



Long-Term SDLC Blueprint for a Personal or Small-Team Application

Executive summary

This report proposes a full, end-to-end Software Development Life Cycle (SDLC) optimised for a **personal or small-team (1-10; assumed 1-3 initially)** **web application** that must remain maintainable for years, keep technical debt low, and support later evolution from a **modular monolith (single repo, single deployment, single database)** into independently deployable services if/when justified. The baseline assumption ("monolith first") is strongly supported by industry experience: even advocates of microservices acknowledge a meaningful "microservice premium" (operational and coordination costs) and the difficulty of getting service boundaries correct early. 1

The core idea is to **front-load "change-enabling" practices** (clear module boundaries, contracts, migrations discipline, observability, automated delivery, security controls) while keeping implementation lightweight enough for a solo developer. This aligns with a pragmatic "build and iterate quickly without locking in accidental complexity" stance and is consistent with modern guidance on evolutionary design, safe interface change, and continuous delivery. 2

Top recommendations (high leverage)

- **Architect for evolution, not distribution:** build a modular monolith with strict internal boundaries and "published interfaces" between modules. This reduces later decomposition pain and matches "monolith first" rationale. (*Effort: Medium; Benefit: High; Trade-off: upfront discipline and some boilerplate.*) 3
- **Treat your API as a contract:** write OpenAPI-first (OpenAPI latest published version is tracked by the OpenAPI Initiative), keep DTOs explicit, and use safe versioning patterns. (*Effort: Medium; Benefit: High; Trade-off: contract maintenance.*) 4
- **Automate quality with a test portfolio (pyramid-shaped):** many unit tests, fewer integration tests, minimal E2E; focus tests on domain logic and boundary contracts. (*Effort: Medium; Benefit: High; Trade-off: initial investment.*) 5
- **Ship safely and repeatedly:** implement CI/CD that runs linting, tests, builds, migrations, and deploy steps. This aligns to continuous delivery principles and reduces long-term drift. (*Effort: Medium; Benefit: High; Trade-off: pipeline upkeep.*) 6
- **Operate intentionally:** add structured logs, metrics, tracing context, plus minimal SLOs/alerts. Focus on the "signals that represent user experience". (*Effort: Medium; Benefit: High; Trade-off: some instrumentation overhead.*) 7
- **Security-by-default (finance sensitivity):** align baseline controls to OWASP Top 10 and OWASP ASVS; use strong password storage guidance, secure session practices, TLS, and dependency scanning. (*Effort: Medium-High; Benefit: High; Trade-off: scope control needed.*) 8

Assumptions, scope, and operating principles

Scope and objectives

This SDLC blueprint is designed to:

- Provide a full SDLC tailored for personal/small-team projects with long-term maintainability and low technical debt as primary goals. ⁹
- Prioritise practices proven in industry and workable for small teams (monolith/modular monolith first), while including when and how to split into services. ¹⁰
- Cover architecture, code organisation, data ownership, API design, testing, CI/CD, deployment, observability, security, compliance notes, documentation, release management, migrations, and team/process practices. ¹¹
- Provide concrete artefacts: checklists, templates, directory structures, OpenAPI/DTO templates, CI pipelines, logging/metrics schemas, testing matrices, and a migration playbook for service extraction. ¹²

Key attributes and dimensions

Assumptions used throughout (adjust as needed):

- Team size: **1-3 initially** (guidance generalises to 1-10).
- Deployment baseline: **single repo + single deployment + single database** initially. ¹³
- Platforms: **web backend + web frontend** (examples for Django, FastAPI, Node/Next.js). ¹⁴
- Data sensitivity: finance implies higher risk; implement security controls aligned to OWASP Top Ten and stronger assurance with OWASP ASVS (select an ASVS level appropriate to your threat model). ¹⁵
- Availability/scalability: moderate availability; adopt SLO thinking and scale only when signals justify it. ¹⁶
- Budget: limited; prefer open-source tooling and low-cost managed services; apply “Twelve-Factor” principles for portability and reduced ops burden. ¹⁷
- Time horizon: years; treat technical debt explicitly (deliberate/prudent vs reckless/inadvertent) and pay “interest” down continuously. ⁹

Operating principles that keep technical debt low

1. **Optimise for change:** design for safe refactoring through modular boundaries and contracts before designing for distribution. ³
2. **Automate everything you repeat:** releases and migrations should be intentional and reproducible, not artisanal. ¹⁸
3. **Prefer reversible decisions:** use feature flags and “parallel change / expand-migrate-contract” for risky changes. ¹⁹
4. **Measure outcomes:** use DORA metrics to monitor delivery performance and SLOs to anchor reliability to user experience. ²⁰
5. **Security is an SDLC property, not a phase:** integrate secure development practices and supply-chain controls (NIST SSDF, SLSA) proportionally. ²¹

SDLC walkthrough with phase checklists

This section gives a practical SDLC that you can run as a loop (plan → build → verify → release → operate → learn). It is aligned with modern continuous delivery thinking (ship changes safely, quickly, sustainably) rather than heavyweight stage-gates. ²²

Requirements and product definition

A small team needs **just enough process**: clarity of intent, explicit constraints, and a small backlog that supports iteration.

Recommended artefacts (lightweight, high impact):

- A **one-page product brief**: target users, core workflows, success metrics, non-goals. (*Effort: Low; Benefit: High; Trade-off: planning time.*)
- A **risk register** for finance features: transaction correctness, privacy, fraud/abuse scenarios, recovery expectations. (*Effort: Low; Benefit: High; Trade-off: none.*)
- A “**definition of done**” that includes tests, docs, migration plan, and observability hooks. (*Effort: Low; Benefit: High.*)
- A minimal **issue tracker** (GitHub Issues, GitLab Issues) with labelled epics/features/bugs. (*Effort: Low; Benefit: Medium; Trade-off: discipline.*)

Checklist (requirements phase)

- Capture critical domain invariants in plain language (e.g., “a posted transaction cannot change amount; corrections are new transactions”).
- Decide data sensitivity boundaries (what is confidential vs public; what must be encrypted/ hashed; what must never be logged). ²³
- Identify “published interfaces” early: external API endpoints, internal module interfaces, and data export formats. ²⁴

Design and architecture

For a small team, the goal is not perfect architecture; it is **architecture that remains easy to change**. The best-known failure mode is over-optimising early for scale (e.g., premature microservices), incurring the “microservice premium” before it is justified. ¹

Checklist (design phase)

- Choose baseline: modular monolith, single repo, single DB. ¹³
- Define modules around business capabilities (not technical layers) as a first approximation of bounded contexts. ¹
- For each module: document responsibilities, “does not own”, and the APIs it exposes (internal and external).
- For risky changes, plan using expand-migrate-contract (parallel change). ²⁵
- Decide how you will document architecture decisions (ADRs). (*Effort: Low; Benefit: Medium-High.*) ²⁶

Implementation

Implementation practices that pay dividends in small teams are those that reduce rework: consistency, explicit interfaces, static analysis, and disciplined migrations.

