

Stone to Soil Process

This is a copy of a conversation between ChatGPT & Anonymous.



how many years does it take stone to completely become soil

The process of a **stone turning into soil** is called **weathering**, and it can take **hundreds to millions of years**, depending on several factors. Here's a breakdown:

Typical Time Ranges:

- **Soft rocks (like limestone or shale):** May take **hundreds to thousands of years** to significantly break down.
- **Hard rocks (like granite or basalt):** May take **tens of thousands to millions of years**.

Factors That Affect the Speed:

1. **Rock Type:** Softer rocks weather faster.
2. **Climate:**
 - **Wet and warm climates** accelerate chemical weathering.
 - **Cold or dry areas** slow the process.
3. **Biological Activity:** Roots, microbes, and burrowing animals speed up breakdown.
4. **Mechanical vs. Chemical Weathering:**
 - **Mechanical** (physical) weathering breaks rocks into smaller pieces.
 - **Chemical** weathering alters the minerals and helps form true soil.

Final Stage:

Once weathered particles mix with **organic material**, they form **soil**, which itself evolves over **hundreds to thousands of years** into rich, fertile layers.

Would you like a simple diagram showing the stages from rock to soil?



VI. Processes

Execute the app as one or more stateless processes

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

In the simplest case, the code is a stand-alone script, the execution environment is a developer's local laptop with an installed language runtime, and the process is launched via the command line (for example, `python my_script.py`). On the other end of the spectrum, a production deploy of a sophisticated app may use many process types, instantiated into zero or more running 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 – with many processes of each type running, chances are high that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local (e.g., memory and filesystem) state.

Asset packagers like `django-assetpackager` use the filesystem as a cache for compiled assets. A twelve-factor app prefers to do this compiling during the build stage. Asset packagers such as `Jammit` and the Rails asset pipeline can be configured to package assets 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 twelve-factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as `Memcached` or `Redis`.

The section you've shared—“**VI. Processes**”—comes from the [Twelve-Factor App methodology](#), which guides the development of scalable and maintainable web applications. Here's a simplified summary of the key points from this principle:

✓ Summary of Twelve-Factor Principle VI: Processes

1. Stateless Processes:

- Each app runs as **one or more independent, stateless processes**.
- Processes don't retain any memory or disk state between requests.

2. Persisting Data:

- Any persistent data (e.g., user sessions, files, or databases) must be stored in **external services** like databases, object storage, or caches (`Redis`, `Memcached`).

3. Temporary Local Storage:

- Local memory or file systems can be used **temporarily** (e.g., for processing), but nothing should depend on them being persistent.

4. No Sticky Sessions:

- Apps must not store session data in memory expecting the same user to hit the same process again.
- Use shared, external session stores (like `Redis`) instead.

5. Asset Compilation:

- Static asset packaging and compilation should happen **during the build stage**, not at runtime.

Why It Matters:

- Ensures **scalability** (easily add or remove processes).
- Improves **resilience** (process failures don't result in data loss).
- Makes your app **cloud-native friendly**, especially in environments like Kubernetes or Heroku.

Would you like a visual flowchart or diagram to represent this stateless process model?



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:5000/` to access the service exported by their app. 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, such as Tornado for Python, Thin for Ruby, or Jetty for Java and other JVM-based languages. 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.

Here's a concise summary of **Twelve-Factor Principle VII: Port Binding**:

Summary: Port Binding

1. Self-contained Service:

- A twelve-factor app **does not depend** on an external web server (like Apache or Tomcat).
- Instead, it **includes its own web server** as a library (e.g., Flask/Tornado in Python, Thin in Ruby, Jetty in Java).

2. Exports via Port:

- The app **binds to a port** (e.g., `5000`) and listens for incoming requests—**this is how it exposes services**.

3. Works in All Environments:

- In local development: access via `http://localhost:5000/`.
- In production: a routing layer (like NGINX or a load balancer) maps public URLs to internal app ports.

4. Protocol-Agnostic:

- Not limited to HTTP—apps can export **any service** over a port (e.g., Redis, XMPP, gRPC).

5. Service Composition:

- This model allows **microservices** or backing services to communicate over URLs/ports, enabling **decoupled architectures**.

