Single-Deploy Modular Monorepo (SDMM) Engineering Principles & Guidelines

AzzCraft, Inc. (艾之舟科技)

January 2026

目录

Status: General-purpose reference for implementation and AI-assisted development
Scope: Any codebase using workspace packages (pnpm/yarn/npm workspaces, Nx/Turbo optional) to split a frontend into small, contract-driven packages while keeping each deployed frontend as a **single runtime artifact**.

Normative language

The key words **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, and **MAY** are to be interpreted as described in RFC 2119.

---

1 Purpose

SDMM is a build-time modularization pattern:

- Split a codebase into **small, independently editable packages** (API, contracts, shared logic, UI, features).
- Keep each product frontend as **one deployed application** (one SPA/SSR artifact), avoiding micro-frontend runtime complexity.
- Make boundaries explicit so that **AI agents** can implement isolated changes safely without breaking integration.

SDMM is not a "folder refactor". It is a **boundary-and-contract refactor** that aims to reduce:

- accidental coupling
- inconsistent error/auth behavior
- bundle bloat from unmanaged shared code
- integration regressions caused by unclear ownership

---

2 Non-goals

SDMM is **not**:

1) A micro-frontend architecture (separate deployments + runtime composition).

2) A justification to duplicate domain logic in multiple packages.
3) A replacement for contract tests and end-to-end validation.
4) A guarantee of faster builds: SDMM can slow builds if boundaries are unstable or tooling is weak.

---

3 Definitions

3.1 App shell and composition root

- **App shell**: the runnable application entry for a deployable artifact (e.g., `apps/web`). It owns bootstrap, routing, global providers, and cross-package composition.
- **Composition root**: the place where the app chooses implementations and wires dependencies.

In SDMM there is **one app shell per deployed artifact**. A single repo MAY contain multiple app shells (e.g., `apps/customer-web`, `apps/admin-web`). Each app shell is still "single-deploy".

3.2 Package

A **package** is a workspace module under `packages/*` that is:

- build-time linked into an app shell build
- versioned and reviewed as an independent unit of change
- small enough that an AI agent can safely modify it with minimal context

Packages do not imply separate runtime deployments.

3.3 Contract

A **contract** is anything multiple packages (or systems) must agree on. Contracts include:

- data: request/response DTO shapes, error payload shapes, event/message schemas
- identifiers: route IDs, permission codes, feature flags
- semantics: auth/session expectations, retry/timeout policy, pagination behavior, streaming lifecycle rules

Important distinction:

- A **contract (concept)** can include semantics and rules.
- A **`contracts` package (code)** MUST contain only the **representations** of contracts (types/schemas/constants), not the behavioral implementations.

Behavior that enforces a contract lives in `api` / `core` / app shell and is validated via tests.

3.4 Boundary

A **boundary** is the enforced line between packages. Boundaries are defined by:

- allowed imports
- public exports
- ownership and responsibility

SDMM only works when boundaries are real and enforced.

---

4 Core reasoning: efficiency - when SDMM helps vs hurts

4.1 Why SDMM usually avoids "microservice-style" efficiency loss

SDMM typically does **not** add runtime latency because it does **not** introduce:

- extra network hops
- extra backend calls
- separate runtime deployments of frontend parts

Most costs are build-time (tooling, packaging) and organizational (ownership discipline).

4.2 What can regress runtime performance in SDMM

Runtime regressions occur when the split causes:

- **duplicate runtime copies** (two Reacts, two Vue runtimes, duplicate UI library copies)
- loss of tree-shaking due to careless exports ("barrel bloat")
- accidental de-optimization of code splitting (everything becomes eagerly imported)
- duplicated large assets/fonts or repeated CSS injection

4.3 What can regress developer/CI efficiency in SDMM

Developer/CI regressions occur when:

- boundaries are unstable (constant cross-package churn)
- "shared" packages become dumping grounds (`core` grows without discipline)
- import boundaries are not enforced (entropy: "monolith with folders")
- tasks are not cached (no incremental build/test strategy)
- packages have inconsistent tooling versions/configs

---

5 Reference architecture

5.1 Workspace layout

```
repo-root/
  apps/
    web/                      # app shell (single deploy)
    # other app shells may exist; each is single-deploy

  packages/
    contracts/                # shared types + schemas + contract constants
    foundation/               # OPTIONAL: shared pure runtime helpers (below core/ui)
    api/                      # network I/O: clients + endpoint wrappers
    core/                     # shared non-visual logic + policies (I/O-free by default)
    ui/                       # reusable UI components + design primitives
    features/
      feature-a/
      feature-b/
      ...
```

`foundation/` is optional but strongly recommended if you want **shared helpers** without creating a `ui -> core` dependency.

