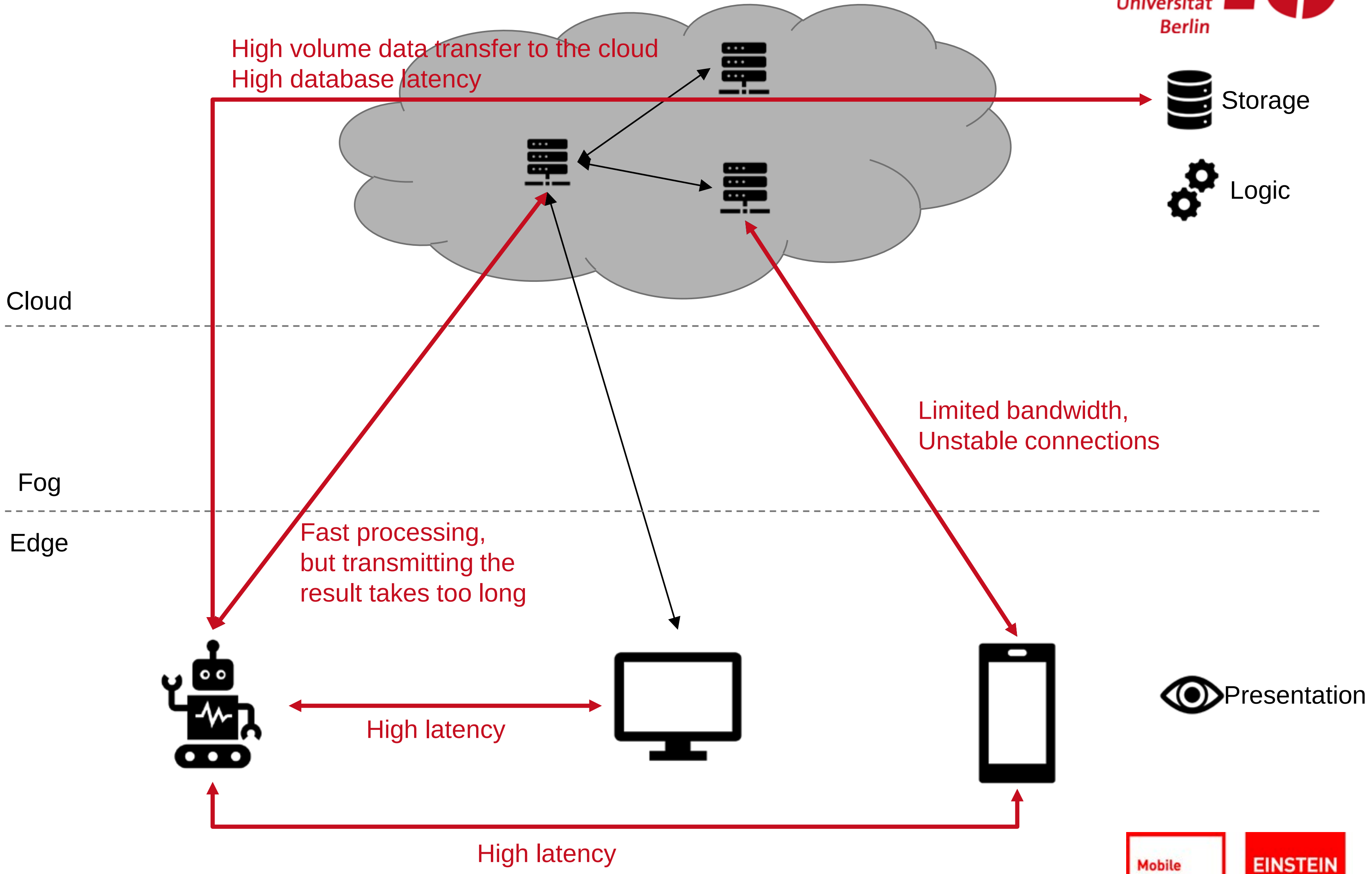
Cons

- Limited bandwidth
- Higher latency between nodes
- Unreliable connections
- Difficulty of infrastructure automation
- Geographic distribution
- …

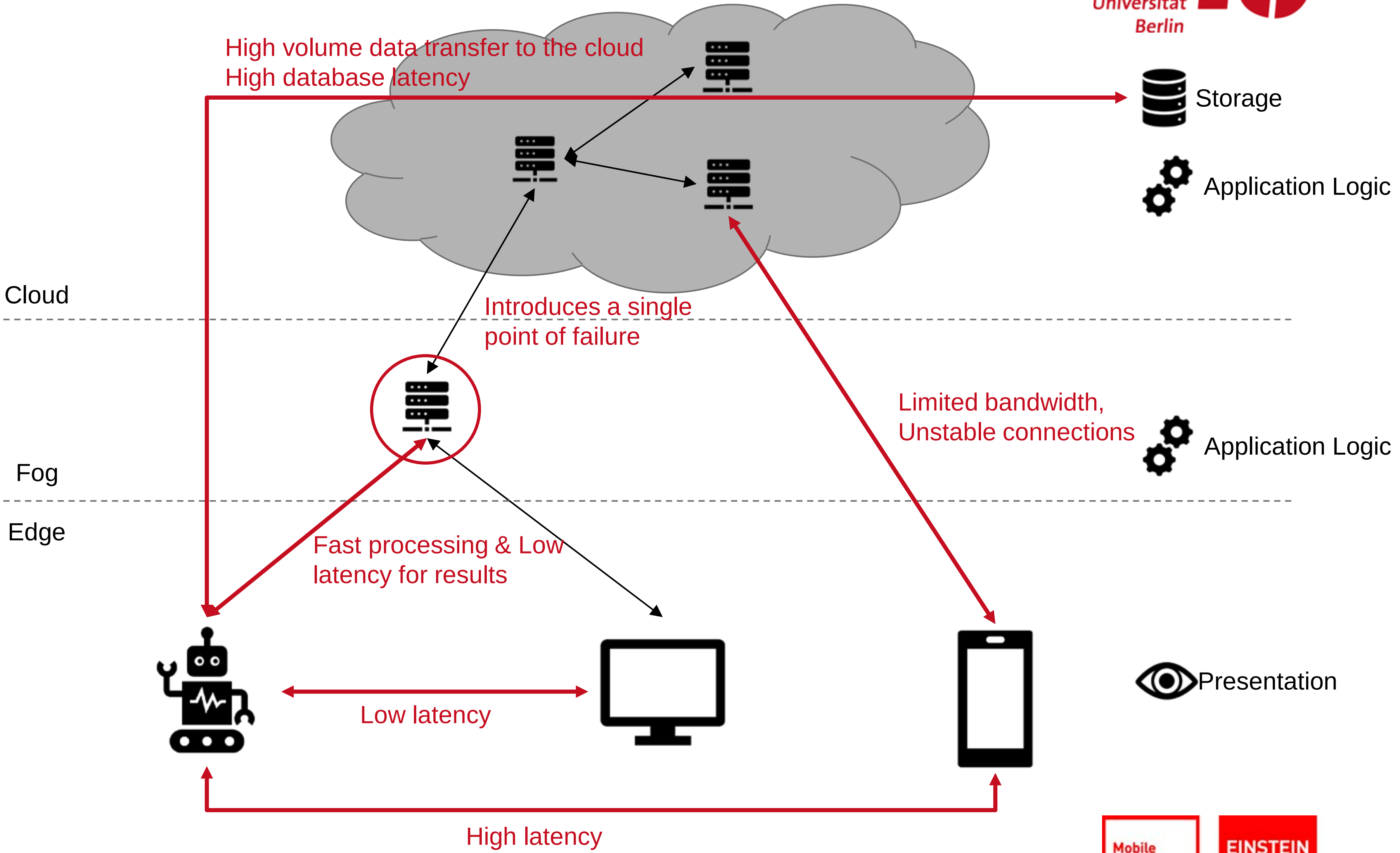
Loose Coupling

- Important design principle in distributed systems
- Components communicate via well defined interfaces
- Components can be replaced arbitrarily as long as they implement the specified interface(s) correctly
- Faults should be limited to the failed component and should not affect the overall distributed system
- Use standardized protocols and data types (RESTful, JSON, XML, …)

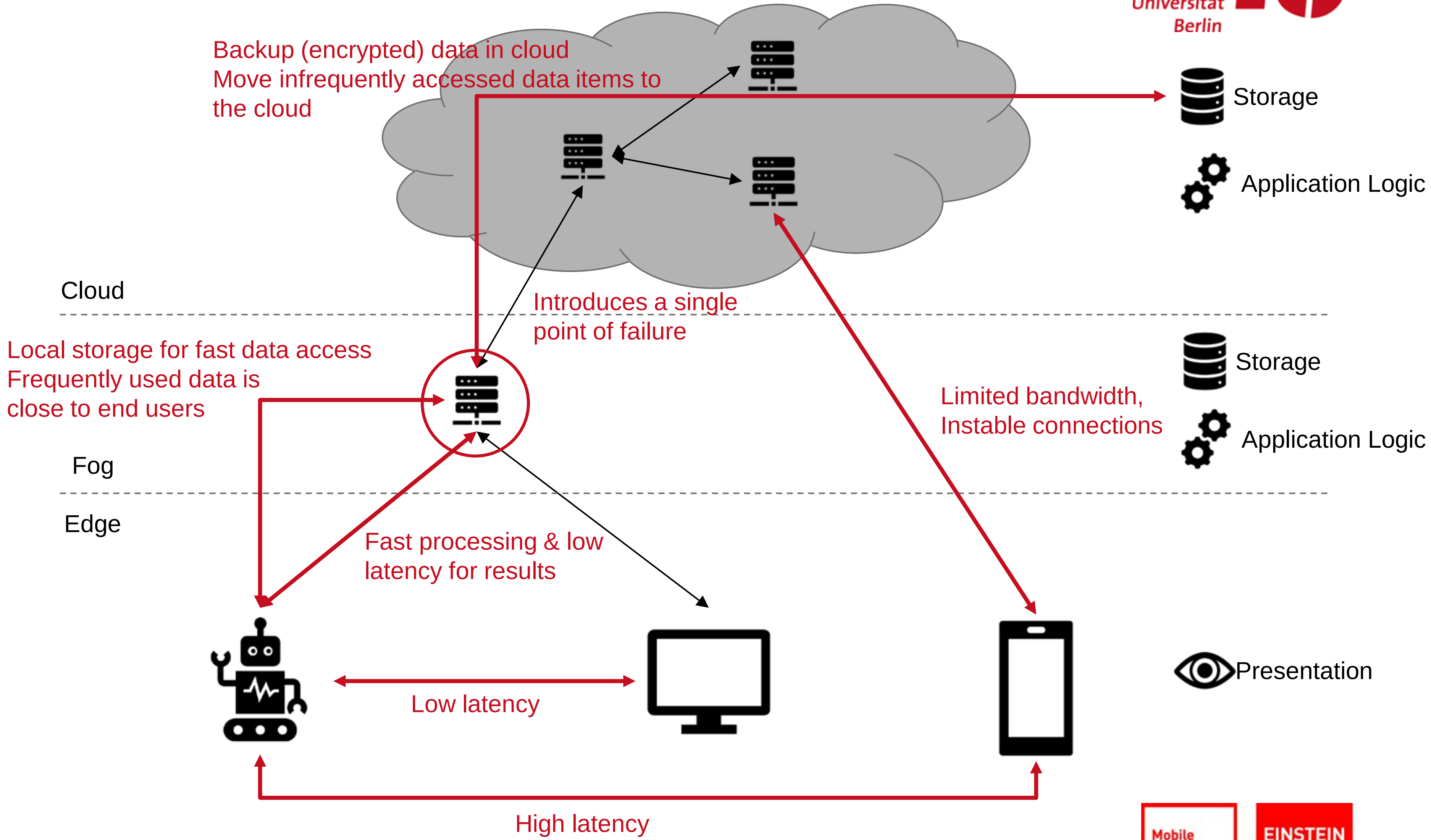
Fog Architecture = Cloud Architecture?