Why It Matters:

- Promotes **microservice-friendly** design.
- Makes your app **environment-agnostic** (can run locally, in containers, or cloud-native platforms).
- Simplifies **deployment** by avoiding dependency on external server containers.

Would you like an infographic showing how an app binds to a port and routes requests?



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.

Scale is expressed as running processes, workload diversity is expressed as process types.

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/evented 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.

Twelve-factor app processes should never daemonize or write PID files. Instead, rely on the operating system's process manager (such as systemd, a distributed process manager on a cloud platform, or a tool like Foreman in development) to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns.

Here's a concise summary of **Twelve-Factor Principle VIII: Concurrency**:

✓ Summary: Concurrency

1. Processes as First-Class Citizens:

- The twelve-factor app **scales out using multiple stateless processes**, not just threads or internal concurrency.
- Each **process type handles a specific workload** (e.g., web, worker, scheduler).

2. Horizontal Scalability:

- Instead of scaling a single process "vertically" (more CPU/memory), the app **scales "horizontally"** by adding more processes.
- These processes can run on **multiple machines or containers**.

3. Process Formation:

- The **combination of process types and quantity** defines the app's *process formation*.
- Example:

```
ini

web=2
worker=4
scheduler=1
```

4. No Daemonizing:

- Processes should **not daemonize** themselves or manage PID files.
- Rely on external tools (e.g., **systemd**, **Foreman**, or **cloud orchestration platforms**) to manage lifecycle, logs, and restarts.

5. Supports Threading and Async:

- Internal concurrency (e.g., threads, async I/O) is fine, but **it doesn't replace process-level scalability**.

💡 Why It Matters:

- Enables **reliable, flexible scaling**.
- Makes your app **cloud-native ready**.

- Simplifies operational concerns—**restart, logs, scaling**, etc., are handled by the platform or tooling.

Would you like a visual example of process formation showing how different types of workloads are separated and scaled?



explain simply

Sure! Here's a **simple explanation** of the Twelve-Factor App's **Concurrency** principle:

🔄 VIII. Concurrency – Scale by Adding Processes

Think of your app like a **restaurant kitchen**:

- You have different **roles**: a chef for cooking, a server for taking orders, and a dishwasher.
- If the restaurant gets busy, you **hire more chefs**, more servers, or more dishwashers.

📌 In the same way:

- Your app should have different **processes** for different jobs:
 - One to handle web requests (like the server).
 - One to do background tasks (like the cook).
- When more work comes in, you **add more of those processes** (like hiring more staff).

🚫 Don't let the app run as one big block or try to manage everything inside itself.

✅ Instead, let the **system or platform (like Heroku, Docker, or systemd)** start, stop, and manage those processes for you.

👉 Simple Rule:

Handle different tasks with different types of processes, and scale by adding more of them as needed.

Would you like an analogy or picture to go along with this?



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.

For a worker process, graceful shutdown is achieved by returning the current job to the work queue. For example, on RabbitMQ the worker can send a NACK; on Beanstalkd, the job is returned to the queue automatically whenever a worker disconnects. Lock-based systems such as Delayed Job need to be sure to release their lock on the job record. Implicit in this model is that all jobs are reentrant, which typically is achieved by wrapping the results in a transaction, or making the operation idempotent.

Processes should also be robust against sudden death, in the case of a failure in the underlying hardware. While this is a much less common occurrence than a graceful shutdown with SIGTERM, it can still happen. A recommended approach is use of a robust queueing backend, such as Beanstalkd, that returns jobs to the queue when clients disconnect or time out. Either way, a twelve-factor app is architected to handle unexpected, non-graceful terminations. Crash-only design takes this concept to its logical conclusion.

Here's a **simple explanation** of **IX. Disposability** from the Twelve-Factor App:

IX. Disposability – Fast Start & Clean Exit

Think of your app's process like a **light bulb** —

 You should be able to **turn it on or off quickly and safely**.

What it means:

1. Fast Start:

- The app should start in a **few seconds**.
- This helps when scaling up (adding more instances) or redeploying quickly.

