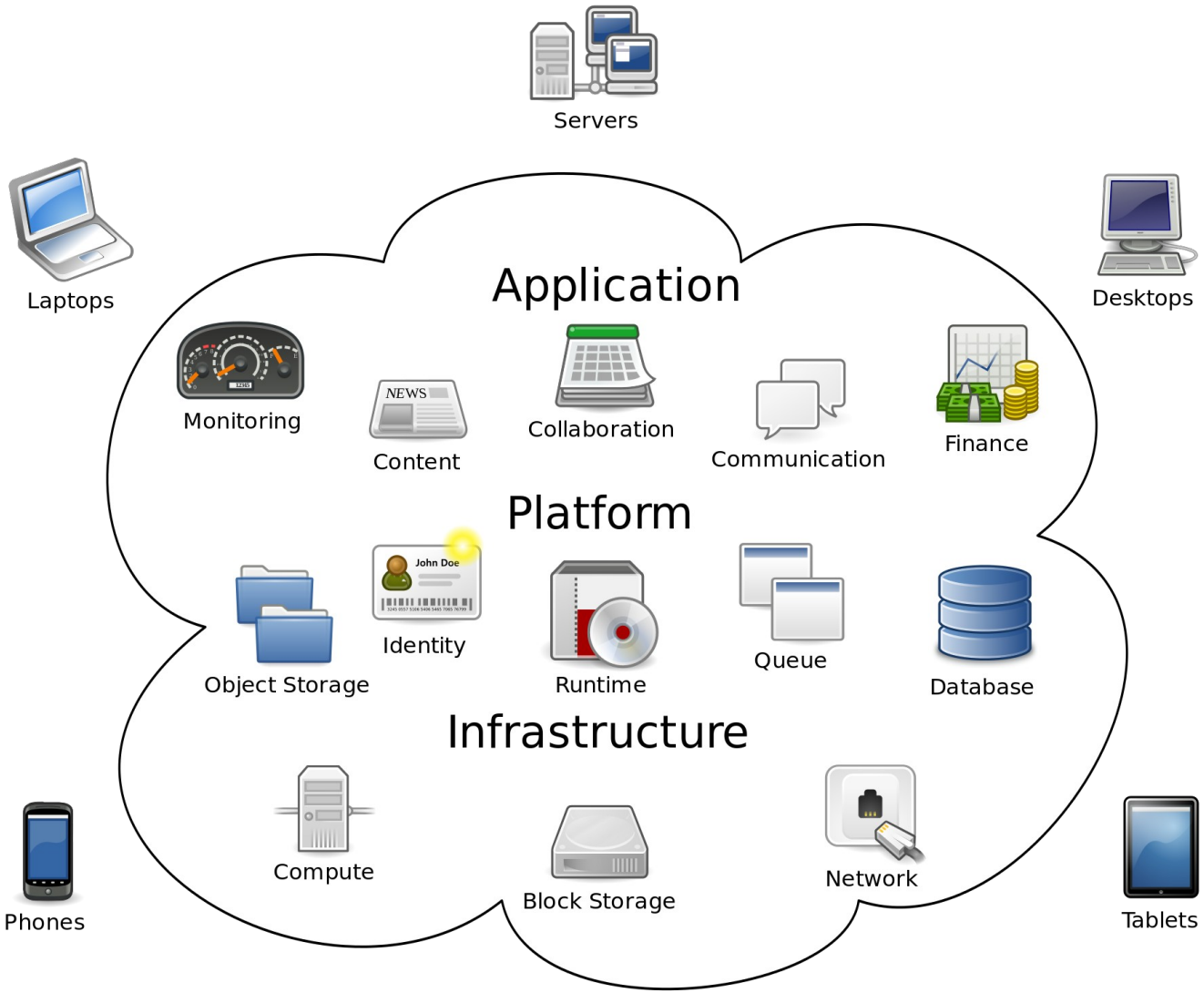
Enterprise Architecture

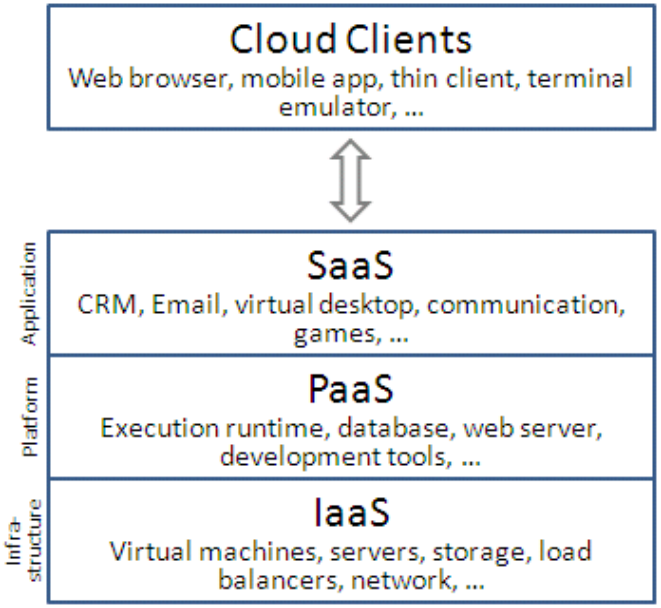
Cloud Native Apps

Deploying to cloud
What do you know about clouds?