Introduce Fog Nodes & Split Application Logic

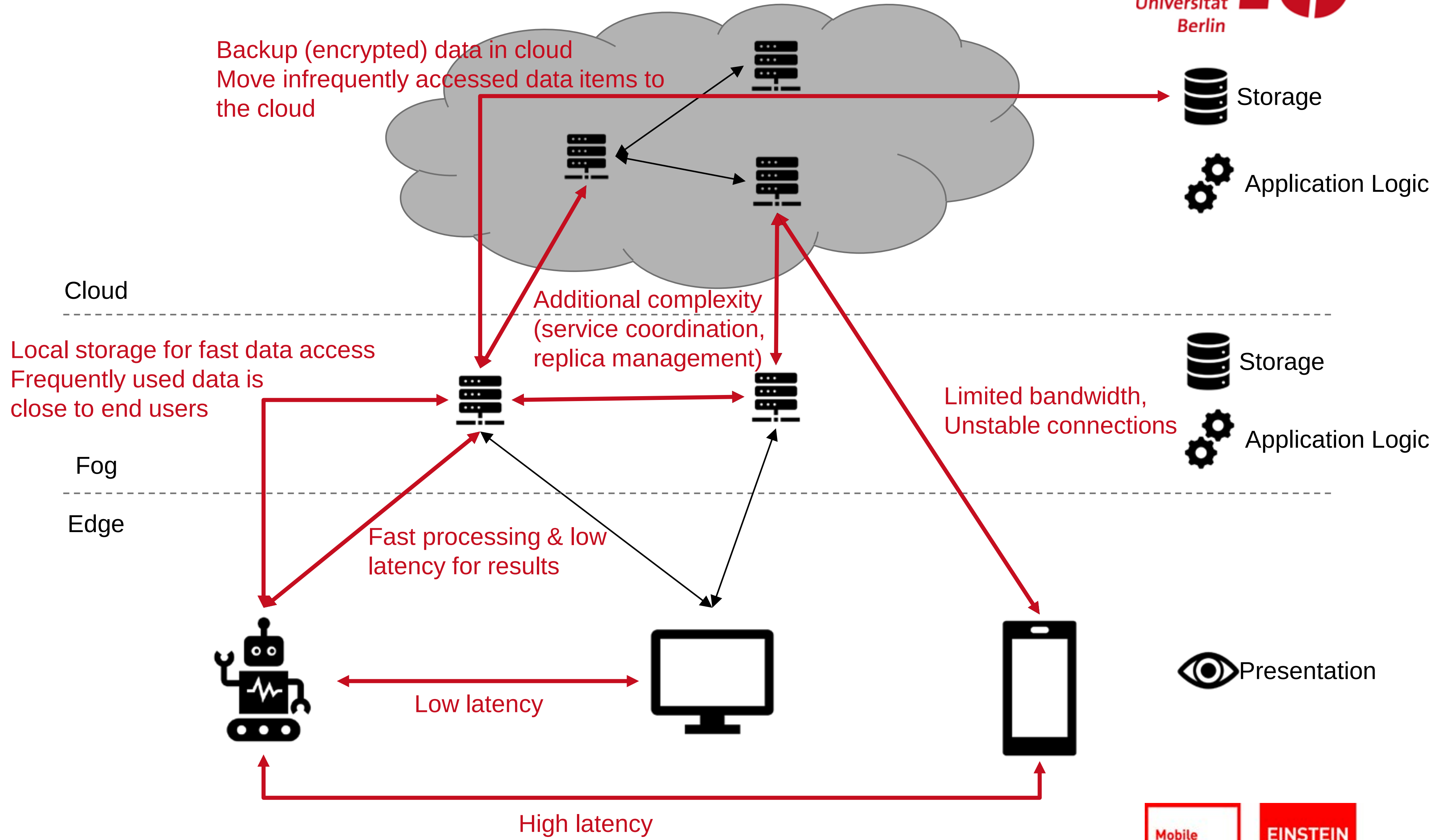

High volume data transfer to the cloud
High database latency
Storage
Application Logic
Cloud
Introduces a single point of failure
Limited bandwidth, Unstable connections
Application Logic
Fog
Edge
Fast processing & Low latency for results
Low latency
Presentation
High latency

Replicate Storage (Parts) to Fog Nodes

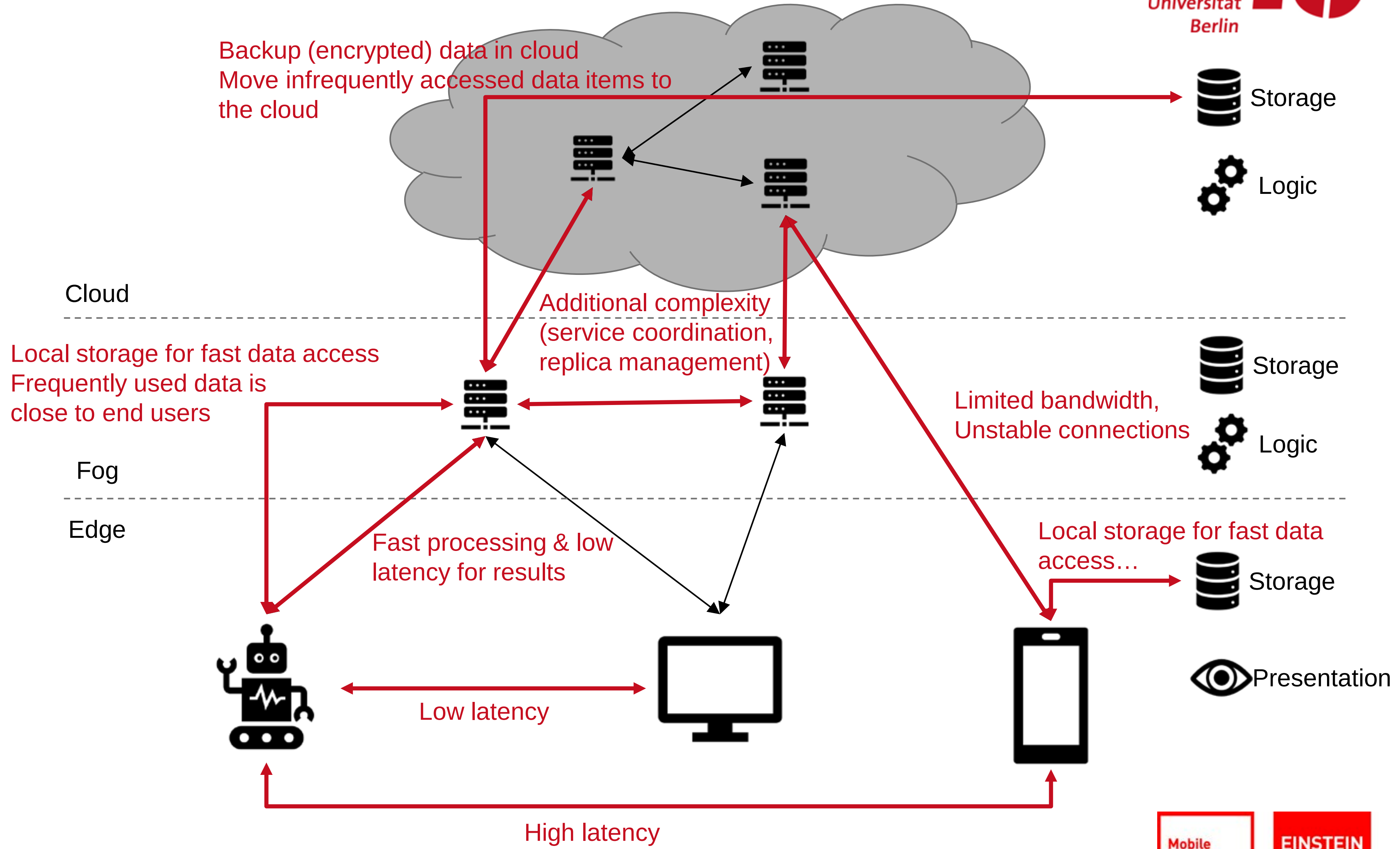


Backup (encrypted) data in cloud
Move infrequently accessed data items to the cloud
Storage
Application Logic
Cloud
Introduces a single point of failure
Local storage for fast data access
Frequently used data is close to end users
Storage
Limited bandwidth, Instable connections
Application Logic
Fog
Edge
Fast processing & low latency for results
Presentation
Low latency
High latency

Eliminate SPOF through Replication


Backup (encrypted) data in cloud
Move infrequently accessed data items to the cloud
Storage
Application Logic
Cloud
Additional complexity (service coordination, replica management)
Local storage for fast data access
Frequently used data is close to end users
Storage
Limited bandwidth, Unstable connections
Application Logic
Fog
Edge
Fast processing & low latency for results
Presentation
Low latency
High latency

Replicate Storage (Parts) to Edge Devices


Backup (encrypted) data in cloud
Move infrequently accessed data items to the cloud
Cloud
Additional complexity (service coordination, replica management)
Local storage for fast data access
Frequently used data is close to end users
Fog
Limited bandwidth, Unstable connections
Edge
Fast processing & low latency for results
Local storage for fast data access…
Storage
Logic
Storage
Logic
Storage
Presentation
Low latency
High latency

Cloud vs. Fog Architecture

Microservices do not run as a cluster

- Data is geo-replicated (full data set in the cloud, subset near the edge)
- Application logic is broken down into parts, replicated, and distributed across nodes

Depending on use case, each microservice may resemble

- Thick client with aggressive caching and server backend (e.g., mobile)
- (Geo-distributed) stream processing pipeline (e.g., IoT)
- …

Application Design

COMMUNICATION

Basic Styles of Communication

Synchronous (=> RPC)

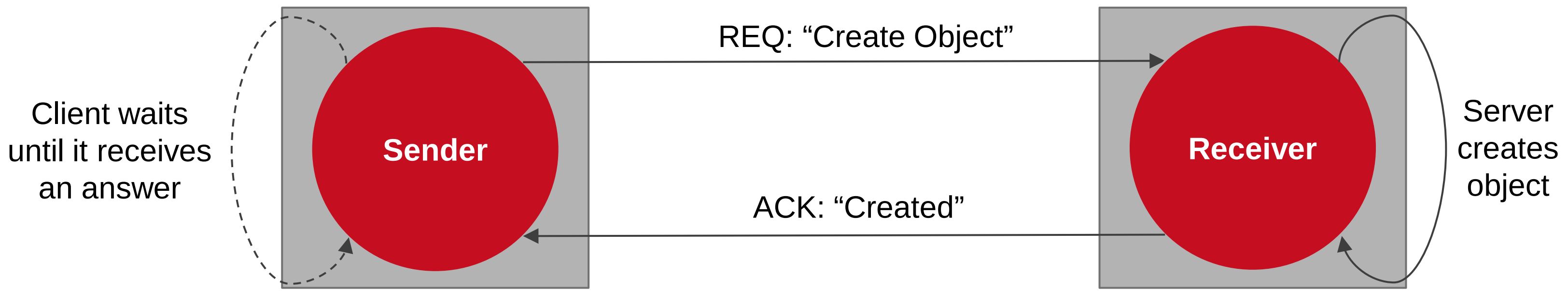
cs.

Asynchronous

- PTP Messaging
- PubSub

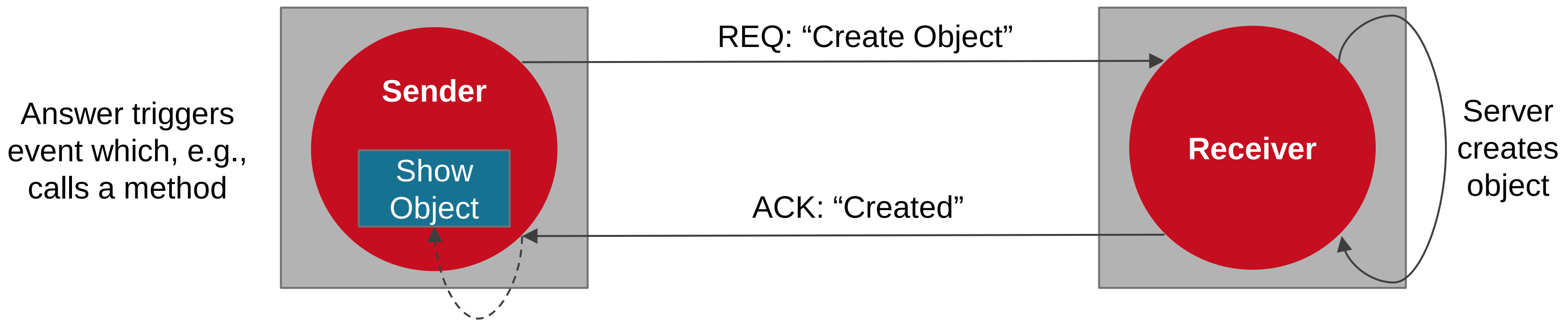
Synchronous Communication

- Sender waits for an answer (will not send another message until it receives an answer from the Receiver)
- Code sends a message (or calls a function) and blocks until an answer (or return value) is received.
- Examples: phone call, method call in Java



