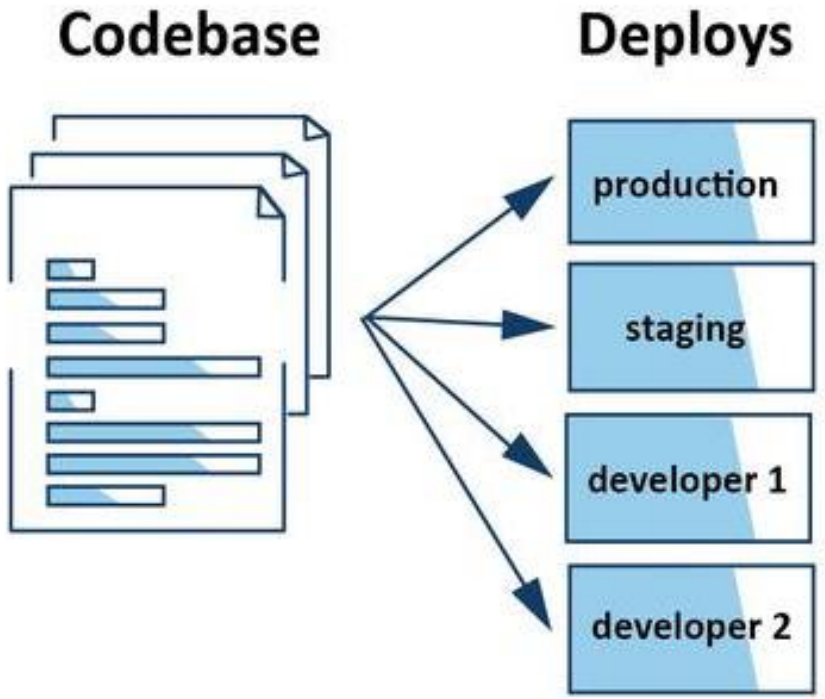
12-FACTOR APP: CONFIG (FACTOR III)

What is 12-Factor Config?

- Configuration is everything that varies between deploys (dev/stage/prod): credentials, endpoints, toggles.
- Store config in the environment, not in code.
- Why it matters
  - Portability across environments
  - Security: no secrets in VCS
  - Deploy the same build to multiple environments



Principles & Boundaries

- Single codebase; multiple deploys via environment values.
- No env-specific branches or hard-coded constants in the code.
- Runtime injection: values provided by the platform (container, CI, PaaS).
- Keep config small, flat, and explicit (strings).

Config vs Code: What Goes Where?

Config: varies per deploy

- Credentials: database passwords, API keys, tokens
- Addresses of backing services (DATABASE_URL, caches, queues)
- Per-deploy values: ports, hostnames, log level

Code: same in every deploy

- Business rules: password length, username format
- Algorithm parameters, e.g. hash iterations
- Routes and wiring between internal modules

Litmus test: *could the codebase be made open source right now without leaking a single credential?*

Anti-Pattern: Environment Groups

```
# ❌ One config file per environment: grows with every new deploy
config/development.json   {"db": "localhost",  "debug": true}
config/staging.json       {"db": "staging-db", "debug": false}
config/production.json    {"db": "prod-db",    "debug": false}
config/qa.json, demo.json, alice-laptop.json ...   ← keeps multiplying

# ✅ Independent variables, each set by the platform for each deploy
DATABASE_URL=postgresql://...
DEBUG=false
LOG_LEVEL=INFO

# Each value is managed on its own: no files to keep in sync,
# and no "which environment am I in?" logic in the code.
```

Reading Environment Variables in Pure Python

```
import os

DB_URL = os.getenv("DATABASE_URL")  # None if not set
SECRET_KEY = os.environ["SECRET_KEY"]  # KeyError if missing (fail fast)

# Provide safe defaults for non-secret toggles
DEBUG = os.getenv("DEBUG", "false").lower() == "true"

if not DB_URL or not SECRET_KEY:
    raise RuntimeError("Missing required configuration: DATABASE_URL and/or SECRET_KEY")
```

Local Development: .env with python-dotenv

```
# pip install python-dotenv
from dotenv import load_dotenv
import os

load_dotenv()  # loads variables from a .env file into process env

# Now use as normal
S3_BUCKET = os.environ["S3_BUCKET"]
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")
```

Configuration as a Typed Settings Object (pydantic)

```
# pip install pydantic
from pydantic import BaseModel, Field, ValidationError
import os

class Settings(BaseModel):
    database_url: str = Field(..., alias="DATABASE_URL")
    secret_key: str = Field(..., alias="SECRET_KEY")
    debug: bool = Field(False, alias="DEBUG")
    timeout_s: int = Field(10, alias="TIMEOUT_S")

def load_settings() -> Settings:
    try:
        # pydantic reads from a dict; we pass os.environ
        return Settings(**os.environ)
    except ValidationError as e:
        raise RuntimeError(f"Invalid/missing config: {e}")

settings = load_settings()
```

