#DemocratizeAI

fuse | machines

Python Ecosystem: Building a Twelve-Factor App

AI Fellowship 2025

Introduction

Imagine you're building a web application that needs to

- run reliably,
- Scale easily to handle many users,
- Easier for multiple developers to work on and deploy.

Scalable

Reliable

Collaborative

Introduction

- A methodology for building modern, scalable, and maintainable software applications,
- Developed by Heroku, based on best practices for cloud-native applications,
- Enables rapid development, deployment, and scaling in cloud environments,
- Ensures applications are scalable, portable across environments, and easier to maintain.

The Need for 12 Factor Apps

To Overcome Challenges in Modern Software Development

- Diverse deployment environments – cloud, on-premise, hybrid setups
- Demand for continuous delivery – frequent, reliable updates with minimal downtime
- Scalability needs – handle unpredictable workloads and user growth with ease

1. Codebase

One codebase tracked in revision control (like Git), many deploys (e.g., staging, production, developer environments).

Best Practices:

- ❑ Use a version control system (e.g., Git) to manage the codebase.
- ❑ Maintain a single repository per application.
- ❑ Exclude environment-specific configuration files from the codebase.

Bad Practices:

- ❑ Maintaining multiple codebases for different deployments.
- ❑ Not using version control.
- ❑ Including environment-specific configurations in the codebase.

2. Dependencies

Explicitly declare and isolate all dependencies, avoiding reliance on system-wide packages.

Best Practices

- Use a dependency management tool (e.g., pip - python, npm - node)
- Declare all dependencies in requirements.txt file(Python) or package.json(Node.js)
- Configure a virtual environment for isolating dependencies (python-venv)

Bad Practices

- Relying on system wide installed packages (base environment)
- E.g. keeping just **pandas** in requirements.txt instead of **pandas==1.5.2**

3. Config

Store configuration (credentials, environment-specific settings, API keys) in the environment, not in the code.

Best Practices

- Enable the same codebase to deploy across multiple environments unchanged.
- Keep configuration out of the codebase.
- Use environmental variables for configuration(E.g. **os.getenv("API_KEY")** by using **export API_KEY="secretkey"** or using .env file and python-dotenv to load the variables)
- Same codebase for all environments:
  - DEBUG = False if ENV == "production" else True

3. Config

Bad Practices

- Hardcoding configuration values in the source code.
  - DATABASE_URL = "postgres://user:pass@host/db"
  - SECRET_KEY = "hard coded secret”
- Storing sensitive data (e.g., passwords) in the codebase.
  - API_KEY = "sk_test_abc123" # stored directly in source code
- Using different codebases for different environments.
  - dev_config.py vs prod_config.py

4. Backing services

Treat backing services (e.g., databases, caches) as attached resources, accessed via configuration.

Best Practices:

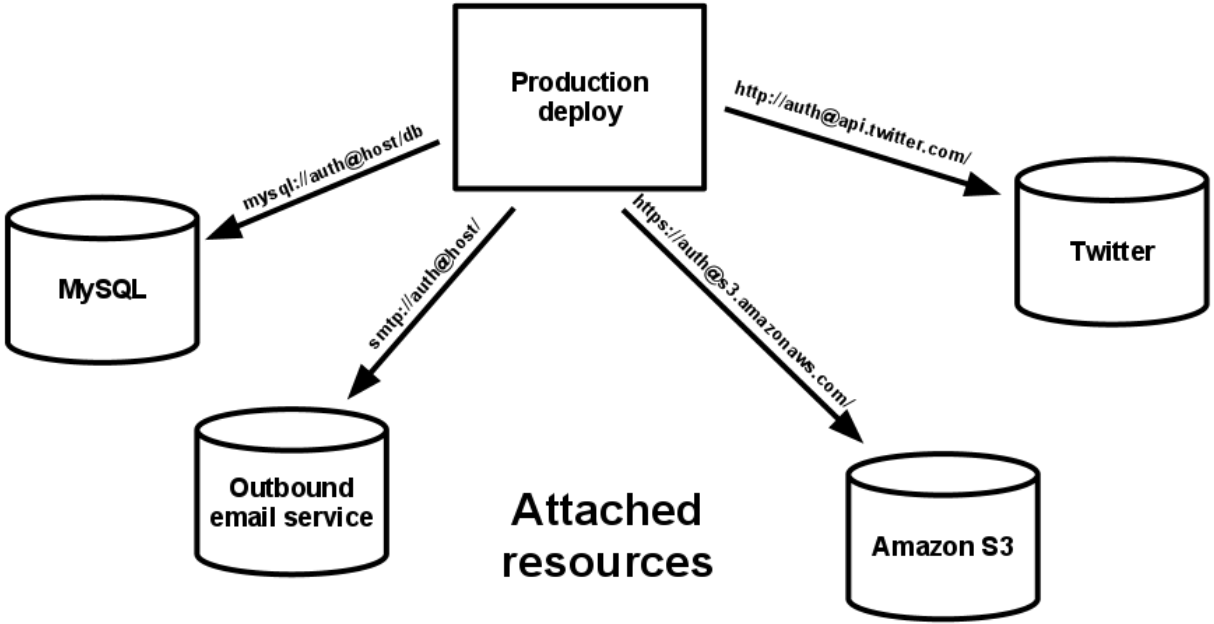
- ❑ Allow easy switching between backing services (e.g., local vs. production)
- ❑ Resources can be attached to and detached from deploys at will,
- ❑ Access services using URLs or locators stored in environment variables
- ❑ Treat services (DBs, caches, queues) as interchangeable resources

Bad Practices:

- ❑ Tightly coupling the application to specific services.
- ❑ Hardcoding service details in the application.