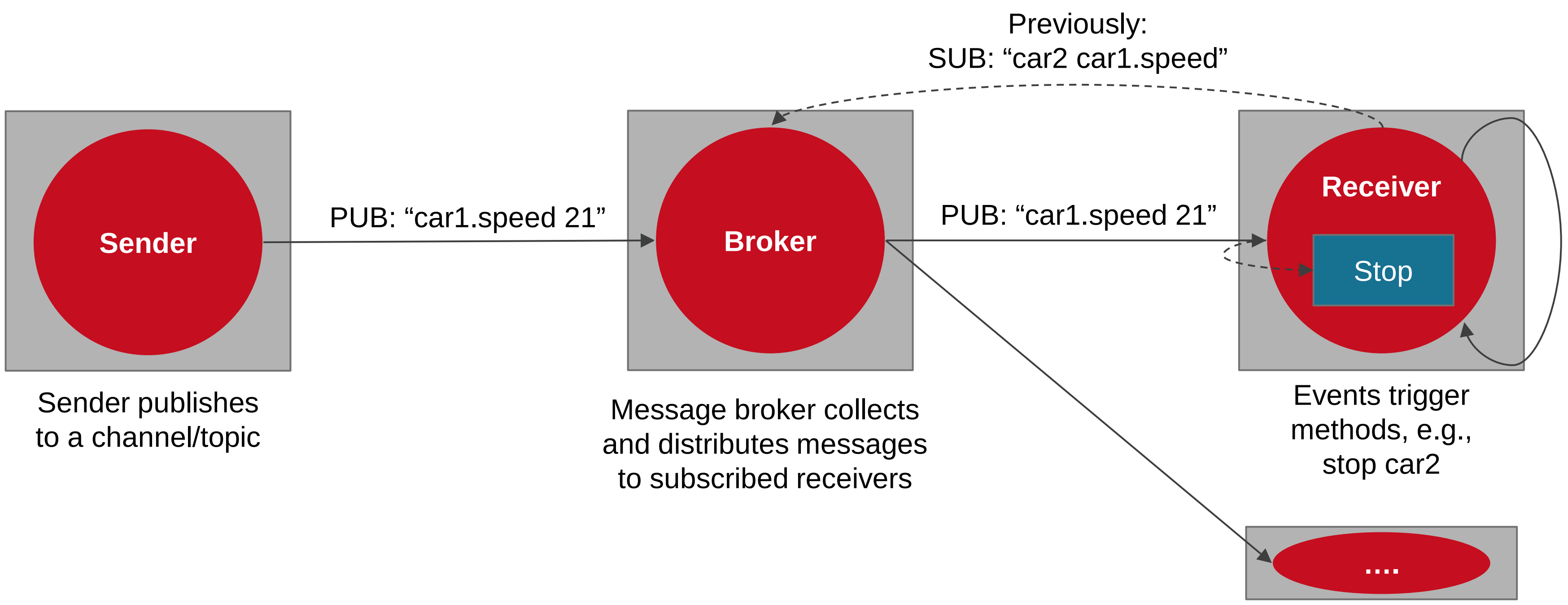
Asynchronous Messaging (PTP)

- Sender doesn’t waits but specifies an event-trigger for the answer
- Code sends a message, a defined function eventually handles the answer.
- Examples: Email, Javascript async/await



Asynchronous Messaging (PubSub)

- Sender publishes to a topic, subscribers receive the messages
- Code either defines a topic and publishes to it; and or subscribes to (multiple) topics and handles incoming messages
- Examples: Mailing list, RSS-Feed


Previously:
SUB: “car2 car1.speed”
Sender
PUB: “car1.speed 21”
Broker
PUB: “car1.speed 21”
Receiver
Stop
....
Sender publishes to a channel/topic
Message broker collects and distributes messages to subscribed receivers
Events trigger methods, e.g., stop car2

Cloud Communication

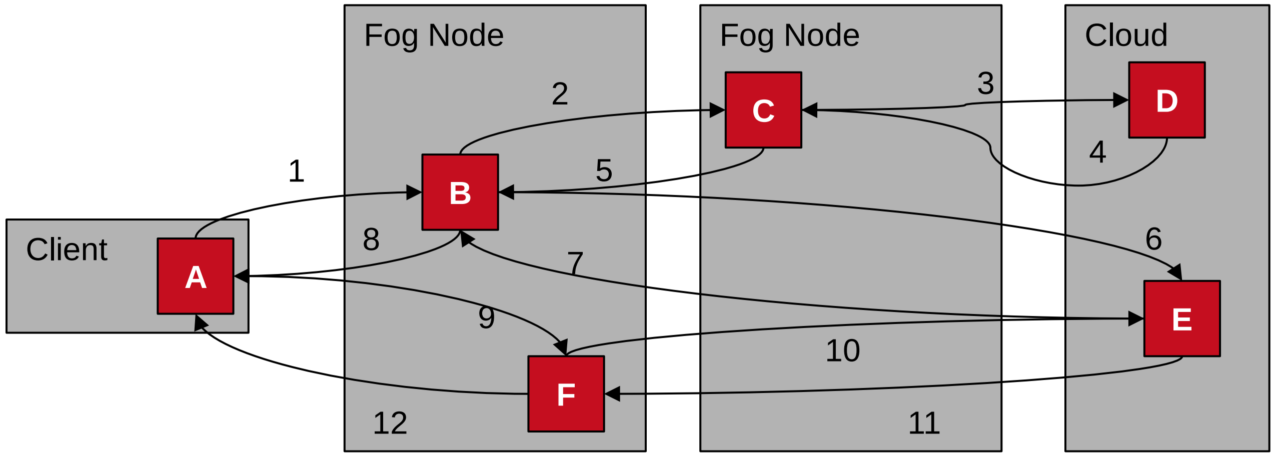
Relies either on messaging services (e.g., SQS), cluster-based messaging systems (e.g., Kafka), or synchronous HTTP calls (particularly REST calls for microservices)

Neither is an option for fog computing:

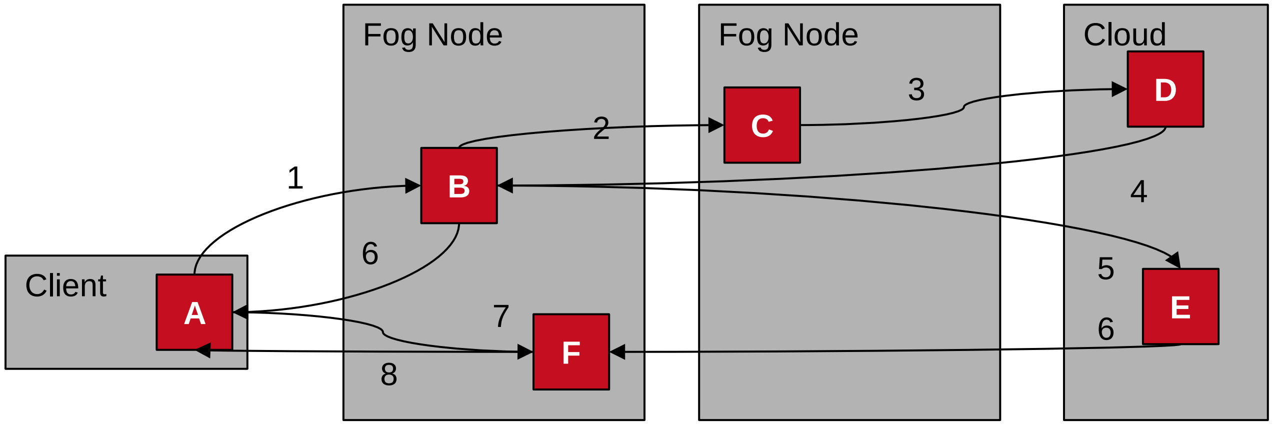
- Messaging services run too far from the edge (at the moment)
- Cluster-based messaging systems are too heavy-weight for the edge
- Synchronous calls don‘t work
  - Geo-distribution (Performance, Time-outs)
  - Failures
  - Sending data from A to B to C

=> Use either messaging systems designed for geo-distribution or light-weight messaging libraries (maybe light-weight RPC libraries for local calls)

Advantages from Asynchronous Messaging



Also removing indirection



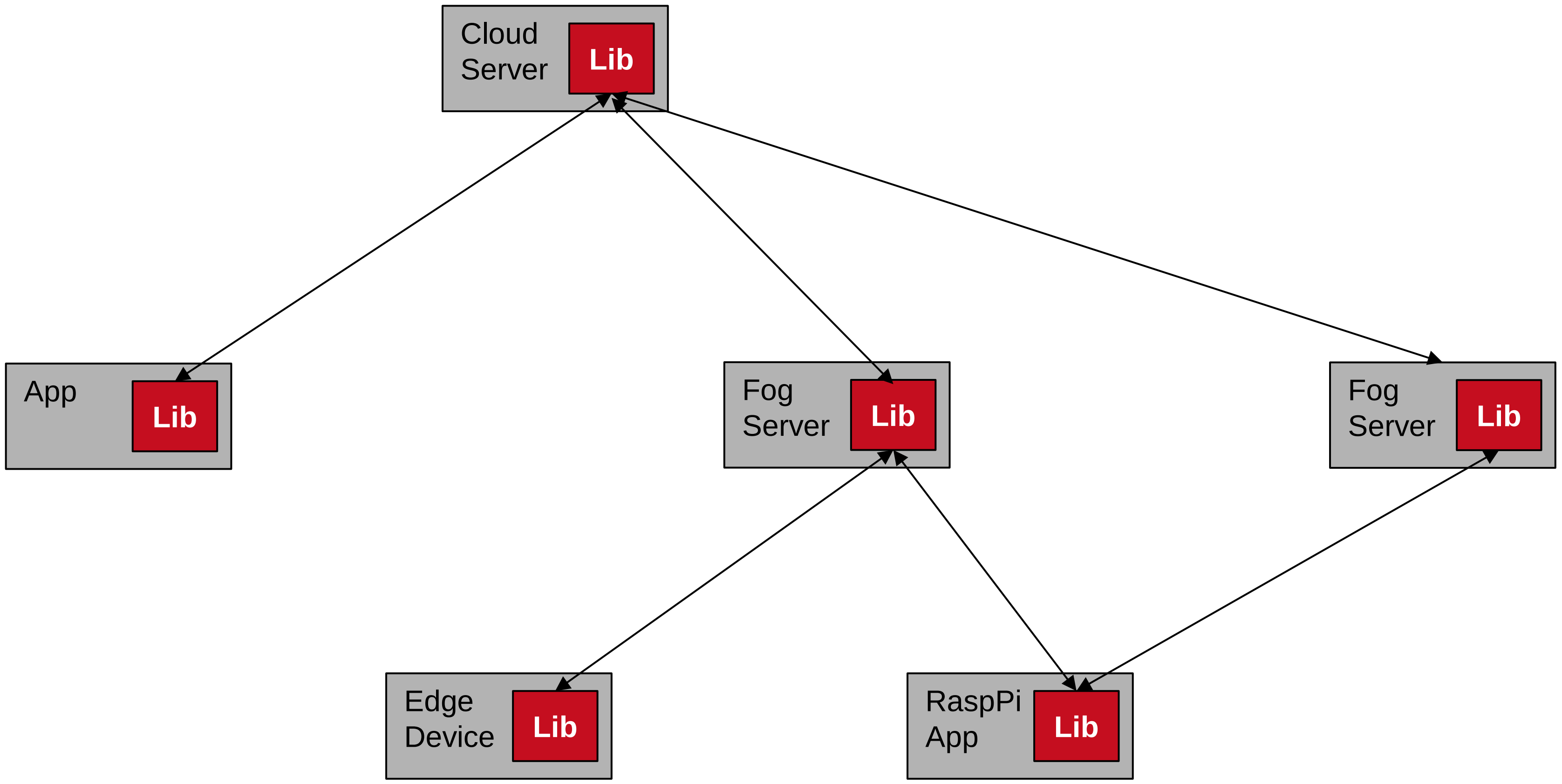
http://zguide.zeromq.org/page:all

Example: ZeroMQ (ØMQ)

- Versatile, asynchronous messaging library
- Supported communication patterns:
  - Synchronous Request-Reply
  - Publish-Subscribe
  - Pipeline
  - Asynchronous Request-Reply
  - …
- Each Pattern has its own Components
- Supports communication:
  - Within a single process
  - Inter-process
  - TCP

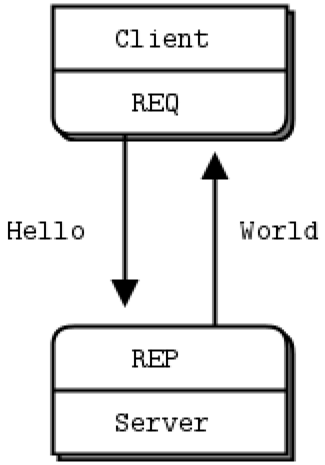
[zeromq.org/]

