Cloud Native Software Development

Software development doesn't exist in a vacuum. The software will need to run somewhere. While not every software needs to, or indeed will, scale to Google levels, a little forethought can go a long way towards surviving the first contact with the customer hordes.

The enemy of scalability: state

Any application that does not store or access data is easy to scale. Simply run more copies of it and put a load balancer in front of it.

State can rear its ugly head in obvious, and in more subtle ways. The obvious point state is, for example, your database. When you have data in your database that's a form of state. You can run as many application servers as you want, if the database can't handle the load you're done. As we discussed in the third lecture scaling databases can be quite tricky since the consistency model may not permit scaling.

A second point of contention is the ability to store data. Integrating an object storage instead of simply writing data to the disk will, again, help with scaling.

A more subtle point of contention may be the storage of sessions. Sessions are usually built by sending the user a session ID via a cookie and then storing a blob of data on the disk or in the database. The problem is not only the storage, but also that moving from a local filesystem to a database may create a race condition that may present a security issue with your application.

A similar issue arises when near-realtime data exchange is desired between a large amount of users (e.g. a chat application). While a single server can scale to tens of thousands of users, that server is neither redundant, nor will it serve an endless number of users. PubSub systems can help with that.

The Twelve Factor App

Once we survive the first night after the launch of our application it often becomes time to deal with long-term architectural problems. The 12 factor app is a concept that collects the current best practices of writing applications for the cloud. Keep in mind that these are just guidelines and you should never make a religious tyrade out of following these guidelines. Instead, apply common sense and be pragmatic about what to do.

1. Codebase

This first guideline is pretty simple: keep your application in a version control system. While this should be the default in 2020 it unfortunately still bears saying.

2. Dependencies

This recommendation deals with dependencies. Almost every programming language ecosystem nowadays has a dependency manager to handle downloading dependencies. These dependencies should be declared explicitly in the configuration file for the dependency manager (e.g. `package.json`, `composer.json`, `pom.xml`, etc.)

Furthermore, these dependency managers often create a lock file (`npm-package-lock.json` , `composer.lock` , etc.). They record the exact versions, and sometimes hashes of the third party library. This ensures that the same exact version is installed in the development and production builds.

3. Configuration

While in older times configuration files had many formats, modern, container-based applications distinctly move towards environment variables. Environment variables are a cross-platform way to provide variables to an application and give a DevOps engineer a flexible way to configure a containerized application.

4. Backing services

External databases, caches, etc. should be treated as such. This means that the connection options (server name, url, username, password, etc) should be configurable via environment variable and should also be replacable by a system administrator by simply reconfiguring the container.

Testing the testability is especially important with services that are not as well standardized as a MySQL database. For example, many S3 implementations only support Amazon's version and cannot be configured to use alternative providers. An application following this recommendation should be tested against at least one other S3 provider.

5. The build process

The build recommendation says that the build process should be separated from the runtime configuration and the release process. In practice this means that you don't bake your configuration into your container. It also means that each release should have a version number or date, allowing you to roll back to a specific version.

6. Stateless processes

The processes recommendation says that your application should run as a single program without any shared data between different copies of your application. Any state should be outsourced to an external database or system as mentioned before.

7. Network configuration

The port binding recommendation says that an application is standalone and does not require an external webserver to run. (Typically PHP applications require a bit of legwork to satisfy this requirement.)

8. Process handling

The concurrency recommendation defines a number of rules for process handling. It requires that it should be possible to run multiple copies of the application, optionally across several machines. It also requires that the application should not daemonize.

While this may seem as trivial with a green field project, it is definitely a challenge for legacy application.

9. Disposability

This recommendation states that an application should have a fast startup and should also shut down quickly when receiving the TERM signal. Applications should also be able to recover from unexpected crashes.

10. Development/production gap

The dev/prod parity recommendation postulates that developers of apps should strive to have a continuous method of deployment that allows for rapid rollout of updates. This lowers the work a developer needs to go through to deploy and therefore promotes smaller change sets.