Flask Example: Config from Environment

```
from flask import Flask
import os

def create_app():
    app = Flask(__name__)
    app.config.update(
        SQLALCHEMY_DATABASE_URI=os.environ["DATABASE_URL"],
        SECRET_KEY=os.environ["SECRET_KEY"],
        JSON_SORT_KEYS=False,
    )
    return app

# gunicorn 'app:create_app'
```

FastAPI Example: Dependency-Injected Settings

```
from fastapi import FastAPI, Depends
from pydantic import BaseModel, Field
import os

class Settings(BaseModel):
    model_path: str = Field(..., alias="MODEL_PATH")
    threshold: float = Field(0.5, alias="THRESHOLD")

def get_settings() -> Settings:
    return Settings(**os.environ)

app = FastAPI()

@app.get("/healthz")
def healthz(settings: Settings = Depends(get_settings)):
    return {"status": "ok", "model_path": settings.model_path}
```

Node.js Example: Same Rules, Different Runtime

```
// config.js: read and validate once, at startup
const missing = ["DATABASE_URL"].filter((key) => !process.env[key]);
if (missing.length > 0) {
  console.error(`Missing required env vars: ${missing.join(", ")}`);
  process.exit(1);
}

export const config = {
  databaseUrl: process.env.DATABASE_URL,
  port: Number(process.env.PORT ?? 3000), // safe default: not a secret
};
```

```
// ⚠ Without the check, pg's new Pool({ connectionString: undefined })
//   silently connects to localhost:5432: wrong database, no error.
```

Secrets & Sensitive Values

- Do not commit secrets to Git; treat .env as local-only and git-ignored.
- Commit a .env.example listing every variable (with placeholder values) so others know what to set.
- Prefer platform secret stores (injected as env at runtime).
- Rotate secrets by updating environment values; no code changes required.
- Limit blast radius: use least-privilege credentials per environment.

Anti-Pattern: Hard-Coded Secrets

```
# ❌ Never do this
API_KEY = "sk_live_REDACTED"  # committed to Git
DB_URL = "postgresql://user:pass@prod-db:5432/app"

# Hidden cost: rotating requires code changes and redeploy of a new build.
```

Anti-Pattern: Insecure Defaults for Secrets

```
import os

# ❌ Looks harmless and "works everywhere", including production
app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY", "dev-only-secret")

# Forget SECRET_KEY in production → the app still starts, signing
# sessions with a key anyone can read in your Git history.

# ✅ Defaults only for non-secrets; secrets must be provided
app.config["SECRET_KEY"] = os.environ["SECRET_KEY"]  # KeyError if missing
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")             # safe default
```

Better: Fail Fast When Required Config Is Missing

```
import os

REQUIRED = ["DATABASE_URL", "SECRET_KEY"]
missing = [k for k in REQUIRED if k not in os.environ]
if missing:
    raise SystemExit(f"Missing required env vars: {missing}")
```

Feature Flags & Safe Defaults

```
import os

FEATURE_NEW_RECOMMENDER = os.getenv("FEATURE_NEW_RECOMMENDER", "false").lower() == "true"

def recommend(user_id: str):
    if FEATURE_NEW_RECOMMENDER:
        return recommend_v2(user_id)
    return recommend_v1(user_id)
```

Database Connections from Env

```
import os
import psycopg2  # pip install psycopg2-binary

conn = psycopg2.connect(os.environ["DATABASE_URL"])
cur = conn.cursor()
cur.execute("SELECT 1;")
print(cur.fetchone())
cur.close()
conn.close()
```

ML Settings: Paths, Registry, and Tracking URIs

```
import os
import mlflow  # pip install mlflow

MLFLOW_TRACKING_URI = os.getenv("MLFLOW_TRACKING_URI", "file:./mlruns")
MODEL_REGISTRY = os.getenv("MODEL_REGISTRY", "./models")
ARTIFACT_STORE = os.getenv("ARTIFACT_STORE", "./artifacts")

mlflow.set_tracking_uri(MLFLOW_TRACKING_URI)
mlflow.set_experiment(os.getenv("MLFLOW_EXPERIMENT", "default"))

with mlflow.start_run(run_name="train"):
    # ... log metrics, params, artifacts
    pass
```