ZeroMQ: Architecture



ZeroMQ: Synchronous Request-Reply

One requests, one answer

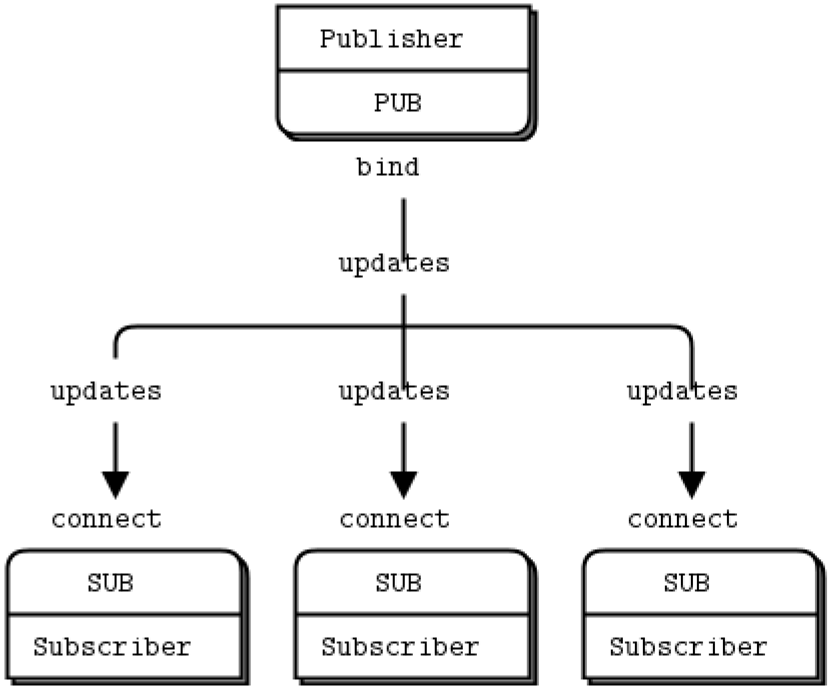


[zeromq.org/]

ZeroMQ: Publish-Subscribe

Server pushes messages to a set of clients

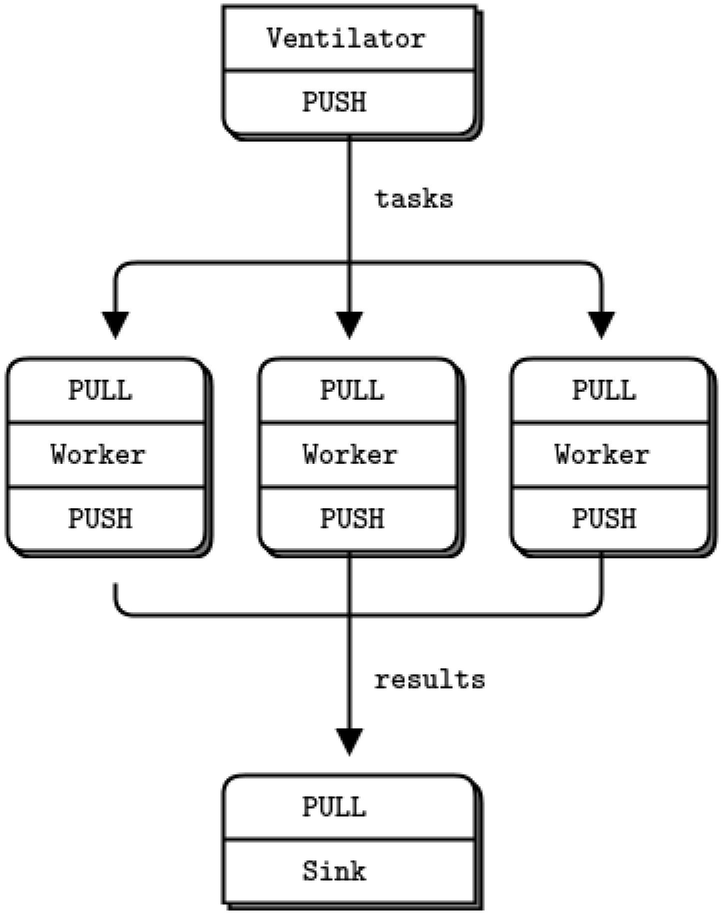
- Clients have to subscribe in advance
- Clients may miss some messages during the connection process



[zeromq.org/]

ZeroMQ: Pipeline

Distribute tasks and collect results

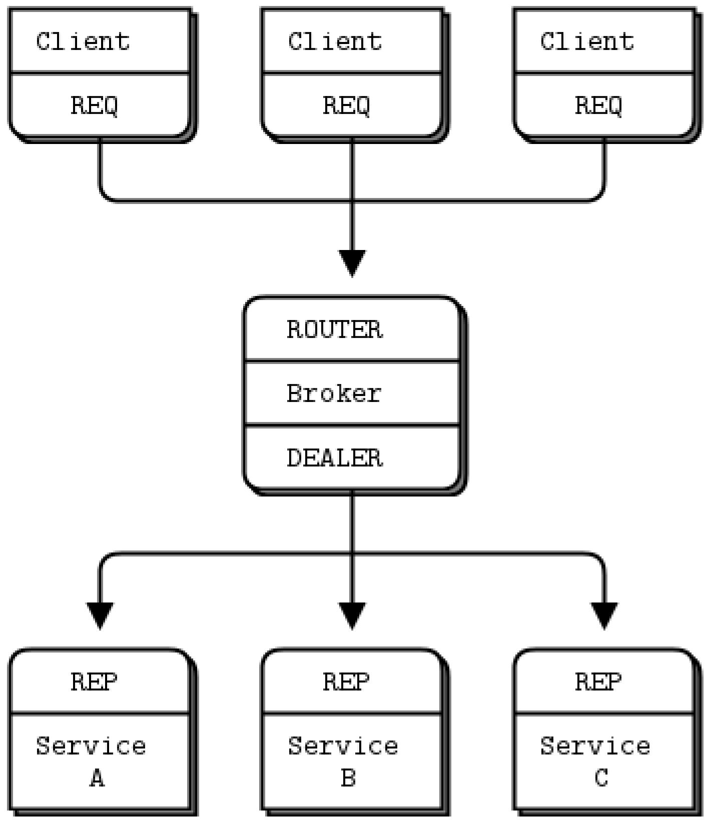


[zeromq.org/]

ZeroMQ: Asynchronous Request - Response

Message Broker connects Clients and Workers

- Responsible for routing & buffering messages



[zeromq.org/]

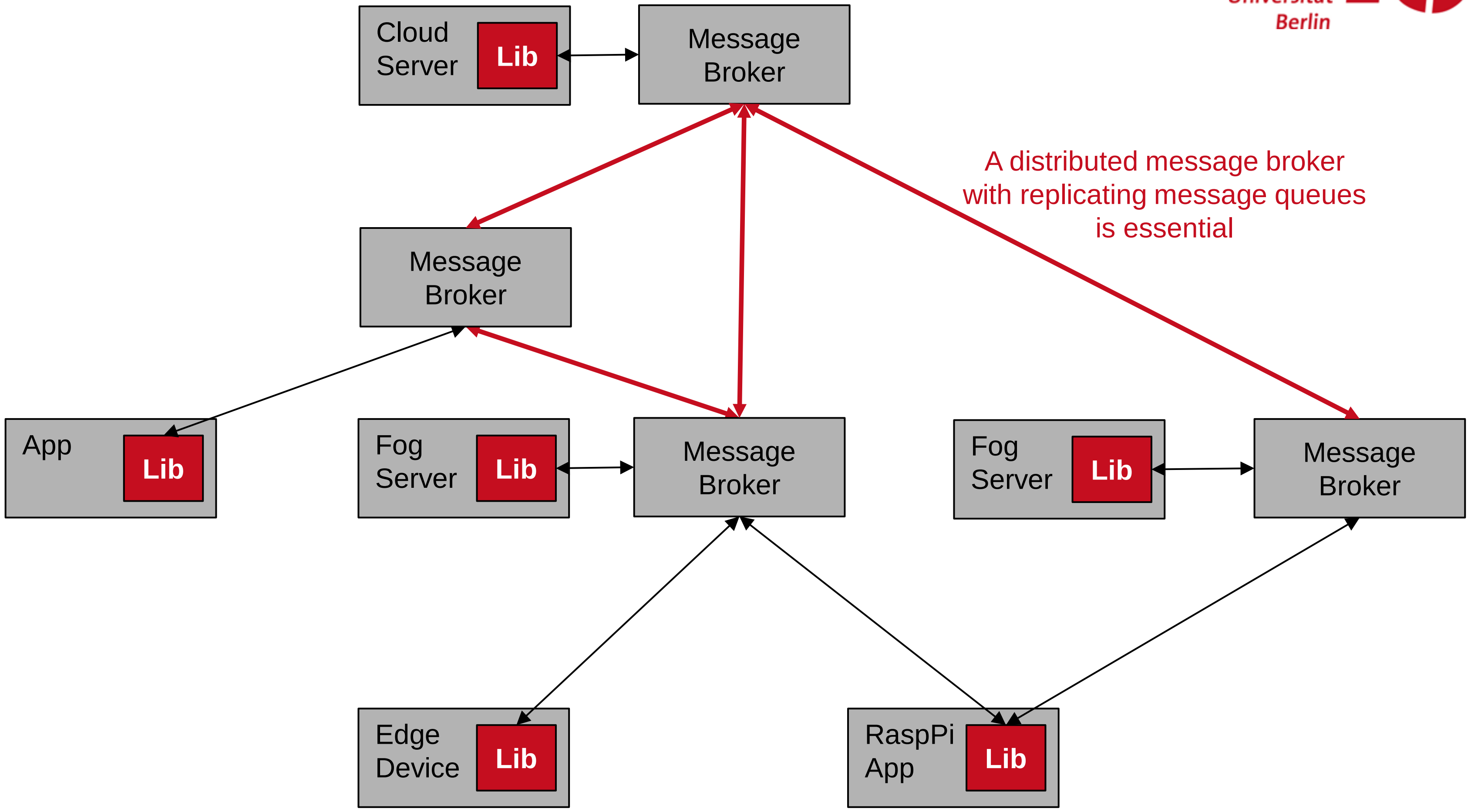
Example: RabbitMQ

- Open Source Message Broker
- Can be deployed in distributed configurations
- Features:
  - Work queues: Distribute tasks among workers
  - Publish/Subscribe: Send messages to many consumers at once
  - Routing: Receiving messages selectively
  - Topics: Receiving messages based on a pattern (topics)
  - RPC: Request-reply pattern

Note: Be careful how far you distribute the broker instances geographically.

[www.rabbitmq.com]

RabbitMQ Architecture



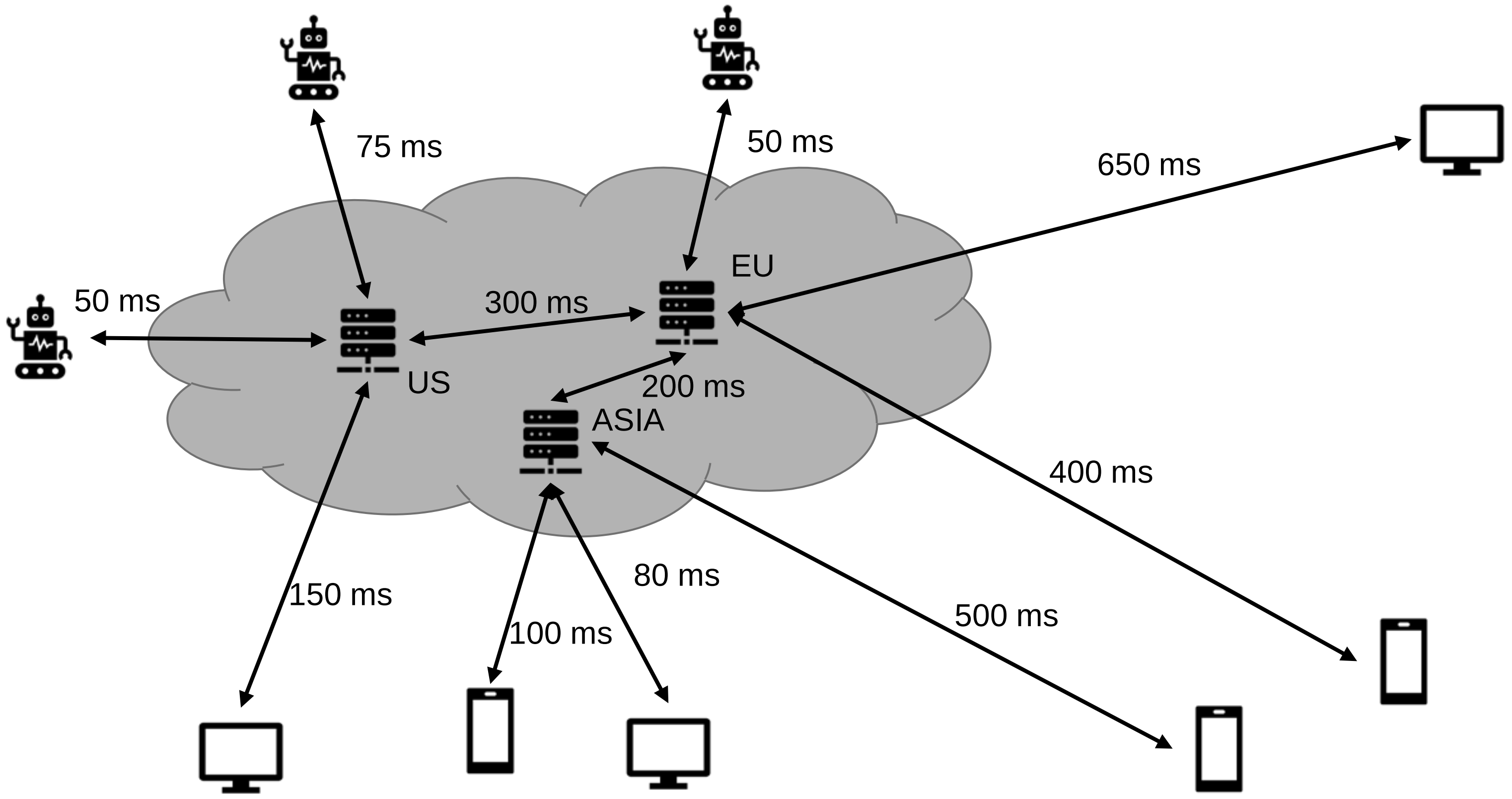
A distributed message broker with replicating message queues is essential

