Cloud native applications, infrastructure and patterns

Part 1: Origins and main characteristics

Renaud Lachaize & Thomas Ropars

Univ. Grenoble Alpes

M2 GI

January 2024

Main references

- B. Scholl, T. Swanson, P. Jausovec. **Cloud native: Using containers, functions, and data to build next-generation applications**. O'Reilly, 2019.

- J. Garrisson, K. Nova. **Cloud-native infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment.** O'Reilly, 2017.

- Cloud Native Computing Foundation (CNCF) Web site: https://www.cncf.io

- Google Site Reliability Engineering (SRE) resources:
  - https://sre.google
  - Free eBooks: https://sre.google/books/

Introduction

- **During the first era of Cloud computing, most efforts were focused on facilitating the migration of existing applications** (and their legacy code bases) to Cloud platforms.
  - "Lift and shift" to IaaS (Infrastructure as a Service)
  - Migration of domain-specific applications (e.g., Web applications) to PaaS (Platform as a Service)

- Over the past ~decade, a number of principles have emerged for shaping the design of cloud-based applications and their underlying infrastructure.
- These **new applications have been designed from the ground up in order to take into account the specific characteristics (challenges and opportunities) of modern Cloud platforms and technologies**.

“Cloud native”: a definition

“**Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds.** Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with **robust automation**, they allow engineers to **make high-impact changes frequently and predictably** with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.”

Source: CNCF Cloud Native Definition v1.0, 2018 [emphasis added]

(https://github.com/cncf/toc/blob/master/DEFINITION.md)

“Cloud native”: another definition

“A cloud native application is engineered to run on a platform and is designed for resiliency, agility, operability, and observability.

- **Resiliency** embraces failures instead of trying to prevent them; it takes advantage of the dynamic nature of running on a platform.
- **Agility** allows for fast deployments and quick iterations.
- **Operability** adds control of application life cycles from inside the application instead of relying on external processes and monitors.
- **Observability** provides information to answer questions about application state.”

“Cloud native applications acquire these traits through various methods. It can often depend on where your applications run and the processes and culture of the business.

The following are common ways to implement the desired characteristics of a cloud native application:

Microservices **Health reporting** **Telemetry data** **Resiliency** **Declarative, not reactive**"

(Source: J. Garrisson and K. Nova. Cloud native infrastructure. O’Reilly, 2017.)

Health reporting

- The developers of an application know best what it means for this application to be in a “healthy state”.
- Letting infrastructure providers figure out by themselves the current health of an application often results in fragile designs.
- **A cloud native application should expose its own “health check” interface.**
  - Such an interface can **typically be implemented as a Web endpoint** that returns a health status via an HTTP return code.
  - In addition to error codes, the absence of a prompt reply can also be interpreted as a symptom of a failed task or communication problem.
- **Application may have more than two states (“healthy” / “unhealthy”)**
  - For example, the application may be starting up or shutting down.
  - Giving previse feedback to the infrastructure can help it operate the application in a more robust and efficient way (e.g., distinction between states such as “ready”, “starting (not ready yet to receive traffic)” and “failed”.

Telemetry data

- Not the same thing as health reporting (although some data may overlap) – they serve different purposes.
- **Telemetry data provide information about business objectives**.
  - Sometimes named "**service level indicators**" (SLIs) or "key performance indicators" (KPIs)
  - These data can be **used to check if an application is meeting its "service level objective"** (SLO).
- Telemetry data is often stored in a time series database.
- Examples of questions and metrics ("RED"):
  - **Rate**: "How many requests per second does my application receive?"
  - **Errors**: "Are there any errors?"
  - **Duration**: "How long does it take to obtain a response?"
- Can be used to raise alerts about the global (application-level) behavior.
- Not the same thing as "logs" (logs are mostly used for debugging).

Service Levels: SLAs, SLOs & SLIs (1/3)

- These notions are complementary and all related to the level/quality of service provided to the users/clients using a service (internal or external “customers”).