5.2 Package roles - responsibility boundaries

- `contracts`
  - Owns shared **types/schemas/constants** for contracts.
  - MUST remain dependency-light.
- `foundation` **(optional)**
  - Owns shared **pure runtime utilities** used across layers:
    - formatting helpers that are truly generic
    - small deterministic string/number/date helpers
    - environment-agnostic parsing helpers
  - MUST NOT contain domain policy or application lifecycle logic.
- `api`
  - Owns all **network I/O** (HTTP/SSE/WS wrappers, uploads/downloads).
  - Normalizes errors and enforces response contracts.
- `core`

  - Owns shared **non-visual logic and policies**:
    * auth/session helpers (as policy, not routing)
    * permission evaluation helpers
    * shared deterministic domain utilities (prefer `foundation` for generic helpers)
    * optional global state stores (when truly cross-feature)

- `ui`
  - Owns reusable **presentation components** and design primitives.

- `features/*`
  - Own page-level user flows: routes, screens, feature stores, feature-local components.

- `apps/*`
  - Own composition: routing composition, bootstrap, global providers, global CSS and assets, configuration injection.

---

6 Foundational principles

6.1 P1 — Contracts first, implementation second

If multiple packages depend on a behavior, define the contract first:

- encode it as types/schemas/constants (in `contracts`)
- document semantics (edge cases, invariants)
- add at least one boundary test (contract test or stable mock)

Do not start feature implementation until the boundary contract is explicit.

6.2 P2 — Single owner per cross-cutting concern

A cross-cutting concern MUST have a single owner package. Examples:

- response envelope decoding and error mapping -> `api`
- auth/session policy helpers -> `core` (routing remains in app shell)
- permission evaluation utilities -> `core`
- reusable components and theme primitives -> `ui`
- shared DTO types and identifiers -> `contracts`

Duplicate implementations are treated as defects unless explicitly justified.

6.3 P3 — Features own user flows, not infrastructure

Feature packages MUST NOT:

- create their own HTTP clients
- implement their own global auth redirect logic
- reinterpret response envelopes

Features SHOULD:

- call `api`
- use `core`/`foundation` for shared logic
- use `ui` for shared components

6.4 P4 — Acyclic dependency graph, enforced

Allowed imports:

- `apps/*` -> may import from any internal package.
- `features/*` -> may import from:
  - `contracts`
  - `foundation` (if present)
  - `api`
  - `core`
  - `ui`
  - itself
- `ui` -> may import from:
  - `contracts`
  - `foundation` (preferred)
  - external UI libs
  - it MUST NOT import from `core` or from any feature.
- `core` -> may import from:
  - `contracts`
  - `foundation`
  - (rarely) `api` if truly unavoidable; prefer inversion (interfaces/events) instead.
  - NOTE: If `core` needs to reason about API failures, it MUST do so via shared error representations defined below both (`contracts`/`foundation`), not by importing `api`-owned error classes.
  - it MUST NOT import from `ui` or features.
- `api` -> may import from:

  - `contracts`
  - `foundation`
  - it MUST NOT import from `core`, `ui`, or features.

- `contracts` -> MUST NOT import from other internal packages.
- `foundation` (if present) -> MUST NOT import from `api`/`core`/`ui`/features; it MAY import from `contracts`.

MUST: No feature-to-feature imports.
If two features need shared code, promote it into `ui`, `core`, `foundation`, or `contracts` as appropriate.

Strict rule: keep pure logic separated from external side effects To avoid "shared layer entropy" and test flakiness, SDMM distinguishes **pure logic** from **side effects**.

- **Pure logic** = deterministic computation. Given the same inputs, it returns the same outputs.

  - no network
  - no persistence
  - no global mutable state
  - no direct clock/time reads if determinism matters (inject time via `now()`)
  - no DOM access

- **Side effects** = interacting with the environment or introducing nondeterminism.

