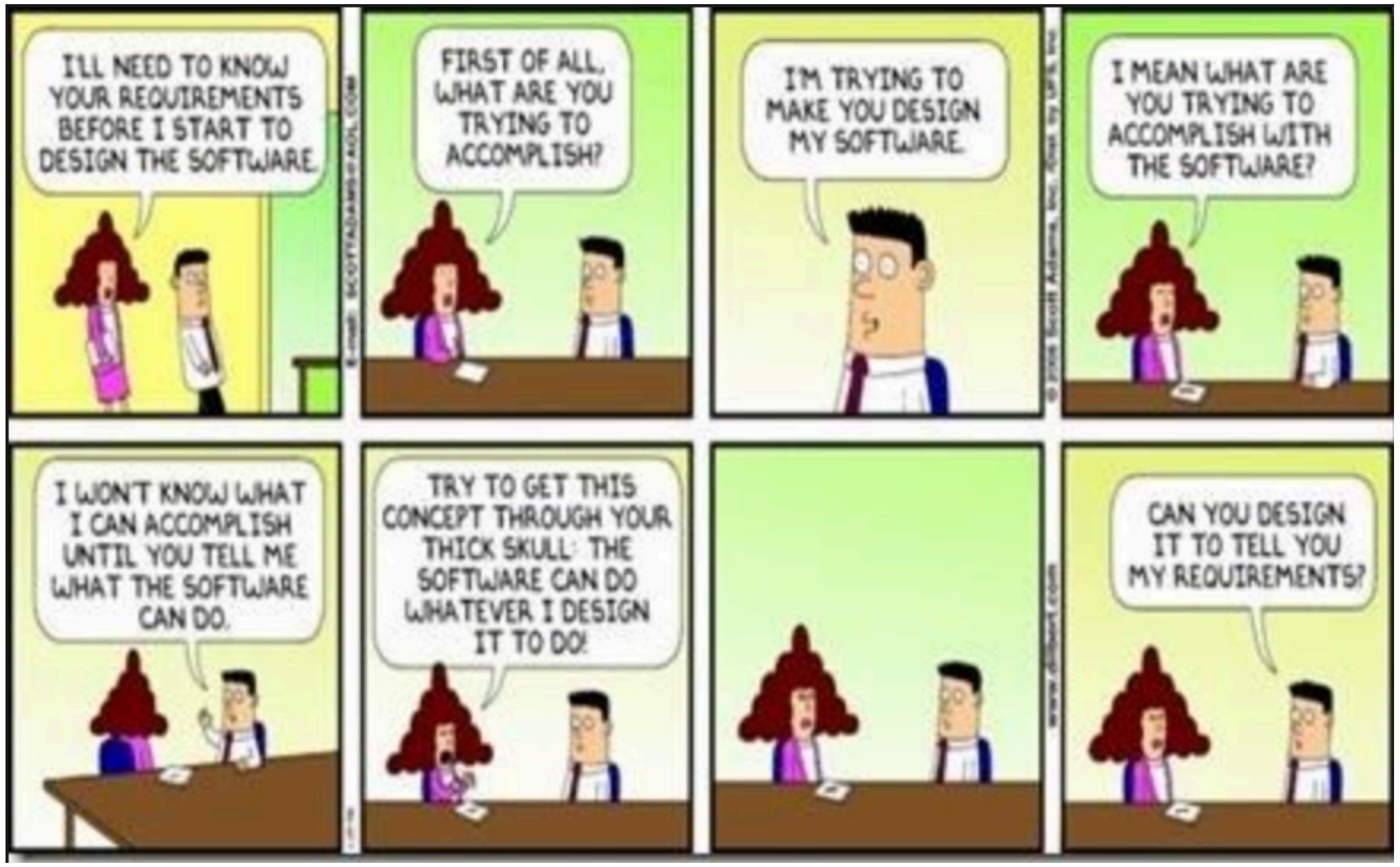
DHBW Informatik 4. Semester Wahlpflicht Cloud Computing Chapter 5 Micro Services

Juergen Schneider


I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

Very well documented in:

Microservices From Design to Deployment

by
Chris Richardson, Floyd Smith NGINX

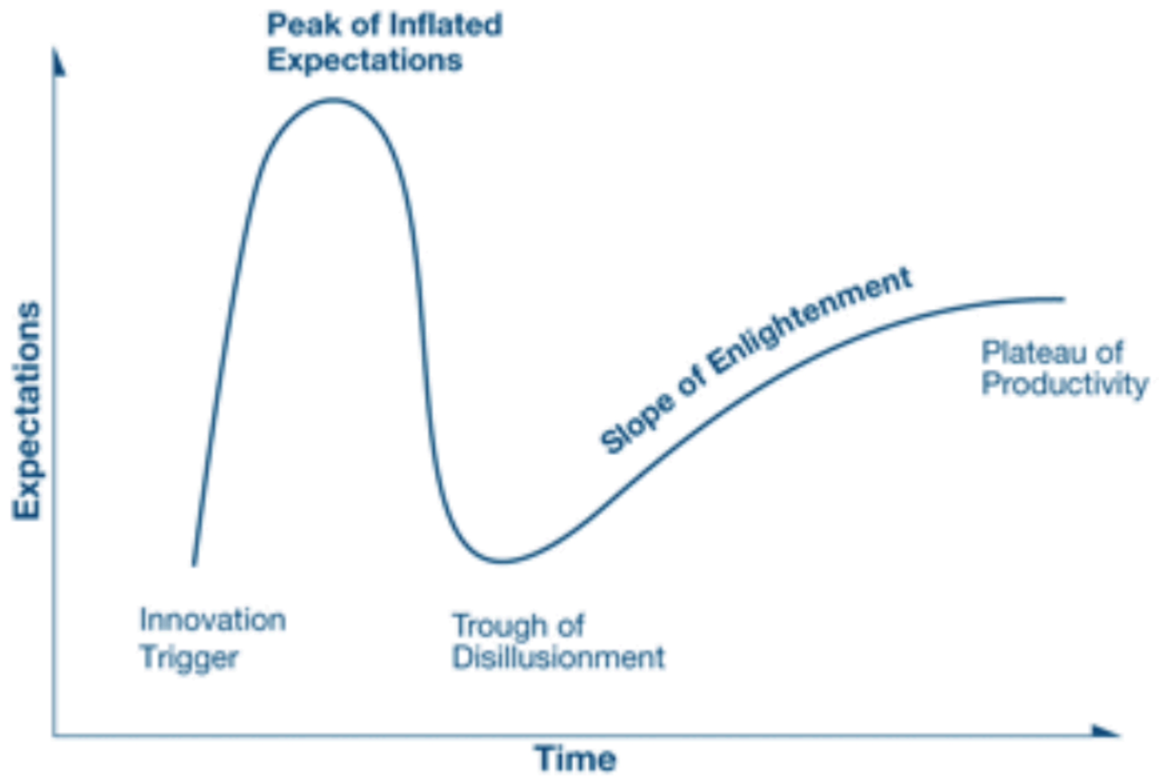
https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/

We use it as our layout here…..

Agenda

- What are Microservices
- Why Microservices
- Microservices Pros and Cons
- 12 Factor App

What are Microservices ?