- The quotes below are taken from Google’s SRE book: https://sre.google/sre-book/service-level-objectives/

- **SLI: Service Level Indicator**
  - **Definition:** “a carefully defined quantitative measure of some aspect of the level of service that is provided”.
  - **Examples:** request latency, system throughput, end-to-end latency, error rate, availability, data durability

Service Levels: SLAs, SLOs & SLIs (2/3)

- **SLO: Service Level Objective**
  - **Definition:** “a target value or range of values for a service level that is measured by an SLI”
  - **A natural structure for SLOs is thus SLI ≤ target, or lower bound ≤ SLI ≤ upper bound**.
  - For example: average search request latency should be less than 100 milliseconds.

  - **Why?** “Without an explicit SLO, users often develop their own beliefs about desired performance, which may be unrelated to the beliefs held by the people designing and operating the service. This dynamic can lead to both over-reliance on the service, when users incorrectly believe that a service will be more available than it actually is, and under-reliance, when prospective users believe a system is flakier and less reliable than it actually is.”

  - **Choosing a set of SLOs can be nontrivial.** For example, several SLIs might be connected behind the scenes:
    - Higher throughput often leads to higher latencies.
    - Many services have some performance cliff/drop beyond some input load threshold

Service Levels: SLAs, SLOs & SLIs (3/3)

- **SLA: Service Level Agreement**
  - **Definition:** “an explicit or implicit contract with your users that includes consequences of meeting (or missing) the SLOs they contain. The consequences are most easily recognized when they are financial—a rebate or a penalty—but they can take other forms.”
  - “An easy way to tell the difference between an SLO and an SLA is to ask "what happens if the SLOs aren’t met?": if there is no explicit consequence, then you are almost certainly looking at an SLO.”
  - “Whether or not a particular service has an SLA, it’s valuable to define SLIs and SLOs and use them to manage the service.”

- For more details, examples and advice:
  - https://cloud.google.com/blog/products/devops-sre/sre-fundamentals-sli-vs-slo-vs-sla
  - https://sre.google/sre-book/service-level-objectives/

Resiliency (1/3)

- **Resilience to failures:**
  - is generally the most important characteristic for an application.
  - is partially managed by the infrastructure but cloud native applications must also handle some part of that responsibility.
  - relies on two main aspects: **design for failure** and **graceful degradation**.

Resiliency (2/3)

- **Design for failure**
  - The SLO specifies the uptime guarantees for a given service.
  - Failure are almost unavoidable (over time) in any complex system.
  - A cloud native application is built with the assumption that failures will happen (although not all precise types/scenarios of failures can be anticipated).
  - Some (severe) kinds of failures cannot be addressed by the application (e.g., network partitions) and should be handled by the cloud platform.

Resiliency (3/3)

- **Graceful degradation**
  - Cloud native apps must be designed to handle excessive load (even though the cloud platform may also help).
  - Graceful service degradation consists in **servicing all/most requests** (i.e., providing available service) **yet with a lower quality of responses** (e.g., with less accuracy or less data – partial answers), in order to lower the request processing costs.

Declarative, not reactive (1/2)

- **Cloud native (distributed) applications should rely on the infrastructure to achieve some kinds properties instead of trying to manage them directly.**
  - For such properties, the application should **declare the desired outcome** and let the infrastructure reach this outcome. The steps to be taken are decided by the infrastructure.
  - This approach allows building simpler and more robust applications (and assemblies of applications/services).

- **Declarative communications**
  - **Applications trust that the network infrastructure will deliver the messages.**
  - The infrastructure can leverage a number of techniques to improve communication resiliency and efficiency, such as load balancing, load shedding, service discovery, retries and timeouts, and circuit breaking.
  - Such infrastructure-level features can be embedded within application using transparent proxies (e.g., implemented as “sidecar” containers).

Declarative, not reactive (2/2)

- **Declarative state**
  - Here, the “state” of an application refers to behavioral or structural properties.
    - For example : the number of replicas for a given service.
  - Old approaches are based on imperative configuration: a description of a sequence of steps to perform in order to reach a given state.
  - In contrast, **with a declarative approach, an administrator describes only the desired state of the distributed application** (not the steps to reach it).

  - Expected benefits:
    - **Fewer errors**. Since a declarative configuration describes an expected result, its impact can be immediately understood.
    - Allows leveraging usual software development tools for manipulating configurations: source versioning, unit testing.
    - **Provides simple support for configuration rollback** in case of problem (i.e., reversible operations).