2. Graceful Shutdown:

- When told to stop, the app:
 - **Stops accepting new work.**
 - **Finishes what it's currently doing.**
 - **Then shuts down cleanly.**

3. Handles Crashes:

- If it crashes suddenly (like a power cut), it **should not lose data**.
- Use queues (like Redis or RabbitMQ) that **save the job** if the app goes down.

🎯 Why this matters:

- Your app becomes **resilient** and **easy to manage**.
- It's ready for **cloud scaling**, **auto restarts**, and **rolling deployments**.

💡 Simple rule:

Your app should be ready to start and stop at any time without breaking things.

Would you like a diagram showing the startup and shutdown flow?



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.

Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.

Make the tools gap small: keep development and production as similar as possible.

Summarizing the above into a table:

Traditional app Twelve-factor app

Time between deploys Weeks Hours

Code authors vs code deployers Different people Same people

Dev vs production environments Divergent 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. Some examples are in the table below.

Type Language Library Adapters

Database Ruby/Rails ActiveRecord MySQL, PostgreSQL, SQLite

Queue Python/Django Celery RabbitMQ, Beanstalkd, Redis

Cache Ruby/Rails ActiveSupport::Cache Memory, filesystem, Memcached

Developers sometimes find great appeal in using a lightweight backing service in their local environments, while a more serious and robust backing service will be used in production. For example, using SQLite locally and PostgreSQL in production; or local process memory for caching in

development and Memcached in production.

The twelve-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. Differences between backing services mean that tiny incompatibilities crop up, causing code that worked and passed tests in development or staging to fail in production. These types of errors create friction that disincentivizes continuous deployment. The cost of this friction and the subsequent dampening of continuous deployment is extremely high when considered in aggregate over the lifetime of an application.

Lightweight local services are less compelling than they once were. Modern backing services such as Memcached, PostgreSQL, and RabbitMQ are not difficult to install and run thanks to modern packaging systems, such as Homebrew and apt-get. Alternatively, declarative provisioning tools such as Chef and Puppet combined with light-weight virtual environments such as Docker and Vagrant allow developers to run local environments which closely approximate production environments. The cost of installing and using these systems is low compared to the benefit of dev/prod parity and continuous deployment.

Adapters to different backing services are still useful, because they make porting to new backing services relatively painless. But all deploys of the app (developer environments, staging, production) should be using the same type and version of each of the backing services.

Here's a **simple explanation** of **X. Dev/Prod Parity** from the Twelve-Factor App:

X. Dev/Prod Parity – Keep Dev and Production Alike

"What works on your laptop should work the same way in production."

What it means:

Your development (dev), testing (staging), and live (production) environments should be:

- **As similar as possible**
- To avoid surprises when deploying

3 Common Gaps (and how to close them):

| Gap Type | Traditional App | Twelve-Factor App |
|---------------|-----------------------------|--|
| Time | Deploy code every few weeks | Deploy code within minutes or hours |
| People | Devs write, Ops deploy | Same people write & deploy |
| Tools | Different tools in dev/prod | Same stack in all environments |

Example:

Bad practice:

- Dev uses **SQLite** for local DB

- Production uses **PostgreSQL**

This may cause bugs that don't appear in dev but crash in production.

Good practice:

- Use **PostgreSQL everywhere** (dev, staging, production)

Why it matters:

- Fewer bugs after deployment
- Easier to test realistically
- Enables fast, safe, **continuous delivery**

Golden Rule:

"Use the **same tools, configs, and services** in all environments."

Would you like a simple checklist to ensure dev/prod parity in your own projects?



XI. Logs

Treat logs as event streams

Logs provide visibility into the behavior 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 behavior.

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. These archival destinations are not visible to or configurable by the app, and instead are completely managed by the execution environment. Open-source log routers (such as Logplex and Fluentd) are available for this purpose.

The event stream for an app can be routed to a file, or watched via realtime tail in a terminal. Most significantly, the stream can 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 behavior over time, including:

Finding specific events in the past.

Large-scale graphing of trends (such as requests per minute).
Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

Here's a **simple explanation** of **XI. Logs – Treat Logs as Event Streams**:

XI. Logs – Logs are Live Streams of Events

"Don't save logs to files. Just stream them."

