The Evolutionary Architect

Contents

[An Evolutionary Vision for the Architect 2](#_Toc59287804)

[Zoning 2](#_Toc59287805)

[A Principled Approach 3](#_Toc59287806)

[Strategic Goals 3](#_Toc59287807)

[Principles 3](#_Toc59287808)

[Heroku’s 12 Factors App 3](#_Toc59287809)

[Practices 9](#_Toc59287810)

[Combining Principles and Practices 9](#_Toc59287811)

[A Real-World Example 10](#_Toc59287812)

[The Required Standard 10](#_Toc59287813)

[Monitoring 10](#_Toc59287814)

[Interfaces 10](#_Toc59287815)

[Architectural Safety 11](#_Toc59287816)

[Governance Through Code 11](#_Toc59287817)

[Exemplars 11](#_Toc59287818)

[Tailored Service Template 11](#_Toc59287819)

[Technical Debt 11](#_Toc59287820)

[Exception Handling 11](#_Toc59287821)

[Governance and Leading from the Centre 11](#_Toc59287822)

# An Evolutionary Vision for the Architect

The software architects should behave like architects, or rather than town planners. Everything cannot be foreseen, and so rather than planning for any eventuality, we should plan to allow everything that will happen, and so rather than plan for any eventuality, we should plan to allow for change by avoiding the urge to overspecifiy everything.

# Zoning

What are our zones??

These are our service boundaries, or perhaps coarse-grained group of services.

As architects, we need to worry more about what happens between the zones than what happens inside the zones. *We need to think more about how our services talk to each other or ensuring that we can properly monitor the overall health of our system.*

Within each service, you may be OK with the team who owns that zone picking a different technology stack or data store.   
Concerns here could be:

1. It becomes harder to hire people or them between teams if you have 10 different technology stacks to support.
2. If each team picks a completely different data store, you may find yourself lacking enough experience to run any of them at scale.

Between services, things could get messy.

For example, if one service decides to expose REST over HTTP, another makes use of protocol buffers, a third uses Java RMI, then integration can become a nightmare as consuming service have to understand and support multiple styles of interchange.

“Be worried about what happens between the boxes and be liberal in what happens inside.”

# A Principled Approach

“Rules are for the obedience of fools and the guidance of wise men.”

## Strategic Goals

Strategic goals should speak to where your company is going, and how it sees itself as best making its customers happy. These will be high-level goals and may not include technology at all. They could be defined at a company level or a division level.

## Principles

Principles are rules you have made in order to align what you are doing to some larger goal and will sometimes change.

You probably do not want loads of these. Fewer than 10 is a good number – small enough that people can remember them, or to fit on small posters.

The more principles you have, the greater the chance that they overlap or contradict each other.

### Heroku’s 12 Factors App

#### Who Should Use Heroku’s 12-factor app??

The 12-factor app is a methodology for building software-as-a-service apps that:

* Use declarative formats for setup automation, to minimize time and cost for new developers joining the project.
* Have a clean contract with underlying OS, offering maximum portability between execution environments.
* Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration.
* Minimize divergence between development and production, enabling continuous deployment for maximum agility.
* And can scale up without significant changes to tooling, architecture or development practices.

The 12-factor methodology can be applied to apps written in any language, and which use any combination of backing services.

#### The Twelve Factors

[The Twelve-Factor App (12factor.net)](https://www.12factor.net/)

##### Codebase

(One codebase tracked in revision control, many deploys)

A 12-factor app is always tracked in 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, or any set of repos who share a root commit.

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

1. If there are multiple codebases, it is not an app – it is a distributed system. *Each component is a distributed system in an app. And each can individually comply with twelve-factor.*
2. 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.*

##### Dependencies

(Explicitly declare and isolate dependencies/*bundling*/*vendoring*)

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. Also, 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.

|  |  |  |
| --- | --- | --- |
| Language | Dependency Declaration | Dependency Isolation |
| Ruby | Gemfile | bundle exec |
| Python | Pip | virtualenv |
| C | Autoconf | static linking |

##### Config

(Store config in the environment)

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.

“A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.”

**The twelve-factor app stores config in environment variables.**

Another aspect of config management is grouping. Sometimes apps batch config into named groups named after specific deploys, such as *development*, *test* and *production* environments.