“The twelve-factor app”

- One of the first historical milestones that lead to the design principles of Cloud-native applications.

- Context: **A (short) document / manifesto written in 2012 by the staff of Heroku, a Cloud provider for applications based on the Platform-as-a-Service (PaaS) paradigm**.
  - “The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and indirectly witnessed the development, operation, and scaling of hundreds of thousands of apps.”

- Available online at: https://12factor.net

- We will summarize it in the next slides, using mostly verbatim quotes.

“The twelve-factor app” – Motivation (1/2)

A synthesis of experience and observations on a wide variety of SaaS apps in the wild.

A triangulation on ideal practices for app development, paying particular attention to :

- The **dynamics of the organic growth of an app** over time,
- the **dynamics of collaboration between developers** working on the app’s codebase,
- and **avoiding the cost of software erosion**.
  - Slow deterioration of software over time that will eventually lead to it becoming slow, faulty or unusable.
  - Typical cause: the software suffers from a lack of updates with respect to the changing environment in which it resides.

Source: Most of the contents of this slide are borrowed/adapted from https://12factor.net

"The twelve-factor app" – Motivation (2/2)

Goals:

- "**to raise awareness of some systemic problems** seen in modern applications development,
- **to provide a shared vocabulary** for discussing those problems,
- and **to offer a set of broad conceptual solutions** to those problems with accompanying terminology."

Source: Most of the contents of this slide are borrowed/adapted from https://12factor.net

The 12 factors – Overview

I) Codebase: One codebase tracked in revision control, many deploys.

II) Dependencies: Explicitly declare and isolate dependencies.

III) Config: Store config in the environment.

IV) Backing services: Treat backing services as attached resources.

V) Build, release, run: Strictly separate build and run stages.

VI) Processes: Execute the app as one or more stateless processes.

VII) Port binding: Export services via port binding.

VIII) Concurrency: Scale out via the process model.

IX) Disposability: Maximize robustness with fast startup and graceful shutdown.

X) Dev/prod parity: Keep development, staging, and production as similar as possible.

XI) Logs: Treat logs as event streams.

XII) Admin processes: Run admin/management tasks as one-off processes.

Source: Most of the contents of this slide are borrowed/adapted from https://12factor.net

“Beyond the 12-factor application”

A revised list of guidelines proposed by Kevin Hoffman (Pivotal) in 2016.

(Freely available booklet from Pivotal/O’Reilly: https://content.pivotal.io/ebooks/beyond-the-12-factor-app)

- Strongly inspired by the original 12-factor manifesto from Heroku.
  - Revisits some factors
  - Adds some new factors
  - Changes the order of some factor to highlight a sense of priority
- **Overall: 15 factors**

- **Warning: “**Rather than adopting an all-or-nothing approach, learning where and when to compromise on the guidelines […] is probably the single most important skill to have when planning and implementing cloud-native applications.”

“Beyond the 12-factor application” - List

1) One codebase, one application
2) API first
3) Dependency management
4) Design, build, release and run
5) Configuration, credentials, and code
6) Logs
7) Disposability
8) Backing services
9) Environment parity
10) Administrative processes
11) Port binding
12) Stateless processes
13) Concurrency
14) Telemetry
15) Authentication and authorization

Infrastructure as Code (IaC)

- **A set of tools and methodologies aimed at achieving automated and reproducible deployments of software configurations on (physical or virtual) machines.**
  - Promotes the usage of machine-readable configuration description files, rather than interactive configuration tools.
  - The configuration descriptions can be managed using a version control system.
  - Abstracts away and leverages low-level APIs (e.g., the machine provisioning API of a given cloud vendor, or the configuration setup of a given operating system).
  - Can address complex deployments requirements (e.g., multi-step coordination of dependencies between machines/services).
- **Typically based on a declarative approach:**
  - Goes beyond mere scripting of configuration steps.
  - Administrator defines the desired target configuration, and the tools perform the necessary actions to reach this target state, regardless of the initial state (and the potential intermediate failures).
- **Examples of tools**: Terraform, Saltstack, Ansible.

Infrastructure as Software

- An infrastructure management approach that goes further than “infrastructure as code”.