Side effects are not all equal. SDMM uses two buckets that matter for package boundaries:

1) **External I/O (system effects)**

   - network calls (HTTP/SSE/WS)
   - persistence (cookies/localStorage/IndexedDB/cache storage)
   - filesystem access (Node/server runtimes)
   - service worker registration
   - analytics emission / telemetry exporting
   - global process state and cross-tab/process coordination

2) **View I/O (view effects)**

   - DOM APIs (measurement, focus, scroll, observers)
   - browser events and event listeners
   - timers and scheduling (`setTimeout`, `requestAnimationFrame`, debouncing)

Rules:

- **MUST:** Pure logic MAY be promoted into more shared, lower-level packages (`foundation` / `core`) when it stays pure.
- **MUST NOT:** External I/O MUST NOT be promoted into `foundation` / `core` / `ui`.
  - External I/O belongs in `api` (or explicitly named infra packages), orchestrated by features/app shell.
- **MAY:** View I/O is allowed in `ui` and feature packages because it is part of presentation behavior.
  - View I/O MUST be localized (component/hook scope), lifecycle-correct (cleanup on unmount), and MUST NOT run at import time.
  - View I/O MUST NOT be used as a backdoor to perform External I/O (e.g., `ui` starting background polling or writing persistent storage).

If code mixes pure logic with effects:

- split it: keep the pure core in `foundation/core`, keep the effectful adapter in `ui`/feature/app shell.

6.5 P5 — Public API only - no deep imports

Across package boundaries, imports MUST be from the package root only:
OK: `import { fetchCourses } from '@org/api'`
NOT OK: `import { fetchCourses } from '@org/api/src/course/fetch'`

Define the public surface explicitly Each package MUST have a single entry (e.g., `src/index.ts`) exporting:

- stable public functions/types/components
- nothing internal

If you intentionally expose multiple public entrypoints (subpaths), they MUST be declared as public API (e.g., via export maps) and treated as contracts.

6.6 P6 — Backward-compatible behavior is the default

During modularization, runtime behavior SHOULD remain stable by default.
Behavior changes MUST be treated as product changes:

- documented
- reviewed
- tested

6.7 P7 — Make packages small, cohesive, and replaceable

A package SHOULD be:

- owned by a clear team/person
- understandable with local context
- replaceable without rewriting the whole repo

If a package grows beyond "AI-safe" size, split it by responsibility, not by file count.

6.8 P8 — The app shell is the composition root

MUST: Only the app shell wires together:

- router + route guards
- global providers (auth context, query client, i18n)
- infrastructure configuration (API base URLs, token providers, loggers)
- global error handling and notifications policy
- telemetry setup

MUST NOT: Packages create app-global behaviors at import time (e.g., registering router guards, installing plugins, starting timers, mutating global singletons), unless explicitly coordinated through the app shell.

MUST: If a package needs global registration, it MUST expose an explicit `registerX(appContext)` (or `install(app)`) function, and the app shell MUST call it during bootstrap.

6.9 P9 — Prefer explicit dependency injection over hidden globals

SHOULD: Prefer explicit construction over implicit module singletons.

Recommended approach (best balance of ergonomics and testability):

1) Implement a factory:

- `createX(config) -> X` (pure construction; no hidden globals)

2) (Optional convenience) Provide a default instance configured by the app shell:

- `configureX(config)` sets a default instance
- exports that rely on the default MUST fail fast if not configured

MUST: If you provide a `configureX()` convenience singleton, you MUST also provide a factory (`createX()`).

MUST: Libraries/packages SHOULD NOT require global configuration to be import-safe. **MUST:** Tests MUST be able to create isolated instances without relying on global mutable module state.