Checklist (implementation phase)

- Enforce formatting and linting via CI (not developer heroics). ²⁷
- Keep configuration out of code (12-factor config); keep logs as event streams (12-factor logs). ²⁸
- Use code review even for solo: review your own PR after a break; keep changes small (faster reviews, lower risk). ²⁹
- Prefer trunk-based or GitHub flow for small teams; avoid long-lived branches. ³⁰
- Track technical debt explicitly (e.g., “debt” labels; debt budget per iteration). ⁹

Testing and verification

Testing should be portfolio-based, not dogmatic. A pyramid-shaped mix (many unit tests, fewer integration tests, minimal E2E) is a widely-cited starting point, and large-scale practitioners emphasise avoiding excessive end-to-end tests due to cost and brittleness. ⁵

Checklist (testing phase)

- Unit-test domain rules first; integration-test DB and boundary behaviour next; reserve E2E for a few critical journeys. ⁵
- Include contract tests if you plan future service splitting (API contracts become service boundaries). ³¹
- Gate merges on automated tests; avoid “works on my machine” divergence (12-factor dev/prod parity). ³²

Release and deployment

Release should be repeatable and intentional; the SDLC should make it safe to deploy often, even if you choose not to. Release engineering at scale highlights the value of explicitly defining the steps from source to production—small teams benefit even more because you cannot afford fragile, manual releases. ³³

Checklist (release phase)

- CI pipeline produces a build artefact (container image recommended), runs DB migrations, deploys, and performs smoke checks. ³⁴
- Use feature flags for incomplete features, but manage them carefully (flags add complexity and must be removed). ³⁵
- Tag releases and maintain a changelog (SemVer as a useful convention if you publish APIs). ³⁶

Operations and maintenance

Operations is where long-term maintainability is won or lost: without observability, safe changes become guesswork. SRE guidance formalises this with metrics tied to user experience (SLIs/SLOs), and emphasises monitoring as a way to understand system behaviour and support reliable operation. ³⁷

Checklist (operations phase)

- Structured logs + request IDs + trace context; never log secrets or sensitive values. ³⁸
- Metrics with clear types (Prometheus counters/gauges/histograms/summaries). ³⁹
- Backups tested by restore drills (finance data implies “recovery correctness” is part of product).
- Dependency scanning and patch cadence (software supply-chain risk). ⁴⁰

Architecture and modularity patterns for long-term evolution

Monolith first, but with service-minded boundaries

A “monolith first” strategy is repeatedly observed as the successful path: most success stories start with a monolith then break up, and many systems built as microservices from scratch struggle due to the complexity premium and early boundary mistakes. ¹

A modular monolith is therefore a **strategic compromise**: one deployment and (initially) one database, but with module boundaries designed as if the modules *could* become services later. That “design as if it might become remote” mindset works because refactoring across service boundaries is harder than inside a monolith. ³

Recommendation: “service-shaped modules”

Rationale: reduces future migration cost; preserves speed now. ⁴¹

Trade-offs: requires discipline (no cross-imports, clear interfaces). ²⁴

Effort: Medium. Benefit: High.

Monolith vs microservices trade-offs

A small team should treat microservices as an investment with ongoing operational cost rather than a default. Microservices are independently deployable services communicating via lightweight mechanisms and require automation to work well; this has benefits in large/complex systems but is not free. ⁴²

Dimension	Modular monolith (baseline)	Microservices (later option)
Delivery speed (early)	High (single build/deploy path) ⁴³	Often slower initially due to “microservice premium” (more pipelines, more ops) ¹
Boundary correctness	Easier to refactor boundaries internally ¹³	Harder to change boundaries once distributed ⁴⁴
Operational complexity	Lower (one runtime, fewer moving parts) ¹³	Higher (service discovery, monitoring, tracing, versioning, rollout coordination) ⁴⁵
Scaling	Scale the whole app or key components horizontally; can be enough for moderate needs ³²	Fine-grained scaling per service, often with more infrastructure ⁴⁶
Reliability isolation	Failures can cascade within one process if not controlled	Better isolation if designed well, but more failure modes (networks, timeouts) ⁴⁷

Dimension	Modular monolith (baseline)	Microservices (later option)
Data management	Simpler transactions (single DB) <small>48</small>	Distributed data consistency challenges; requires patterns (sagas, outbox, etc.) <small>49</small>

❑image_group❑{"layout":"carousel","aspect_ratio":"16:9","query":[{"modular monolith architecture diagram","microservices architecture diagram","C4 model diagram example","strangler fig pattern diagram"],"num_per_query":1}❑

Practical modular monolith patterns

Published internal APIs (“module contracts”)

Treat internal module boundaries like published interfaces; changing them should be planned with “parallel change” (expand → migrate → contract). 25

Effort: Medium. Benefit: High. Trade-off: more explicit interfaces.

Feature flags for controlled rollouts

Feature toggles allow behaviour changes without deploys, but they introduce complexity and require lifecycle management (avoid “flag debt”). 35

Effort: Medium. Benefit: Medium-High. Trade-off: complexity and cleanup.

Architecture diagrams and decision records

Use light standards: C4 for diagrams and ADRs for decisions. These are designed to make architecture communication consistent and keep decision history transparent. 50

Effort: Low. Benefit: Medium-High. Trade-off: paper-cut overhead unless kept brief.