Application Design

GEO-AWARENESS

Geo-Awareness in the Cloud

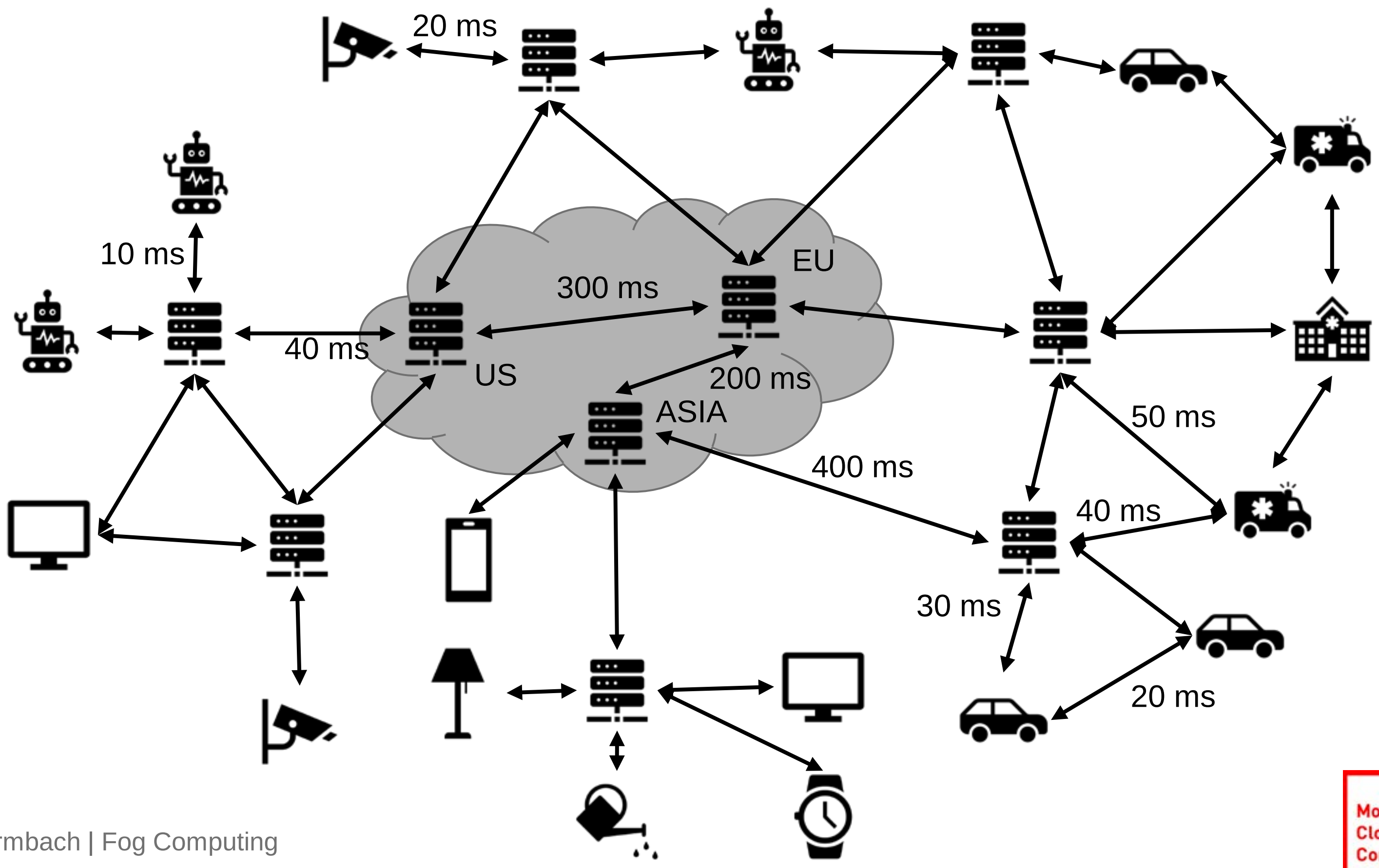
- Limited to large regions (e.g., counties)
- High latency if the closest data center is quite far



80 ms

Introducing Fog Nodes

- Fast connection to nearby fog nodes but limited bandwidth to cloud
- Access point(s) of mobile devices must be adapted based on its location



Geo-Awareness

Infrastructure needs to expose location and network topology explicitly (beyond regions)

An application

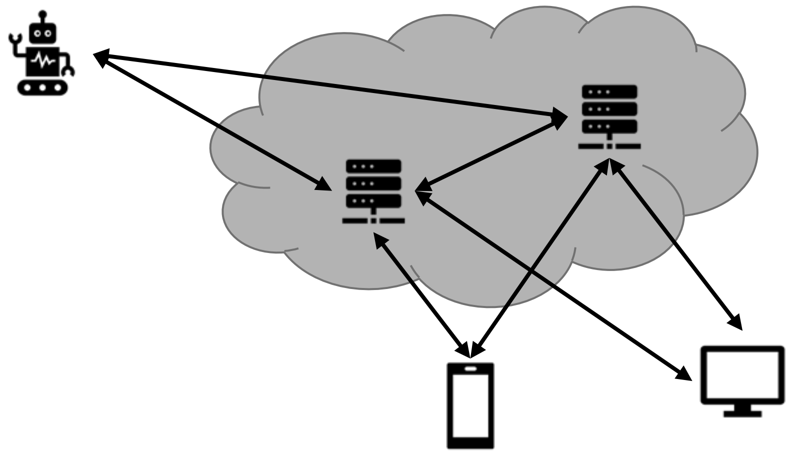
- Must to be aware of ist deployment location
- Needs to handle client movement (handover to other edge device)
- Must be prepared to move components elsewhere (=> stateless application logic)
- Must move data when necessary
- May not rely on the availability of remote components

Application Design

FAULT-TOLERANCE

Fault-Tolerance in Cloud Applications

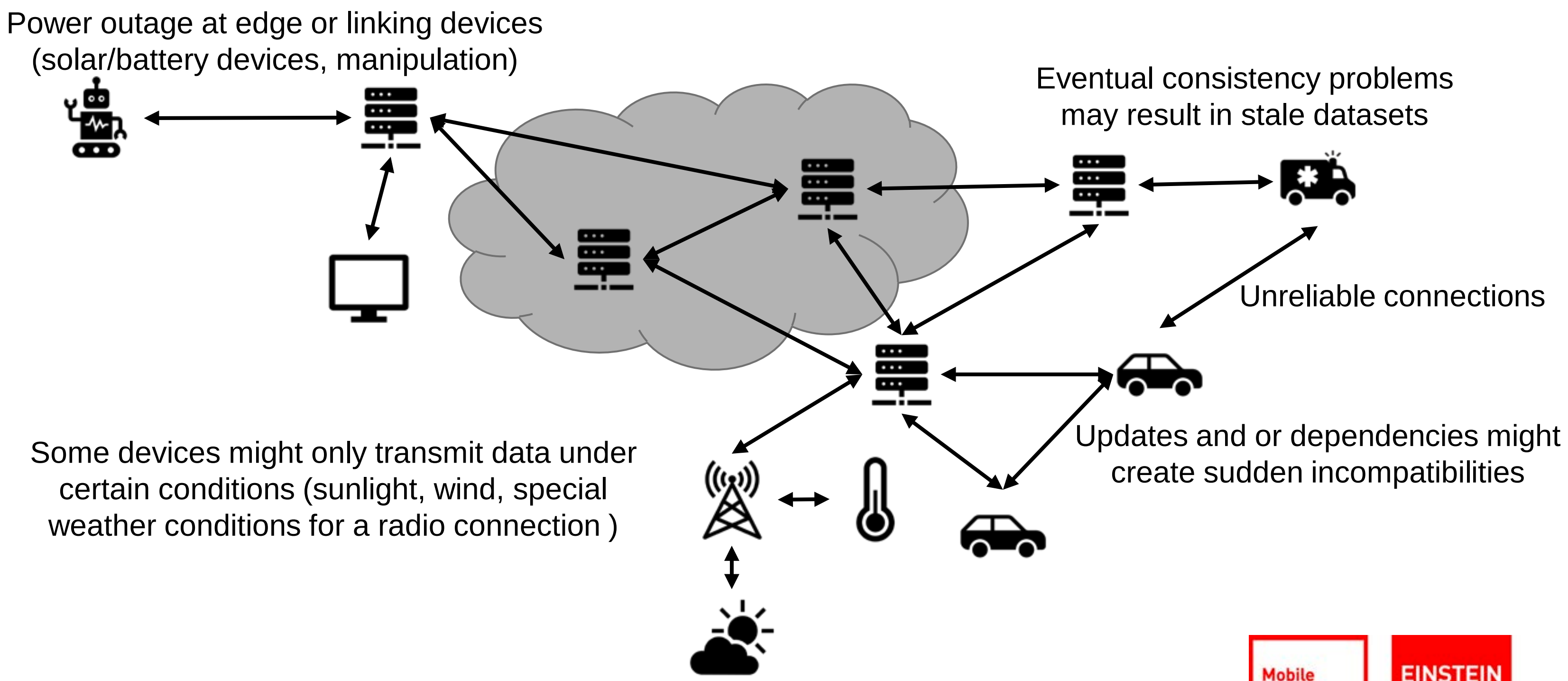
- Redundant Servers
- Retry-on-error principle (with other service instance)
- Monitoring services and its workload, auto-scaling
- Chaos-Monkey which randomly shuts down services to check if the system adapts and catches the outage



Fault-Tolerance in Fog Applications

Prevalence of faults depends on the number of nodes:

- Systems and/or its components fail continuously
- Connecting infrastructure fails or operates with reduced quality


Power outage at edge or linking devices (solar/battery devices, manipulation)
Eventual consistency problems may result in stale datasets
Unreliable connections
Updates and or dependencies might create sudden incompatibilities
Some devices might only transmit data under certain conditions (sunlight, wind, special weather conditions for a radio connection)

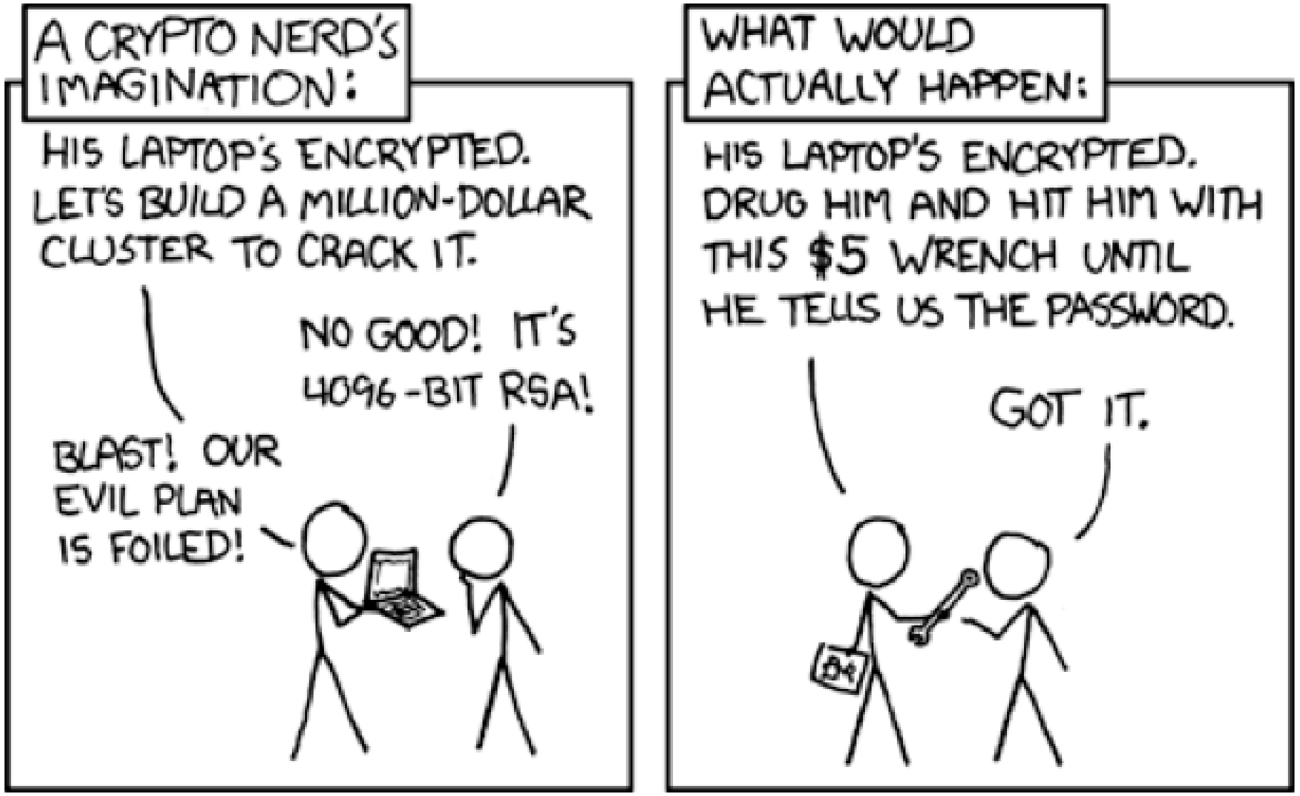
Fault-Tolerance in Fog Applications

- Buffer messages until its receiver is available again
- Expect data staleness and ordering issues
- Cache data aggressively
- Compress data items as much as possible on unreliable connections (small time frames must suffice to transmit the entire message)
- Plan with incompatibility, constantly monitor software versions on devices
- …