If you choose a configured default instance pattern:

- configuration MUST be single-assignment in production (fail fast on reconfigure)
- a `resetForTests()` hook MAY exist, but it is not sufficient if tests run concurrently in the same process
- in test-intensive or concurrency-heavy environments, prefer instance-based injection (factory + explicit passing) as the default

---

7 Contract rules

7.1 Contract sources of truth

MUST: For each contract, identify exactly one source of truth:

- backend-owned API contracts -> backend schema/spec (mirrored into `contracts`)
- frontend-owned identifiers (route IDs, permission codes) -> `contracts` + app shell routing
- UI component contracts -> `ui` public API (props/events) + story/tests

7.2 Contract evolution: refactors vs versioning

The word "versioning" is overloaded; SDMM uses two modes:

1) **Refactor-mode (single deploy, single change set)**
   When all consumers are updated together (typical within one repo + one app build), treat contract changes as **refactors**:

   - prefer additive changes
   - update all usages in the same change set
   - keep behavior stable unless intentionally changed

2) **Versioning-mode (independent deploy cadence or external consumers)**
   When consumers cannot be updated atomically (separate services, multiple deploy pipelines, third-party clients), treat contract changes as **versioned**:

   - introduce explicit v2/v3 (URL versioning or header/content negotiation)
   - maintain a compatibility window
   - measure adoption and deprecate intentionally

MUST: A boundary that crosses independent deploy lifecycles MUST use versioning-mode.

7.3 Contract drift prevention

MUST: Prevent silent drift by enforcing at least one of:

- contract tests at boundaries

- runtime schema validation at boundaries for untrusted inputs (when practical)
- shared mocks generated from `contracts` and used by features

---

8 `api` package guidelines (I/O layer)

8.1 Responsibilities

`api` owns:

- HTTP client configuration (base URL, interceptors, headers)
- request/response decoding (including envelopes)
- typed endpoint wrappers grouped by domain
- error normalization (mapping raw transport/library errors into shared error representations defined in `contracts` and/or `foundation`)
- upload/download helpers
- streaming wrappers (SSE/WS) where applicable
- request dedup/caching policy if used

8.2 Configuration injection

Because `api` SHOULD NOT import app-specific state (router/stores), it MUST be configurable from the app shell.

Recommended pattern:

- `api.configure({ baseUrl, getAuthToken, onAuthError, onError, logger, now })`

Where:

- `getAuthToken` returns a token (or `undefined`) and MAY be async
- `onAuthError` handles auth failure at the app level (redirect/reset)
- `onError` is an optional hook for centralized error reporting
- `now` is an injectable clock for testability

MUST: `api` must be usable in tests without a browser/router/store by replacing hooks with test doubles.

Reference implementation shape (illustrative):

```
export type Awaitable<T> = T | Promise<T>

export type GetAuthToken = () => Awaitable<string | undefined>

export interface ApiConfig {
```

```
  baseUrl: string
  getAuthToken?: GetAuthToken
  onAuthError?: (err: unknown) => void
  onError?: (err: unknown) => void
  now?: () => number
}

export interface ApiClient {
  request<T>(path: string, init?: RequestInit): Promise<T>
  // Add domain wrappers on top, or expose typed per-domain modules.
}

// 1) Factory (preferred for tests and isolation)
export function createApiClient(cfg: ApiConfig): ApiClient {
  // return an instance that closes over cfg; no hidden globals
  throw new Error('implement')
}

// 2) Convenience singleton (for app shell ergonomics)
let _defaultClient: ApiClient | undefined
let _configured = false

export function configureApi(cfg: ApiConfig): void {
  if (_configured) {
    throw new Error('configureApi called more than once')
  }
  _defaultClient = createApiClient(cfg)
  _configured = true
}

export function api(): ApiClient {
  if (!_defaultClient) {
    throw new Error('API not configured. Call configureApi() in app shell bootstrap.')
  }
  return _defaultClient
}

// Optional: only for unit tests that must use the singleton.
export function resetApiForTests(): void {
  _defaultClient = undefined
```

```
  _configured = false
}
```

MUST: If `getAuthToken` can refresh tokens, it MUST implement single-flight behavior (avoid stampedes) and it MUST be safe to call on every request.

MUST: Configure `api` during application bootstrap (in the app shell) before any feature invokes endpoints.

8.3 Timeouts, cancellation, and long-running operations

MUST: Every request has an explicit timeout policy (global default + per-endpoint overrides).

Classify operations:

- **Interactive** (user waiting): strict time budget, clear loading UX, cancellation when possible
- **Background** (user can navigate away): job-based patterns, resumability
- **Compute/generation/media**: explicit long timeout, progress hints, idempotency keys, best-effort cancellation

MUST: If the UI can trigger the same long-running action repeatedly, define dedup/idempotency strategy explicitly.

