- -12 Factor App - -

Core Guidelines

to Cloud Ready Solutions

Rajeev Gupta
Java Trainer & Consultant

What it is any way, 12 Factor App?

- Set of principals
- A collection of guidelines
- Methodologies
- Software engineering practices

That helps one in ...

- Designing a perfect software architecture for large scale/volume use
- Running a fluent engineering process for complex ever changing business workflows
- Making sure the timely release of new features, changes, fixes
- Maintaining a software solution over time

https://12factor.net/

The main Internet resource online on the topic with all the primary details.

A little background first …

- The methodology was drafted by developers at **Heroku**
- Was first presented by **Adam Wiggins** circa 2011

https://www.heroku.com/

- The principals were laid down as foundation to the Heroku platform, but later due to their generic and platform independent implementation they are released as fundamental guidelines for any cloud ready application

-- Heroku --

- Heroku is a cloud platform as a service (PaaS) supporting several programming languages, starting from Ruby, and now Java, Node.js, Scala, Clojure, Python, PHP, and Go
- A polyglot platform, allowing developers to build, run and scale applications in a similar manner across most languages
- Heroku was acquired by Salesforce.com in 2010

How it impacts on a software solution?

Easy maintenance, and reusability	Using declarative formats for setup automation
Maximize portability between environment	Using clean contract with the underlying operating system
Cloud ready architecture	Fulfilling all the cloud based solution requirements
Maximize automation	By minimize divergence between development and production
Provides scaling and resilience	Using distributed system's design patterns
Platform independent support	Provide identical rules for all kind of software tech stacks, languages

So, what happens if I donot follow these principals?

- Will most probably end up
  - Designing a rigid software ...
    - With too much dependencies on the environment and setup
    - No room for future extensions
    - No or little flexibility towards third party integrations
  - Creating a monolithic software
  - Lossing capability to scale for large volume access and/or data processing
  - Lossing resilience to various failures
  - Decreasing software delivery efficiency and efficacy
  - Creating an inefficient engineering process and/or its execution
  - Etc ...

Who should follow these guidelines?

- Ofcourse software developers ... developing
  - SAAS (Software As A Service) solutions
  - Distributed software solutions
  - Cloud application solutions
  - Online Internet based services
    - Online games / gaming services
    - Social networking services
    - Internet application support services
  - Enterprise solutions
  - On-prem solutions
  - Large scale, and/or highly resilient solutions

The 12 factors … The list

I	**Codebase** One codebase tracked in revision control, many deploys	VII	**Port Binding** Export services via port binding
II	**Dependencies** Explicitly declare and isolate dependencies	VIII	**Concurrency** Export services via port binding
III	**Config** Store config in the environment	IX	**Disposibility** Maximize robustness with fast startup and graceful shutdown
IV	**Backing Services** Treat backing services as attached resources	X	**Dev/Prod Parity** Keep development, staging, and production as similar as possible
V	**Build, Release, Run** Strictly separate build and run stages	XI	**Logs** Treat logs as event streams
VI	**Processes** Execute the app as one or more stateless processes	XII	**Admin Processes** Run admin/management tasks as one-off processes

I Codebase

One codebase tracked in revision control, many deploys

Do's

- An app codebase must be stored in a repository managed by a VCS such as Git
- Must be 1-to-1 correlation between the codebase and the app
  - Means, a distributed app can have multiple codebases, one for each distributed module
- **Fact:** A codebase can trigger multiple deploys ... where a deploy is a running instance of app ... deploys shares code but at different versions

Don'ts

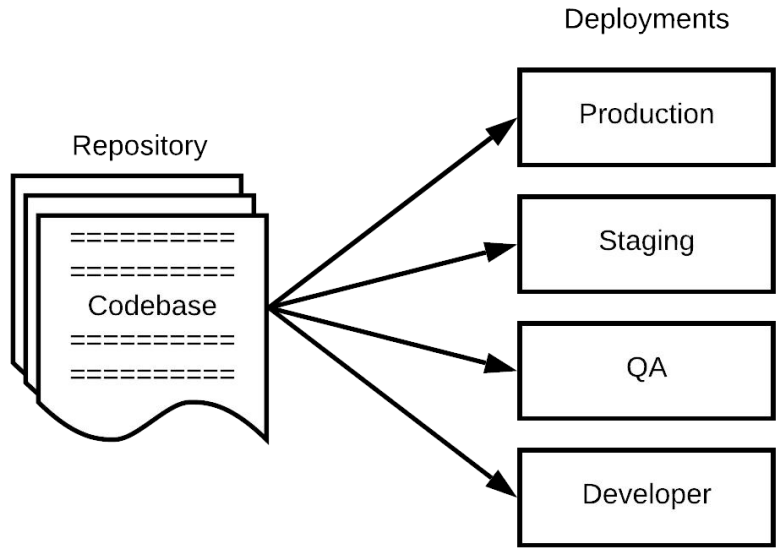
- Multiple apps cannot share code ... such code must be factored out as shared libraries
- Hence there will be no separate codebase for each deploy

Definitions ...

Codebase ➲ The source code for the application that will be compile or interpreted before execution.

Repository (repo) ➲ Codebase managed within a VCS (Version Control System) system.

Deploy ➲ An installation of the app with a specific configuration for a specific use case and/or audience.



II Dependencies

Explicitly declare and isolate dependencies

Do's

- The software codebase should declares all of its dependencies, along with their specific versions (if required) in a dependency declaration manifest
- The software codebase during build process should uses dependency manager, aka packaging system, to fetch all required dependencies from their sources and maintain them in a local repository
- A dependency isolation tool is used to prevent accidental import of unwanted dependencies or their other versions from the shared environment
- The full and explicit dependency specification is applied uniformly to both production and development.
- **Fact:** In Java world, Maven, Gradle are the dependency managers

Don'ts

- A software codebase does not relies on the implicit existence of dependencies in the build environment

Definitions ...

Dependency ➲ Any library module used by the app which is either developed/managed by the developer or by a 3rd party vendor requires. A dependency is identified by its name, version, vendor, and download source.

Dependency Manifest ➲ A document within the codebase that lists all the dependencies for the app. E.g. pom.xml, build.gradle, gemfile, etc.

Dependency Manager ➲ A separate module which is part of the build toolchain that processes the dependency manifest, manages and download all the required dependencies from their sources before the codebase is taken to the compilation. E.g. Maven, Gradle, PIP, etc.

Dependency Isolation ➲ A separate module which is part of the build toolchain which provides dependencies isolation for each app within an environment such that a dependency within an isolation (virtual environment) cannot be used by another app from the other isolation. E.g. Virtualenv, etc.

III Config

Store config in the environment

Do's

- An app requires strict separation of config from its code
- Config can also be made available from a file separately define in the system registry like /etc away from the app, but it is not a generic way of passing config, as it is not possible in other non-Linux environments like Windows
- An app should get config from the environment as enviroment variables as it is much more platform independent way
- Various app packaging and containerization runtimes, and orchestration systems provides facility to define config for the app thru environment variables based on the deploy type

Don'ts

- An app should not store config within code files, or as separate files within codebase
- An app should not retrive config with tech stack and/or language specific APIs
- An app should not use config groups like production, staging, qa, dev config as it make it quite rigid and prune to various issues

Definitions ...

Config ➲ A collection of parameters that hold values that are used by the app's codebase in 1) making strategic decisions, 2) integration with backing services, 3) communication, 4) processing requests, 5) formating outputs, etc. An app’s config is everything that is likely to vary between deploys (staging, production, developer environments, etc).