What it means:

- Your app should **just print logs** (like using `console.log()` or `print()`)
- Let the **platform** (e.g. Heroku, Docker, cloud providers) handle **where the logs go**

Don't:

- Write logs to files (like `log.txt`)
- Try to manage or move logs from your app code

Do:

- Write logs to **stdout** (standard output)
- Example:

```
bash
print("User logged in")
```

- The system will automatically collect these logs and:
 - Show them live on your terminal (in dev)
 - Send them to tools like **Splunk**, **Logstash**, or **Datadog** (in production)

Why it's useful:

- Easier to **debug** and **monitor**
- Works well with tools that:
 - Search logs (e.g. errors)
 - Graph trends (e.g. traffic spikes)
 - Send alerts (e.g. if too many crashes)

Summary:

"Your app writes logs. The system handles them."

Would you like examples of good logging practices in Python or JavaScript?



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.

The same dependency isolation techniques should be used on all process types. For example, if the Ruby web process uses the command `bundle exec thin start`, then a database migration should use `bundle exec rake db:migrate`. Likewise, a Python program using `Virtualenv` should use the vendored `bin/python` for running both the Tornado webserver and any `manage.py` admin processes.

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.

Here's a **simple explanation** of **XII. Admin Processes – Run admin tasks as one-off processes**:

XII. Admin Processes – Separate Tasks for Special Jobs

"For setup or fixes, run separate short-lived processes, not long-running ones."

What it means:

Sometimes, you need to do **admin tasks** like:

- Running database migrations (``python manage.py migrate``)
- Fixing bad data (``scripts/fix_bug.py``)
- Testing app code in a REPL (``python``, ``rails console``)

These should be run as **one-off commands, not built into the main app.**

Key Points:

- Run admin tasks in the **same environment** as your app (same code, config, and dependencies).
 - Use the **same tools and setup** (e.g. ``venv``, ``bundle exec``, etc.).
 - Never hardcode special admin logic into the long-running app (like your web server).
-

Examples:

In development:

```
bash

python manage.py migrate
```

In production (via SSH or a control panel like Heroku CLI):

```
bash

heroku run python manage.py migrate
```

Why this matters:

- Keeps your main app clean and focused
 - Avoids code/config mismatches
 - Makes fixes and setup scripts easy to test and run
-

Summary:

"For fixes or setup, run short one-time commands in the same setup as your app."

Want a simple chart summarizing all 12 factors next?



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:5000/` to

access the service exported by their app. 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, such as Tornado for Python, Thin for Ruby, or Jetty for Java and other JVM-based languages. 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.

Here's a **simple explanation** of **VII. Port Binding – Export services via port binding**:

VII. Port Binding – Let Your App Speak on a Port

"Your app should run as a self-contained service and **listen on a port** for requests."

What it means:

Instead of relying on an external web server like Apache or Tomcat to "host" your app, a **Twelve-Factor App includes its own web server** and binds directly to a port.

Key Points:

- Your app should run by itself, not **inside another server**.
- It should **bind to a port** (e.g., `5000`) and listen for incoming requests.
- You can access it at something like:
`http://localhost:5000/`
- In production, a **router or proxy (like NGINX)** can forward traffic from the public internet to that port.

Tools that do this:

| Language | Web Server |
|----------|--------------------------|
| Python | Gunicorn, Flask, Tornado |
| Ruby | Puma, Thin |
| Java | Jetty, Undertow |

Bonus:

Your app can also **be a service** for other apps by exposing its port (like Redis or a custom API).

Summary:

"Don't depend on external web servers. Let your app **run its own** and expose itself via a **port**."

Want a summary cheat sheet with all 12 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.

In the simplest case, the code is a stand-alone script, the execution environment is a developer's local laptop with an installed language runtime, and the process is launched via the command line (for example, `python my_script.py`). On the other end of the spectrum, a production deploy of a sophisticated app may use many process types, instantiated into zero or more running 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 – with many processes of each type running, chances are high that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local (e.g., memory and filesystem) state.