- **“Infrastructure as code” relies on a static description of the target infrastructure.**
  - There is limited support for managing the evolution of the infrastructure.
  - Risks of frequent configuration drifts.

- In contrast, **“infrastructure as software” is a continuously running service** that:
  - Builds and maintains a representation of the current state of the infrastructure
  - Monitors the infrastructure and the desired state specified by the administrator (declarative approach)
  - Mutates the infrastructure (and its representation), in order to reach (or maintain) the desired state.
  - Is based on a reconciler pattern (control loop) to converge towards the desired state.

- **Container orchestrator systems (like Kubernetes) are an incarnation of this approach.**

Immutable infrastructure

- A popular approach for the management of configuration modifications within a cloud infrastructure.
- **Main idea:**
  - One should **avoid mutating the existing/deployed infrastructure.**
  - Instead, **it is better to allocate/deploy (from scratch) a new version of the infrastructure (and then decommission the old one).**
  - Metaphor: “treat infrastructure resources (e.g., VMs) like cattle, not like pets!” (a.k.a. “phoenixes vs. snowflakes”)
- Based on some key observations:
  - Lower risks of configuration drifts/problems if we always start from a known, consistent state.
  - This approach is more heavyweight but nonetheless viable, because the creation/destruction of virtualized resources (e.g., VMs or containers) can be made efficient and fully automated.
- Expected benefits: improved configuration consistency, predictability and reliability.
- Limitations: **mainly suited to stateless components.**

Stateless vs. stateful components (1/2)

- One of the major traits of cloud native applications is related to how they handle application state.

- **Some definitions:**
  - By “state”, here, we mean the information that must be retained by a service/component after it has finished processing a request/job.
  - A **stateless component** does not retain any state after the completion.
  - A **stateful component** retains state, either temporarily (**“session state”**) or permanently (**“persistent state”**)

Stateless vs. stateful components (2/2)

- In order to handle scalability, autoscaling, and fault-tolerance in a flexible and efficient way, cloud native applications are based on design principles that strive to:
  - **Dissociate as much as possible the stateful parts and the stateless parts in an application** (“externalized state”)
  - If appropriate, **use distinct stateful components to store different types of state information** (see the “microservices” paradigm discussed later)
    - Different types of storage components and/or different instances of the same component
- In other words, compute and storage layers are decoupled, at a fine granularity.

CNCF: Cloud Native Computing Foundation (1/2)

- A non-profit foundation (part of the Linux Foundation), with many industrial members
- **The foundation's mission is to make cloud native computing ubiquitous through open-source projects.**
- Main responsibilities:
  - Stewardship of the projects
  - Fostering the growth and evolution of the ecosystem
  - Promotion of the underlying technologies, and approach to application definition and management, including: events and conferences, marketing (SEM, direct marketing), training courses and developer certification
  - Serve the community by making the technology accessible and reliable.
- **Useful links:**
  - Web site: https://www.cncf.io
  - **List of projects ("interactive landscape"): https://landscape.cncf.io**

CNCF: Cloud Native Computing Foundation (2/2)