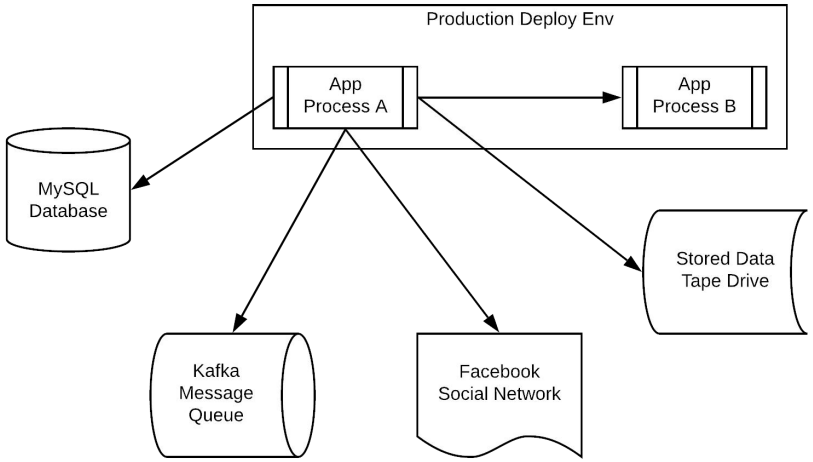
Evironment Variable ➲ A collection of key-value pairs defined in the environment. A process receives these variables from it's parent process (creator). Hence these variables are not being fetched by the process, but rather handed over from it's parent that get those from it's parent and so on. These variables can hold all sort of data like information of the OS, environment, dependencies, and configuration for the process itself.

IV Backing Services

Treat backing services as attached resources

Do's

- The app should always assume backing services as resources whom life cycle management is the responsibility of the app itself
- If an app process utilizes the services of another process, then the other process becomes the backing service for the first one
- The code for an app should make no distinction between local and third party services ie resources managed by dev vs managed by others
- The service access parameters must be part of the app's config so that each deploy can have a choice to attached to other service of same type
- An app should have logic and functionality to handle each cases regarding the resource attachment and detachment scenarios



Definitions ...

Resource ➲ A helper acquired thru request or created by the app to perform certain task.

Backing service ➲ A backing service is any resource the app consumes over the network as part of its normal operation; e.g. datastores, messaging/queueing systems , SMTP services, and caching systems. A backing service can be managed by the app developer e.g. database, or by some third party e.g. email gateway. A backing service is pretty much different from an embedded resource, as with the later one, one can have assumption regarding availability and resilience. A backing service can be attached and detached at it's own will or due to other uncontrollable circumtances.

Don'ts

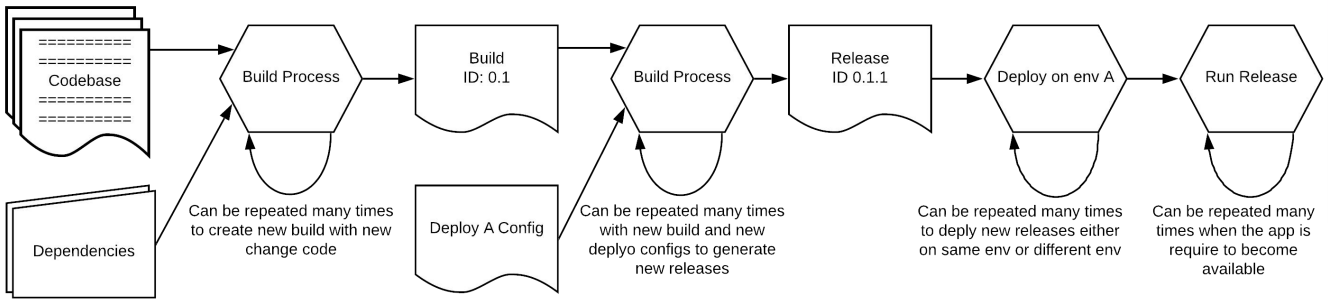
- An app must not assume a resource (either local or third party) as always available, and/or resilient to failures

V Build, Release, Run

Strictly separate build and run stages

Do's

- **Fact:** An app codebase transform into a deploy thru three steps; 1) build, 2) release, and 3) run
- Hence all the stages must be taken as separate processes that can be run on their convinent times; “build” when the developer pushes new changes, “release” when deploy is required, and “run” when the app is required to become available
- The “run” can run multiple times with the same “release” on the environment, “release” can run multiple times with same “build” for different deploy type configs, and “build” can run multiple times
- The “build” process will use its own ID to tag build artifacts, whereas “release” process will either use its own ID or add suffix to the build ID
- The build and release artifacts should be stored separately for future access and reference



Definitions ...

Build ➲ The build process takes the code from the repo, acquire all the dependencies from their sources, and generates an app build that can be use for release. The process can take code thru compilation and optimization processes.

Release ➲ The release process takes the app build and based on the deploy type acquire the necessary app config and generates a release build, that will be ready for execution in its deploy environment.

Run ➲ It is a process of actually executing the app, and creating its process(es).

BUILD = codebase + dependencies + assets
RELEASE = **BUILD** + config
RUN = run process against **RELEASE**

Don'ts

- An app must not have a unified process that can build, create release, and does the run on the target
- A CI pipeline must not implements the continous delivery and continous deployment phases.

VI Processes

Execute the app as one or more stateless processes

Do's

- **Fact:** An app is executed in its deployment environment as one or a collection of processes
- A business workflow must be designed as a process instead as a thread
- An app process(es) must be designed to run statelessly and share nothing ... Any state that requires persistent must be handled by the backing services (e.g. database)
- An app process should identify the incoming request's session or context rather than assuming the previous one
- Should save the sessions in a database rather than holding it in its memory
- Uses single transaction only caches

Don'ts

- An app's process not necessarily get all the subsequent requests from the same session/context
- An app's process must not rely on sticky session feature
- An app's process cannot assume itself as fault free, and thus can get terminated on number of conditions other than the code itself

Definitions ...

Process ➲ The running of a release in a deploy environment.

App State Management ... The app is designed such that,

- if it's process crashed, terminated abruptly, there won't be any loss of state or data except the processing of the current requests.
- if app's multiple instances are be launch in parallel, they will not need to be aware of each other
- there will be no synchronization requirements between it's process instances.
- any instance can be brought down any time without any need to backup any state or service data.

With Stateful Design ...

- The app processes reduces the overall app resilience.
- Increases the process liability.
- Makes the process unexpendable, reduces desposibility.
- Requires to have state syncing and consistency assurance procedures.
- Reduces the process overall efficiency and efficacy.
- Makes an app harder to upgrade in deployment environments.