The Gartner Hype Cycle

- Yet another Hype ?
- Yet another SOA ?
- What about data ?
- How to operate ?
- How to scale ?
- Where does it fit and where does it not ?

Microservice architecture, or simply **microservices**, is a method of developing software systems that tries to focus on building **single-function modules with well-defined interfaces and operations** (microservices).
In other words: Microservice Architecture try to tackle the big monolith application complexity (by solving some of them and maybe adding others) ;-) ..
Some characteristics :

- Microservices are at most self contained (no dependency to any other particular microservice (e.g. if data from outside is needed it is a question of an service interface call)
- Microservices scale independently
- Microservices deploy independently
- Microservices will be developed independently
- Many Microservices (e.g. 600 at Netflix) build a business app
- Exploit cloud characteristics (elasticity) (Cloud Computing in mind thinker…)
- Future trend is to 'use' existing cloud service in the cloud (rather build your own) (SaaS via API)
- Netflix, eBay, Amazon, Twitter, PayPal, (and their Web Apps)

How big is Micro

- The debate will go on …

1. Can be developed and maintained by a team of (3 to 9) people
2. Is understood by one developer
3. A microservice could re-written in weeks
4. Should fit into a container
5. Two many micro services increases operation costs

Reference: Referat Hanna Siegfried, Nahku Saidy, DHBW Stuttgart TINF17ITA

Large monolithic Apps versus many small Microservices

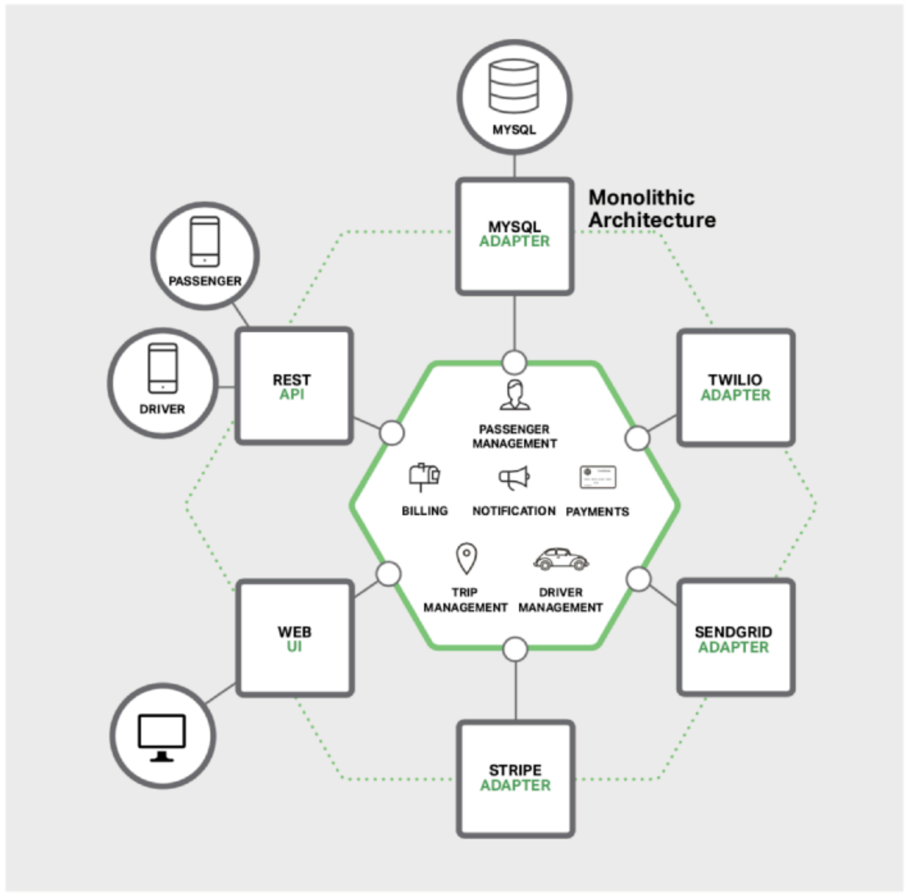


Figure 1-1. A sample taxi-hailing application.

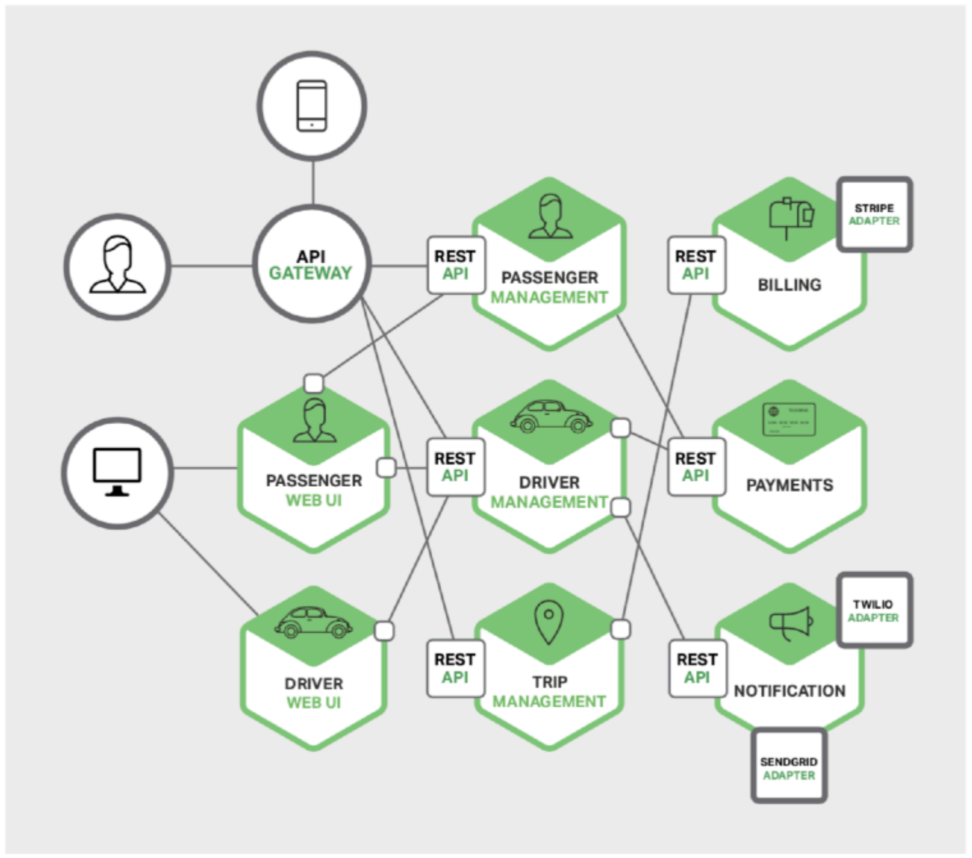


Figure 1-2. A monolithic application decomposed into microservices.