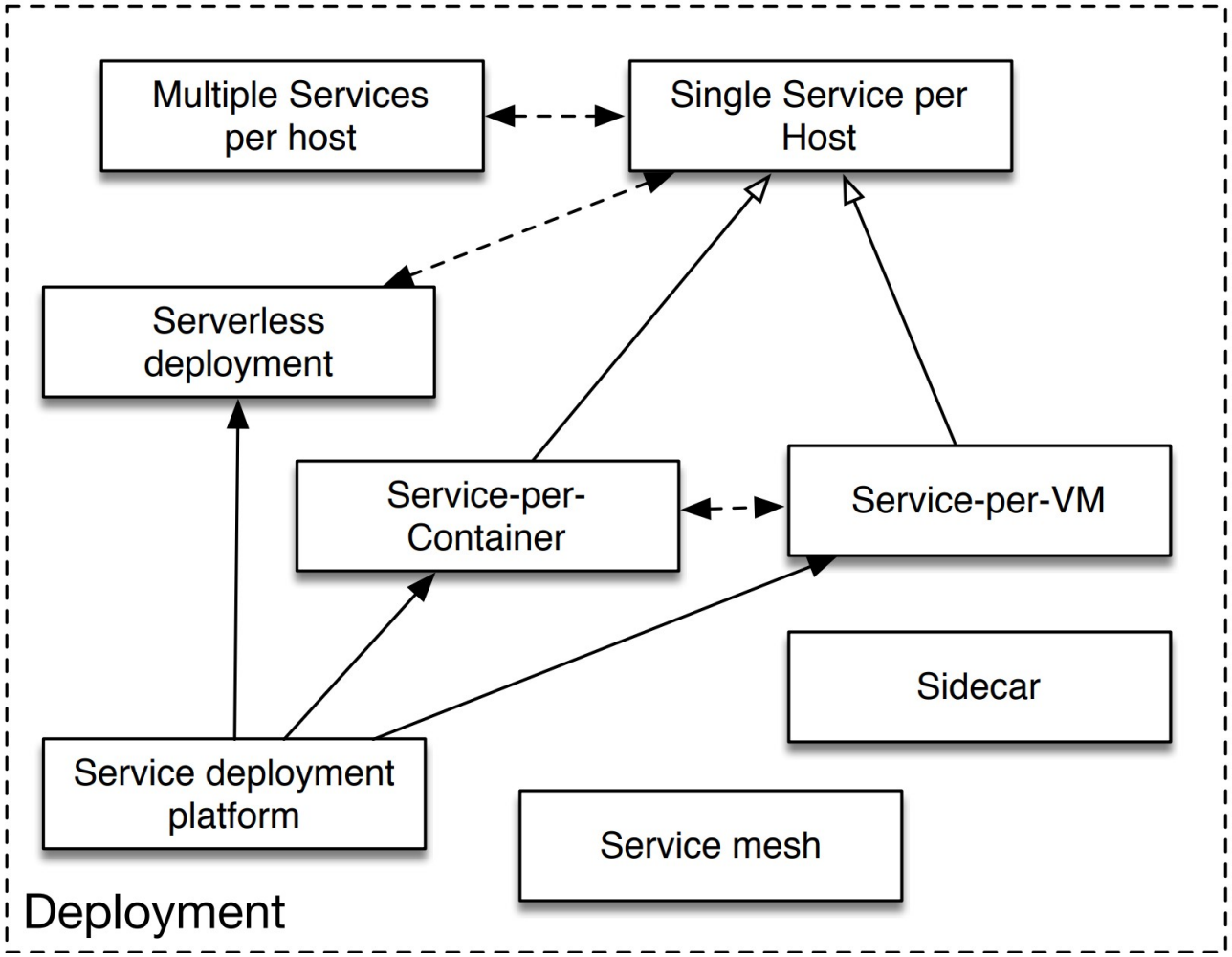


Cloud computing

Service types



Infrastructure patterns



Highly recommend to check it → https://microservices.io/patterns/index.html

Migrating to the cloud ?

- Lift and shift vs cloud Native
  - https://akfpartners.com/growth-blog/migrating-to-the-cloud-lift-and-shift-versus-cloud-native

Serverless pattern

- Let’s watch: Serverless Architecture Explained
  - https://www.youtube.com/watch?v=RzsaM6kL1FU

You decided on containers,
How to orchestrate (~1000s) of them ?