VII Port Binding

Export services via port binding

Do's

- An app should be fully self-contained; means it does not requires runtime injection of a web server/container
- An app should only bind to a TCP/UDP port rather than the complete address set i.e. IP address and TCP/UDP port.
- An app port binding should be configurable; not hard coded in the codebase
- **Fact:** Apps with port binding will bring flexibility of getting run within the same environment provided that all the other processes are binded to different unique ports
- An app thru dependency declaration, should add an embedded web server/container to the release bundle that gets port binding config from the environment
- **Fact:** Apps with port binding and embedded web server/container enables to have a distributed architecture of multiple processes acting as backing services to others

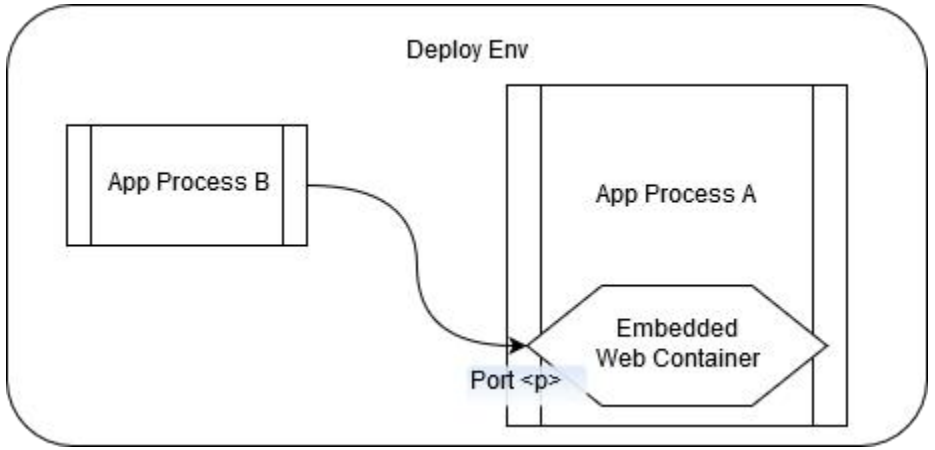
Don'ts

- An app's process must not tie up itself with the IP address along with port

Definitions ...

Port ➲ A virtual interface for the TCP/UDP protocol on which the app communicates with the other app.

Port binding ➲ A procedure in which an app process associates its communication interface with a TCP port, and other processes then establish comm link to the first process on its binded port.



VIII Concurrency

Scale out via the process model

Do's

- An app should run their business workflow at the process level; not at the thread level
- The app should use thread level concurrency for specific smaller tasks like communication, accessing data or resources, or crunching numbers or data into information
- Hence the main business workflow concurrency is achieved by instantiating more processes on same or separate environments, and within the business workflow various parallelism among the subtasks is acheived thru threads
- **Fact:** Concurrency thru process enables isolation of one business workflow from others; gives failure and starvation protection to each business process from others; provides an effective and efficient way of scaling load, and maximizing resilience towards failures
- An app should be configured to manage its output streams, crash recovery, auto-start policy on system restart/reboot

Don'ts

- An app process neither should deamonize nor create PID files; but rather behave like normal processes that can be controlled from the central process manager like systemd in Linux

Definitions ...

Concurrency ➲ Ability to run similar tasks in parallel to load balance, and to scale up the solution capability to take on more load or data volume for processing. Concurrency can be achieve either at process level or at thread level.

Thread ➲ A thread is a unit of execution within a process that can run independently and in parallel. A process can have from one to *N* number of threads, all running in parallel, and can share and access each other memory and resources.

Process level concurrency ➲ A type of concurrency where the task parallelism is achieve by running two or more processes, and each process executes the tasks.

Thread level concurrency ➲ A type of concurrency where the task is executed in a thread, and multiple threads are spawned to parallelize tasks execution within a single process boundry.

IX Disposibility

Maximize robustness with fast startup and graceful shutdown

Do's

- An app process should be design in a way that it can be tear down, terminated, and restart again in a moment's notice
- An app process should have minimum startup time
- An app process should shut down gracefully on terminate signal
- An app process should also be robust against sudden failure, and should be architectured to handle unexpected, non-graceful termination without lossing the inprogress workload requests
- **Fact:** An app with the above characteristics enables fast elastic scaling, rapid deployment of code and config changes, and robustness of production deploys
- For a worker process, the jobs taken from the queue should have a state thru which the process can mark a job in progress, and later delete the job or mark it done when it is completed; whereas in case of a job in progress state for more than a predefine time, can be retried by other worker process

Definitions ...

Disposibility ➲ An app's process characteristic that makes it expandable which further enables the process manager and system orchestrator to start and teardown these app's processes on demand without going into various issues of data/state synchronization, resource integration, etc.

An app ...
⟶ with stateless design,
⟶ providing business workflow execution at process level,
⟶ and that takes configuration from the environment
are more elastic in nature and thus can be easily terminated and restarted immediately without considering their previous runtime state.

Don'ts

- Most of the usual resources like memory, open files, pipes, FIFOs are automatically released when a process terminates in any condition. However there are resources that requires an explicit release procedure like TCP sockets. In today's world, the frameworks usually take care of these resources, but if the app is handling the sockets, then it should take care of its graceful shutdown.

X Dev / Prod Parity

Keep development, staging, and production as similar as possible

Do's

- An app and its engineer process should be design to favor continous integration, continous delivery, and continous deployment
- An app's engineering process should minimize the dev and production gap to the maximum by
  - Making features/changes/fixes small so that they can be developed in less time, and requires lesser time in testing
  - Making release deployment more efficient and effective so that new builds with new features/changes/fixes can be deployed quickly on the target environments
  - Trying to make dev and production environment as similar as possible by using infrastructure that can spawn or create environments from a template on will to create any required environments like dev, QA, staging, etc.
- Use mock layers for the backing services for an apps to do functional testing

Don'ts

- Donot allow local deploys for integration testing; always use spawnable environments

Definitions ...

Dev -> Prod parity ➲ An app usually have gaps between development and production due to the following reasons:

- Development completion period ... Time taken by the developers to complete a feature, a change or a fix, and have it reviewed and pushed to the codebase.
- Release deployment ... Time taken by the devops / CD to deploy the new build on production site that often include testing, reviewing, etc.
- Environment dissimilarities ... The changes between the dev and production environment like changes in tech stack, servers, backing services, or their versions, etc.

Continous integration ➲ A process that automates the task of creating builds, validating the build thru unit tests, and automated functional testing.

Continous delivery ➲ A process that automates the task of creating releases for production enviroments, validating the release thru automated integration and UAT testing on staging environment.

Continous delivery ➲ A process that automates the task of deploying the tested release on production environment with a strategy the favors smooth transition from old version to the newer one.

XI Logs

Treat logs as event streams

Do's

- **Fact:** Logs provide visibility into the behavior of a running app; have no fixed beginning or end, but flow continuously as long as the app is operating
- **Fact:** Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services
- **Fact:** Logs in their raw form are typically a text format with one event per line
- An app should write its logs to its output stream
- An app output stream should be configurable from the environment so as to redirect logs as per requirement
- An app's log must follow a definite pattern with one line per log entry
- An app's log should be created as they will be pushed to an analyzer thru an aggregator for further analysis, and event processing

Don'ts

- An app should never concern with managing its logging stream
- An app should never output its logs to files

Definitions ...

Logs ➲ Logs are the explicitly printed/written statements from the app's code to document start and end of events and situations. These statements are outputted to a single resource in a uniform structure/pattern in textual format; usually one statement per line. The output resource is hoped to keep these log statements for an enough longer period where they can be further process and reviewed.

Loging ➲ A processing of outputting log statement to the logging device/resource.

Event ➲ Events are somethings that happens at a given time or place representing some case, consequence, or an effect.

Streams ➲ A flow of data units of same structure and/or size that undergoes same/identical processing.

Logs as event streams ➲ A stream of logs statements that represents chronologically sorted events in the processes. Each statement must be uniform in structure.

XII Admin Processes

Run admin/management tasks as one-off processes

Do's

- **Fact:** An app often comes with various one-off administrative processes for housecleaning, and maintenance tasks like cleaning temporary/used/deleted/malformed data, etc
- **Fact:** These one-off admin processes are run within the production environment where the actual apps deploys and runs using the app's config
- The one-off admin processes should be implemented as separate processes from app's own processes
- Each one-off admin process should have its own separate app
- The one-off admin processes apps can be bundled with the actual app bundle, but they should be kept separately from the normal app, and their executable access should also be limited to fewer users

Don'ts

- Hence a one-off admin process should not be made part of the actual app that can be design to run on certain time and situation

Definitions ...