- Above is a monolith app with different adapters
- Good things (potentially the bad things for the other approach)
  - Only one thing to monitor and operate on
  - Data managed consistently with a single or few Databases (Data can be protected and HA/DR is easier)
  - Fast program to program calls
  - Because this has been built in the past it is 'easier' or less expensive to maintain (we know what we do and we have the optimised infrastructure)
  - Version dependencies solved in the app (could be resolved during build time)

- Many small apps (different languages) loosely coupled via data driven API (e.g. REST).. 12 factor Apps
- Good things (potentially the bad things for the other approach)
  - Each service can be lifecycle managed independently
  - Faster deployment (only the smaller thing must be started)
  - Scaling is more efficient (only the parts needed will scale)
  - Intrinsic HA (horizontally scaling on demand)
  - Services can be distributed over many servers
  - Some services can be bought from the cloud
  - Better suited for continuous delivery
  - Less 'dirty' interfaces, higher separation of module scope
  - Support of running in different version must be coded in the services

Scalability…. the many dimensions

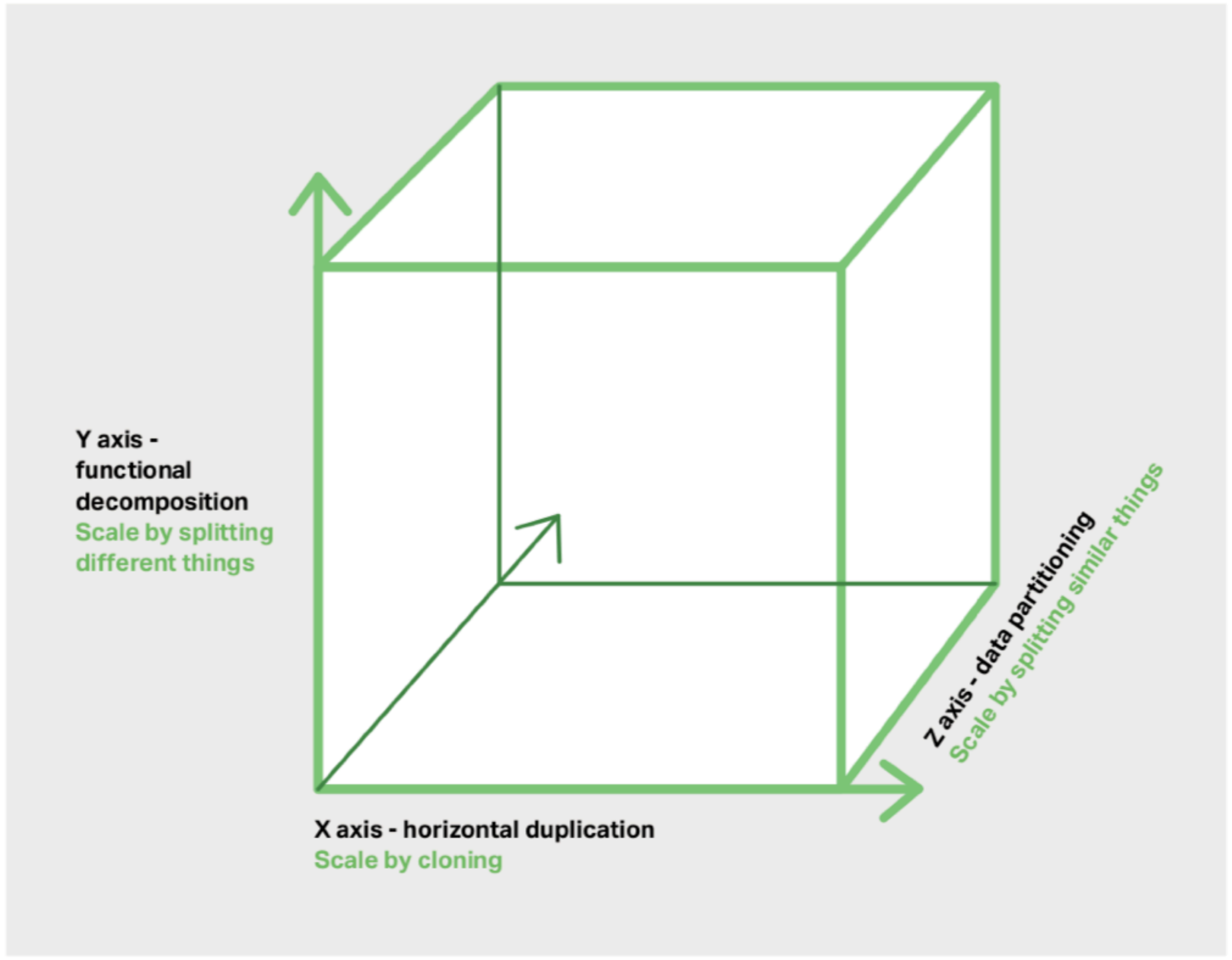

Y axis - functional decomposition
Scale by splitting different things
Z axis - data partitioning
Scale by splitting similar things
X axis - horizontal duplication
Scale by cloning

Figure 1-3. The Scale Cube, used in both development and delivery.

Actually we have 4 dimensions

- X axis, horizontal (or vertical) scaling one or instances of the same thing to get more CPU, Memory, Storage, Network Power. X itself has 2 dimensions
  - Vertically : scale by allocating more to one instance
  - Horizontally : more instances
- Y axis, scale by different things (our Microservice model), smaller independent pieces can be efficiently scaled). The different things then scale horizontally or vertically
- Z axis similar to the X axis, but driven from data. Example: split the processing of HR management by using different instances based on the person ID.
- The problem of unclear workload demand (extreme high ups and downs and time unknown) can be mitigated thru cloud based deployment automation

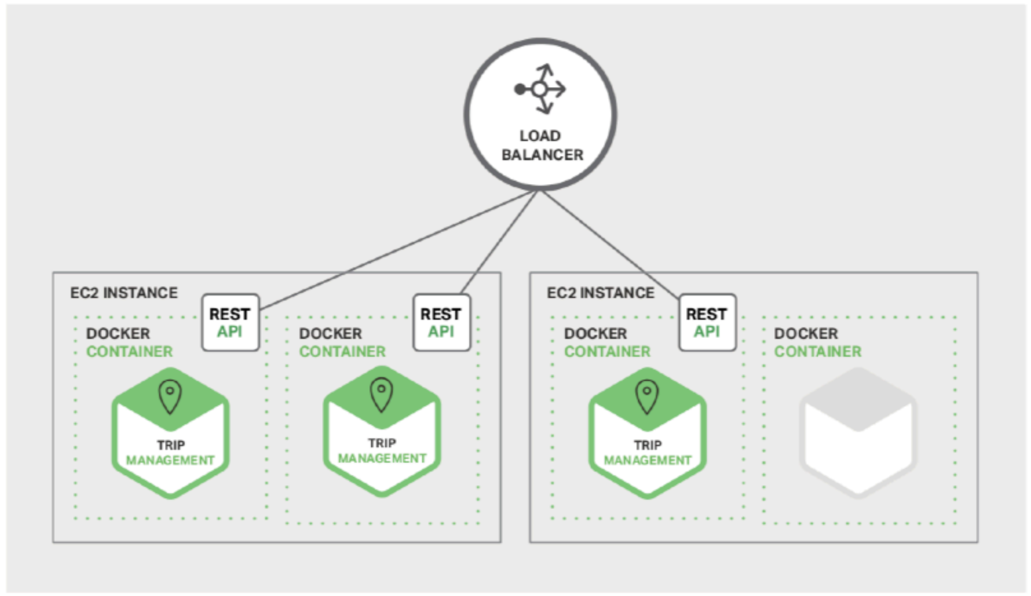


Figure 1-4. Deploying the Trip Management service using Docker.