Modular monolith diagram snippet

Renderable Mermaid suggestion (adapt names to your finance + dev domains):

```
flowchart TB
    subgraph UI[Frontend]
        Web[Web UI (Next.js)]
    end

    subgraph App[Backend (Single Deployable Unit)]
        API[HTTP API Layer]
        subgraph Core[Core / Shared]
            Auth[AuthN/AuthZ]
            Audit[Audit Logging]
            Notif[Notifications]
        end
        subgraph Finance[Finance Module]
            FinAPI[Finance Internal API]
            FinDomain[Domain Rules]
            FinRepo[Persistence Adapter]
        end
        subgraph Dev[Dev Module]
            DevAPI[Dev Internal API]
        end
    end
```

```

    DevDomain[Domain Rules]
    DevRepo[Persistence Adapter]
end
API --> FinAPI --> FinDomain --> FinRepo
API --> DevAPI --> DevDomain --> DevRepo
FinAPI --> Audit
DevAPI --> Audit
Auth --> API
end

subgraph DB[(Single Database - PostgreSQL)]
Tables[(Schemas/Tables)]
end

FinRepo --> Tables
DevRepo --> Tables
Web --> API

```

This supports later extraction because modules communicate via explicit internal APIs, not direct cross-imports or cross-table writes. [51](#)

Directory structure templates

These are examples designed for modularity and testability. They are deliberately “boring” because small teams benefit from convention and predictability. [29](#)

Django modular monolith layout

Django differentiates “project” vs “apps”; each app is a Python package, and settings are designated via an environment variable (pattern-friendly for multi-environment config). [52](#)

```

repo/
backend/
  manage.py
  pyproject.toml
  config/
    settings/
      base.py
      dev.py
      prod.py
    urls.py
    asgi.py
    wsgi.py

core/
  apps.py
  domain/
    policies.py
  services/
    auth_service.py

```

```

audit_service.py
api/
    serializers.py
    views.py
    urls.py

finance/
    apps.py
    domain/
        models.py          # domain objects / invariants
        rules.py           # pure business rules
    services/
        commands.py       # state changes (write)
        queries.py         # read models/materialised views (read)
    api/
        serializers.py
        views.py
        urls.py
    migrations/

dev/
    apps.py
    domain/
        models.py
        rules.py
    services/
        commands.py
        queries.py
    api/
        serializers.py
        views.py
        urls.py
    migrations/

tests/
    unit/
    integration/
    e2e/
frontend/
infra/

```

Guidance: keep domain rules “pure” (easy to unit-test); keep APIs thin; route orchestration in services.

53

FastAPI modular monolith layout

FastAPI explicitly supports structuring larger applications with routers; you build route modules then include them in the main application.

54

```

repo/
  backend/
    pyproject.toml
  app/
    main.py          # create FastAPI(), include routers
    core/
      config.py      # env-based config
      security.py
      audit.py
      observability.py
  finance/
    api.py          # APIRouter
    schemas.py      # Pydantic DTOs
    domain.py       # invariants/rules
    service.py      # use-cases
    repo.py         # DB adapter
  dev/
    api.py
    schemas.py
    domain.py
    service.py
    repo.py
  db/
    session.py
    migrations/     # Alembic
  tests/
    unit/
    integration/
    contract/
    e2e/

```

Pydantic models provide explicit schema/validation via `BaseModel`, making DTO patterns natural with FastAPI. [55](#)

Alembic supports autogeneration of migrations, but (as with Django) generated migrations require review to avoid unsafe diffs. [56](#)

Next.js frontend layout (brief)

Next.js documents conventions for project structure in the `app` directory and route handlers; use feature folders for domain-aligned UI components and pages. [57](#)

```

frontend/
  app/
    (auth)/
      login/
        page.tsx
    finance/
      dashboard/
        page.tsx

```

```
transactions/
  page.tsx
dev/
  projects/
    page.tsx
components/
  ui/
  finance/
  dev/
lib/
  api-client/
  auth/
  public/
```

Data, API contracts, and integration design

Default database choice for finance: relational + transactional correctness

For a finance-oriented domain, a relational database with well-understood transactional guarantees is typically the safest default. PostgreSQL documentation explicitly highlights transaction guarantees and durability expectations using a banking example, which aligns closely to finance correctness needs. ⁵⁸
Effort: Low (choose Postgres early). Benefit: High. Trade-off: relational modelling discipline.

Schema migrations as a first-class discipline

Treat migrations as version control for your schema; Django explicitly frames migrations this way (“makemigrations” packages changes; “migrate” applies them). ⁵⁹

For cross-component change or future service extraction, use **expand-migrate-contract** (parallel change) to implement backwards-incompatible changes safely. ⁶⁰

Recommendation: migration policy

Rationale: schema drift is one of the fastest ways to create long-term debt. ⁶¹

Trade-offs: slower changes upfront; requires discipline around deprecating old schema.

Effort: Medium. Benefit: High.

Data ownership rules (monolith now, services later)

To enable later splitting, adopt a simple rule early:

- **One module owns the write-path for a set of tables (or schema).**
- Other modules consume data through the owning module’s interface (service API or internal service function), not by directly writing tables.

This is essentially “service boundaries without the network”; it reduces coupling and prepares you for extraction patterns like strangler fig and anti-corruption layers. ⁶²

Effort: Medium. Benefit: High. Trade-off: fewer “easy joins” across domains.

API design: contract-first OpenAPI + HTTP semantics

OpenAPI provides a standard, language-agnostic interface description so humans and tools can understand an API without inspecting code; the OpenAPI Initiative maintains the published spec (latest published version is listed on the official spec site). ⁶³

HTTP semantics—including status codes and method properties like idempotence—are defined by the IETF standard (RFC 9110). Treat these as “canonical truth” to avoid ad hoc behaviour. ⁶⁴
JSON as a data interchange format is standardised in RFC 8259. ⁶⁵

Recommendation: OpenAPI-first, DTO-first

Rationale: contracts reduce accidental coupling and simplify future service extraction and client evolution. ⁶⁶

Trade-offs: versioning decisions become explicit; you must maintain the spec.

Effort: Medium. Benefit: High.