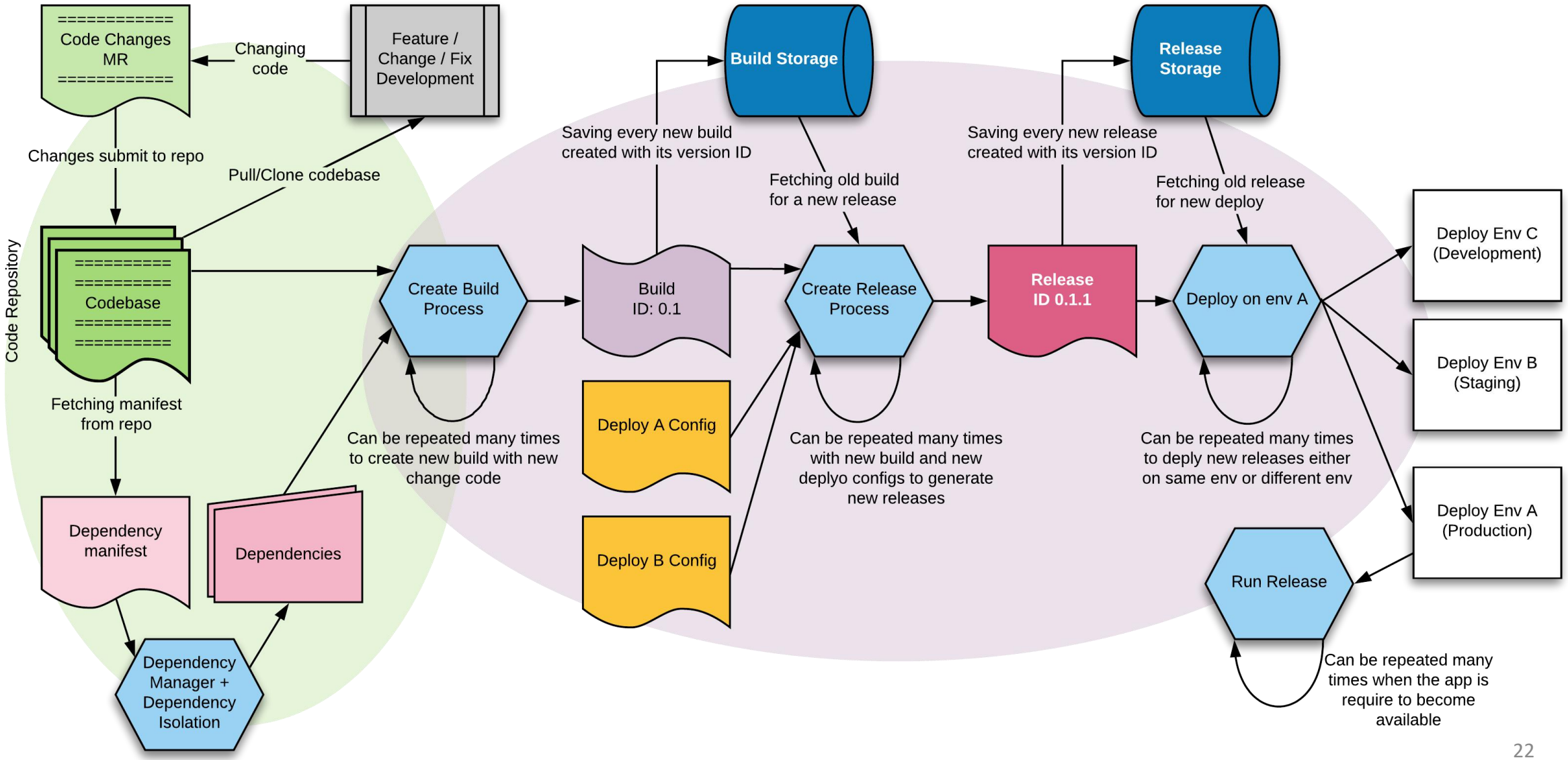
One-off process ➲ A kind of process that terminates when it is done with its specific task. Means it doesnot runs forever. It runs separately from the actual app, either by the user having permission, and/or by the system on certain events (as scheduled job).

Admin processes ➲ Admin processes are one-off processes that have no direct relation to any business workflow within the app, means they donot perform any app's business responsibility, but rather are executed to support them or make them more efficient. E.g. processes for data migration (import/export), database cleaning, process maintenance, etc.

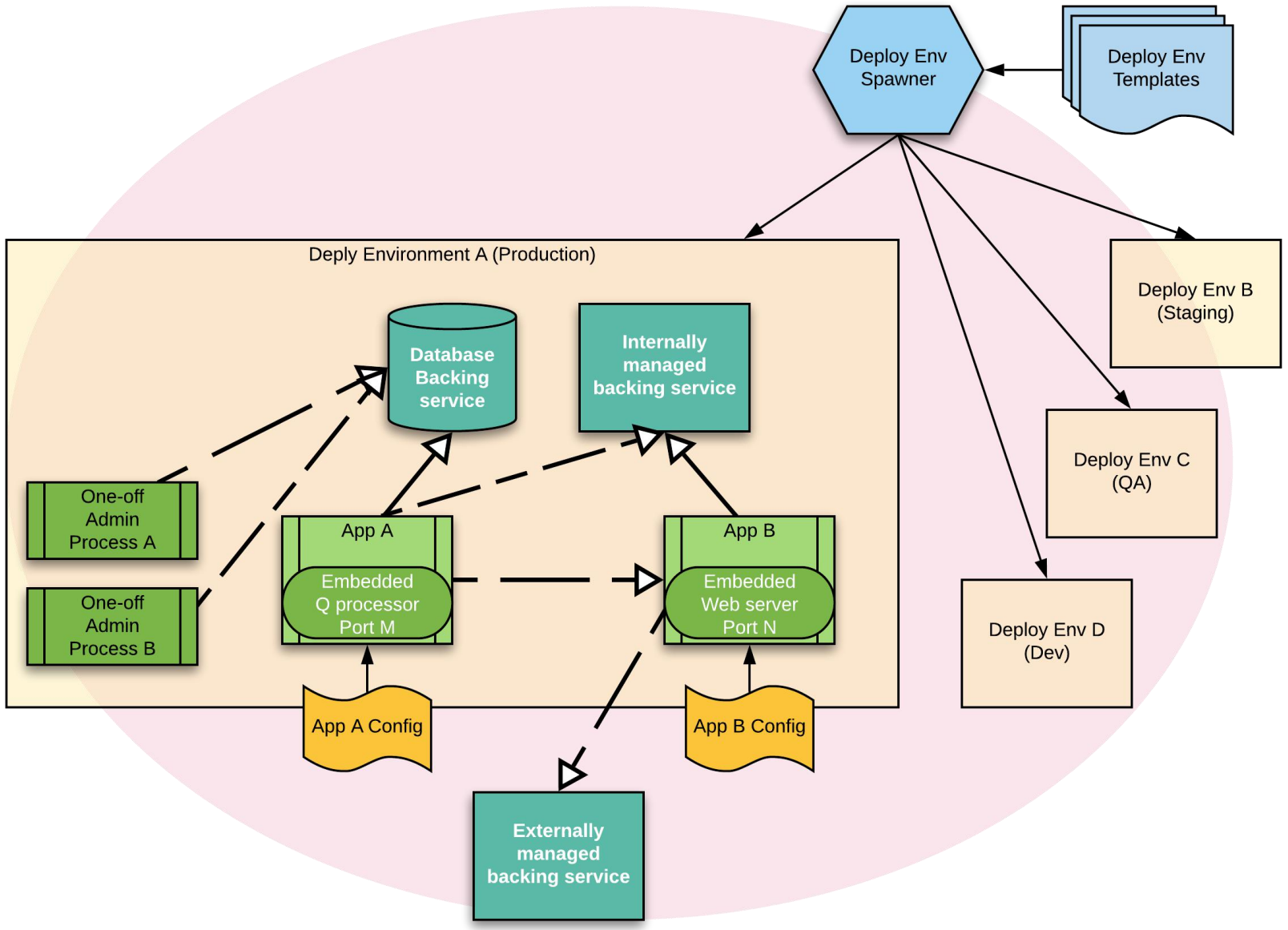
Benefits of having admin processes ...