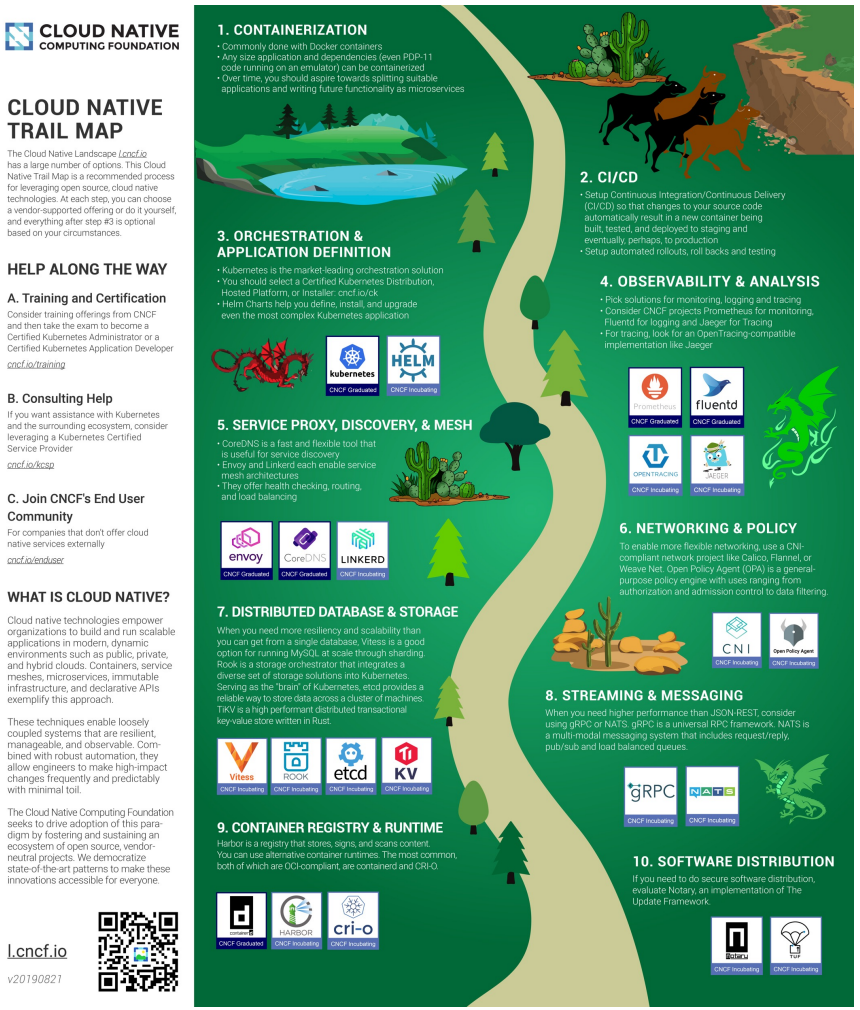
“The CNCF will strive to adhere to the following principles:

- **Fast is better than slow.** The foundation enables projects to progress at high velocity to support aggressive adoption by users.
- **Open**. The foundation is open and accessible, and operates independently of specific partisan interests. […] the foundation’s technology must be available to all according to open-source values.
- **Fair.** The foundation will avoid undue influence, bad behavior or “pay-to-play” decision-making.
- **Strong technical identity.** The foundation will achieve and maintain a high degree of its own technical identify that is shared across the projects.
- **Clear boundaries.** The foundation shall establish clear goals, and in some cases, what the non-goals of the foundation are to allow projects to effectively co-exist, and to help the ecosystem understand where to focus for new innovation.
- **Scalable.** Ability to support all scales of deployment, from small developer centric environments to the scale of enterprises and service providers. This implies that in some deployments some optional components may not be deployed, but the overall design and architecture should still be applicable.
- **Platform agnostic.** The specifications developed will not be platform specific such that they can be implemented on a variety of architectures and operating systems.”

CNCF Cloud native trail map

- Available from: https://github.com/cncf/landscape/blob/master/README.md#trail-map
- “Provides an overview for enterprises starting their cloud native journey.”
- “[Describes] a recommended process for leveraging open-source, cloud-native technologies.”
- 10 steps:

1. Containerization	2. CI/CD
3. Orchestration & application definition	4. Observability & analysis
5. Service proxy, discovery & mesh	6. Networking & policy
7. Distributed database & storage	8. Streaming & messaging
9. Container registry & runtime	10. Software distribution



Monitoring distributed systems (1/11)

- **Monitoring – a definition**:
  - "Collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes."
  - Different types of monitoring:
    - **Black-box monitoring** (externally visible behavior as a user would see it)
    - **White-box monitoring** (based on metrics exposed by the internals of the system)
    - It is often important to combine both types
- **Why monitoring?**
  - Analyzing long-term trends (e.g., growth of number of users, data volume)
  - Making comparisons (over time or over different configurations)
  - Alerting
  - Building dashboards to answer basic questions about a service
  - Conducting retrospective analysis (debugging)

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (2/11)

- **More generally:**
  - **Monitoring and alerting allow noticing when a system is broken or when something is about to break.**
  - An effective alerting system must have a **very good signal-to-noise ratio**. This often implies that an alerting system should be based on simple and robust rules and principles.
  - Monitoring a complex application is a significant effort.