##### 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,

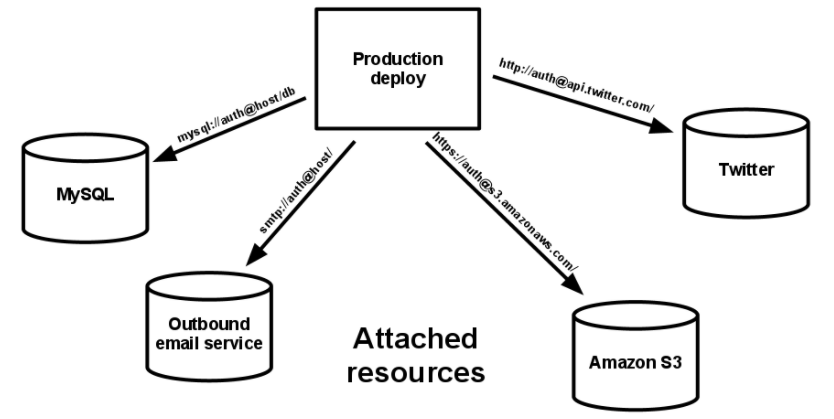
1. Datastores (such as MySQL or CouchDB)
2. Messaging/queueing systems (such as RabbitMQ or Beanstalkd)
3. SMTP services for outbound email (such as Postfix)
4. Caching systems (such as Memcached)

Backing services like the databases are traditionally managed by the same systems administrators who deploy the app’s runtime.

Examples include,

1. SMTP services (such as Postmark)
2. Metrics-gathering services (such as New Relic or Loggly)
3. Binary Asset services (such as Amazon S3)
4. 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 URL or local/credentials stored in the config.**

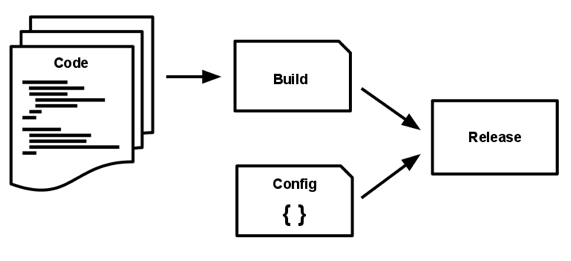
A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third-party without any changes to the app’s code. Only the resource handle in the config needs to change.

##### Build, Release, Run

(Strictly separate build and run stages)

**The 12-factor app uses strict separation between the build, release and run stages.**

A codebase is transformed into a (non-development) 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* andcompiles 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 runs the app in the execution environment, by launching some set of the app’s *processes* against a selected release.

Deployment tools typically offer release management tools, most notably the ability to roll back to a previous release.

Every release should always have a unique release ID. Builds are initiated by the app’s developers whenever new code is deployed.

Runtime execution, by contrast, can happen automatically in case such as server reboot, or a crashed process being restarted by the process manager. Therefore, the run stage should be kept to a few moving parts as possible.

The build can be more complex, since errors are always in the foreground for a dev who is driving the deploy.

##### Processes

(Execute the app as one or more stateless processes)

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

The memory space or filesystem of the process can be used a brief, single-transaction cache. *The 12-factor app never assumes that anything cached in memory or on disk will be available on a future request or job – with many processes of each type running.*

*A 12-factor app prefers to do this compiling during the build stage.*

Some web systems rely on “sticky sessions” – that is, caching user session data in memory of the app’s process and expecting future requests from the same visitor to be routed to the same process.

Sticky sessions are a violation of 12-factor and should never be used or relied upon.

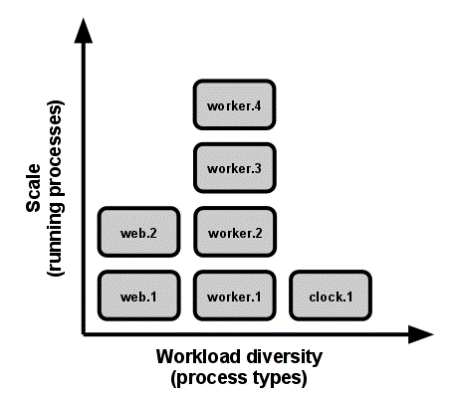
##### Port Binding

(Export services via port binding)

**The 12-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution n environment to create a web-facing service.** The we app exports HTTP as a service by binding to a port, and listening requests coming in on that port.