Assuming service-per-container approach

Container cluster orchestration

- Docker swarm mode
- Kubernetes (k8s)
- DC/OS
  - Mesos (Marathon)

Example deployments

- Got time ? , Watch:
  - Microservices + Events + Docker = A Perfect Trio
    - https://www.youtube.com/watch?v=sSm2dRarhPo
  - Introduction to Microservices, Docker, and Kubernetes
    - https://www.youtube.com/watch?v=1xo-0gCVhTU

What’s a cloud native app again?

Traditional 12 factors

- Created by Heroku engineers to share their experience with the cloud apps
- Let’s check them out
  - https://12factor.net/

Beyond the 12 factors

- Beyond the Twelve-Factor App | Kevin Hoffman
- Upcoming guidelines are from :
  - https://jimmysong.io/posts/high-level-cloud-native-from-kevin-hoffman/

1. One codebase, one App

- Single version-controlled codebase, many deploys
- Multiple apps should not share code
  - Microservices need separate release schedules
  - Upgrade, deploy one without impacting others
- Tie build and deploy pipelines to single codebase

2. API first

- Service ecosystem requires a contract
  - Public API
- Multiple teams on different schedulers
  - Code to contract/API, not code dependencies
- Use well-documented contract standards
  - Protobuf IDL, Swagger, Apiary, etc
- API First != REST first
  - RPC can be more appropriate in some situations

3. Dependency Management

- Explicitly declare dependencies
- Include all dependencies with app release
- Create immutable build artifact (e.g. docker image)
- Rely on smallest docker image
  - Base on scratch if possible
- App cannot rely on host for system tools or libraries

4. Design, Build, Release, Run

- Design part of iterative cycle
  - Agile doesn't mean random or undesigned
- Mature CI/CD pipeline and teams
  - Design to production in days not months
- Build immutable artifacts
- Release automatically deploys to environment
  - Environments contains config, not release artifact

5. Configuration, Credentials, Code

- “3 Cs” volatile substances that explode when combinded
- Password in a config file is as bad as password in code
- App must accept “3 Cs” from **environment** and only use harmless defaults
- Test - Could you expose code on Github and not reveal passwords, URLs, credentials?

6. Logs

- Emit formatted logs to stdout
- Code should not know about destination or purpose of log emissions
- Use downstream log aggregator
  - collect, store, process, expose logs
  - ELK, Splunk, Sumo, etc
- Use **structured** logs to allow query and analysis
  - JSON, csv, KV, etc
- Logs are not metrics

7. Disposability

- App must start as quickly as possible
- App must stop quickly and gracefully
- Processes start and stop all the time in the cloud
- Every scale up/down disposes of processes
- Slow dispose == slow scale
- Slow dispose or startup can cause availability gaps

8. Backing Services

- Assume all resources supplied by backingservices
- Cannotassume mutable file system
  - “Disk as a Service” (e.g. S3, virtual mounts, etc)
- Every backing service is bound resource
  - URL, credentials, etc-> environment config
- Host does not satisfy NFRs Non functional requirements
  - Backing services and cloud infrastructure

9. Environment Parity

- “Works on my machine”
  - Cloud-native anti-pattern. Must **work everywhere**
- Every commit is candidate for deployment
- Automated acceptance tests
  - Provide no confidence if environments don’t match

10. Administrative Processes

- Database migrations
- Run-once scripts or jobs
- Avoid using for batch operations, consider instead:
  - Event sourcing
  - Schedulers
  - Triggers from queues, etc
  - Lambdas/functions

11. Port Binding

- In cloud, infrastructure determines port
- App must accept port assigned by platform
- Containers have internal/external ports
  - App design must embrace this
- Never use reserved ports
- Beware of container “host mode” networking

12. Stateless Processes

- What is stateless?
- Long-term state handled by a backing service
- In-memory state lives onlyas long as request
- Requests from same client routed to different instances
  - “Sticky sessions” cloud native anti-pattern

13. Concurency

- Scale horizontally using the process model
- Build disposable, stateless, share-nothing processes
- Avoid adding CPU/RAM to increase scale/throughput
- Where possible, let platform/libraries do threading
    - Many single-threaded services > 1 multi-threaded monolith

14. Telemetry

- Monitor apps in the cloud like satellite in orbit
- No tether, no live debugger
- Application Perf Monitoring (APM)
- Domain Telemetry
- Health and system logs

15. Authentication & Authorization

- Security should never be an afterthought
- Auth should be explicit, documented decision
  - Even if anonymous access is allowed
  - Don’t allow anonymous access
- Bearer tokens/OAuth/OIDC best practices
- Audit all attempts to access

OK, let’s see that in real life

Mastering Chaos

- Mastering Chaos - A Netflix Guide to Microservices | Josh Evan
  - https://www.youtube.com/watch?v=CZ3wIuvmHeM