Asset packagers like `django-assetpackager` use the filesystem as a cache for compiled assets. A twelve-factor app prefers to do this compiling during the build stage. Asset packagers such as `Jammit` and the Rails asset pipeline can be configured to package assets 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 twelve-factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as `Memcached` or `Redis`.

Here's a **simple explanation** of **VI. Processes – Execute the app as one or more stateless processes**:

What It Means:

Your app should run as one or more **stateless processes**, meaning that **each process doesn't store data** about previous requests or jobs.

Key Points:

- **Stateless:** Every process should **not** keep any memory of previous actions. Each request is treated independently.
- **No reliance on local memory or file storage:** Don't store data or session information in the app's local memory or file system. Instead, store data in an **external, stateful service** (like a database).
- **Short-term caching:** Processes can use memory or the file system briefly for temporary storage (e.g., caching the results of a calculation), but this **doesn't last** beyond a process restart.

Why Stateless is Important:

- **Reliability:** If the process is moved, restarted, or multiple instances are used, there won't be any problems with missing or inconsistent data.
- **Scaling:** Stateless processes allow easy scaling by adding or removing processes without worrying about data being lost.

Example:

Imagine you're downloading a file, processing it, and saving the result in a database:

- **Download and Process:** Do this in the memory for fast access.
- **Persist:** Save the final result in a database (not the app's memory or filesystem).

Avoiding Sticky Sessions:

Don't use **sticky sessions**, where user data is cached in the app's memory, because that ties requests from a user to a specific process. Instead, use **external storage** for session data (like **Redis** or **Memcached**).

Summary:

"Processes should be **stateless**. Any data that needs to persist must go to an external, stateful service like a database."

Let me know if you want more details or examples!



V. Build, release, run

Strictly separate build 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 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.

Code becomes a build, which is combined with config to create a 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.

Deployment tools typically offer release management tools, most notably the ability to roll back to a previous release. For example, the Capistrano deployment tool stores releases in a subdirectory named releases, where the current release is a symlink to the current release directory. Its rollback command makes it easy to quickly roll back to a previous release.

Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100). Releases are an append-only ledger and a release cannot be mutated once it is created. Any change must create a new release.

Builds are initiated by the app’s developers whenever new code is deployed. Runtime execution, by contrast, can happen automatically in cases such as a server reboot, or a crashed process being restarted by the process manager. Therefore, the run stage should be kept to as few moving parts as possible, since problems that prevent an app from running can cause it to break in the middle of the night when no developers are on hand. The build stage can be more complex, since errors are always in the foreground for a developer who is driving the deploy.

V. Build, Release, Run – Strictly Separate Build and Run Stages

What It Means:

The process of getting your app into production is broken down into **three separate stages: build, release, and run**. These stages are **distinct** and **cannot overlap**, ensuring a **smooth deployment pipeline**.

Key Stages Explained:

1. Build Stage:

- **What Happens:** The code is **compiled** into an executable bundle (a build). This includes fetching **dependencies** and **compiling** any necessary binaries or assets.
- **Example:** Your code repo is transformed into a build artifact, ready to be deployed.

2. Release Stage:

- **What Happens:** The **build** from the build stage is combined with the app’s **configuration** (settings, environment variables). This results in a **release**, which contains both the app code and config, making it ready for execution.
- **Example:** The build now contains all the necessary config (like database URLs, API keys, etc.), ready to run on any environment.

3. Run Stage:

- **What Happens:** The app's processes are started in the execution environment. The app runs using the selected release.
- **Example:** This is where your app is **actively running**, serving requests or processing jobs.

Why This Separation is Important:

- **Clear Workflow:** Each stage is handled separately, allowing developers to troubleshoot and test each part independently.
- **Consistency:** No changes can be made to the code during the **run stage**, preventing errors from sneaking in once the app is live.
- **Rollback:** Deployment tools (like **Capistrano**) allow you to easily **rollback** to a previous release if something goes wrong.

Release ID and Immutability:

- Every release should have a **unique ID** (e.g., timestamp or version number) so you can track and **rollback** easily.
- **Immutable Releases:** Once a release is created, it **cannot be changed**. Any new changes require a **new release** to be created.