In 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. This happens entirely in *user space*, i.e., within app’s code.  
The contract with the execution environment is a binding to a port to serve requests.

##### Concurrency

 (Scale out via the process model)  
**In the 12-factor app, processes are a first-class citizen.** Processes in the 12-factor app take strong cues from *UNIX process model for running server daemons.*

Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*.

This does not exclude individual processes from handling their own internal multiplexing, via threads inside the runtime VM, or the async/evented model found in tools such as *EventMachine, Twisted or Node.js*. But an individual VM can grow so large, so the application must also be able to span multiple processes on multiple physical machines.

The *share-nothing, horizontally partitionable nature of 12-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*.

*12-factor app processes should never daemonize or write PID files*. Instead, rely on the OS’s process manager to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns.

##### Disposability

(Maximize robustness with fast startup and graceful shutdown)

**The 12-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.

1. *Process should strive to minimize startup time.*   
   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.
2. *Processes shut down gracefully when they receive a SIGTERM signal from the process manager.*Implicit in this model is that HTTP requests are short, or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost.
3. *For worker process, graceful shutdown is achieved by returning the current job to the work queue.*
4. *Processes should also be robust against sudden death, in the case of a failure in the underling hardware.*

##### Dev/Prod Parity

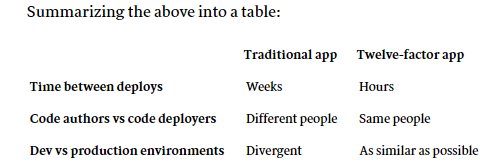
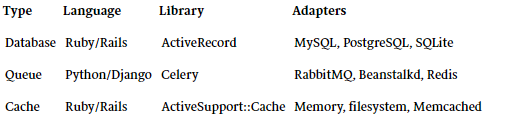
(Keep development, staging and production as similar as possible)

Historically, there have been substantial gaps between development and production. These gaps manifest in 3 areas:

1. The time gap: A dev may work on code that takes days, weeks or even months to go into production.
2. The personnel gap: Devs write code, ops engineer deploys it.
3. The tools gap: Devs may be using a stack like nginx, SQLite and OS X, which the production deploy uses Apache, MySQL and Linux.

**The 12-factor app is designed for continuous deployment by keeping the gap between development and production small.**

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

Backing services, such as the app’s database, queueing system, or cache, is one area where dev/prod parity is important. Many languages offer libraries which simplify access to the backing service, including *adapters* to different types of services.

**The 12-factor developer resists the urge to use different backing services between development and production,** even when adapters theoretically abstract away any differences in backing services.

All deploys of the app should be using the same type and version of each of the backing services.

##### Logs

(Treat logs as event streams)

Logs are the streams of aggregated, time-ordered events collected from the output streams of all running processes and backing services.

**A 12-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`.

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. The archival destinations are not visible to or configurable by the app, and instead are completely managed by the execution environment.   
Example of open-source log routers – *Logplex, Fluentd.*

The event stream for an app can be routed to a file or watched via real-time tail in a terminal. Most significantly, the stream ca be sent to a log indexing and analysis system such as *Splunk*, or a general-purpose data warehousing system such as *Hadoop/Hive*.   
These systems allow for great power and flexibility for introspecting an app’s behaviour over time, including:

1. Finding specific events in the past,
2. Large-scale graphing of trends (such as requests per minute),
3. Active alerting according to user-defined heuristics (such as alert when the quantity of errors per minute exceeds a certain threshold)

##### Admin Process

(Run admin/management tasks as one-off processes)

**12-factor strongly favours languages which provide a REPL shell out of the box, and which make it easy to run one-off scripts.**

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

1. Running database migrations.
2. Running a console to run arbitrary code or inspect the app’s models against the live database.
3. Running one-time scripts committed into the app’s repo.

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.

The same *dependency isolation* techniques should be used on all process types.

## Practices

Our practices are how we ensure our principles are being carried out. *They are a set of detailed, practical guidance for performing tasks*.