- **A monitoring system must address two main questions:**
  - Symptom: **"What is broken?"**
  - Cause: **"Why?"** (root cause or intermediate cause)
  - Making the distinction between these questions is important to achieve a good signal-to-noise ratio

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (3/11)

- **Black-box monitoring and white-box monitoring are complementary**
  - **Black-box monitoring is symptom-oriented**; it **reports current problems**, causing trouble right now (as opposed to future/predicted problems). Useful for raising alerts about real problems.
  - **White-box monitoring**, based on the inspection of the system internals (e.g., logs) allows the **anticipated detection of upcoming/imminent problems** (such as failures masked by retries).
  - In a multi-layered system, one person's symptom is another person's cause (e.g., slow database accesses in a web application). Therefore white-box monitoring is sometimes symptom-oriented and sometimes cause-oriented.
  - In the case of telemetry for debugging purposes, white-box monitoring is essential (e.g., is my database really slow or is it due to the network?).

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (4/11)

- **The 4 "golden signals" of monitoring:** *latency, traffic, errors, saturation*
  - "If you measure all four golden signals and page a human when one signal is problematic (or, in the case of saturation, nearly problematic), your service will be at least decently covered by monitoring."

  - **Latency**
    - Definition: **The time required to service a request**.
    - It is important to distinguish between the latency of successful requests and the latency of failed requests.
      - A failed request may have a very low latency and "pollute" the statistics of successful requests.
      - A slow error is even worse than a false error.
      - It is important to track error latency (instead of just filtering out errors).

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (5/11)

- **The 4 “golden signals” of monitoring (continued)**
  - **Traffic**
    - Definition: **A measure of the demand placed on a service, using a high-level system-specific metric.**
      - Example 1 – Web service: typical metric is the number of HTTP requests per second (possibly broken down by category (e.g., static vs dynamic requests-
      - Example 2 – Media streaming service: typical metrics are number of concurrent sessions and network I/O rate

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (6/11)

- **The 4 “golden signals” of monitoring (continued)**
  - **Errors**
    - Definition: **The rate of requests that fail, either explicitly or implicitly.**
    - Examples of implicit failures:
      - Request returns HTTP code 200 (OK) but wrong content
      - Request fails by policy: you consider that a response time > 1 second is unacceptable
    - Catching (and measuring) different types of errors generally requires the combination of several monitoring techniques/probes.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (7/11)

- **The 4 “golden signals” of monitoring (continued)**
  - **Saturation**
    - Definition: **The estimation of how “full” a service is (or will soon become)**, with an emphasis on the most limiting resource (e.g., in a memory-constrained system, show memory).
    - In many systems, the performance degrades before reaching 100% of utilization. Therefore, defining a utilization target is essential.
    - In complex systems, saturation can be supplemented with higher-level load measurement (e.g., can the system handle 2x the current load?).
    - Latency increases are often a leading indicator of saturation.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (8/11)

- **Some methodological recommendations**

  - **Do not focus only/mainly on mean values**
    - A given metric may be very imbalanced between different machines (e.g., CPU utilization) or different requests (e.g., latency) and the average may hide some issues.
    - Instead, **pay special attention to the statistical distribution of your measurement and the tail (i.e., high-percentile) values.**
    - If a given high-level task involves several services, the 99th percentile for the latency of a service can become the median latency for your high-level task!
    - **A simple way to monitor and visualize such distributions is to collect request counts bucketed by latencies** (suitable for rendering a histogram), rather than actual latencies, and to choose an exponential distribution for the histogram boundaries. For example: how many requests did I serve that took between 0 ms and 10 ms, between 10 ms and 30 ms, between 30 ms and 100 ms, etc.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (9/11)

- **Some methodological recommendations (continued)**

  - **Choose an appropriate resolution for your measurements**
    - **A naïve choice can result in poor results in both ways: too coarse (useless) or unnecessarily fine-grained (costly without additional benefits)**
    - **Different aspects of a system require measurements at different granularities**.
      - Example 1: Observing CPU load over a time span of one minute will not allow detecting most load spikes. A finer granularity is necessary.
      - Example 2: For the goal of achieving 99.9% annual uptime (in other words, no more than 9 hours of aggregated downtime per year), checking the health of a service (or the saturation of a resource like the disk space) more than once or twice per minute is probably unnecessary.
    - **High-resolution measurements may be expensive to collect, store and analyze.** Think carefully about your needs and how you can optimize the aggregation/summarization/retention in order to achieve a good trade-off between accuracy and cost.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (10/11)