Application Design

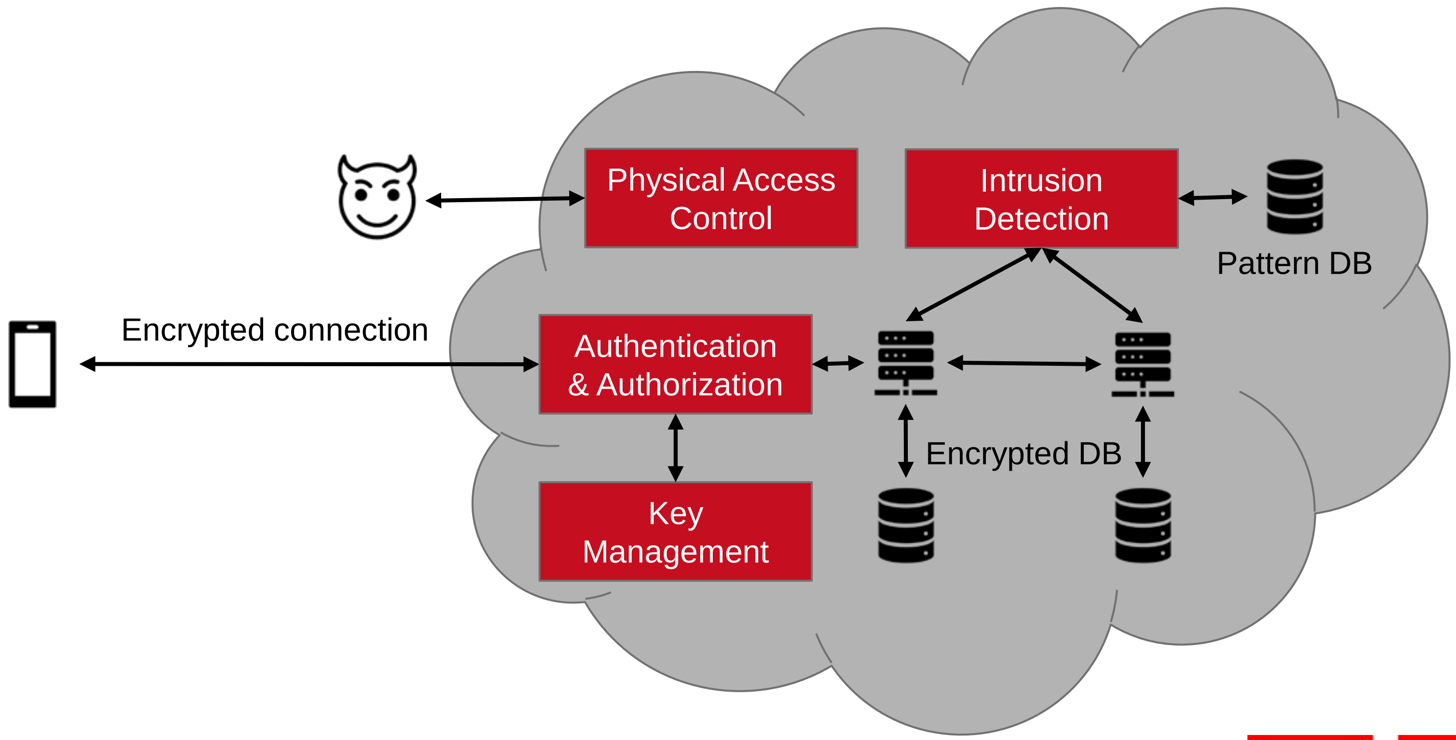
SECURITY

Security by Obscurity


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

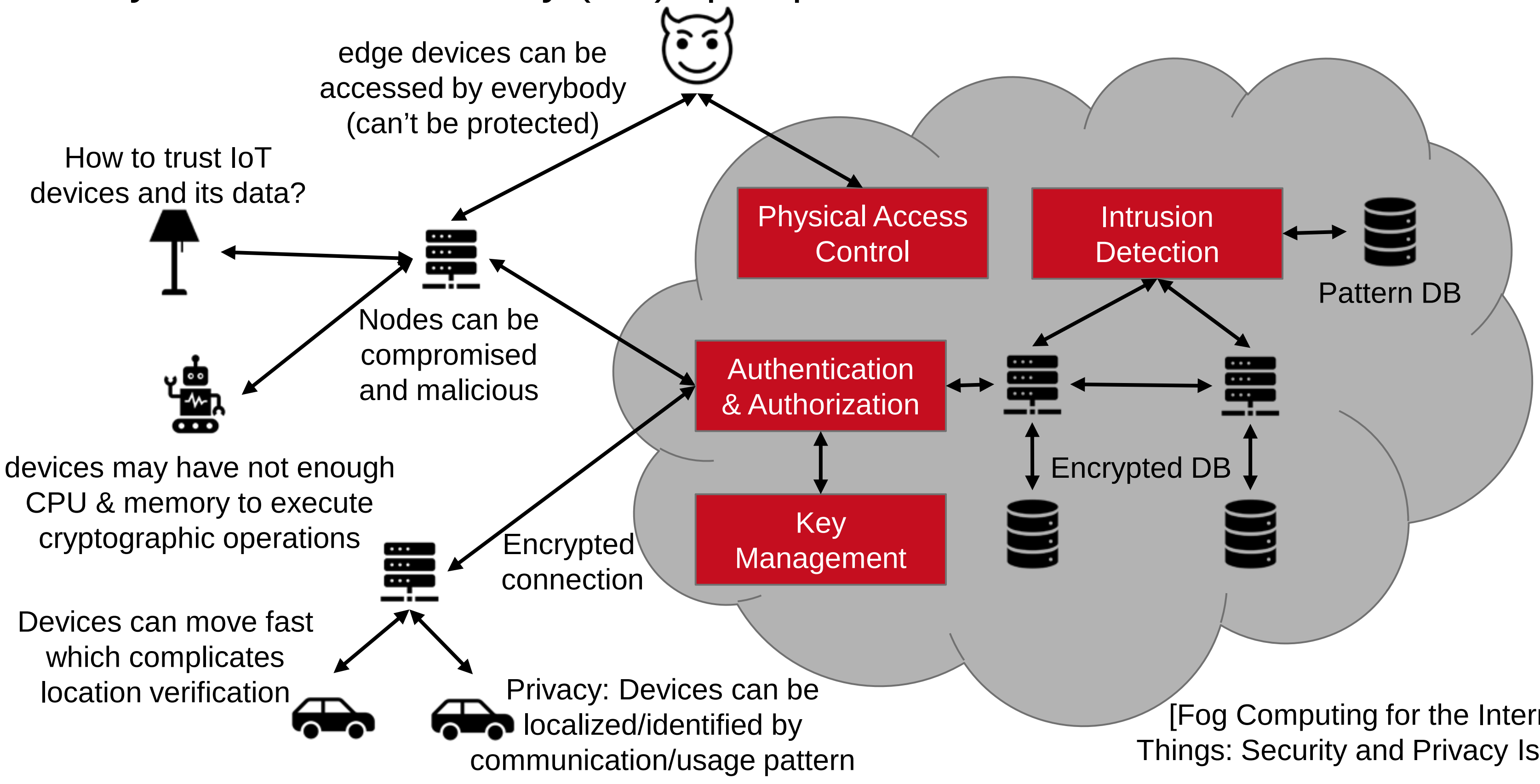
Cloud Security

- Many established algorithms, systems, designs ensure security
- Applied correctly, cloud applications are as secure as it gets



Fog Security?

Many vulnerabilities, many (still) open problems



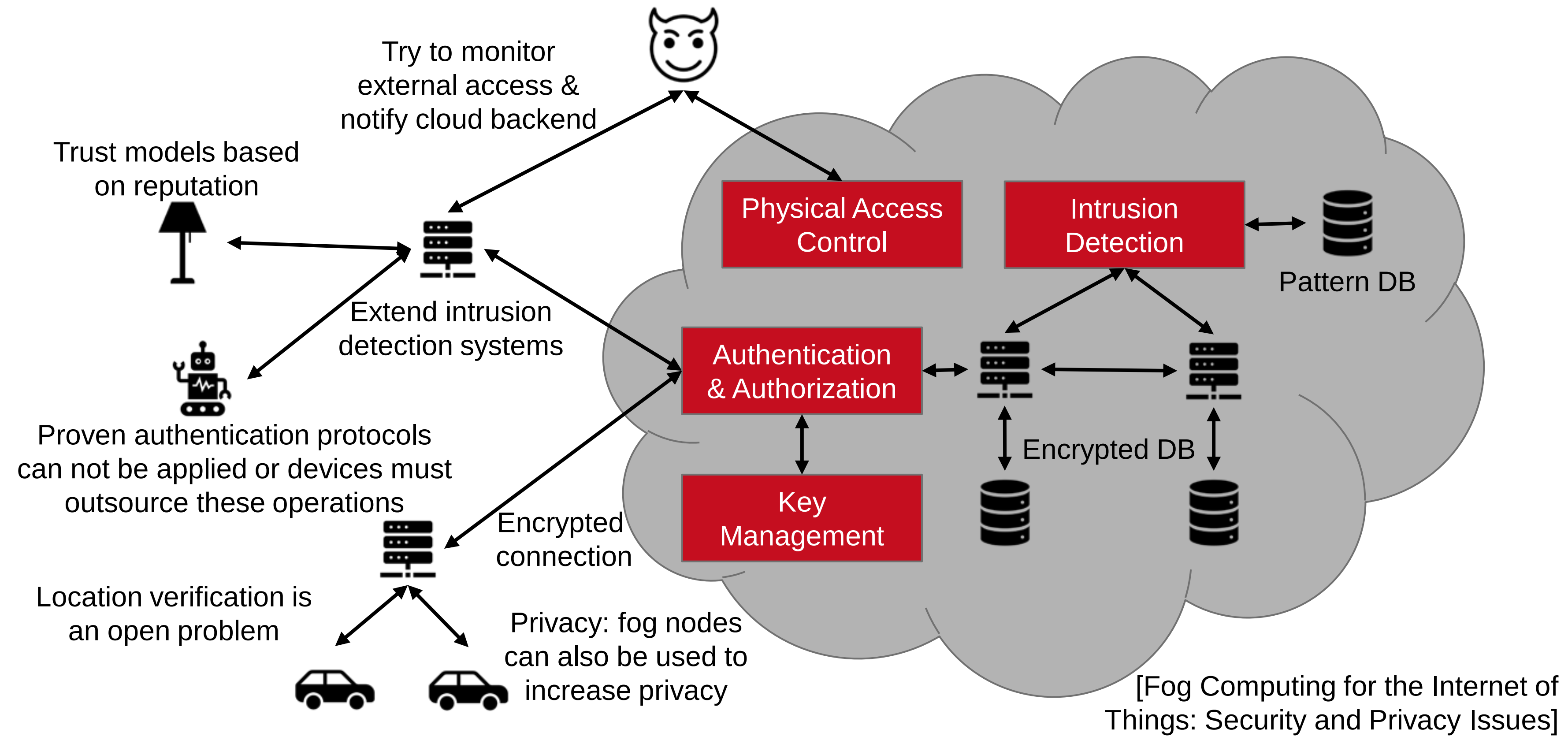
edge devices can be accessed by everybody (can't be protected)
How to trust IoT devices and its data?
Nodes can be compromised and malicious
devices may have not enough CPU & memory to execute cryptographic operations
Encrypted connection
Devices can move fast which complicates location verification
Privacy: Devices can be localized/identified by communication/usage pattern
Physical Access Control
Intrusion Detection
Pattern DB
Authentication & Authorization
Key Management
Encrypted DB