- Microservices typically scale using the various levels of loadbalancer capabilities

Again… what is different in the cloud when we build apps

- You don't 'own' the infrastructure and there are many more user using it….
  - 'Some' (but not full) admin access to Cloud services (e.g. Kubernetes) (Cloud User admin versus Full admin).
  - Operating System (and potential dependencies to it)
  - New Infrastructure after a 'restart'
    - We have a 'cattle' farm
    - App Data on local file systems are lost

- A 'lift and shift' application migration must tolerate those changes
- A 're-structuring' of a App towards Microservices should exploit cloud features
  - Function decomposition (typically smaller than a traditional monolith, -> Done by the app developer)
  - Horizontal scaling. (Done by cloud platform)
  - Understand the microservice platform
    - Service Registry and Discovery (Done by cloud platform)
    - Location and Redundancy
    - Operations

12 factor App
Some guideline to build Apps in the Cloud

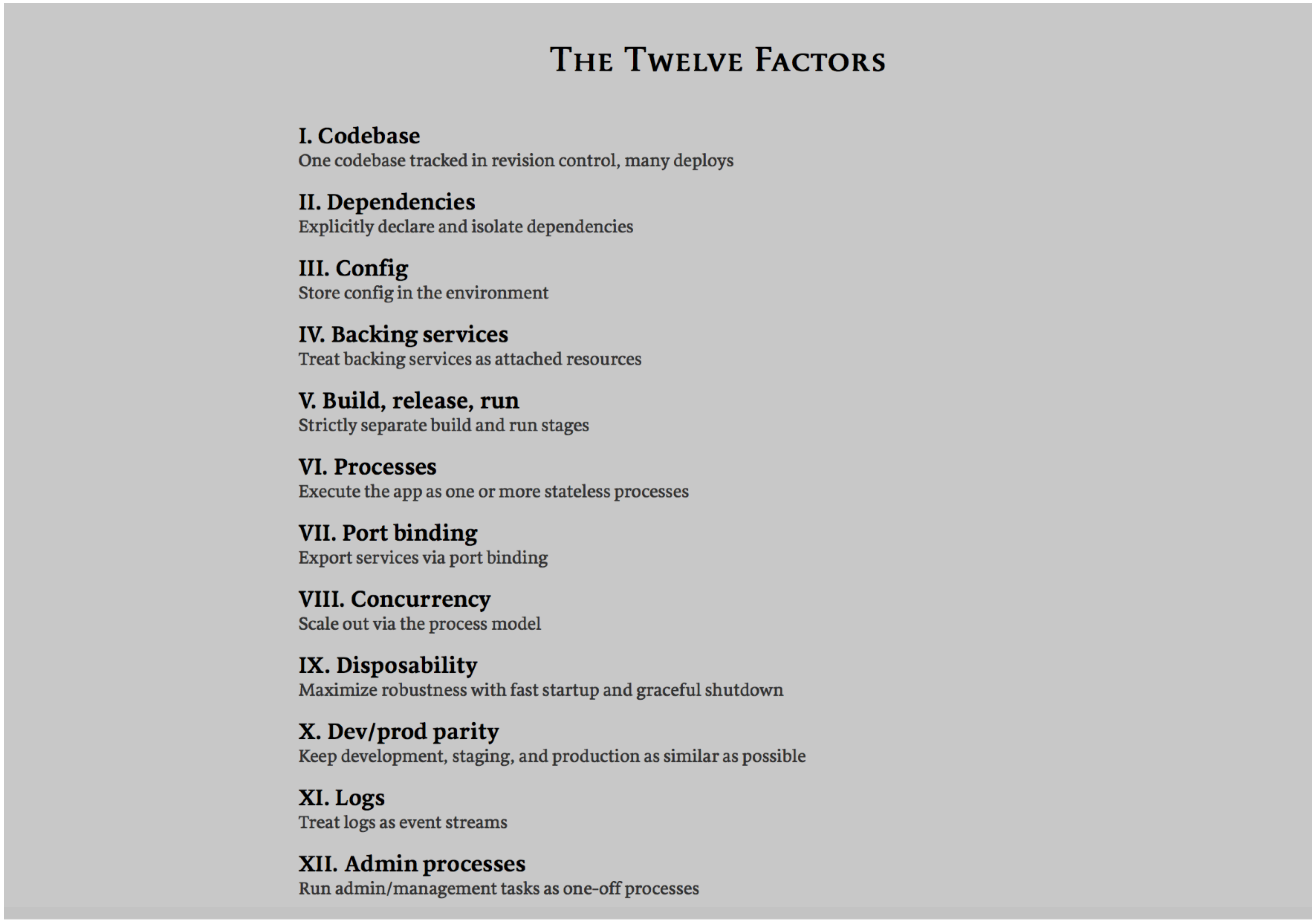
- A 12 factor App
  - enables **Build/Release and Deploy Automation** using an external (from the code) declarative approach.
  - requests to **minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
  - can **scale up** without significant changes to tooling, architecture, or development practices.
  - Enables the usage of MicroServices

12 factors are identified by a set of 'web' developers (e.g. Heroku) to consider building apps in this space.

- Talk, **12 Factor Apps: A Scorecard**
  - By Matt Momen, GE Digital
  - https://www.youtube.com/watch?v=Q_S0pGsKav0