Automating with Deployments:

- **Builds** are initiated by developers when new code is ready to be deployed.
- **Run** happens automatically in production (e.g., if the server restarts or a crashed process is restarted).

Summary:

"Separate the build, release, and run stages. The build creates the codebase, the release combines the build with configuration, and the run starts the app in production."

Let me know if you need further clarification!



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

who deploy the app's runtime. 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.

Each distinct backing service is a resource. For example, a MySQL database is a resource; two MySQL databases (used for sharding at the application layer) qualify as two distinct resources. The twelve-factor app treats these databases as attached resources, which indicates their loose coupling to the deploy they are attached to.

A production deploy attached to four backing services.

Resources can be attached to and detached from deploys at will. For example, if the app's database is misbehaving due to a hardware issue, the app's administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

IV. Backing Services – Treat Backing Services as Attached Resources

What Are Backing Services?

A **backing service** is any service that your app consumes over the network as part of its normal operations. These services provide essential functionality but are separate from the core application code. Examples include:

- **Datstores:** e.g., MySQL, CouchDB
- **Messaging/Queueing Systems:** e.g., RabbitMQ, Beanstalkd
- **SMTP for Email:** e.g., Postfix for outbound emails
- **Caching Systems:** e.g., Memcached

In addition to locally managed services, you might also use third-party services such as:

- **Email services:** e.g., Postmark, SendGrid
- **Metrics services:** e.g., New Relic, Loggly
- **File storage:** e.g., Amazon S3
- **APIs:** e.g., Twitter, Google Maps, Last.fm

Loose Coupling of Backing Services

The app doesn't distinguish between **local** and **third-party** backing services. **Both are treated the same** and are accessed via a **resource handle** (URL or other credentials) stored in the app's config.

- **Example:** You can **swap** a local **MySQL database** with a **third-party managed one** (e.g., Amazon RDS) without making any changes to the app's code. Only the database's URL or credentials in the config need to be updated.

Key Characteristics of Backing Services:

1. **Resources Are Separate:** Every service is considered a distinct resource. For instance, using **two MySQL databases** for sharding would mean treating them as **two separate resources**.
2. **Flexibility:** A backing service can be **attached or detached** to/from a deploy as needed.
 - Example: If a database goes down, an administrator could easily replace it with a new one, restore from backup, and attach it to the app without changing the app's code.
3. **Loose Coupling:** Backing services are loosely coupled to the app itself, meaning the app doesn't directly control the backing service but only interacts with it via configuration.

Practical Example:

- In a **production deploy** of the app, it could be attached to multiple backing services (e.g., one for MySQL, one for Redis, etc.), all accessed via their respective resource handles.
- If one backing service, like the database, needs to be swapped due to an issue, the app **does not require code changes**. The only action required is to **change the resource handle** in the app's config to point to the new service.

Summary:

"Treat every backing service (database, email system, cache, etc.) as an attached resource, allowing easy swap-outs between local and third-party services with no changes to app code."

Let me know if you need further clarification or have more questions!



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.

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.

Note that this definition of “config” does not include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

Another approach to config is the use of config files which are not checked into revision control, such as `config/database.yml` in Rails. This is a huge improvement over using constants which are checked into the code repo, but still has weaknesses: it’s easy to mistakenly check in a config file to the repo; there is a tendency for config files to be scattered about in different places and different formats, making it hard to see and manage all the config in one place. Further, these formats tend to be language- or framework-specific.

The twelve-factor app stores config in environment variables (often shortened to env vars or env). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called “environments”) named after specific deploys, such as the development, test, and production environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as staging or qa. As the project grows further, developers may add their own special environments like joes-staging, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

In a twelve-factor app, env vars are granular controls, each fully orthogonal to other env vars. They are never grouped together as “environments”, but instead are independently managed for each deploy. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

III. Config – Store Config in the Environment

What is Config?

In the context of a **twelve-factor app**, **config** refers to all values that vary between different environments (such as **development**, **staging**, or **production**) but are not part of the core code itself. This includes:

- **Resource handles:** for databases, Memcached, and other backing services
- **Credentials:** for external services like Amazon S3, Twitter API, etc.
- **Per-deploy values:** like the **canonical hostname** of the app’s deploy

What is NOT Config?