Example API contract template (OpenAPI)

Below is a minimal OpenAPI 3.2-style skeleton you can expand. The point is to establish conventions: error format, versioning approach, auth scheme, idempotency header where needed. ⁶⁷

```
openapi: 3.0.3
info:
  title: Example API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /finance/transactions:
    post:
      summary: Create a transaction
      description: >
        Finance-critical endpoint; supports retries via idempotency key.
      parameters:
        - in: header
          name: Idempotency-Key
          required: false
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/CreateTransactionRequest"
    responses:
      "201":
        description: Created
        content:
          application/json:
```

```

    schema:
      $ref: "#/components/schemas/Transaction"
"400":
  description: Validation error
  content:
    application/json:
      schema:
        $ref: "#/components/schemas/Error"
components:
  schemas:
    CreateTransactionRequest:
      type: object
      required: [amount, currency, occurred_at]
      properties:
        amount:
          type: integer
          description: Minor units (e.g., cents)
        currency:
          type: string
        occurred_at:
          type: string
          format: date-time
        description:
          type: string
    Transaction:
      type: object
      required: [id, amount, currency, occurred_at, created_at]
      properties:
        id: { type: string }
        amount: { type: integer }
        currency: { type: string }
        occurred_at: { type: string, format: date-time }
        created_at: { type: string, format: date-time }
    Error:
      type: object
      required: [code, message, request_id]
      properties:
        code: { type: string }
        message: { type: string }
        request_id: { type: string }

```

DTO patterns (Pydantic / Django)

Pydantic models are classes inheriting from `BaseModel`, designed to define schema and validation—use them as boundary DTOs even if your internal domain model differs. (68)

Example (FastAPI/Pydantic style):

```

from pydantic import BaseModel, Field
from datetime import datetime

```

```

from typing import Optional

class CreateTransactionRequest(BaseModel):
    amount_minor: int = Field(..., ge=0)
    currency: str = Field(..., min_length=3, max_length=3)
    occurred_at: datetime
    description: Optional[str] = None

class TransactionDTO(BaseModel):
    id: str
    amount_minor: int
    currency: str
    occurred_at: datetime
    created_at: datetime

```

Idempotency and finance-critical endpoints

HTTP defines idempotent *methods*, but finance often needs idempotency guarantees even for operations that create resources. For payment-like or transaction-creating endpoints, a practical industry approach is an **Idempotency-Key** header so retries do not duplicate effects; Stripe documents this pattern explicitly for idempotent requests (including that the same key should return the same result on retries). ⁶⁹

Effort: Medium. Benefit: High. Trade-off: server-side storage for idempotency keys and careful semantics.

Quality engineering: testing strategy, CI/CD, and release management

Testing strategy for small teams: portfolio, not perfection

A “practical test pyramid” frames the essential idea: tests come in buckets of different granularity, and teams should generally have far more low-level tests than broad-stack tests. ⁷⁰

Large-scale practice also suggests a rough initial split (e.g., 70/20/10 unit/integration/E2E) as a starting point, recognising the exact mix varies by system. ⁷¹

Recommendation: domain-first unit tests + targeted integration + minimal E2E

Rationale: reduces long-term cost; keeps feedback fast. ⁵

Trade-offs: some bugs emerge only at full-stack; mitigate with a small number of E2E smoke tests.

Effort: Medium. Benefit: High.

❑image_group❑{"layout":"carousel","aspect_ratio":"16:9","query":["software testing pyramid diagram unit integration e2e","contract testing diagram","DORA metrics diagram deployment frequency lead time"],"num_per_query":1}❑

Suggested testing pyramid chart and data points

Suggested chart: **stacked bar chart** (or pyramid illustration) showing proportions.

Data points to plot (starting heuristic):

- Unit: **70%**
- Integration: **20%**
- End-to-end: **10%** 72

Test matrix (actionable)

Test type	Primary goal	What to test (examples for finance)	Recommended emphasis	Notes
Unit	Validate domain rules fast	rounding rules; ledger invariants; state transitions; permission checks	High 73	Pure functions / domain services
Integration	Verify DB + adapters	transaction atomicity; constraints; migrations; repository correctness	Medium 74	Use real Postgres in CI if possible
Contract	Stabilise boundaries	OpenAPI schema compatibility; consumer/provider expectations	Medium (if planning split) 75	Builds “service readiness”
E2E	Prevent catastrophic regressions	login → create transaction → report; dev workflow create task → mark done	Low 71	Keep a few, keep them stable

Sample finance-critical unit tests (illustrative)

- “Posting a transaction updates account balance with correct sign and currency rules.”
- “Corrections create compensating entries; original is immutable.”
- “Duplicate Idempotency-Key returns same transaction ID and does not change balances.” 76

CI/CD pipeline template (GitHub Actions)

GitHub Actions defines workflows in YAML; this is the official workflow syntax. 27
The following template assumes containerised deployment (Docker) and a simple target (VPS via SSH). Dockerfile semantics are documented in Docker’s reference. 77

This pipeline is intentionally “small-team realistic”: lint + tests + build image + migrate + deploy + smoke check.

```
name: ci-cd

on:
  push:
    branches: [ "main" ]
  pull_request:

  jobs:
```

```

test:
  runs-on: ubuntu-latest
  services:
    postgres:
      image: postgres:16
      env:
        POSTGRES_USER: app
        POSTGRES_PASSWORD: app
        POSTGRES_DB: app_test
    ports:
      - 5432:5432
    options: >-
      --health-cmd "pg_isready -U app"
      --health-interval 10s
      --health-timeout 5s
      --health-retries 5
  env:
    DATABASE_URL: postgresql://app:app@localhost:5432/app_test
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-python@v5
    with:
      python-version: "3.12"
  - name: Install
    run: |
      python -m pip install -U pip
      pip install -r backend/requirements.txt
  - name: Lint
    run: |
      ruff check backend
      ruff format --check backend
  - name: Unit + Integration tests
    run: |
      pytest -q backend/tests/unit backend/tests/integration
  - name: Migration safety check
    run: |
      python backend/manage.py makemigrations --check --dry-run
      python backend/manage.py migrate --noinput

build_and_deploy:
  needs: [ test ]
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'
  steps:
    - uses: actions/checkout@v4
    - name: Build image
      run: |
        docker build -t app:${{ github.sha }} backend
    - name: Push image to registry
      run: |
        echo "Push to GHCR/Docker Hub here"

```

```

- name: Deploy (SSH)
  env:
    SSH_HOST: ${secrets.SSH_HOST}
    SSH_USER: ${secrets.SSH_USER}
    SSH_KEY: ${secrets.SSH_KEY}
  run: |
    echo "Use ssh + docker compose pull/up here"
- name: Smoke check
  run: |
    echo "curl /healthz and fail if not 200"

```

Notes:

- The migration step mirrors Django's "migrations as schema version control." [59](#)
- If using Alembic, replace the migration commands with `alembic upgrade head` and enforce review for autogenerate output. [78](#)

Release and versioning conventions (small-team pragmatic)

- Use **SemVer** if you publish an API or distribute clients; it formalises how version numbers convey compatibility. [79](#)
- Treat interface changes as a design/migration problem; apply parallel change for breaking changes even inside a monolith to rehearse service evolution. [25](#)
- Track delivery health with DORA metrics (deployment frequency, lead time, change failure rate, time to restore/repair). [80](#)

Deployment, observability, security, compliance, and governance

Deployment models and tooling (budget-aware)

Default recommendation: Docker + Compose for a VPS

Rationale: reproducible environments, small-team friendly, low cost; Compose defines multi-container services, networks, and volumes using the Compose spec. [81](#)

Trade-offs: you operate the host (patching, backups, monitoring).