- https://codersociety.com/blog/articles/twelve-factor-app-methodology

12 factors as overview. https://12factor.net



The Twelve Factors

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

Factor 1 one codebase, many deploys

I. Codebase

One codebase tracked in revision control, many deploys

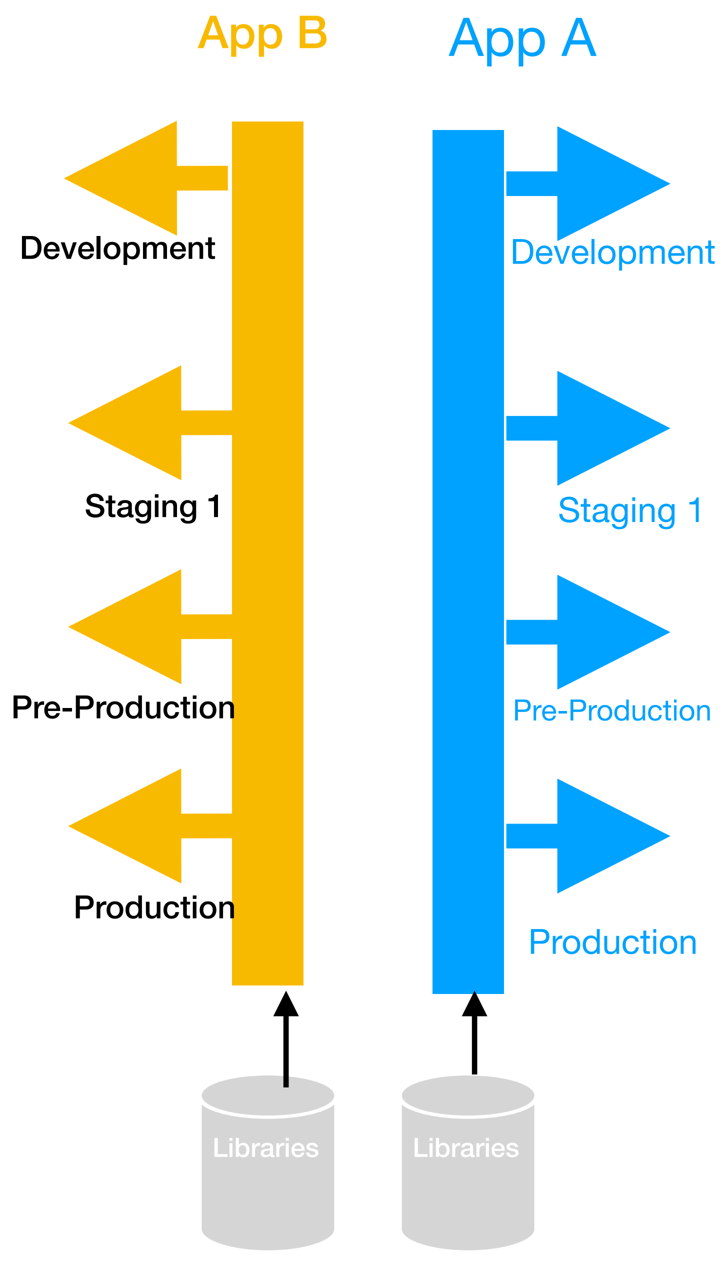
A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a *code repository*, often shortened to *code repo* or just *repo*.
A *codebase* is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).

There is always a one-to-one correlation between the codebase and the app:

- **If there are multiple codebases, it's not an app – it's a distributed system.** Each component in a distributed system is an app, and each can individually comply with twelve-factor.
- **Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.**

There is **only one codebase per app, but there will be many deploys of the app**. A *deploy* is a running instance of the app. This is typically a production site, and one or more staging sites. Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy.
The codebase is the same across all deploys, although different versions may be active in each deploy. For example, a developer has some commits not yet deployed to staging; staging has some commits not yet deployed to production. But they all share the same codebase, thus making them identifiable as different deploys of the same app.

$$\begin{tikzcd} \text{App} \arrow[r, dash, "1"' near start, "1"' near end] & \text{Codebase} \arrow[r, dash, "1"' near start, "n"' near end] & \text{Deploy} \end{tikzcd}$$



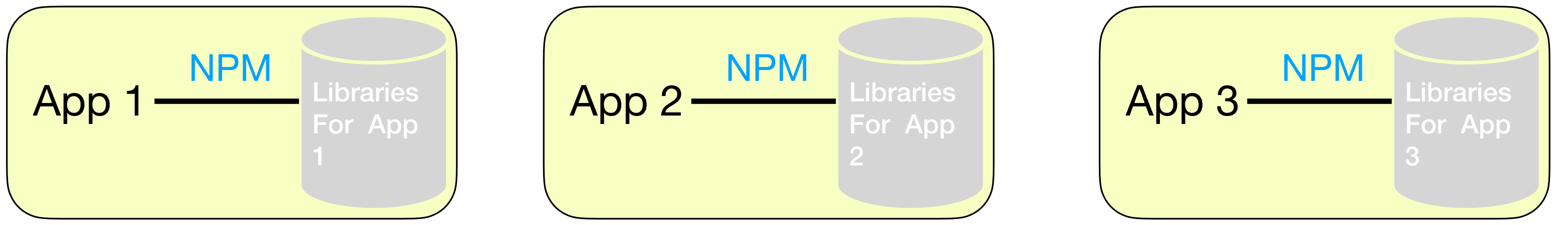
More Factors…

II. Dependencies

Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. *Libraries installed through a packaging system can be installed system-wide (known as "site packages") or scoped into the directory containing the app (known as "vendoring" or "bundling").*

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a *dependency declaration* manifest. Furthermore, it uses a *dependency isolation* tool during execution to ensure that no implicit dependencies "leak in" from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.



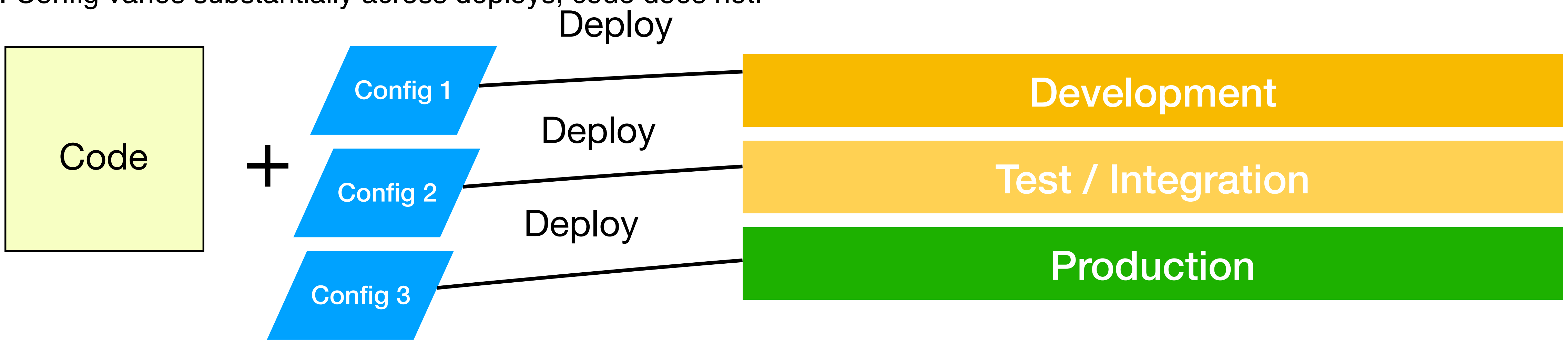
III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not.