8.4 Error model ownership

A common SDMM failure mode is placing "shared" error classes in `api`, then writing retry/auth/session policies in `core` that need to inspect those classes.

That creates one of two bad outcomes:

- `core` imports `api` (violating the intended purity boundary), or
- developers duplicate error representations across packages (drift and inconsistent handling).

MUST: Define the *shape* of normalized errors **below both** producers and consumers:

- Prefer `contracts` for the **data representation** (types, enums, schemas).
- Use `foundation` (optional) for **pure runtime helpers** (constructors, type guards) if needed.

`api` owns the **mapping logic**:

- transport/library error -> normalized error representation
- API envelope failure -> normalized error representation

`core` owns **pure policies** over the normalized representation:

- retry classification: `shouldRetry(err)`
- session/auth classification: `isSessionExpired(err)`
- user messaging policy inputs (but not UI rendering)

MUST NOT: Require `core` (or `ui`) to import error classes/types that are defined in `api`. If an error representation must be shared, move it down into `contracts/foundation`.

SHOULD: Prefer discriminated unions (or tagged objects) over `instanceof` across package boundaries.

Example (illustrative):

```ts
// contracts/errors.ts (representation only)
export type NormalizedError =
  | { kind: 'auth'; status?: number; code?: string; message: string }
  | { kind: 'transport'; message: string; retryAfterMs?: number }
  | { kind: 'api'; code: string; message: string }
  | { kind: 'cancelled'; message?: string }
  | { kind: 'unknown'; message: string; detail?: unknown }

// core/policy.ts (pure)
export function shouldRetry(e: NormalizedError): boolean {
  return e.kind === 'transport'
}

export function isSessionExpired(e: NormalizedError): boolean {
  return e.kind === 'auth' && (e.status === 401 || e.status === 403)
}

// api/errors.ts (I/O layer: mapping only)
export function normalizeError(raw: unknown): NormalizedError {
  // map axios/fetch/etc to NormalizedError
  return { kind: 'unknown', message: 'Unexpected error', detail: raw }
}
```

This pattern keeps the dependency graph acyclic and allows policy evolution without leaking I/O concerns upward.

---

9 `contracts` package guidelines (contract layer)

9.1 What belongs here

`contracts` SHOULD contain:

- DTO types: request/response shapes, shared domain models
- error code enums and typed error payload shapes
- normalized error representations used across packages (as pure data types), when you need shared retry/auth policies
- identifiers: route IDs, permission codes, feature flags
- event/message schemas for streaming
- optional runtime schemas for boundary validation

MUST: Keep `contracts` as representations only.
Behavior like retry logic, auth redirect behavior, caching policy, or envelope parsing MUST NOT live here.

If you need to standardize semantics, encode them as:

- documented rules (this document)
- tested behavior in `api/core`
- constants that parameterize behavior (only if truly stable)

9.2 What must not belong here

`contracts` MUST NOT contain:

- UI components
- network clients
- environment-specific logic (reading `window/document` or process env directly)
- orchestration code

Goal: `contracts` is boring, stable, and dependency-light.

---

10 `core` package guidelines (shared policy + pure logic layer)

10.1 Responsibilities

`core` owns:

- auth/session policy helpers (not routing)
- permission evaluation helpers
- cross-feature non-visual logic and policies
- shared stores only when truly global

10.2 Keep `core` deterministic by default

SHOULD: `core` should be I/O-free by default.
If `core` must perform External I/O, isolate it behind explicit interfaces and keep it rare.

10.3 Framework specificity

`core` SHOULD avoid coupling to specific UI frameworks.
If framework hooks are needed, isolate them into a thin adapter submodule and keep the majority framework-agnostic.

---

11 `ui` package guidelines (shared presentation layer)

11.1 Responsibilities

`ui` owns:

- reusable presentation components
- design tokens/theme primitives (as code and/or CSS variables)
- shared layout blocks (when they are genuinely reusable)
- component-level accessibility patterns

11.2 Rules (avoid hidden coupling)

`ui` components SHOULD be:

- prop-driven
- slot/children-friendly
- stable in public API
- free of feature-specific state

`ui` MUST NOT import feature stores or feature logic.