Effort: Medium. Benefit: High.

For larger scale or multi-service deployments, Kubernetes is a widely adopted open-source platform for managing containerised workloads, but it is usually overkill early for 1–3 developers. [82](#)

Example Docker Compose (backend + DB + metrics)

Compose exists to configure an application's services, networks, and volumes. [81](#)

```

services:
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app
      POSTGRES_DB: app

```

```

volumes:
  - db_data:/var/lib/postgresql/data
web:
  build: ./backend
  environment:
    DATABASE_URL: postgresql://app:app@db:5432/app
  ports:
    - "8000:8000"
  depends_on:
    - db
volumes:
  db_data:

```

Observability: logs, metrics, traces as first-class SDLC outputs

Logs: 12-factor recommends treating logs as event streams; do not treat logfiles as your interface—write to stdout and let tooling aggregate. ⁸³

Security logging should follow OWASP guidance (avoid sensitive data; protect integrity; log meaningful security events). ⁸⁴

Metrics: Prometheus recommends adding instrumentation through client libraries; metrics types and best practices are documented in Prometheus docs. ⁸⁵

Grafana is widely used for Prometheus visualisation and alerting, with official integration guidance. ⁸⁶

Tracing and correlation: OpenTelemetry defines a vendor-neutral framework for traces/metrics/logs, and documents context propagation to correlate trace IDs and span IDs across boundaries—including log correlation. ⁸⁷

Recommendation: “structured telemetry baseline”

Rationale: without it, correctness and performance regressions become hard to diagnose; with it, you can evolve architecture more safely. ⁸⁸

Trade-offs: initial work to standardise fields and instrumentation.

Effort: Medium. Benefit: High.

Structured logging schema (example)

This schema is designed to be: searchable, correlatable, and safe.

- “Trace/Span correlation” aligns with OpenTelemetry guidance. ⁸⁹
- “Security logging vocabulary and sensitivity constraints” align with OWASP logging guidance. ⁸⁴

Example JSON log event:

```
{
  "ts": "2026-02-10T12:34:56.789Z",
  "level": "INFO",
  "service": "backend",
  "module": "finance",
  "event": "finance.transaction.posted",
}
```

```

    "request_id": "req_01HR...",
    "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736",
    "span_id": "00f067aa0ba902b7",
    "actor": { "user_id": "u_123", "role": "owner" },
    "entity": { "type": "transaction", "id": "txn_456" },
    "result": "success",
    "latency_ms": 27
}

```

Metrics schema (example): key metrics for finance + dev domains

Prometheus metric types (counter/gauge/histogram/summary) and when to use them are defined in Prometheus docs. [39](#)

Finance domain metrics (examples)

- Counter: `finance_transactions_posted_total{currency="GBP"}`
- Counter: `finance_idempotency_replays_total`
- Histogram: `finance_post_transaction_seconds_bucket` (latency distribution)
- Gauge: `finance_unreconciled_items`

Dev domain metrics (examples)

- Counter: `dev_tasks_created_total`
- Counter: `dev_tasks_completed_total`
- Histogram: `dev_task_cycle_time_seconds_bucket` (optional; calculated)

Minimal instrumentation snippet (Python)

Prometheus recommends exposing metrics via an HTTP endpoint using a client library. [85](#)

```

from prometheus_client import Counter, Histogram

transactions_posted = Counter(
    "finance_transactions_posted_total",
    "Total number of posted finance transactions",
    ["currency"]
)

post_latency = Histogram(
    "finance_post_transaction_seconds",
    "Latency for posting a transaction (seconds)"
)

def post_transaction(currency: str, fn):
    with post_latency.time():
        result = fn()
    transactions_posted.labels(currency=currency).inc()
    return result

```

Security checklist (finance-oriented baseline)

This checklist combines OWASP risk awareness (Top Ten), OWASP ASVS as a requirements catalogue, and practical secure engineering guidance (sessions, password storage, logging). [90](#)

Authentication and sessions

- Use secure password storage (OWASP Password Storage Cheat Sheet); prefer modern password hashing guidance. [91](#)
- Implement robust session management (timeouts, invalidation, secure cookie flags) per OWASP session guidance. [92](#)
- If using tokens, understand JWT properties (RFC 7519) and treat refresh/revocation carefully; OAuth 2.0 (RFC 6749) is the standard framework for delegated authorisation. [93](#)

Authorisation

- Default-deny and least privilege; test access control paths explicitly (broken access control is a top OWASP category). [94](#)

Transport and storage protection

- Enforce TLS for all traffic; TLS 1.3 is specified by RFC 8446. [95](#)
- Encrypt sensitive data at rest where appropriate (application-level encryption for highly sensitive fields; provider-managed encryption for disks/backups—provider choice is unspecified).
- Maintain audit logs with integrity protections; do not log sensitive data. [84](#)

Input validation and abuse controls

- Validate inputs; prevent injection issues (OWASP Top Ten). [96](#)
- Rate limit authentication and transaction endpoints; monitor anomalies (OWASP logging/monitoring guidance). [97](#)

Supply-chain security

- Use dependency scanning (OWASP Dependency-Check) and patch regularly. [98](#)
- Adopt secure development framework practices proportionally (NIST SSDF) and improve build integrity over time (SLSA). [99](#)

Compliance notes (general, not regime-specific)

- If your app stores or processes personal data, GDPR-like requirements may apply depending on jurisdiction and user base; consult the legal text for obligations and seek legal advice for your context. [100](#)
- If you store/process payment card data directly, PCI DSS sets a baseline of technical/operational requirements; many small teams avoid direct scope by using payment providers and not storing card data. [101](#)