Smaller change sets make tracking down issues with deployments easier, but can be impractical when working with larger, slower moving clients. (e.g. banks, telcos, etc.)

11. Logs

The logs recommendation should simply write logs to the standard output in a standardized form (e.g. JSON). Everything else, like routing or storing logs, is not the applications concern.

In practice each application contains at least some basic logic to filter logs to make it easier to configure the logging level.

12. Tooling

The admin process recommendation concerns itself with the command line tools needed to operate the application. These are things like running database migrations, etc. The recommendation says that these tools should be runnable directly from the applications directory without much extra configuration or installation.

> **Tip**
>
> These factors are, of course, not exhaustive. IBM has published a whole host of additional factors.

Monitoring and metrics collection

The above 12 factors are, by necessity, limited in scope. The author(s) of those 12 factors have taken many things into account, yet left out others.

One of the most important aspects left out of the list above is monitoring. Any developer wanting to working a cloud-native environment needs to consider monitoring. After all, the journey to the cloud is supposed to bring greater stability and facilitate rapid development cycles.

In this section we will discuss the monitoring types and their practical implementation.

Periodic checks

One of the most classic forms of monitoring are periodic checks. Periodic checks run a certain check, for example a connection, against the application. If the connection succeeds the check is green, if not the check is red and an alert is sent out. The check types are varied, ranging from a simple connection attempt to a fully blown interaction with the application.

These checks not only serve the purpose of alerting the administrator but can also work as a trigger for a self-healing system. In a containerized environment, for example, we have several options to implement a self-healing system. You can add the `HEALTHCHECK` directive in the `Dockerfile` for example, or you can implement Liveness, Readiness, and Startup Probles in Kubernetes.

The important question to consider with periodic checks is the rate of false positives. An intermittent network failure may not be a signal for an application failing and can still raise alarms. In order to avoid alert fatigue a well designed monitoring system keeps false alerts to a minimum.

Feature tests

A very special case of periodic checks are full-on feature tests. These feature tests blur the boundaries between development-time software testing and operations. The intention is to run a full application test, such as a Selenium test suite, against a production application. This test could test a full signup, for example. This level of testing ensures that the application keeps working even after deployment.

While not practical in every scenario these tests can ensure a high level of certainty and can also function as a regression test when the infrastructure changes rather then the application.

Metrics collection

Another important aspect of monitoring is metrics collection. Periodic checks can only check what's there, but they cannot indicate a brewing trouble before it happens. Metrics can be an early warning system for a failure in the application. For example, a software rollout can significantly impact signups. While the change may have been entirely intentional it is business critical to observe this happening and alert immediately to a potential defect in the software. Are the loading times too long? Did the layout change make the process too confusing? Metrics can help.

As you will learn in the Prometheus exercise, Prometheus has established itself as a defacto standard for monitoring cloud-native applications. Prometheus makes it exceedingly easy to integrate metrics collection with just about any application. It is not uncommon to see applications include a small webserver that exposes internal metrics in the Prometheus data format. Prometheus then collects these metrics and provides alerting and query services.

The reason why Prometheus has established itself as a standard is its ability to monitor a dynamic number of machines. It first queries the the cloud provider for a list of IP addresses and then queries those IPs for the metrics. As machines are started and stopped Prometheus always keeps track of them.

Log collection

Depending on how well written the application is it may expose more or less useful logs. One thing is certain: error counts can also be an early warning system for latent defects. For example, your access log may contain the individual entries of failed requests. A log aggregation system such as the ELK stack can discover patterns in your logs and give you alerts when something is going south.

Dashboards

No monitoring system is complete without a way to look at what's going on. This niche is filled by dashboards such as Grafana which we will discuss in greater detail in exercise 5.

Microservices

The proliferation of containers has brought about a change in software design aswell: microservices. Microservices is a concept where, instead of building one monolithic application one builds a tiny service for each need. These services are connected across the network, usually via HTTP.