[Fog Computing for the Internet of Things: Security and Privacy Issues]

Also: Fog is typically a cooperative effort

Fog Security

- Upcoming solutions focus on early attack detection
- Accepting & handling weaknesses rather than preventing/eliminating them


Try to monitor external access & notify cloud backend
Trust models based on reputation
Physical Access Control
Intrusion Detection
Pattern DB
Extend intrusion detection systems
Authentication & Authorization
Encrypted DB
Proven authentication protocols can not be applied or devices must outsource these operations
Key Management
Encrypted connection
Location verification is an open problem
Privacy: fog nodes can also be used to increase privacy

[Fog Computing for the Internet of Things: Security and Privacy Issues]

Cloud vs. Fog Application Engineering

Application Design

- Architecture
- Communication
- Geo-awareness
- Fault-Tolerance
- Security

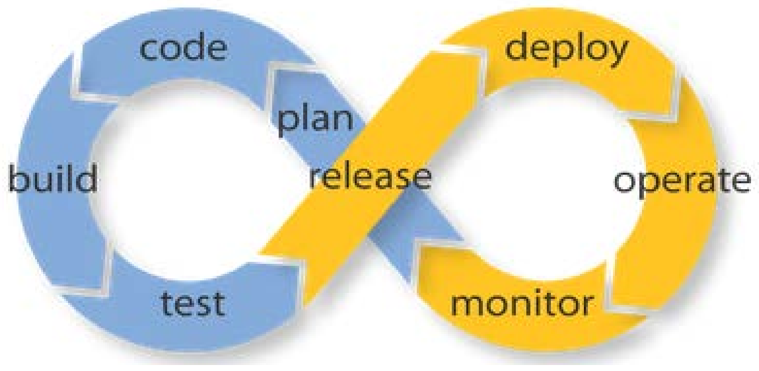
Development Process

- Testing
- Deployment

WORKED FINE IN DEV
OPS PROBLEM NOW
quickmeme.com

Dev and Ops

Dev: Agility, many versions and features



DevOps: Many stable releases

Ops: Stability, few releases

DevOps in a nutshell

DevOps most generally is about software and business engineering

…emphasizing short, iterative planning and development cycles

- supporting changing requirements as projects evolve
- enabling fast and frequent delivery of high-quality functionality

…improving communication and collaboration between different teams

- focusing on small, highly decoupled tasks
- Communicating via language-agnostic APIs

overall establishing development practices that leverage frequent code commits, automated verification and builds, and early problem detection

Development Process

TESTING

Testing

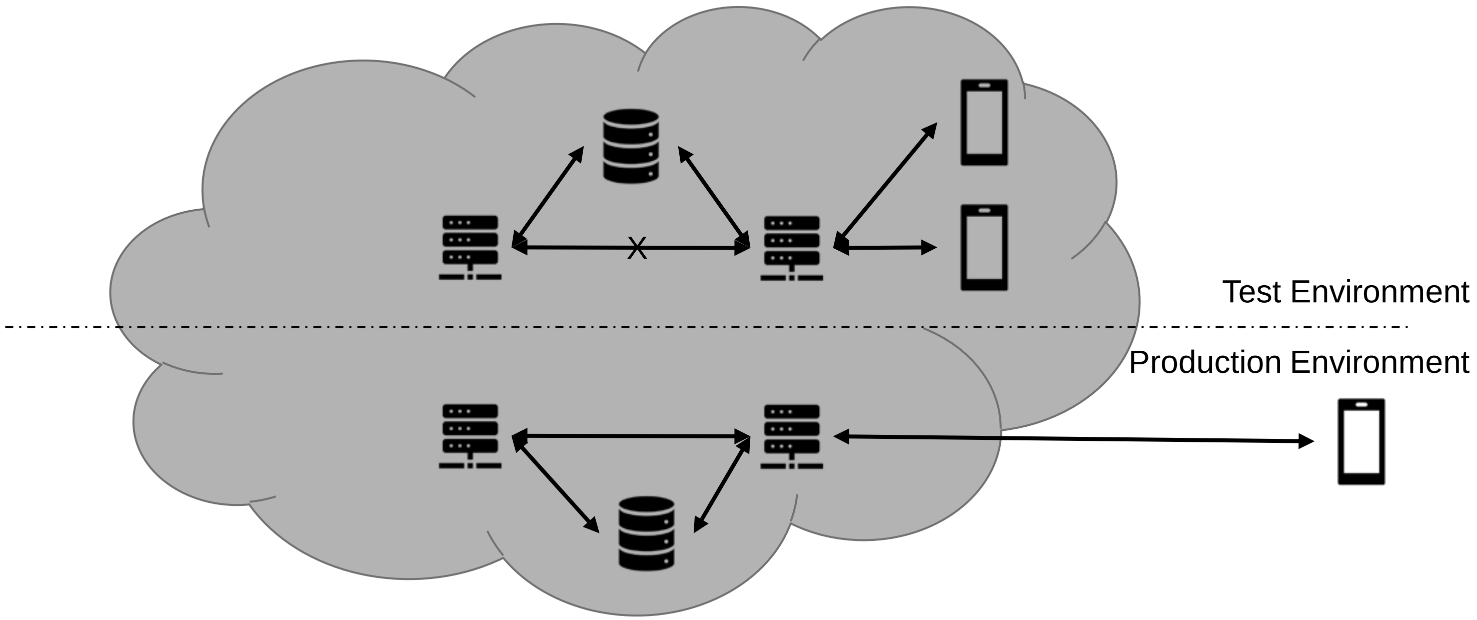
Important part of software engineering process: Make sure that the system works as intended

Phases:

- (Unit Testing)
- Integration & Live Testing
  - Canary Testing
  - Dark Launches
  - A/B Testing
  - …

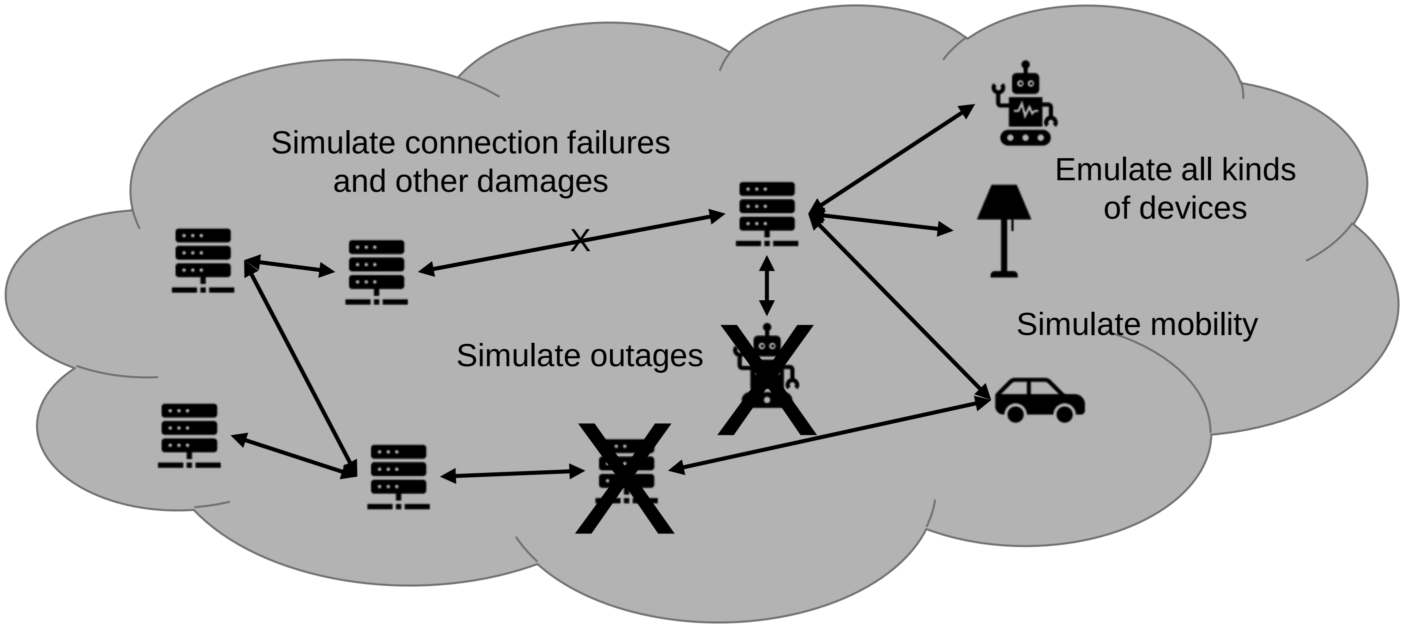
Cloud Integration Tests

- Setup (virtual) testbed and check whatever is required
- Mock services, data, devices
- Evaluate corner-cases which usually should not exist in production



Fog Integration Tests

- Testbed creation is difficult because physical infrastructure and devices cannot be duplicated
- (Partial) Solution: virtualize & emulate fog environment in the cloud


Simulate connection failures and other damages
Emulate all kinds of devices
Simulate mobility
Simulate outages

[Hasenburg et al., MockFog, 2018]

Live Testing

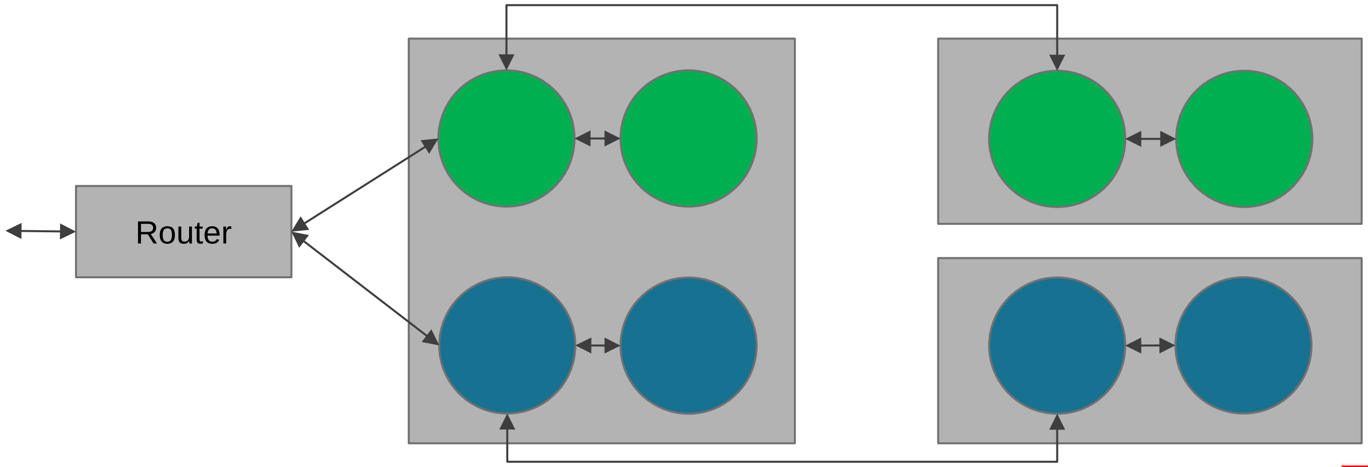
If possible, live testing techniques test new software versions in production

Basic principle: Monitor what happens…
…while rolling out an update gradually
…while directing part of the traffic to old and/or new version

Example: Blue-Green Deployments

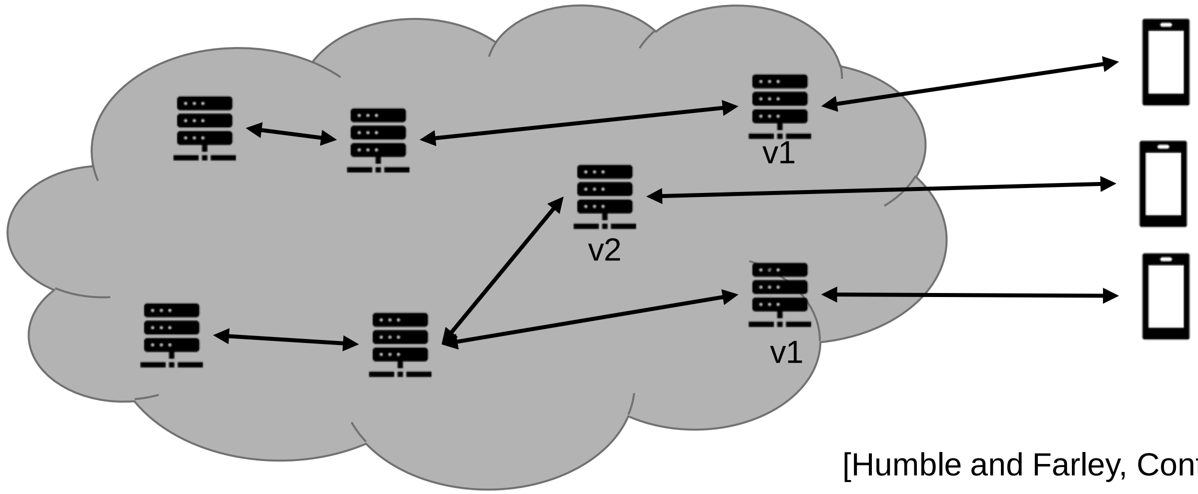
There are two identical versions of the production environment, the blue and the green one