Containers: Injecting Config with Docker Compose

```
# docker-compose.yml: committed to Git, contains NO secret values
services:
  db:
    image: postgres:18-alpine
    environment:
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set it in .env}
  backend:
    image: myuser/auth-app:backend
    environment:
      DATABASE_URL: postgres://app:${POSTGRES_PASSWORD}@db:5432/auth
# ${VAR:?msg} → Compose refuses to start, printing msg, if VAR is unset

# .env: read automatically by Compose, listed in .gitignore
POSTGRES_PASSWORD=change-me

# .env.example: committed, documents which variables to set
POSTGRES_PASSWORD=
```

Environment Variables Are Not Fully Secret

```
# Anyone with access to Docker can read a container's environment
$ docker inspect auth-db-1 --format '{{json .Config.Env}}'
["POSTGRES_PASSWORD=change-me", "PATH=/usr/local/sbin:...", ...]

# Safer for secrets: mount them as files (many images support *_FILE)
services:
  db:
    image: postgres:18-alpine
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets: [db_password]
secrets:
  db_password:
    file: ./db_password.txt   # git-ignored, never baked into the image
```

Frontend Config: Build vs Runtime

Build time: baked in

- Bundlers replace values during the build (e.g. Vite's import.meta.env.VITE_API_URL)
- One build per environment breaks "build once, deploy many"
- The bundle is public: no secrets in it

Runtime: injected at start

- Relative URLs behind a reverse proxy: fetch("/api/...") → nginx → backend
- The nginx image fills /etc/nginx/templates/*.template from env vars at start
- The same image runs in every environment

Build, Release, Run (12-Factor tie-in)

- Build (create artifact) is environment-agnostic.
- Release = Build + Config (environment variables).
- Run executes a specific release; promoting to prod means switching config, not rebuilding.

Config Across the CI/CD Pipeline

Stage	Where config lives	Example
Local development	.env (git-ignored) + committed .env.example	DATABASE_URL for npm run dev
CI: GitHub Actions	Repository secrets and variables	DOCKERHUB_TOKEN (secret), DOCKERHUB_USERNAME (variable)
Image: Docker Hub	Nowhere: the image carries no config	auth-app:backend runs unchanged everywhere
Deploy machine	.env next to the deploy compose file, shell env	POSTGRES_PASSWORD, WEBHOOK_SECRET

Least privilege: Read & Write token for CI, Read-only token for the deploy machine.

Common Pitfalls & Mitigations

- Pitfall: multiple .env files drifting → Mitigation: treat .env as local only; source of truth is the platform.
- Pitfall: implicit defaults hide errors → Mitigation: require critical values and validate types.
- Pitfall: secret leakage in logs → Mitigation: redact and never print sensitive values.

More Pitfalls & Mitigations

- Pitfall: "false" is a non-empty string, so it is truthy → Mitigation: parse booleans and numbers explicitly, or use a typed settings object.
- Pitfall: env vars visible via docker inspect or process listings → Mitigation: mount secrets as files (*_FILE) or use a secrets manager.
- Pitfall: frontend values baked in at build time → Mitigation: relative URLs or runtime injection.

Case Study: Auditing Our Auth App

Config item	Where it is now	Verdict
Database password	Hard-coded twice in docker-compose.yml	❌ Secret committed to Git
DATABASE_URL, PORT (backend)	Read from process.env	⚠ If unset, silently uses localhost:5432
SECRET_KEY (Flask app)	os.environ.get(..., "dev-only-secret")	❌ Insecure default
Backend address (nginx)	Hard-coded in nginx.conf, baked into the image	⚠ Config inside the image
API URL (frontend)	Relative /api path	✅ Nothing baked into the build
WEBHOOK_SECRET (deploy server)	Env var; refuses to start without it	✅ Fails fast
Docker Hub credentials (CI)	GitHub secrets and variables	✅ Out of the repo
Playwright baseURL	Hard-coded localhost:5173	⚠ Can't test other deployments

Exercise: Make Our App 12-Factor

- Move every secret out of docker-compose.yml into a git-ignored .env; commit a .env.example
- Make the backend refuse to start when DATABASE_URL is missing
- Make the backend address in nginx configurable when the container starts
- Make the Playwright baseURL configurable, to test any deployment
- **Done when:**
  - **Public-repo test:** git grep -i password finds no real secret
  - **Same-image test:** one backend image runs against two databases, changing only env vars

Key Takeaways

- Config is whatever changes between deploys; everything else is code
- One environment variable per value, not one file per environment
- Secrets never enter Git or the image: inject them at runtime
- Validate at startup and fail fast; defaults only for non-secrets
- Build once: the same image runs in CI, staging and production
- Related: IV Backing services · V Build, release, run · X Dev/prod parity