- They have very little code, only for the task at hand.
- They can be executed anytime and in any order when there is a need.
- There execution is isolated from the app's own processes, hence any failure either in admin process or app's own process donot impact other; and similarly either of them cannot starve other, neither there will be any security issues.
- Since admin processes are separate, they can by-pass various business logic and validations on data access for speedier processing.

12 Factor App Engineering Process



12 Factor App Deploy Environment



Deploy env templates ➲ that can be use to clone new environments

Env spawner ➲ that can create new env from the templates by cloning them and applying declared transformations

4 deploy envs ➲ which are production (blowup), staging, QA and dev, all cloned by the spawner from their respective templates

App A and App B ➲ are actual business apps with an embedded listner (queue consumer, web container respectively) binded to specific ports M and N respectively

App A and App B Configs ➲ are the externalized configurations provided from outside and injected into App A and B as env variables at deploy time before execution

One-off Admin Processes A and B ➲ are the two admin apps separate from the actual business apps A and B, working with the actual backing services

Backing services ➲ are divided into two; managed, and externally managed, are those managed by the dev and by the 3rd party respectively. Here business App B is also a backing service to App A.

12 Factor App Software Design

Deploy environment ➲ is the required infrastructure for app deployment with (production) configuration.

App A and App B ➲ are the actual business apps that implements certain business concerns or workflows.

App A and App B Configs ➲ are the configs for the respective business apps A and B provided separately with the environment based on the environment type.

One-off Admin Process A ➲ Is one of the one-off admin app run separately from the business apps but shipped with the same build bundle and utilizes the same backing services, as well as the business app config.

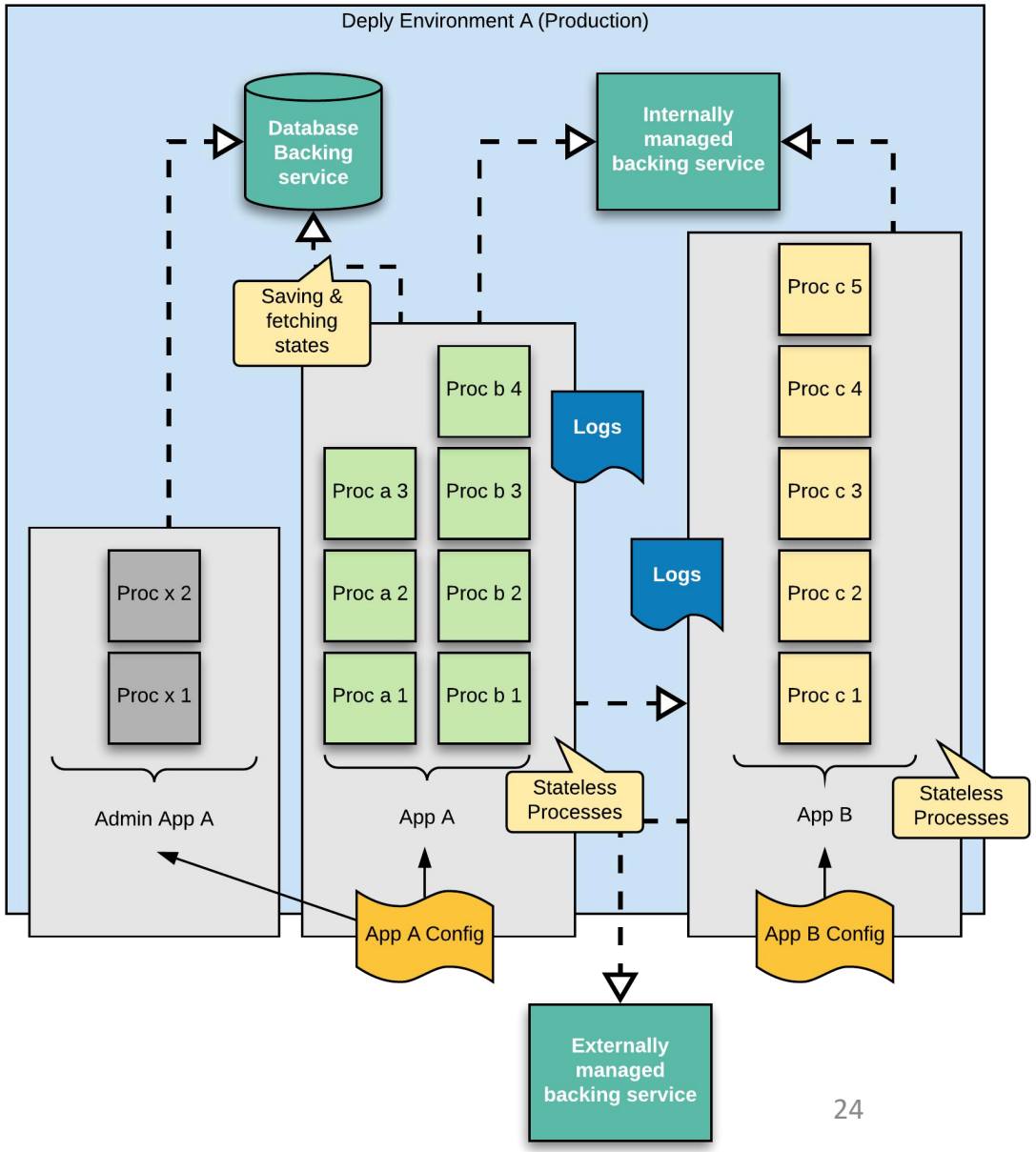
Apps processes ➲ The business apps based on the single responsbility principal (SRP) can further breaks into a set of processes where each process implements a single responsibility. Here App A is compose of two processes, whereas App B is compose of one process. An admin app can also mapped to one or more processes. Here it is mapped to one process.

Concurrency ➲ The parallelism in the processing of multiple business workflows is achieved by running two or more processes within the app based on the load requirement, and capability. Similarly admin app can have two or more process instances based on the load requirements. These process instances can be run on same or different computing node.

Statelessness & state management ➲ is one of the core design element in the business apps. Since an app is design to be stateless, all its process instances are then stateless. Hence due to that the env manager can tear down an instance or start a new one based on the requirement without worrying about the state management. If the process working with the state (as of App A), the changed state is immediately pushed to the backing service (database) that keeps the states for all the instances. Hence with this rudimentary form of state management, the instances get free from the state synchronization and state consistency management.

Logging ➲ is done by all the process instances of each App A and B, and these are aggregated into a single stream and reroute to an admin console and/or to an analysis software.