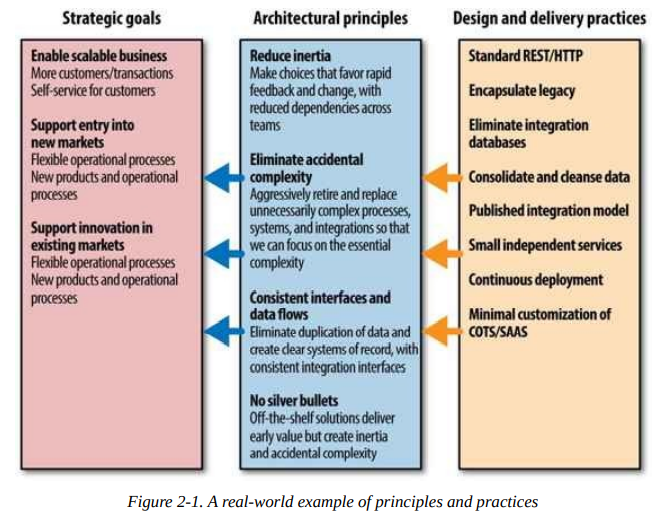
Practices could include coding guidelines, the fact that all log data needs to be captured centrally, or that HTTP/REST is the standard integration style. Due to their technical nature, practices will often change more often than principles.

## Combining Principles and Practices

One person’s principles are another’s practices.

For a small enough group, perhaps a small team, combining principles and practices might be OK. However, for larger organizations, where the technology and working practices may differ, you may want a different set of principles in different places, if they both map to a common set of principles.

## A Real-World Example



# The Required Standard

When you are working through your practices and thinking about the trade-offs you need to make, one of the core balances to find is how much variability to allow in your system.

One of the keyways to identify what should be constant from service to service is to define what a well-behaved, good service looks like.

We need to find the balance between optimizing for autonomy of the individual microservice without losing sight of the bigger picture. Defining clear attributes that each service should have is one way of being clear as to where that balance sits.

## Monitoring

It is essential that we can draw up coherent, cross-service views of our systems health. This must be a system-wide view, not a service-specific view.   
It might be chosen to adopt a push mechanism, where each service needs to push this data into a central location.

For metrics, this might be Graphite and for health, it might be Nagios.

## Interfaces

Picking a small number of defined interface technologies helps integrate new consumers.

## Architectural Safety

We must ensure that our services shield themselves accordingly from unhealthy, downstream calls.

The more services we have that do not properly handle the potential failure of downstream calls, the more fragile our systems will be. This means you will probably want to mandate as a minimum that each downstream service get its own connection pool, and you may even go as far as to say that each also uses a circuit breaker.

Playing by the rules is important when it comes to response codes, too.

# Governance Through Code

Two techniques which I have seen work here are using exemplars and providing service templates.

## Exemplars

If you have a set of standards or best practices you would like to encourage, then having exemplars that you can point people to is useful. Ideally, these should be real-world services you have that get things right, rather than isolated services that are just implemented to be perfect examples.

## Tailored Service Template

By tailoring a service template for one’s own set of development practices, one ensured that teams could get going faster, and that developers must go out of their way to make their services badly behaved.

Defining the practices, one uses should be a collective activity, so ideally one’s team should take joint responsibility for updating this template.

# Technical Debt

The technical vision exists for a reason. If we deviate from this reason, it might have a short-term benefit but a long-term cost.

The deviation is a trade-off which can be understood by the concept of technical debt.

# Exception Handling

It might be worth capturing decisions in a log which allow system which could deviated from the principles and practices. If enough exceptions are found, it may eventually make sense to change the principle or practice to reflect a new understanding of the world.

# Governance and Leading from the Centre

Governance ensures that enterprise objectives are achieved by evaluating stakeholder needs, conditions and options; setting direction through prioritisation and compliance and progress against agreed-on direction and objectives.

COBIT-5

If one of the architect’s jobs is ensuring there is a technical vision, then governance is about ensuring what we are building matches this vision and evolving the vision if needed.

Architects should not partake this work alone. Instead, a properly functioning governance group can work together to share the work and shape the vision.

Governance is a group activity. It could be an informal chat with a small enough team, or a more structured regular meeting with formal group membership for a larger scope. This group needs to be led by a technologist, and to consist predominantly of people who are executing the work being governed. This group should also be responsible for tracking and managing technical risks.