On its face this solves the problem of creating an incoherent giant ball of code with code quality. However, experience has shown that this is not true. **A team or company that is not able to write a good quality monolith is also not able to design and maintain microservices properly.**

To write a well designed system, no matter if microservices or monolith, needs to have clean API boundaries between the different parts of the system. Unfortunately, there is no simple way to achieve clean boundaries as there is a host of literature on Clean Code that would exceed the bounds of this course.

Where's the difference? For one, it's the *network*. When you write a monolith you can rely on every call across API boundaries will always, reliably return. With microservices, this is not true. Microservices calls can hang indefinitely. It is possible that the other service simply disappears from the network and you won't even know it. That's why a microservices architecture always needs to include the Circuit Breaker pattern

Second, it's the runtime. If there are several microservices calling each other, each service being *reasonably* fast, the run time can still add up. Imagine 5 microservices calling each other, each woth a maximum runtime of 200ms. That's a one second response time right there!

Third, it's the API design. If you break your API in a monolithic application chances are your compiler will bark at you and refuse to compile the code. If you break compatibility across microservices possibly nothing happens. With badly written microservices that ignore missing parameters, for example, systems can break in a really horrifying manner.

Microservices make sense, and indeed follow Conway's Law:

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. — Melvin E. Conway

Conway formulated this in 1968, long before microservices. It still holds true today: microservices in an organization will always follow the team or company structure. For example, there may be several development teams that enjoy the ability to develop and deploy their application parts independently from each other. Or, there may be one *development* team that needs to gather information from several source systems from different departments.

Microservices are also a good idea where boundaries are needed between applications with different technology stacks (e.g. to pass information from Java to Javascript, etc.)

However, it makes very little sense for a small development team in a startup, using a homogeneous technology stack to use microservices. The changes are this team will be violating API boundaries all over the place and disregard the requirements for building a network-based distributed system.

Ask the Authors

If you want a fun story about how microservices can go horribly wrong, ask the authors in the consulting sessions.

Be careful!

It is easy to go over-board with microservices to the point where you have more Kubernetes manifests than actual program code! Don't forget, you not only have to write them, but also deploy them. If you go for services that are too small or your team is ill-equipped to handle a large number of tiny services in production you may have one of the many horrible microservices failures on your hand.

Loosely typed languages

Be careful with loosely typed languages and systems like JavaScript! These systems are very tolerant towards missing parameters, or parameters with incorrect types. This can lead to some disastrous consequences.

Further reading

While we will not be asking any of the following materials in the test we strongly recommend that you take them as a starting point for your further studies:

- Clean code
  - S.O.L.I.D. principles
  - Getting started in Object-Oriented Programming
  - What people misunderstand about OOP
  - What the ** is an IoC container?
  - The clean code talks: Unit Testing
  - The clean code talks: OO Design for Testability
  - The clean code talks: Don't look for things!
  - How to design a good API and why it matters
- Microservices
  - Principles of Microservices
  - 3 common pitfalls of microservices integration
  - Avoiding Microservice Megadisasters
- Distributed systems
  - Distributed Systems in One Lesson

Service meshes, Frameworks, and Tools

Service meshes are a tool to help with building microservices. As mentioned before, microservices are hard. Not only do you have to take the network into account, not only do you have to make the API work together across many different components, you also have to make sure that components find each other.

Kubernetes makes sure that there are internal load balancers for each service allowing you to do a basic rollout. However, Kubernetes does not retry failed calls, allow for versioning, implement a circuit breaker, etc.

Service meshes like Istio do that. You don't strictly *need* a service mesh to build microservices, but they make the process easier. This is especially true if you run applications developed by different teams with different tooling, where implementing a common service registration would be hard. We recommend watching Building cloud-native applications with Kubernetes and Istio, by Kelsey Hightower for more details on how Istio works.

Alternatively, you can of course use the tools afforded by your framework. If you are in the Java world, you could use Spring Cloud, for example. Spring Cloud will let you register your services with a wide variety of backend services.

However, keep in mind: microservices are hard. As a developer you will be curious and willing to learn about them, of course. However, your curiosity may not serve the company's interest as much as you may think it does.