1. Deploy new version to blue environment
2. Smoke tests, etc., against the blue system
3. Switch the router configuration to the blue system
4. The blue environment becomes the production system
   (or switch back if there are errors)



Example: Canary Releasing

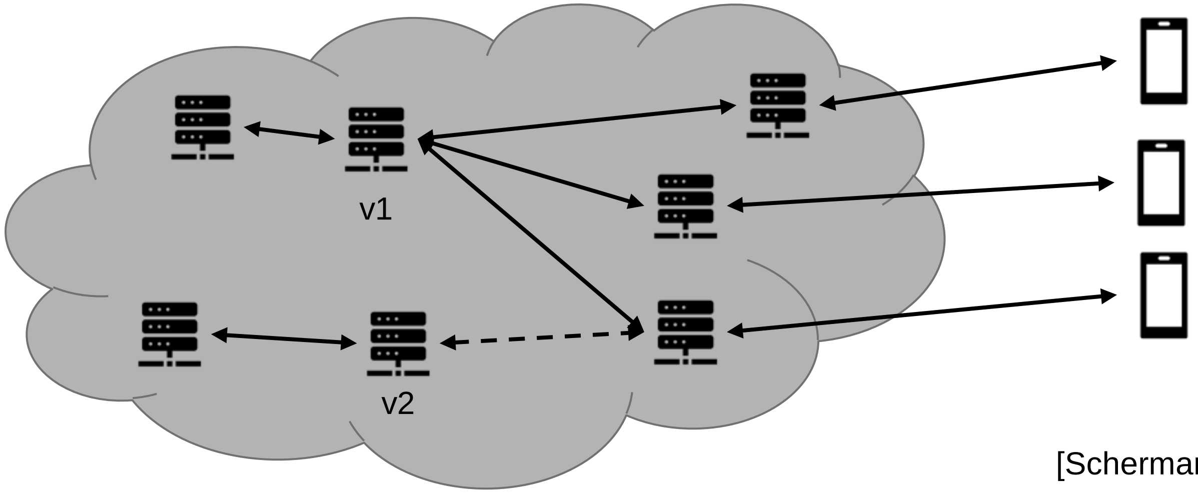
- Rollout of a new version only to a subset of production servers
- Any problem with this new version will not affect the majority of users
- Benefits:
  - Easy to revert: stop routing users to canary servers
  - Use it for A/B-Testing: Decide on user-defined criteria whether to keep the old or new version (e.g., revenue, quality of results, …)
  - Check capacity requirements by incrementally increasing the load



[Humble and Farley, Continuous Delivery, 2011]

Example: Dark/Shadow Launches

- Functionality is deployed in a production environment without being visible or activated for the users
- Some or all production traffic is duplicated and routed to the shadow versions as well
- Observing the shadow version without impacting any user



[Schermann et al., Bifrost, 2016]

Fog Live Testing

Can continue to be used in the cloud part or on intermediary nodes (>= single rack)

Difficult on edge devices which

- may not have the capacity to run two versions in parallel
- may have safety requirements which make canary releases impossible

⇒ Find a separate solution for the edge part

- Mock edge devices in the cloud
- Have a dedicated (physical!) testbed
- …

Development Process

DEPLOYMENT

Cloud Deployment: Infrastructure as Code (IaC)

“Infrastructure as code is an approach to using cloud era technologies to build and manage dynamic infrastructure. It treats the infrastructure, and the tools and services that manage the infrastructure itself, as a software system, adapting software engineering practices to manage changes to the system in a structured, safe way.“

- [Kief Morris, Infrastructure as Code]

Principles of IaC

- Reproducibility: Effortlessly and reliably rebuild any element of an infrastructure
- Consistency: Two infrastructure elements providing a similar service should be nearly identical (some configuration parameters or IP address may vary)
- Repeatability: Any action you carry out on your infrastructure should be repeatable
- Disposability: Assume that any infrastructure element can and will be destroyed at any time
- Service continuity: Make sure that a service is always able to handle requests, in spite of what might be happening to the infrastructure
- Self-testing systems: Effective automated testing
- Self-documenting systems: Minimal documentation outside of scripts
- Small changes: Incremental changes
- Version all the things: Versioning of infrastructure configurations

Challenges in IaC

- Configuration drift:
  - Configurations are consistent when they're created, but become outdated/incompatible over time
- Snowflake servers
  - Special servers which can't be touched or reproduced
- Jenga infrastructure
  - Infrastructure which can be easily disrupted and which is hard to fix
  - There is the possibly apocryphal story of the data center with a server that nobody had the login details for, and nobody was certain what the server did. Someone took the bull by the horns and unplugged the server from the network. The network failed completely, the cable was re-plugged, and nobody touched the server again." [Kief Morris, Infrastructure as Code]
- Automation fear
  - Automation tools are not running in an automatic schedule
- Erosion
  - Even without new requirements, the infrastructure will decay over time
  - System upgrades, disk filling with files, crashing processes, …

Solutions I

- Do not run routine tasks manually
  - Manual setup, update, etc., result in configuration drift and snowflake servers
- Do not run scripts manually
  - Teams create scripts for routine work but run them by hand
  - Babysitted scripts suggest that there are not enough controls to make the script save to run
  - Better: Include error detection and handling in script; run script by (event) trigger or on a schedule
- Run automation frequently
  - Infrequently runs lead to automation fear, configuration drift and snowflake servers

Solutions II

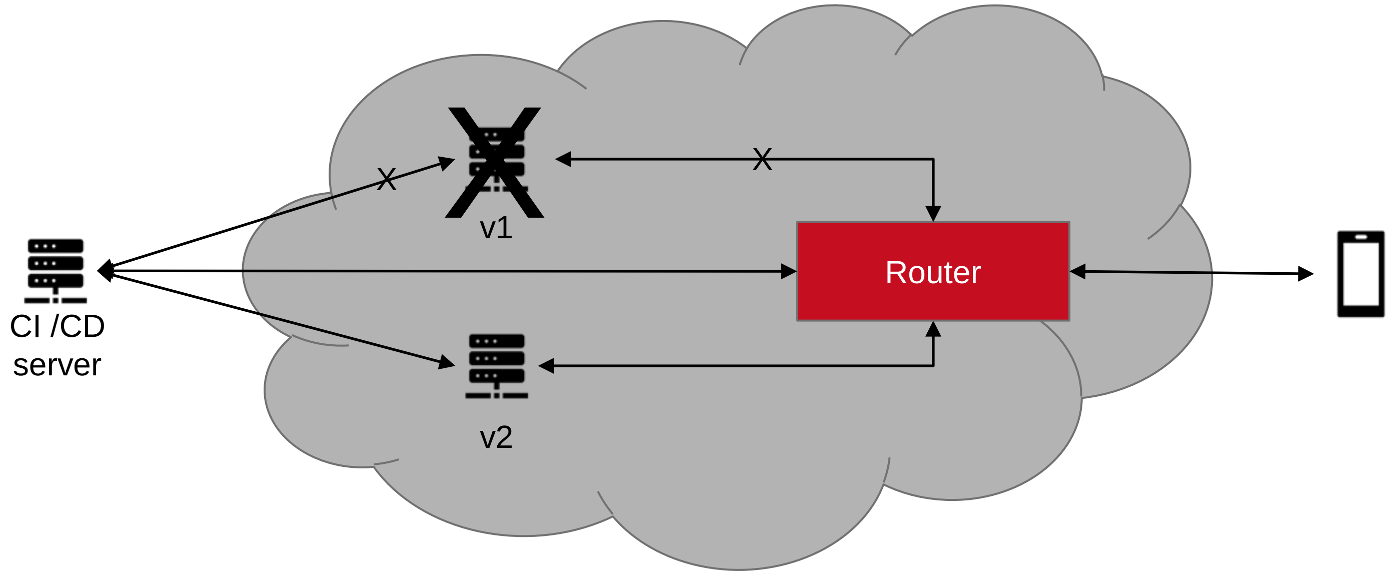
- Never bypass automation
  - These changes will cause later automation fails or may revert important fixes

- Do not apply configuration changes directly to important systems
  - “Editing production server configuration by hand is like playing Russian Roulette with your business. Running untested configuration scripts against a groups of servers is like playing Russian Roulette with a machine gun.”

- [Kief Morris, Infrastructure as Code]

Deploying Cloud Applications

- Changes are pushed to devices/instances through IaC
- New virtual devices are created, configured , and deployed with the new version; old instances are disconnected terminated



Deploying Fog Applications

For fog applications this only works for the cloud end (and possibly for smaller cloud-like data centers).

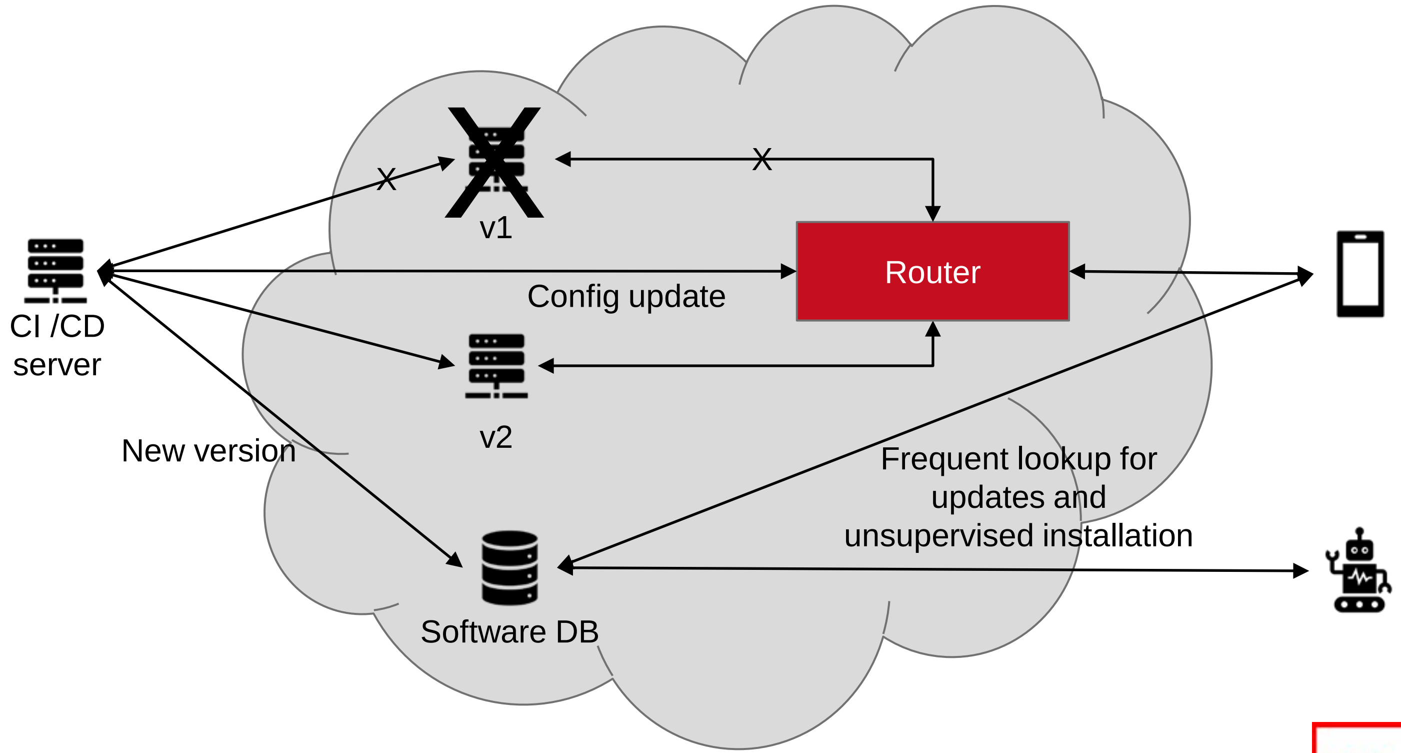
Edge devices often need to be physically connected once to deploy the first version.

One strategy (there might be others): Use an app store-like approach

- Update is sent to a central software repository
- Deployed application frequently checks for updates and self-updates if necessary => pull approach

Deploying Fog Applications

- Plan with incompatibilities and different versions on devices
- Use versioned interfaces!



Summary

Fog applications share some similarities with cloud applications.

Some changes are needed in Application Design regarding

- Architecture
- Communication
- Geo-awareness
- Fault-Tolerance
- Security

and in the Development Process regarding

- Testing
- Deployment.

Questions?