More Factors…

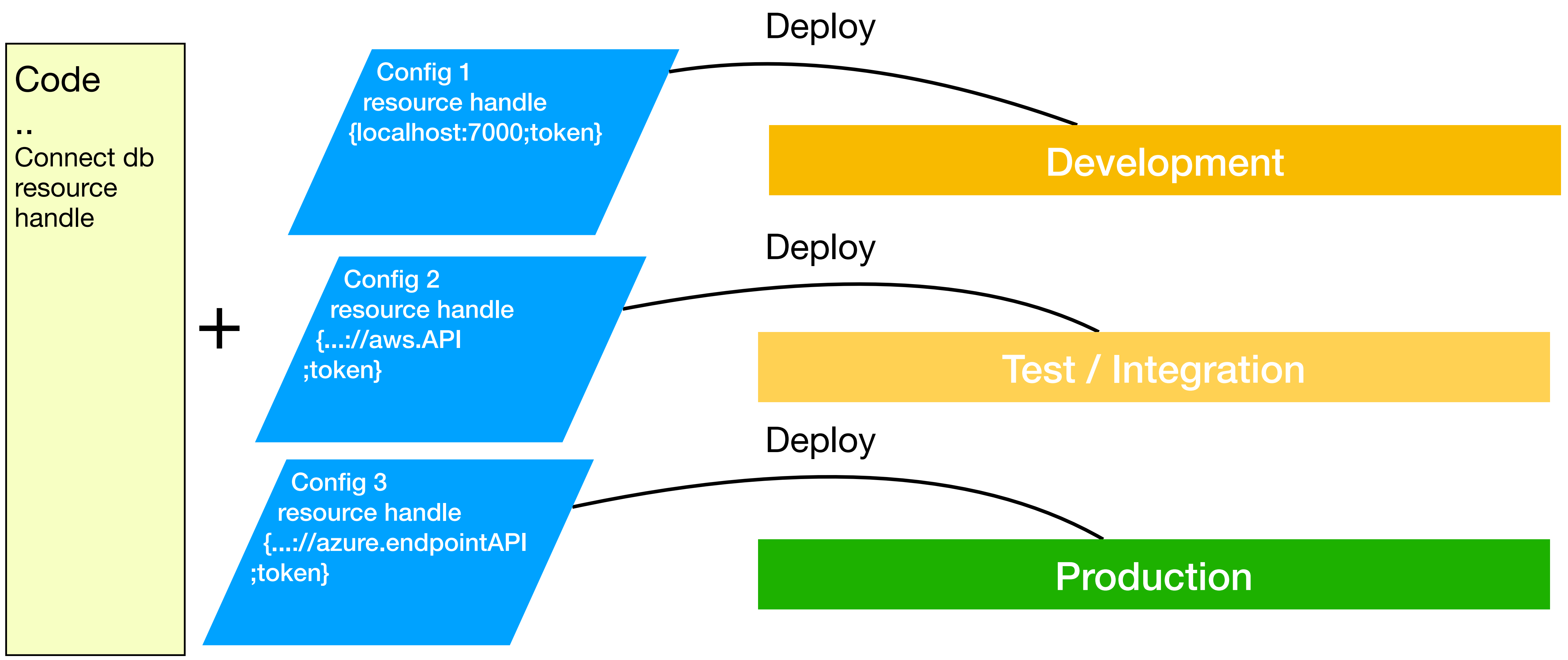
IV. Backing services

Treat backing services as attached resources

A *backing service* is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

Backing services like the database are traditionally managed by the same systems administrators as the app's runtime deploy. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as Postmark), metrics-gathering services (such as New Relic or Loggly), binary asset services (such as Amazon S3), and even API-accessible consumer services (such as Twitter, Google Maps, or Last.fm).

The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.


Code
..
Connect db resource handle
+
Config 1 resource handle {localhost:7000;token}
Config 2 resource handle {...://aws.API ;token}
Config 3 resource handle {...://azure.endpointAPI ;token}
Deploy
Deploy
Deploy
Development
Test / Integration
Production

More Factors…

V. Build, release, run

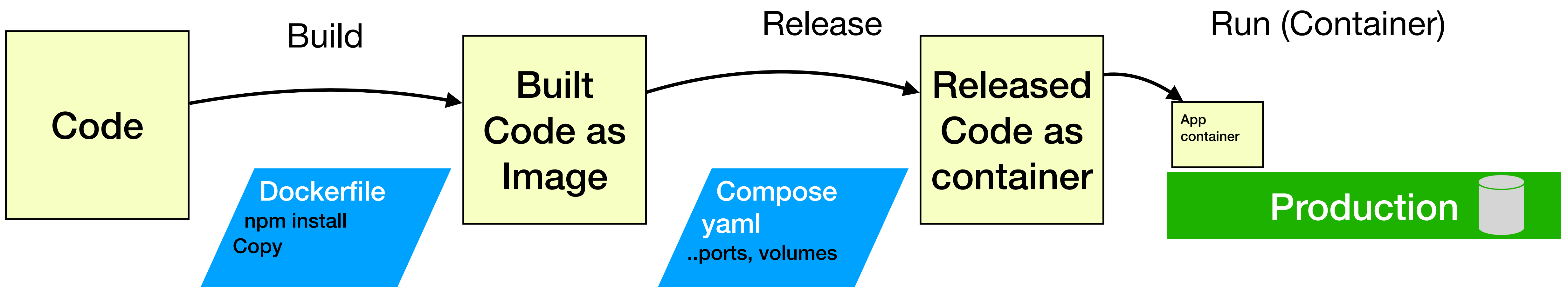
Strictly separate build and run stages

A codebase is transformed into a deploy through three stages:

- The ***build** stage* is a transform which converts a code repo into an executable bundle known as a *build*. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors dependencies and compiles binaries and assets.
- The ***release** stage* takes the build produced by the build stage and combines it with the deploy's current config. The resulting *release* contains both the build and the config and is ready for immediate execution in the execution environment.
- The ***run** stage* (also known as "runtime") runs the app in the execution environment, by launching some set of the app's processes against a selected release.

The twelve-factor app uses strict separation between the build, release, and run stages. For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.

For example. Docker based Apps



Potentially more than 1 revision

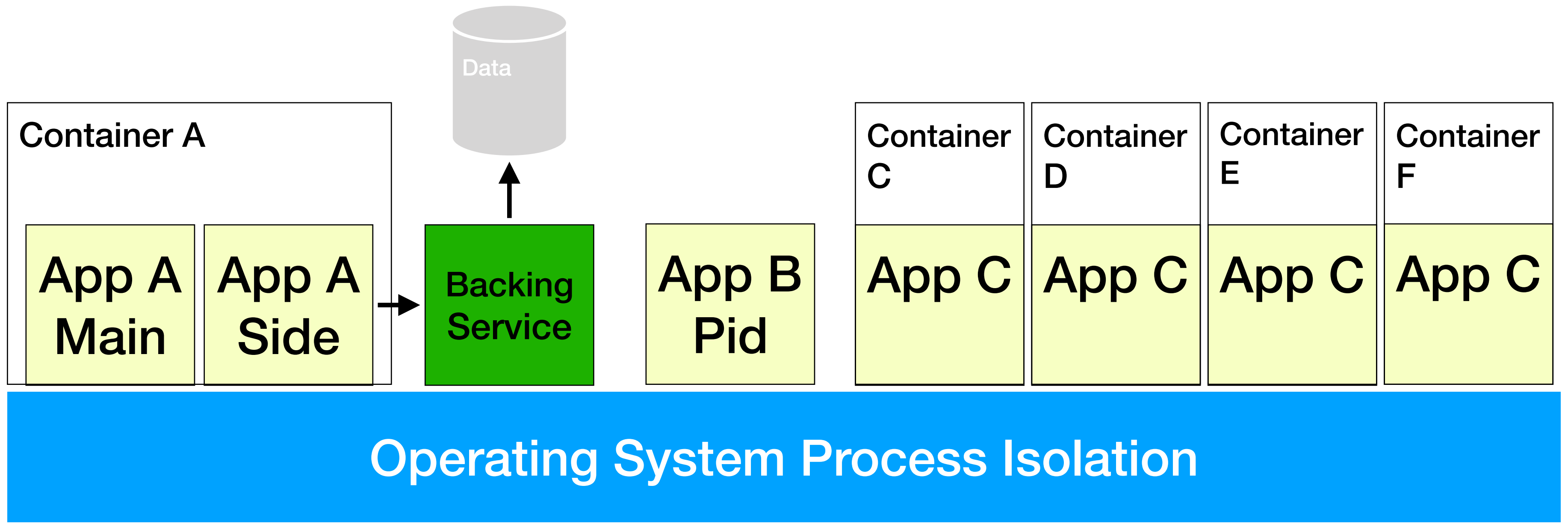
More Factors…

VI. Processes

Execute the app as one or more **stateless processes**

The app is executed in the execution environment as one or more *processes*.

Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.
The memory space or filesystem of the process can be used as a brief, single-transaction cache. For example, downloading a large file, operating on it, and storing the results of the operation in the database. The twelve-factor app never assumes that anything cached in memory or on disk will be available on a future request or job.



A process doesn't necessarily need to be container based isolated

More Factors…

VII. Port binding

Export services via port binding

Web apps are sometimes executed inside a Webserver container. For example, PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat.

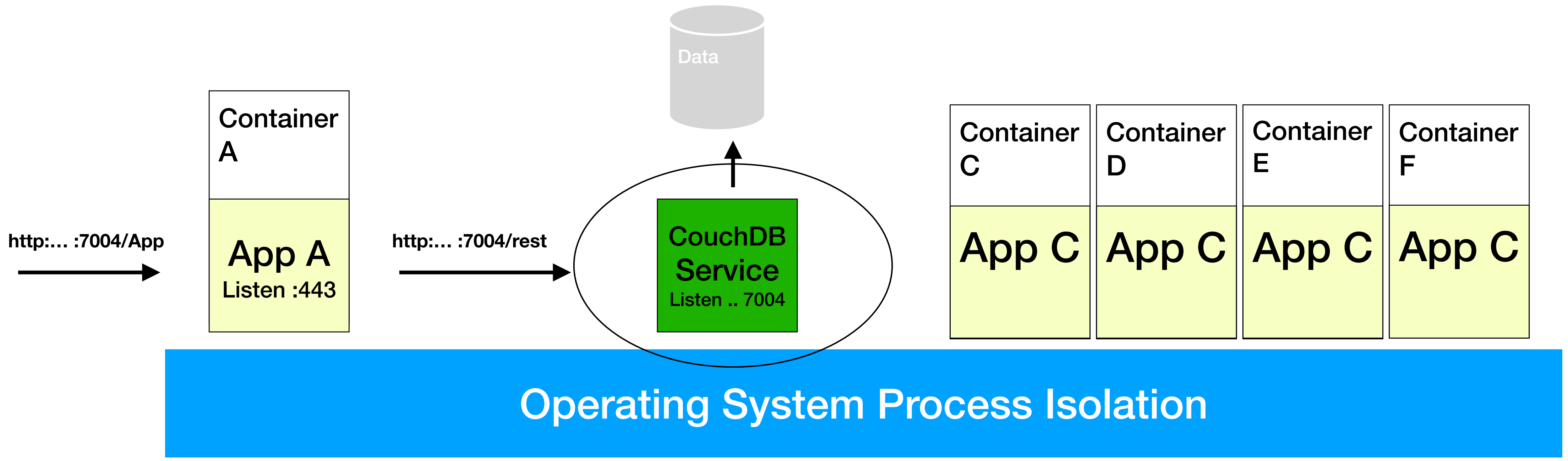
The twelve-factor app is completely self-contained and does not rely on runtime injection of a Webserver into the execution environment to create a web-facing service. The web app **exports HTTP as a service by binding to a port**, and listening to requests coming in on that port.

In a local development environment, the developer visits a service URL like `http://localhost:7004` to access the service exported by the used service app. In a production deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

This is typically implemented by using dependency declaration to add a Webserver library to the app, such as Tornado for Python, Thin for Ruby, or Jetty for Java, Express for Node.js. This happens entirely in *user space*, that is, within the app's code. The contract with the execution environment is binding to a port to serve requests.

HTTP is not the only service that can be exported by port binding. Nearly any kind of server software can be run via a process binding to a port and awaiting incoming requests. Examples include ejabberd (speaking XMPP), and Redis (speaking the Redis protocol).

Note also that the port-binding approach means that one app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config for the consuming app.