Backing services ➲ are divided into two; managed, and externally managed, are those managed by the dev and by the 3rd party respectively. Here business App B is also a backing service to App A.



So, it promotes ...

1) An engineering process/strategy that favors ...

- cleaner and leaner codebase utilizing repository capabilities to seggregate deploy type code
- quicker delivery to deploys; every deploy is like hot-fix
- quicker fixes to bugs and issues
- enhanced testing quality by enabling devs to test on production similar environments

2) A software deployment strategy that enables ...

- easy setup of deploy environments thru the use of spawnable environments
- heterogeneity in deployment environments
- similar or identical dev and production environments
- flexibility in deployment strategy
- efficient configuration injection
- isolation between actual and admin processes

3) A software architecture that supports ...

- extensibility in apps
- single responsibility apps
- statelessness
- concurrency thru scaling processes horizontally
- tech agnostic logging that can be relayed to any other system
- high disposibility; the ability to start and shutdown immediately

It enforces distributed architecture design patterns ...

#	Design Patterns	
1	Automation	Enforced thru separate build, release, run processes, and minimized dev-prod parity. This includes ... • Continous integration, continous testing, continous delivery, continous deployment
2	Software architecture	Enforced thru architecture characteristic requirements like ... • Software compartmentization • Responsibility segregation • Stateless, self-contained processes
3	Deployment patterns	Enforced thru the use of spawnable environments that means using app containers or virtual machines along with env orchestration solutions
4	Configuration externalization	Enforced to make configuration external to the app and providing it from external source mainly as environment variables
5	Communication resilience patterns	Enforced to handle all failure cases with backing services by implementing patterns such as: • Retry, timeout, fallback, circuit breaker, fail fast, isolation, bulkheads, location transparency
7	Service tracing and log aggregation	Enforced thru logging to process output stream and letting the environment to route the stream to external processors (analyzers)

Application containers

Definition

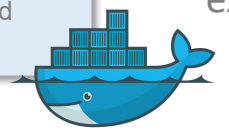
➲ A container being a standard unit of software, packages up code and all its dependencies along with the required configuration so the application runs quickly and reliably from one computing environment to another.

- Hence a container is just an environment for the target application that contains the application, its runtime, its dependent libraries, configurations, and other resources.
- A container usually shares OS and its system libraries with the host environment.
  - In case of having containers on a different OS, then emulation is being used. For instance setting up Linux based containers on Windows OS.
- An application in a container with the matching host OS gives almost the same performance as it is deployed directly on the host environment.
- An containerized application can be serialized in an image, and then the image can be replicated and deployed on other hosts as per need.
  - Hence a service containerized can be saved as an image, and then can be used later for repeated number of deployments.

- Containers provides isolation to the application such that
  - an application cannot see others in the other containers
  - an application cannot impact / affect others applicaitons
  - an application cannot access resources in other application containers
  - an application can have dedicated runtime and dependencies for specific version

For example ...
- Docker containers
- CoreOS Rkt (Rocket)
- Mesos containerizer
- Etc

Docker ➲ is an application container software that not only provides the capability of a containerization, but also management. Docker is developed primarily for Linux, where it uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs).



Application containers help implements some of the principals of 12 factors ... which are:

- **Config** → by providing a mechanism to define env variables as part of the container definition that becomes available to the containerized app when launched.
- **Backing service** → by providing a strategy to deploy all internal backing services as containerized apps so that they can be redeploy again and again.
- **Processes** → by providing process manager thru which each app process can be managed externally.
- **Port binding** → by providing port mapping facility thru which the containerized app ports can be mapped to other ports.
- **Dev/Prod parity** → by providing a mechanism to create images of builds and then spawn containers from them on need basis. Hence easing the process of creating environments.
- **Logs** → by providing standard output stream collection facility; hence all the standard output can be routed to external processors.

Container Orchestrators

Container orchestrators help implements most of the principals of 12 factors. Since they manages containers and their images, hence all the principals enforced by containers (discussed previously) are also being enforced by them indirectly. The principals they support are:

- **Backing Services** → by providing functionality to define backing services, and their availability requirements that includes how scalable and resilient they should be. Moreover using advance features one can easily define integrations to these backing services. And reusing them among two or more environments.
- **Build, Release, Run** → by providing support for various strategies for deploying, and executing containerized apps.
- **Processes** → by providing app definition thru which ops can declare apps, and how their processes will be created as containers and how they will be managed.
- **Concurrency** → by providing controls to define process instantiation, scalability, and auto restart capabilities.
- **Disposibility** → by providing support for process startups, shutdown, and their performance measurements.
- **Dev/Prod Parity** → by providing infrastructure to define deployments, and controls to manage them as per need.
- **Logs** → thru functions to capture logs from standard output streams and route them to admin consoles.

Definition

➲ Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments with multiple hosts/nodes.

Software teams use container orchestration to control and automate many tasks:

- Provisioning and deployment of containers
- Defining communication paths
- Defining resilience schemes
- Designing security parameter
- Redundancy and availability of containers
- Scaling up or removing containers to spread application load evenly across host infrastructure
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Allocation of resources between containers
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it

For example ...

- Kubernetes
- Docker Swarm
- Apache Mesos
  - (and Marathon)

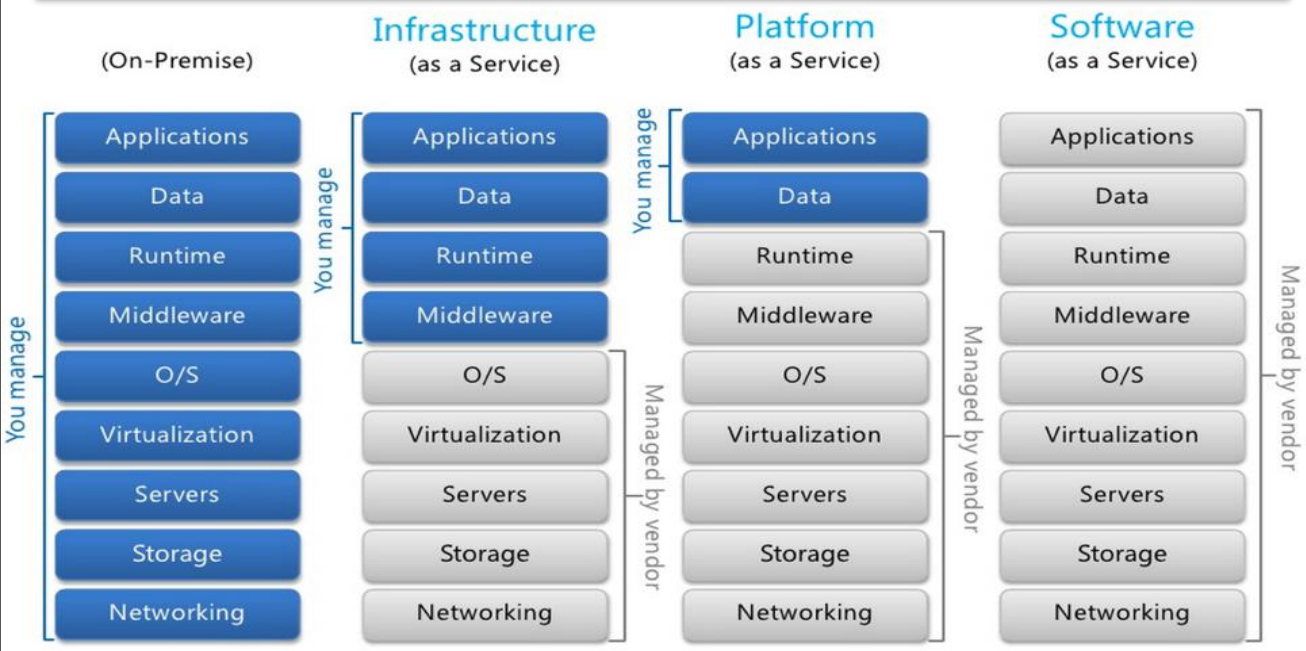
PAAS Solutions

Definition

➲ Platform as a Service (PaaS) or Application Platform as a Service (aPaaS) is a category of cloud computing services that provides a platform with environments set up with app runtime allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.