- **Internal application configuration**, such as **routes** in a framework like Rails (`config/routes.rb`) or module connections in Spring. These configurations do **not** vary between deploys, so they are managed in the code itself.
- **Config files checked into revision control**: e.g., `config/database.yml` in Rails. While this is an improvement over hardcoding constants, it still introduces issues like accidental commits, scattered config files, and language/framework-specific formats.

✓ The Twelve-Factor Approach to Config:

The twelve-factor app **stores config in environment variables (env vars)**. This approach has several benefits:

- **Environment agnostic**: Env vars work across different programming languages and operating systems, unlike custom config files or Java properties.
- **No code changes**: Env vars can be changed without modifying the app's code, allowing seamless configuration changes across environments.
- **Security**: It prevents accidental inclusion of sensitive data (like credentials) in the codebase, which could happen with config files.
- **Centralized control**: Env vars are easy to manage, unlike scattered config files in different formats.

↻ Granularity of Config:

Env vars are **granular** and **independent**, meaning each variable controls one aspect of the configuration. Unlike the older model of grouping config into named environments (e.g., `development`, `production`), the twelve-factor app treats **each config as an individual unit**, avoiding a combinatorial explosion of different environment configurations.

For example:

- **In a twelve-factor app:**
 - **Env var for database**: `DATABASE_URL=postgres://user:pass@host:port/db`
 - **Env var for API keys**: `TWITTER_API_KEY=abcd1234`

These are all handled independently, without grouping them into larger, predefined environments.

💡 Benefits of Storing Config in Environment Variables:

1. **Easily manageable**: Env vars can be quickly updated, and there's no need for code changes.
2. **Separation of concerns**: It enforces a **clear distinction** between code (which remains static) and configuration (which changes per environment).
3. **Scalability**: It can handle scaling up the number of deploys and environments without issues like **conflicting configurations** or **environment explosion**.

🎯 Summary:

"Store configuration in environment variables to ensure a clean separation of concerns, improve security, and provide scalability for your app across different environments."

Let me know if you'd like further clarification!



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.

For example, Bundler for Ruby offers the Gemfile manifest format for dependency declaration and bundle exec for dependency isolation. In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation. Even C has Autoconf for dependency declaration, and static linking can provide dependency isolation. No matter what the toolchain, dependency declaration and isolation must always be used together – only one or the other is not sufficient to satisfy twelve-factor.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app's code with a deterministic build command. For example, the build command for Ruby/Bundler is bundle install, while for Clojure/Leiningen it is lein deps.

Twelve-factor apps also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or curl. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be vendored into the app.

II. Dependencies – Explicitly Declare and Isolate Dependencies

What Does This Mean?

A **twelve-factor app** ensures that **all dependencies** are **explicitly declared** and **isolated** to guarantee consistency across all environments and deployments. Dependencies include:

- **Support libraries:** These are additional tools or packages that your app relies on (like frameworks or utilities).

- **System tools:** External tools like ImageMagick or curl that the app may need to interact with.

How Does This Work?

1. Explicit Declaration:

- Every app dependency is **declared** in a **dependency manifest** (a configuration file). For example:
 - Ruby uses `Gemfile` with **Bundler**.
 - Python uses `requirements.txt` with **Pip**.
 - JavaScript uses `package.json` with **npm** or **yarn**.

2. Dependency Isolation:

- The app **isolates** its dependencies from the system, ensuring that the app won't inadvertently use external libraries that are not explicitly declared.
- For example:
 - Python uses **Virtualenv** to create isolated environments.
 - Ruby uses **Bundler** with the `bundle exec` command to ensure the environment is isolated.
 - C programs use **static linking** to bundle dependencies directly.

By using these tools together, the app ensures that **only the exact dependencies** specified will be used, without any implicit or external dependencies being "leaked" into the app.

Tools and Their Roles:

- **Bundler (Ruby):** Uses a `Gemfile` to declare dependencies and `bundle exec` for isolation.
- **Pip and Virtualenv (Python):** `requirements.txt` for dependency declaration and `Virtualenv` for isolation.
- **Autoconf and Static Linking (C):** Tools to declare and isolate dependencies in C-based applications.
- **npm (Node.js):** `package.json` for dependency declaration and isolation within the app.