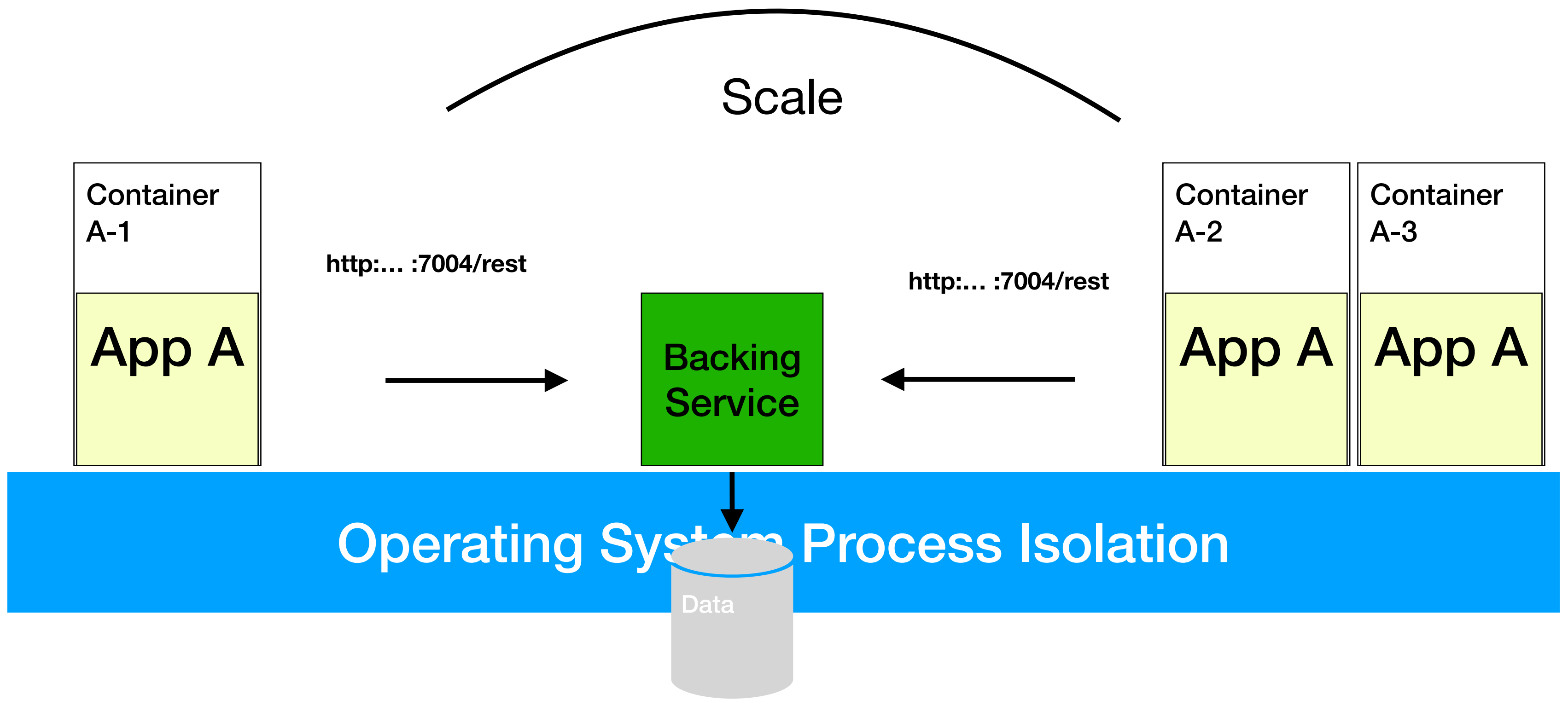
More Factors…

VIII. Concurrency

Scale out via the process model

Any computer program, once run, is represented by one or more processes. Web apps have taken a variety of process-execution forms. For example, PHP processes run as child processes of Apache, started on demand as needed by request volume. Java processes take the opposite approach, with the JVM providing one massive uberprocess that reserves a large block of system resources (CPU and memory) on startup, with concurrency managed internally via threads. In both cases, the running process(es) are only minimally visible to the developers of the app.

In the twelve-factor app, processes are a first class citizen. Processes in the twelve-factor app take strong cues from the unix process model for running service daemons. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.
This does not exclude individual processes from handling their own internal multiplexing, via threads inside the runtime VM, or the async/event model found in tools such as EventMachine, Twisted, or Node.js. But an individual VM can only grow so large (vertical scale), so the application must also be able to span multiple processes running on multiple physical machines.
The process model truly shines when it comes time to scale out. The share-nothing, horizontally partitionable nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation. The array of process types and number of processes of each type is known as the *process formation*.

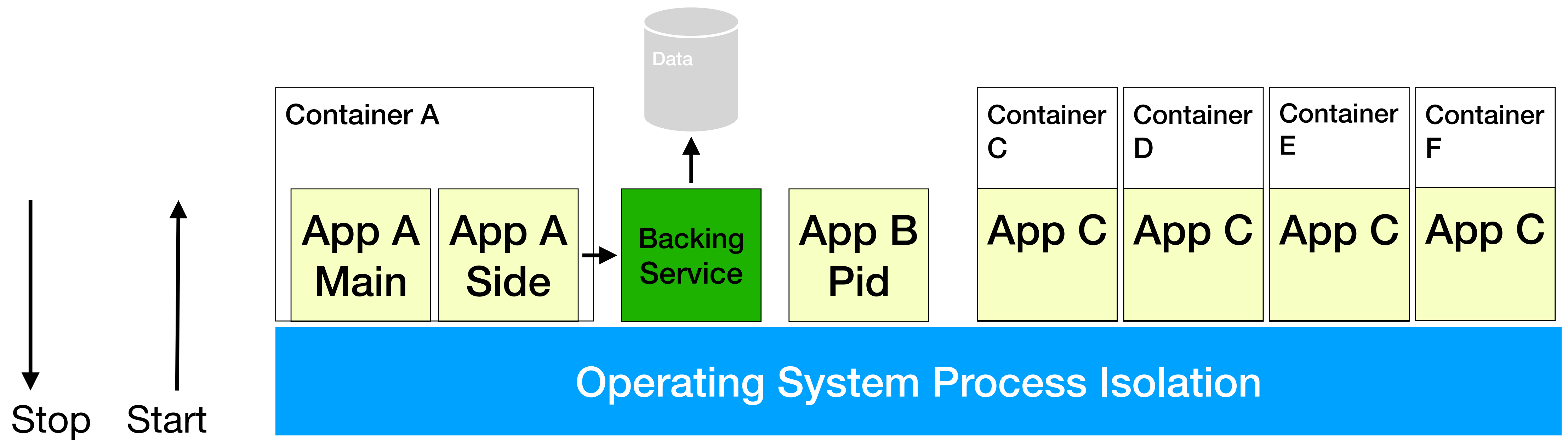


More Factors…

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

The twelve-factor app's processes are *disposable*, meaning they can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.
Processes should strive to **minimize startup time**. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs. Short startup time provides more agility for the release process and scaling up; and it aids robustness, because the process manager can more easily move processes to new physical machines when warranted.
Processes **shut down gracefully when they receive a SIGTERM** signal from the process manager. For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting. Implicit in this model is that HTTP requests are short (no more than a few seconds), or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost.



More Factors…

X. Dev/Prod parity

Keep development, staging, and production as similar as possible

Historically, there have been substantial gaps between development (a developer making live edits to a local deploy of the app) and production (a running deploy of the app accessed by end users). These gaps manifest in three areas:

- **The time gap:** A developer may work on code that takes days, weeks, or even months to go into production.
- **The personnel gap**: Developers write code, ops engineers deploy it.
- **The tools gap**: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.

The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small. Looking at the three gaps described above:

- Make the time gap small: a developer may write code and have it deployed hours or even just minutes later. **(CI / CD**)
- Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behaviour in production. **(Dev/Ops)**
- Make the tools gap small: keep development and production as similar as possible.

More Factors…

XI. Logs

Treat logs as event streams

Logs provide visibility into the behaviour of a running app. In server-based environments they are commonly written to a file on disk (a “logfile”); but this is only an output format.

Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services. Logs in their raw form are typically a text format with one event per line (though backtraces from exceptions may span multiple lines). Logs have no fixed beginning or end, but flow continuously as long as the app is operating.

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to `stdout`. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behaviour.

In staging or production deploys, each process’ stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival.

XII. Admin processes

Run admin/management tasks as one-off processes

The process formation is the array of processes that are used to do the app’s regular business (such as handling web requests) as it runs. Separately, developers will often wish to do one-off administrative or maintenance tasks for the app, such as:

- Running database migrations (e.g. `manage.py migrate` in Django, `rake db:migrate` in Rails).
- Running a console (also known as a REPL shell) to run arbitrary code or inspect the app’s models against the live database. Most languages provide a REPL by running the interpreter without any arguments (e.g. `python` or `perl`) or in some cases have a separate command (e.g. `irb` for Ruby, `rails console` for Rails).
- Running one-time scripts committed into the app’s repo (e.g. `php scripts/fix_bad_records.php`).

One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

Twelve-factor strongly favors languages which provide a REPL shell out of the box, and which make it easy to run one-off scripts. In a local deploy, developers invoke one-off admin processes by a direct shell command inside the app’s checkout directory. In a production deploy, developers can use ssh or other remote command execution mechanism provided by that deploy’s execution environment to run such a process.

A REPL (Read, Evaluate, Print Loop) enables an interactive session with your APP (e.g. a JavaScript Console) in which you can debug your code.

Referenzen

- Microservices
  - https://smartbear.com/learn/api-design/what-are-microservices/
  - https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/
  - https://aws.amazon.com/de/microservices/
  - https://microservices.io/index.html
  - https://www.innoq.com/de/articles/2016/04/microservices-agilitaet/
  - https://www.informatik-aktuell.de/entwicklung/methoden/warum-es-nicht-immer-microservices-sein-muessen.html