Documentation templates (docs-as-code)

Use minimal but structured documents; ADRs are specifically intended to record architecturally significant decisions and their rationale. [26](#)

C4 provides a simple, consistent diagram approach. [102](#)

Recommended templates (copy/paste starters)

README.md (minimum sections)

- What the system does
- Local setup (Docker/Compose; env vars) 103
- How to run tests
- How to deploy
- Links to docs (architecture, runbook)

docs/architecture.md

- C4 diagrams (system/context/container) 104
- Key module boundaries and data ownership rules
- Major dependencies and backing services (12-factor “attached resources” mindset) 32

docs/runbook.md (lightweight on-call notes)

- How to check health (URLs, dashboards)
- Common failure modes and remedies
- How to roll back/deploy
- How to restore from backup

Evolution: decision criteria, migration playbook, and immediate next steps

Decision matrix: keep monolith vs split into services

Service extraction should be driven by **signals**, not fashion. Starting from monoliths has often ended in trouble when done without mature prerequisites, while successful stories typically involve decomposing a monolith once complexity warrants it. 1

When you do split, incremental patterns like strangler fig reduce risk by introducing a façade and routing traffic gradually. 105

Decision matrix (practical signals)

Signal	Meaning	Keep modular monolith when...	Consider splitting when...	Costs if you split	Benefits if you split
Team topology	Organisational scaling	1–3 devs; shared context	Multiple teams, independent roadmaps	More coordination + ops 1	Parallel delivery, ownership
Deployment cadence	Release coupling	One release train is fine	Modules need independent deploys	More pipelines, more monitoring 106	Reduced release risk per domain

Signal	Meaning	Keep modular monolith when...	Consider splitting when...	Costs if you split	Benefits if you split
Boundary stability	Domain understanding	Boundaries changing often	Boundaries stable for months	Harder refactors across services <small>44</small>	Clear contracts
Scaling profile	Performance hotspots	Overall load moderate	One module dominates and isolates scaling need	Operational complexity <small>107</small>	Targeted scaling
Reliability isolation	Blast radius	Failures manageable inside app	One domain needs strict isolation	Distributed failure modes <small>108</small>	Reduced blast radius
Data governance	Compliance / risk	Single DB acceptable	Separation required (audit, legal boundary)	Data migration complexity <small>49</small>	Stronger isolation and control

Migration playbook: splitting Finance out of the modular monolith

This playbook is designed to be reversible and incremental, combining:

- **Strangler fig:** introduce a façade and route gradually. 105
- **Anti-corruption layer:** protect the new service's domain model from legacy coupling. 109
- **Parallel change (expand-migrate-contract)** for interface and schema evolution. 60

Step-by-step plan (actionable)

Phase 1: prepare the monolith for extraction (weeks, not days)

1. **Stabilise module boundaries:** ensure all calls into finance go through a single internal interface (e.g., `FinanceService`). 24
2. **Add contract tests** for the finance interface (OpenAPI schemas + behavioural tests). 75
3. **Add observability** to finance flows: event logs + metrics; ensure trace correlation can work across boundaries later. 110

Phase 2: introduce an external seam (façade/proxy)

1. Create a **façade endpoint** (or routing layer) that can send finance traffic to either in-process finance module or an external service. 111
2. Use feature flags to route a small percentage of requests to the new path (if applicable), managing toggle lifecycle carefully. 35

Phase 3: data strategy (choose one; start simplest)

Option A (recommended for small teams): **shared DB initially, strict ownership**

- New finance service reads/writes its tables in the same DB; monolith accesses finance only via API

(stop direct DB writes from non-finance).

- Later migrate to separate DB using expand-migrate-contract. [112](#)

Option B: **dual writes (temporary) + reconciliation**

- Monolith writes to old store and new store; build reconciliation checks; remove dual write once stable. ThoughtWorks explicitly mentions preemptive double write/database compatibility as a safety technique in distributed deployment contexts. [113](#)

Option C: **event/outbox + replication** (more complex)

- Produce events from monolith; finance service consumes events; converges eventually.
- Higher setup cost; often overkill for 1-3 developers early. [49](#)

Phase 4: migrate functionality gradually

1. Implement finance service with an anti-corruption layer translating from “monolith-shaped” contracts to “finance-shaped” domain model. [109](#)
2. Migrate endpoint-by-endpoint using strangler routing; keep rollback easy (route back to monolith). [114](#)

Phase 5: contract and decommission

1. Once traffic is fully migrated and stable, remove old code paths and flags (contract phase). [25](#)
2. If needed, split the DB last (hardest step); use expand-migrate-contract and backfill scripts; rehearse with restore drills. [115](#)

Rollback strategies (must be designed upfront)

- Routing rollback: toggle traffic back to monolith (strangler façade enables this). [116](#)
- Schema rollback: avoid destructive schema changes until the contract phase; use parallel change. [60](#)
- Data rollback: keep append-only transaction logs where possible in finance; corrections as new entries reduce destructive updates (domain-level rollback). [117](#)

Migration flow diagram snippet (Mermaid)

```
sequenceDiagram
    participant Client
    participant Facade as Strangler Facade/Router
    participant Mono as Modular Monolith
    participant FinSvc as Finance Service
    participant DB as Database

    Client->>Facade: POST /finance/transactions
    alt Before migration (0%)
        Facade-->>Mono: route to in-process finance module
        Mono-->>DB: write transaction
        Mono-->>Facade: 201 Created
    else During migration (partial)
        Facade-->>FinSvc: route to finance service
        FinSvc-->>DB: write transaction (owned tables)
        FinSvc-->>Facade: 201 Created
```

```

else Rollback
  Facade->>Mono: route back to monolith
  Mono->>DB: write transaction
  Mono-->>Facade: 201 Created
end
Facade-->>Client: response

```

This reflects the façade-first incremental migration described in strangler fig guidance. ¹¹⁴

Prioritised immediate next steps (top 10 actions)

These are ordered to maximise “long-term maintainability per hour invested” for a developer starting today.