Why Declare and Isolate Dependencies?

1. Consistent Setup:

- For new developers, it ensures they can **easily set up** the project on their machine without worrying about system-wide packages. They just need to install the language runtime and the dependency manager.

2. Avoid System Tool Dependency:

- The app will not rely on **external system tools** being available. For example, if the app needs to shell out to a tool like `curl` or `ImageMagick`, it should **vend** these tools into the app to ensure compatibility across all systems, regardless of the external environment.

Key Benefits:

- **Deterministic Builds:** With explicit dependency declaration, you can guarantee that **every deployment** will use the same versions of dependencies, avoiding the "it works on my machine" issue.
- **Isolation from System Environment:** By isolating dependencies, you avoid conflicts with system-installed packages or tools, ensuring consistency no matter where the app runs.
- **Simplified Setup:** New developers or environments can quickly get the app up and running by just installing the dependencies through a single command.

Summary:

"Explicitly declare and isolate all dependencies using a manifest and isolation tool to guarantee consistent builds, avoid reliance on system tools, and simplify setup for new developers."

Let me know if you'd like further details or examples!



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).

One codebase maps to many deploys

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.

I. Codebase – One Codebase Tracked in Revision Control, Many Deploys

What Does This Mean?

A **twelve-factor app** operates with a **single codebase**, tracked using a **version control system (VCS)** like Git, Mercurial, or Subversion. This codebase is the foundation for all deployments of the app.

Key Concepts:

1. Codebase:

- A **codebase** refers to the complete set of code managed under version control, typically in a **single repository**.
- If a system uses a **decentralized version control system (DVCS)** like **Git**, it means the app's codebase is tracked by a shared root commit across repositories.

2. Many Deploys:

- **Deploys** refer to running instances of the app (e.g., production, staging, or a developer's local environment).
 - The **same codebase** is used for all deploys, but **different versions** can exist across them:
 - Developers work with different commits than staging or production environments.
-

How It Works:

- There is always **one codebase** for the app.
 - Multiple deploys of the app can exist at any time (e.g., **production, staging, developer environments**), all running the same codebase but potentially at different points in time (with different commits).
 - The codebase is version-controlled and managed via a VCS, ensuring that **every deploy** shares the same history of changes, even if they have different active versions.
-

Why This is Important:

- **Consistency:** Having a **single codebase** means there's only one version of the app's code to maintain, avoiding discrepancies between different parts of the app.
 - **Scalability:** As the app evolves, there may be multiple deploys (for instance, for testing or production), but they all track back to the **same** codebase, simplifying deployment management.
 - **Collaboration:** Developers, testers, and operators can work with the same codebase, making collaboration across teams easier.
-

What It Means for Deployment:

1. Multiple Deploys, Same Codebase:

- Multiple **deployments** (like production or staging) can run concurrently, each with its own version of the code.
- Each environment (e.g., **developer, staging, production**) will have a copy of the codebase that corresponds to its current state, but they are **all deploys of the same codebase**.

2. Shared Code, No Duplication:

- If **shared code** is used across different apps, it should be placed in **libraries** that can be included using a dependency manager (to avoid violating the twelve-factor principle).

3. Version Control:

- The codebase is **version-controlled**, meaning every commit, change, and update to the app is recorded and traceable.

Example:

- **Git Workflow:**
 - A developer has a local Git branch with certain commits.
 - The staging environment may be running the latest stable branch.
 - The production environment might be running the previous stable version.
 - Despite each environment running a different version of the code, they all share the **same codebase**.

Key Takeaways:

- A **single codebase** tracked via a version control system.
- **Multiple deploys** (e.g., production, staging, development) share the same codebase, but may have different versions active.
- Using version control allows tracking all changes and enables **consistent deployment** across environments.

Summary:

"A twelve-factor app is based on a single codebase, version-controlled, which is deployed across multiple environments. The same codebase is shared by all deploys, even if different versions are active."

Feel free to ask if you'd like more details or clarification!