The advantages of PaaS are primarily that it allows for higher-level programming with dramatically reduced complexity; the overall development of the application can be more effective, as it has built-in/self up-and-down ramping infrastructure resources; and maintenance and enhancement of the application is thus easier.

Disadvantages of various PaaS providers as cited by their users include increased pricing at larger scales, lack of operational features reduced control, and the difficulties of traffic routing systems.



PaaS solutions usually build on top of container orchestration solutions, and with that they indirectly provides support for all the 12 factors supported by container orchestration solutions discussed previously.

PaaS solutions provides their own CLI based toolchain thru which devs and ops can further impl. and/or improve 12 factors which are:

- **Codebase** → by integrating code repo with the app deploys and hence making continous deployment an easy process, manageable from the PaaS CLI client.
- **Dependencies** → by handing over the dependency management to PaaS.
- **Config** → by binding config for an app with PaaS own configuration using CLI.
- **Build, Release, Run** → by providing a simple easy way for creating build and release artifacts, and then doing the deploy and run using the PaaS CLI.
- **Disposibility** → by providing an app framework that make sure of the disposibility characteristics for an app.
- **Dev/Prod Parity** → by providing support thru CLI for deploy of app on various environment types with least parity.

(Cont.) PAAS Solutions

Vendor	Service	URL
Google Cloud Platform (GCP)	App Engine	https://cloud.google.com/appengine/
Amazon's AWS	Elastic Beanstalk	https://aws.amazon.com/elasticbeanstalk/
Microsoft's Azure	App Services	https://azure.microsoft.com/en-us/services/app-service/
IBM	Bluemix (Cloud Foundry)	https://www.engineyard.com/
VMware	Pivotal Cloud Foundry	https://www.vmware.com/solutions/photon-pcf.html
SAP	HANA Cloud Platform	https://cloudplatform.sap.com/index.html
Redhat	Openshift	https://www.redhat.com/en/technologies/cloud-computing/openshift
Orcal Cloud Platform (OCP)	Oracle Cloud PaaS	https://cloud.oracle.com/en_US/tryit
Saleforce	Force.com	https://www.salesforce.com/products/platform/overview/
Pivotal	Pivotal Web Services	https://run.pivotal.io/
Heroku	Heroku Dynos	https://www.heroku.com/dynos
Software AG	LongJump	https://www.crunchbase.com/organization/longjump
CenturyLink	AppFrog	https://www.ctl.io/
Engine Yard	Engine Yard	https://www.engineyard.com/
Platform.sh	Platform.sh	https://platform.sh/

Taken from lists:
https://www.techradar.com/news/best-paas-provider
https://www.cbronline.com/cloud/10-of-the-best-paas-providers-4545381/
https://stackify.com/top-paas-providers/

Serverless Solutions

Definition

➲ Serverless computing is a cloud-computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources based on the incoming computation requests. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity. It can be a form of utility computing.

It is one step above PaaS (discuss previously) where the infrastructure is created and starts execution from the time of app deployment, whereas in serverless solutions the infrastructure is only spawned/used when there is a computation request/need. And hence there is often a limitation to terminate the computation within a predefined timeout otherwise the app execution is halted by the platform.

Serverless computing can simplify the process of deploying code into production. Scaling, capacity planning and maintenance operations may be hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as microservices. Alternatively, applications can be written to be purely serverless and use no provisioned servers at all.

Difference from PaaS solutions ➲ The major difference is how the infrastructure is provisioned. In PaaS, the infrastructure (VMs or containers) is provisioned when the app is deployed, whereas in serverless deployment, it is provisioned when the app is requested for computation. Hence based on this, the billing in PaaS started from the time of app deployment, whereas in serverless the billing is done when the app is computing on a request. Moreover the infrastructure in PaaS remains intact through out the app deployment timeperiod, whereas in the case of serverless deployment the infrastructure is tear down once the app computation is completed/ended/terminated.

The other difference is on the resource limitations aspect. In PaaS, usually there is no limitation to the app other than memory and/or storage (sometimes), whereas in serverless there is a lot of limitation on CPU cycles, memory utilization and storage. Hence the app in serverless is expected to complete its processing in a definite time period utilizing not more than declared memory with very little or no harddisk storage.

Serverless solutions provides same supports for 12 Factor apps as PaaS solutions and in some principals exceeds such as:

- **Concurrency** → by parallelizing app's module instead of the whole app, and as per need bases.
- **Disposibility** → by futher breaking the app into self-contained app-functions (modules) that can start and terminate independently on need bases.
- **Dev/Prod Parity** → by making deployments more simple, and lightweight with ease of provisioning based on demand.
- **Admin Processes** → by enabling admin tasks to be implemented as app-functions that can run on separate provisioned context when there is a need.

Some serverless platforms

→ Google's GCP - **Cloud Functions** (https://cloud.google.com/functions/)
→ Amazon's AWS - **Lambda Functions** (https://aws.amazon.com/lambda/)
→ Microsoft's Azure - **Functions** (https://azure.microsoft.com/en-us/services/functions/)

These serverless platform's enables developers to code their apps as discreate, small modules; each one is self contained, and can be executed independently, and can fetch and save data/state from/to a database. Cloud platform often also provide orchestration tools to integrate these serverless apps to various other apps and events.

Programming languages ➲ Java, JavaScript, Python, GoLang, Ruby, etc.

Tools ...

Type	Softwares/Libraries/Frameworks	12 Factor App Support	
VCS	Git, CVS, SVN, etc	Codebase	Use to store app's codebase, and provide management for version control.
Dependency managers + Dependency isolators	Maven, Graddle, PIP, VirtualEnv, etc	Dependency	Use to manage app's dependencies by keeping all required dependencies and updating them as their new versions get available.
Continuous integration	Jenkins, TeamCity, etc	Build-Release-Run	Use to host and run pipelines that are responsble to create builds and releases for certain deploys.
Continuous deployment	Spinnaker, XL Deploy, DeployBot, AWS CodeDeploy,	Build-Release-Run, Dev/Prod Parity	Use to deploy releases on target environments.
Containers, Container orchestrators	Docker, Rkt, Kubernetes, etc	Build-Release-Run, Processes, Concurrency, Dev/Prod Parity, Port Binding	Use to build environment infrastructure for the app's processes.
Embedded servers/containers	Tomcat, Jetty, Flask, Nameko, etc	Processes, Concurrency, Port Binding	Use to contain an app, and expose its interface over some predefined port and protocol.
Cloud, PaaS CLI clients	gcloud, aws, azure-cli, pf, etc	Codebase, Config, Process, Concurrency, Build-Release-Run, Logs	Use to create, push, pull, start, stop an app deployment on the infrastructure.
Scripting languages	Bash, Python, Groovy, Powershell,etc	Build-Release-Run, Dev/Prod Parity, Admin Processes	Use to provide aid in CI/CD pipelines, and to develop admin apps.
Configuration management	Chef, Puppet, Ansible, etc	Config, Build-Release-Run, Dev/Prod Parity	Use to define environment configuration and automate the configuration process through out the deploy environments.
Database systems	Redis, MongoDB, Cassandra, MySQL, Kafka, RabbitMQ, Apache Camel, etc	Backing Services, Processes	Use to hold app's state, and provide other data related facilities.