1. Establish repo structure (modules + core) and enforce boundary rules (“only call through services/interfaces”). (*Effort: Medium; Benefit: High.*) ³
2. Write an initial OpenAPI contract for your public API and adopt explicit DTOs (Pydantic/serialisers). (*Effort: Medium; Benefit: High.*) ¹¹⁸
3. Set up CI (GitHub Actions or GitLab CI) to run lint + unit + integration tests and fail PRs on violations. (*Effort: Medium; Benefit: High.*) ¹¹⁹
4. Implement migrations discipline (Django migrations or Alembic) and define “expand-migrate-contract” as your default for breaking changes. (*Effort: Medium; Benefit: High.*) ¹²⁰
5. Implement structured logs with request IDs and trace correlation fields from day one; adopt OWASP logging safety rules. (*Effort: Medium; Benefit: High.*) ¹²¹
6. Add baseline metrics (Prometheus) and a small dashboard/alerts set (Grafana), focusing on finance-critical operations. (*Effort: Medium; Benefit: High.*) ¹²²
7. Implement core security controls: password hashing per OWASP, session timeouts, TLS everywhere, and access-control tests. (*Effort: Medium-High; Benefit: High.*) ¹²³
8. Add dependency scanning (OWASP Dependency-Check or equivalent) and define a patch cadence; map practices lightly to NIST SSDF / SLSA to guide maturity. (*Effort: Medium; Benefit: High.*) ⁴⁰
9. Adopt a lightweight branching + review practice (GitHub flow or trunk-based; small PRs) and record major design choices as ADRs. (*Effort: Low-Medium; Benefit: Medium-High.*) ¹²⁴
10. Define your first SLOs (availability/latency for key endpoints) and track DORA metrics to prevent delivery degradation as the project grows. (*Effort: Medium; Benefit: High.*) ¹²⁵

References

The sources below are selected to prioritise primary/official and canonical guidance. They are cited inline throughout the report.

- `entity["people","Martin Fowler","software engineer"]`: monolith-first strategy; microservices definition; strangler fig; feature toggles; parallel change; test pyramid; technical debt. ¹²⁶
- `entity["organization","ThoughtWorks","global consultancy"]`: guidance on continuous deployment in distributed systems, including expand/contract and double-write considerations. ¹²⁷
- `entity["organization","OWASP","application security org"]`: OWASP Top Ten (latest release referenced), ASVS, Cheat Sheet Series (password storage, logging, sessions), Dependency-Check. ¹²⁸
- `entity["organization","OpenAPI Initiative","linux foundation project"]`: OpenAPI specification and published versions. ⁴

- Entity["company", "GitHub", "code hosting platform"]¹²⁹: GitHub Actions workflow syntax; GitHub flow.
 - Entity["company", "Docker", "container software company"]¹³⁰: Dockerfile reference; Compose specification/reference.
 - Entity["organization", "Prometheus", "monitoring project"]¹³¹ and Entity["company", "Grafana Labs", "observability company"]¹³²: metrics instrumentation/client libraries; metric types; Prometheus↔Grafana integration and dashboards/alerting.
 - Entity["organization", "OpenTelemetry", "observability project"]¹³³: observability framework and context propagation for cross-signal correlation.
 - Entity["organization", "Google", "technology company"]¹³⁴: SRE online book (SLOs, monitoring, release engineering) and testing guidance (unit/integration/E2E ratio heuristic).
 - Entity["organization", "NIST", "us standards agency"]¹³⁵: Secure Software Development Framework (SP 800-218) and digital identity guidelines (SP 800-63B).
 - Key standards (IETF RFCs): HTTP semantics (RFC 9110), JSON (RFC 8259), JWT (RFC 7519), OAuth 2.0 (RFC 6749), TLS 1.3 (RFC 8446).
 - Entity["book", "Site Reliability Engineering", "google sre book"]¹³⁶ online edition: monitoring, SLOs, release engineering chapters.
 - Entity["book", "Continuous Delivery", "humble and farley 2010"]¹³⁷ (concept site and canonical description of continuous delivery).
 - Entity["book", "Monolith to Microservices", "sam newman"]¹³⁸ (evolutionary migration framing for monolith → services).
 - Cloud provider architecture frameworks (provider-neutral principles): AWS Well-Architected pillars; Google Cloud Well-Architected Framework; Azure Well-Architected Framework.
-

[1](#) [2](#) [3](#) [10](#) [13](#) [41](#) [43](#) [44](#) [51](#) [126](#) **Monolith First**

<https://martinfowler.com/bliki/MonolithFirst.html>

[4](#) [67](#) **OpenAPI Specification v3.2.0**

https://spec.openapis.org/oas/v3.2.0.html?utm_source=chatgpt.com

[5](#) [31](#) [53](#) [70](#) [73](#) **The Practical Test Pyramid**

https://martinfowler.com/articles/practical-test-pyramid.html?utm_source=chatgpt.com

[6](#) [22](#) [136](#) **What is Continuous Delivery? - Continuous Delivery**

https://continuousdelivery.com/?utm_source=chatgpt.com

[7](#) [16](#) [37](#) [125](#) [132](#) **Defining slo: service level objective meaning**

https://sre.google/sre-book/service-level-objectives/?utm_source=chatgpt.com

[8](#) [90](#) [128](#) **OWASP Top Ten Web Application Security Risks**

https://owasp.org/www-project-top-ten/?utm_source=chatgpt.com

[9](#) **Technical Debt**

https://martinfowler.com/bliki/TechnicalDebt.html?utm_source=chatgpt.com

[11](#) **The Twelve-Factor App**

https://12factor.net/?utm_source=chatgpt.com

[12](#) [27](#) [119](#) [129](#) **Workflow syntax for GitHub Actions**

https://docs.github.com/actions/using-workflows/workflow-syntax-for-github-actions?utm_source=chatgpt.com

[14](#) [54](#) **Bigger Applications - Multiple Files**

https://fastapi.tiangolo.com/tutorial/bigger-applications/?utm_source=chatgpt.com

15 OWASP Top 10:2021

https://owasp.org/Top10/2021/?utm_source=chatgpt.com

17 32 The Twelve-Factor App

<https://12factor.net/>

18 33 Google SRE: Role of Release Engineer and Best Practices

https://sre.google/sre-book/release-engineering/?utm_source=chatgpt.com

19 35 Feature Toggles (aka Feature Flags)

https://martinfowler.com/articles/feature-toggles.html?utm_source=chatgpt.com

20 80 DORA's software delivery performance metrics

https://dora.dev/guides/dora-metrics/?utm_source=chatgpt.com

21 99 133 Secure Software Development Framework (SSDF) Version 1.1

https://csrc.nist.gov/pubs/sp/800/218/final?utm_source=chatgpt.com

23 38 84 97 Logging - OWASP Cheat Sheet Series

https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html?utm_source=chatgpt.com