- **Some methodological recommendations (continued)**

  - **Design your monitoring system for simplicity**
    - The sources of potential complexity are never ending.
    - Like all software systems, **monitoring can become so complex that it becomes fragile, difficult to change, and a big maintenance burden.**
    - The rules that catch real and frequent incidents should be as simple, predictable and reliable as possible.
    - Consider the removal of some parts that are not very useful/exercised.
    - It can be tempting to combine monitoring with other facilities related to system inspection (e.g., used for profiling, debugging, crash analysis, etc.) but combining too many parts may result in a global system that is very complex and fragile. Maintaining distinct systems with clear, simple, loosely coupled points of integration is a better strategy.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

Monitoring distributed systems (11/11)

- **Wrap-up**
  - **A healthy monitoring and alerting system must be simple and easy to reason about.**
  - The alerts should **focus primarily on symptoms or imminent real problems.**
    - Cause-oriented should rather be used as aids to debugging problems.
  - Monitoring symptoms is easier the further “up” your stack you monitor.
    - However, monitoring saturation and performance of subsystems such as databases often must be performed directly on the subsystem itself.
  - Adapt your targets to goals that are actually achievable.
  - Make sure that your monitoring supports rapid diagnosis.

Source: The contents of this slide are heavily based on https://sre.google/sre-book/monitoring-distributed-systems/

DORA metrics (1/5)

- **DevOps Research and Assessment (DORA)** is a running research program of Google Cloud that seeks to understand the capabilities that drive software delivery and operations performance.
- In 2020, the DORA team has identified a set of **4 key metrics** that indicate the performance of a software development team:
  - **Deployment Frequency** — How often an organization successfully releases to production
  - **Lead Time for Changes** — The amount of time it takes a commit to get into production
  - **Change Failure Rate** — The percentage of deployments causing a failure in production
  - **Time to Restore Service** — How long it takes an organization to recover from a failure in production
- The set of metrics was **later updated with a 5th metric**: **reliability** — How a team meets or exceed the reliability targets for the software and applications they operate.

DORA metrics (2/5)

- **Deployment frequency**
  - This metric is related to velocity.
  - “For the primary application or service you work on, how often does your organization deploy code to production or release it to end users?”
  - “Useful to to understand how often the team successfully deploys software to production, and how quickly the teams can respond to customers’ requests or new market opportunities.”
- **Lead time for change**
  - This metric is related to velocity.
  - “For the primary application or service you work on, what is your lead time for changes (that is, how long does it take to go from code committed to code successfully running in production)?”
  - “Reflects the efficiency of CI/CD pipelines and visualizes how quickly work is delivered to customers.”

DORA metrics (3/5)

- **Time to restore service**
  - This metric is related to stability.
  - “For the primary application or service you work on, how long does it generally take to restore service when a service incident or a defect that impacts users occurs (for example, unplanned outage, service impairment)?”
  - “Low time to restore service means the organization can take risks with new innovative features to drive competitive advantages and increase business results.”
- **Change fail percentage**
  - This metric is related to stability.
  - “For the primary application or service you work on, what percentage of changes to production or releases to users result in degraded service (for example, lead to service impairment or service outage) and subsequently require remediation (for example, require a hotfix, rollback, fix forward, patch)?”
  - “Useful to gain insights into the quality of the code being shipped. High change failure rate may indicate an inefficient deployment process or insufficient automated testing coverage.”

DORA metrics (4/5)

- **Reliability**
  - This metric is related to operational performance and complementary to the four previous metrics, which are more related to software delivery performance.
  - Assesses the ability of a team to meet or exceed their reliability target.
  - Encompasses several dimensions: availability, latency, performance, and scalability.

DORA metrics (5/5)

For more information:

- https://dora.dev
- https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance?hl=en
- https://www.sumologic.com/glossary/dora-metrics/
- https://github.com/dora-team/fourkeys
- https://docs.gitlab.com/ee/user/analytics/dora_metrics.html
- https://github.com/DeveloperMetrics/DevOpsMetrics