So again, what are the benefits of implementing 12 Factors?

>Cloud ready solution

You can develop applications without getting into hassel of explicitly handling cases and requirements for cloud deployment.

>Ensures scalability and resilience to the design

You will get enormous scalability and resilience capabilities without actually doing any explicit development for that.

>Promotes distributed software solution design patterns

You will automatically get compartmentization in your app with distributed modules implementing distributed design patterns.

>Ease in system maintenance

You will get several advantages in system maintenance by utilizing various tools and softwares that provides automation in testing, delivery, deployment, monitoring, and reporting, making software maintenance an easy job.

(Cont.) So again, what are the benefits of implementing 12 Factors?

> Enables easy and effective automation

You will transform your automation jobs into most effective processes possible that makes efficient deloyments a reality.

> Streamline engineering processes

Your engineer process will become more efficient in making speedier deliveries and more responsive to the change requirements without compromizing the build qualities.

> Increases software quality and delivery efficiency

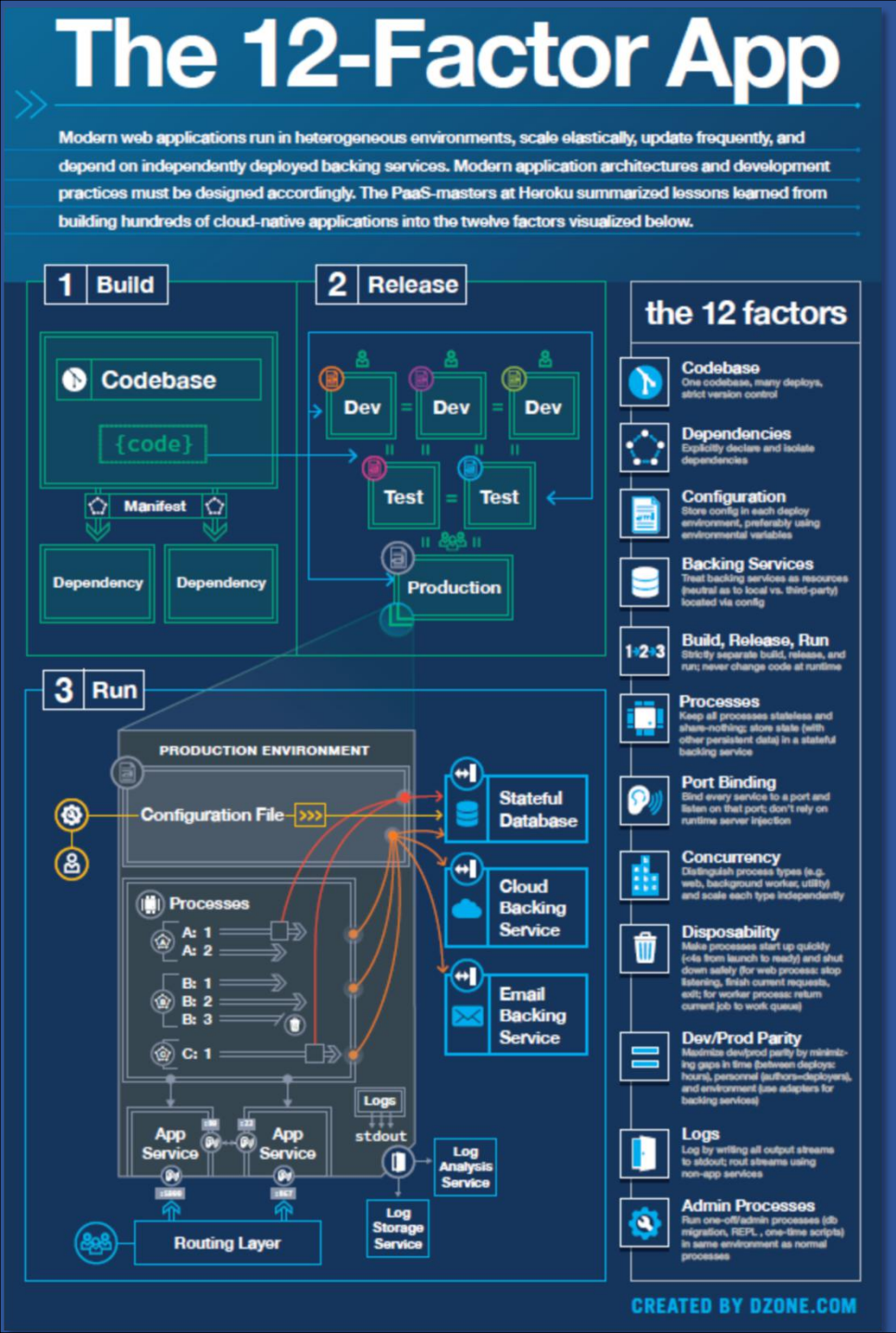
You will be able to increase your app's build quality due to more efficient and effective test automation, app compartmentization, no/minimum dev-prod parity, and quicker delivery mechanism.

> Helps avoid the cost of software erosion

You will be able to make quicker and periodic maintenance on your app that includes adding new features, changes and fixes, hence preventing your app from lossing customers, and keeping their interest alive in your software solution.

For more information on microservice architecture, and how 12 factors are being used there, please refer to my another presentation @

[Understanding MicroSERVICE Architecture with Java Spring-Boot]


The 12-Factor App
Modern web applications run in heterogeneous environments, scale elastically, update frequently, and depend on independently deployed backing services. Modern application architectures and development practices must be designed accordingly. The PaaS-masters at Heroku summarized lessons learned from building hundreds of cloud-native applications into the twelve factors visualized below.
1 Build
Codebase
{code}
Manifest
Dependency
Dependency
2 Release
Dev
Dev
Dev
Test
Test
Production
3 Run
PRODUCTION ENVIRONMENT
Configuration File
Processes
A: 1
A: 2
B: 1
B: 2
B: 3
C: 1
Stateful Database
Cloud Backing Service
Email Backing Service
Logs
stdout
App Service
App Service
Log Analysis Service
Log Storage Service
Routing Layer
the 12 factors
Codebase
One codebase, many deploys, strict version control
Dependencies
Explicitly declare and isolate dependencies
Configuration
Store config in each deploy environment, preferably using environmental variables
Backing Services
Treat backing services as resources (neutral as to local vs. third-party) located via config
Build, Release, Run
Strictly separate build, release, and run; never change code at runtime
Processes
Keep all processes stateless and share-nothing; store state (with other persistent data) in a stateful backing service
Port Binding
Bind every service to a port and listen on that port; don't rely on runtime server injection
Concurrency
Distinguish process types (e.g. web, background worker, utility) and scale each type independently
Disposability
Make processes start up quickly (<6s from launch to ready) and shut down safely (for web process: stop listening, finish current requests, exit; for worker process: return current job to work queue)
Dev/Prod Parity
Maximize dev/prod parity by minimizing gaps in time (between deploys: hours), personnel (authors=deployers), and environment (use adapters for backing services)
Logs
Log by writing all output streams to stdout; rout streams using non-app services
Admin Processes
Run one-off/admin processes (db migration, REPL , one-time scripts) in same environment as normal processes
CREATED BY DZONE.COM

Thank You