24 25 60 Parallel Change

<https://martinfowler.com/bliki/ParallelChange.html>

26 Documenting Architecture Decisions - Cognitect.com

https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions?utm_source=chatgpt.com

28 103 Store config in the environment

https://12factor.net/config?utm_source=chatgpt.com

29 The Standard of Code Review | eng-practices

https://google.github.io/eng-practices/review/reviewer/standard.html?utm_source=chatgpt.com

30 Trunk Based Development

https://trunkbaseddevelopment.com/?utm_source=chatgpt.com

34 77 130 Dockerfile reference | Docker Docs

https://docs.docker.com/reference/dockerfile/?utm_source=chatgpt.com

36 79 Semantic Versioning 2.0.0 | Semantic Versioning

https://semver.org/?utm_source=chatgpt.com

39 Metric types

https://prometheus.io/docs/concepts/metric_types/?utm_source=chatgpt.com

40 98 OWASP Dependency-Check

https://owasp.org/www-project-dependency-check/?utm_source=chatgpt.com

42 45 46 47 106 107 Microservices

https://martinfowler.com/articles/microservices.html?utm_source=chatgpt.com

48 58 117 Documentation: 18: 3.4. Transactions

https://www.postgresql.org/docs/current/tutorial-transactions.html?utm_source=chatgpt.com

49 112 115 Parallel Change

https://martinfowler.com/bliki/ParallelChange.html?utm_source=chatgpt.com

50 102 104 C4 model: Home

https://c4model.com/?utm_source=chatgpt.com

52 Writing your first Django app, part 1

https://docs.djangoproject.com/en/6.0/intro/tutorial01/?utm_source=chatgpt.com

55 68 Models - Pydantic Validation

https://docs.pydantic.dev/latest/concepts/models/?utm_source=chatgpt.com

56 78 Auto Generating Migrations - Alembic's documentation!

https://alembic.sqlalchemy.org/en/latest/generators.html?utm_source=chatgpt.com

57 Getting Started: Project Structure

https://nextjs.org/docs/app/getting-started/project-structure?utm_source=chatgpt.com

59 74 120 Migrations | Django documentation

https://docs.djangoproject.com/en/6.0/topics/migrations/?utm_source=chatgpt.com

61 Evolutionary Database Design

https://martinfowler.com/articles/evodb.html?utm_source=chatgpt.com

62 105 Strangler Fig - Martin Fowler

https://martinfowler.com/bliki/StranglerFigApplication.html?utm_source=chatgpt.com

63 66 75 118 OpenAPI Specification - Version 3.1.0

https://swagger.io/specification/?utm_source=chatgpt.com

64 134 RFC 9110: HTTP Semantics

https://www.rfc-editor.org/rfc/rfc9110.html?utm_source=chatgpt.com

65 RFC 8259 - The JavaScript Object Notation (JSON) Data ...

https://datatracker.ietf.org/doc/html/rfc8259?utm_source=chatgpt.com

69 76 Idempotent requests | Stripe API Reference

https://docs.stripe.com/api/idempotent_requests?utm_source=chatgpt.com

71 72 Just Say No to More End-to-End Tests

https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html?utm_source=chatgpt.com

81 Compose file reference

https://docs.docker.com/reference/compose-file/?utm_source=chatgpt.com

82 Overview

https://kubernetes.io/docs/concepts/overview/?utm_source=chatgpt.com

83 121 Treat logs as event streams

https://12factor.net/logs?utm_source=chatgpt.com

85 122 131 Client libraries

https://prometheus.io/docs/instrumenting/clientlibs/?utm_source=chatgpt.com

86 Grafana support for Prometheus

https://prometheus.io/docs/visualization/grafana/?utm_source=chatgpt.com

87 Documentation

https://opentelemetry.io/docs/?utm_source=chatgpt.com

88 108 135 Chapter 6 - Monitoring Distributed Systems

https://sre.google/sre-book/monitoring-distributed-systems/?utm_source=chatgpt.com

89 110 Context propagation

https://opentelemetry.io/docs/concepts/context-propagation/?utm_source=chatgpt.com

91 123 Password Storage - OWASP Cheat Sheet Series

https://cheatsheetseries.owasp.org/cheatsheets>Password_Storage_Cheat_Sheet.html?utm_source=chatgpt.com

92 Session Management - OWASP Cheat Sheet Series

https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html?utm_source=chatgpt.com

93 RFC 7519 - JSON Web Token (JWT)

https://datatracker.ietf.org/doc/html/rfc7519?utm_source=chatgpt.com

94 A01 Broken Access Control - OWASP Top 10:2021

https://owasp.org/Top10/2021/A01_2021-Broken_Access_Control/?utm_source=chatgpt.com

95 RFC 8446 - The Transport Layer Security (TLS) Protocol ...

https://datatracker.ietf.org/doc/html/rfc8446?utm_source=chatgpt.com

96 A03 Injection - OWASP Top 10:2021

https://owasp.org/Top10/2021/A03_2021-Injection/?utm_source=chatgpt.com

100 General Data Protection Regulation (EU) 2016/679

https://www.legislation.gov.uk/eur/2016/679/contents?utm_source=chatgpt.com

101 Data Security Standard (PCI DSS)

https://www.pcisecuritystandards.org/standards/pci-dss/?utm_source=chatgpt.com

109 Legacy Mimic

https://martinfowler.com/articles/patterns-legacy-displacement/legacy-mimic.html?utm_source=chatgpt.com

111 **114** **116** Strangler Fig Pattern - Azure Architecture Center | Microsoft Learn

https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig?utm_source=chatgpt.com

113 **127** Surviving continuous deployment in distributed systems

https://www.thoughtworks.com/insights/blog/continuous-delivery/surviving-continuous-deployment-distributed-systems?utm_source=chatgpt.com

124 GitHub flow

https://docs.github.com/en/get-started/using-github/github-flow?utm_source=chatgpt.com

137 Monolith to Microservices - Sam Newman

https://samnewman.io/books/monolith-to-microservices/?utm_source=chatgpt.com

138 Operational Excellence Pillar - AWS Well-Architected ...

https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/welcome.html?utm_source=chatgpt.com