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.



5. Build, release, run

Best Practices:

- **Strictly separate the build, release, and run stages of deployment.**
  - **Build:** Transform code into a deployable artifact (e.g., compiling source code)
  - **Release:** Combine the build artifact with configuration settings
  - **Run:** Execute the application in the target environment using the release
- Ensure immutability of releases (once release is created it shouldn't be altered)
- Consistent processes across environments (Same build and release procedures for development, staging, and production)

5. Build, release, run

Strictly separate the build, release, and run stages of deployment.

Bad Practices:

- ❑ Modifying code directly in production
- ❑ Embedding configuration within the codebase (Hardcoding environment-specific settings)
- ❑ Deploying code without a proper build or release process can introduce untested changes and increase the risk of failures.
- ❑ Inconsistent environments

6. Processes

Run the application as one or more stateless processes.

Best practices:

- Each process should be treated as an immutable, replaceable unit of execution
- Execute as stateless, share-nothing processes
  - Data should survive beyond the life of a single process instance i.e. sessions, user uploads, job queues should reside in databases, object stores etc
- Data should persist in backing services rather than in memory or local filesystem
- **Benefits:** Horizontal scalability

6. Processes

Run the application as one or more stateless processes.

Bad practices:

- Storing state in process memory
  - Should avoid storing user sessions, counters in RAM as it hinders horizontal scalability
- Writing durable data to local disk
- Running background jobs and web servers in the same process

7. Port Binding

Export services by binding to a port, making the app self-contained.

Best Practices

- ❑ Use environment variables for port configuration **E.g. os.getenv("PORT")**
- ❑ Document and manage ports explicitly

Bad Practices

- ❑ Relying on orchestration tools to handle ports without app awareness.
- ❑ Multiplexing unrelated services on a single port
- ❑ Hardcoding port numbers
- ❑ Using conflicting or non-standard ports.

8. Concurrency

Scale out by adding more process instances (horizontal scaling).

Best Practices:

- Design for horizontal scaling by adding lightweight processes.
- Scale out with distinct process types (define separate process type to scale independently)
- Use threads or processes within the app for concurrency.
- Maintain stateless, share-nothing processes
  - `gunicorn --workers 4 --bind 0.0.0.0:5000 app.main:app`

Bad Practices:

- Relying solely on vertical scaling (more resources per instance)
- Not preparing for concurrent execution, causing bottlenecks
- Mixing all workloads in one process

9. Disposability

Ensure processes start quickly and shut down gracefully for robustness - No sudden death

- Elastic scaling, faster deployments (rolling updates), quick recovery from crashes, and robust handling of infrastructure changes.

Best Practices:

- ❑ Processes should be disposable – they can be started or stopped at a moment's notice
- ❑ Minimize initialization tasks (e.g., lazy-load non-critical modules)
- ❑ Handle SIGTERM signals for cleanup - Graceful Shutdown

Bad Practices:

- ❑ Heavy initialization (loading heavy model/large datasets) - Long startup times
- ❑ Ignoring shutdown signals
- ❑ Manual restarts over automation
- ❑ Embedding heavy migrations or tasks in startup path

10. Dev/Prod Parity

Keep development, staging, and production environments as similar as possible.

Best Practices:

- ❑ Use consistent backing services across environments
- ❑ Minimize differences in tools and configurations
- ❑ Use containerization (e.g., Docker) for consistency.

Bad Practices:

- ❑ Using different databases or services in dev vs. prod.
- ❑ Allowing significant configuration disparities.
- ❑ Skipping production-like testing environments.

11. Logs

Treat logs as event streams, not files managed by the app.

Best Practices:

- ❑ Write logs to stdout and stderr
- ❑ Use external tools for log aggregation and analysis **E.g. FluentD, Logplex, Cloudwatch**
- ❑ Avoid managing log files within the app

Bad Practices:

- ❑ Writing logs to disk files. (risking data loss)
- ❑ Not centralizing log collection.

12. Admin Processes

Run administrative tasks as one-off processes using the same codebase.

Best Practices:

- ❑ Execute admin tasks as one-off processes independently
- ❑ Use the same codebase and configuration
- ❑ Automate and version admin processes (Scheduled Model Retraining, files cleanup etc)

Bad Practices:

- ❑ Running admin tasks manually without scripts
- ❑ Embedding admin logic in the main app
- ❑ Not tracking admin scripts in version control

fuse | machines

Democratize AI

www.fusemachines.com