11.3 Dependency rule: do not couple `ui` to `core`

SHOULD: Treat `ui` as "dumb" presentation.

- `ui` MUST NOT depend on `core` as a default rule.
- If `ui` needs generic helpers (formatting, parsing), prefer `foundation`.
- If a `ui` component needs policy decisions (permissions, routing, auth state), pass decisions/results as props from the feature/app shell.

If you insist on `ui -> core` in a specific repository:

- you MUST prove `core` is I/O-free and policy-light in the imported area
- you MUST enforce boundaries to prevent `core` from importing `ui` (cycle risk)
- you SHOULD treat this as an exception, not a norm

11.4 Styling and assets strategy (required)

Modular frontends fail most often at styling boundaries. Define this explicitly.

MUST: Choose one primary scoping strategy and enforce it:

- **Scoped styles / CSS Modules**: preferred for package isolation
  - feature-local and UI component styles MUST be scoped
  - global selectors are disallowed except in app shell global stylesheet
- **Utility-first CSS (e.g., Tailwind)**: acceptable if centrally configured and shareable across packages
  - the canonical config/preset MUST live in a shared place (e.g., `ui` or a dedicated `design-tokens` package, or repo root), not only inside an app shell
  - app shells and package tooling (storybook/tests) MUST consume the same preset/-config
  - avoid per-package divergent configs; allow only additive extension in app shell when necessary
- **CSS-in-JS**: acceptable if SSR constraints and runtime costs are understood

MUST: Define ownership:

- global resets, typography baseline, and "page chrome" styles -> app shell
- design tokens (colors, spacing, typography scale) -> `ui` (or `foundation` if purely non-visual constants)
- component styles -> `ui` or feature-local components, but always scoped

MUST: Prevent style collisions:

- no package is allowed to inject global CSS at import time (except app shell)
- class naming conventions or scoping MUST be enforced (lint/build rules)

MUST: Define static asset ownership:

- shared icons/fonts/images used across features -> `ui` (or app shell if truly global)
- feature-specific assets -> feature package
- assets MUST be imported via stable public APIs (no deep imports across packages)

---

12 Feature package guidelines (`features/*`)

12.1 Feature boundaries

Each feature package MUST be "complete" for its domain:

- it can be removed and the app still builds (routes removed)
- it does not require another feature to compile

12.2 Feature public exports

Feature packages SHOULD export only:

- routes/screens registration
- optional menu descriptors (if your app composes menus from features)
- minimal feature-level constants/types

If feature internals are reused by other features, they likely belong in `ui`, `core`, `foundation`, or `contracts`.

---

13 App shell guidelines (`apps/*`)

The app shell owns:

- route composition and guards
- provider composition
- global CSS and assets
- configuration injection for `api` and other infra
- top-level error boundaries and telemetry wiring

Feature packages MUST NOT register global guards or global plugins.

---

14 State management principles

14.1 Where state lives

- truly global state -> `core` or app shell
- feature state -> feature package

14.2 Avoid hidden coupling

Features MUST NOT read/write other feature stores directly.
Shared state must be promoted into `core` (or persisted to the backend).

---

15 Performance and build efficiency principles

15.1 Guarantee singleton framework/runtime dependencies

MUST: Prevent duplicate runtime copies:

- align versions via workspace constraints
- prefer peer dependencies for framework libs in packages that should not bundle them
- use lockfile enforcement/resolutions to avoid drift
- run a bundle/dependency duplication check regularly (CI preferred)

15.2 Deliberate code splitting

SHOULD: Code-split by routes/screens:

- feature pages SHOULD be lazy-loaded
- shared packages SHOULD avoid pulling heavy dependencies into the initial chunk

15.3 Export hygiene (avoid barrel bloat)

MUST: Do not export "everything" from root barrels by default.
Exports are part of your contract; keep them explicit and minimal.

15.4 Build/CI efficiency

SHOULD: Use caching and incremental tasks:

- per-package typecheck/tests
- build caching (Nx/Turbo or CI cache) where available

---

16 Error handling, security, and observability

16.1 One global error policy

MUST: Centralize user-visible error handling policy.
Features may catch and recover, but must not invent new inconsistent policies.

16.2 Correlation IDs and telemetry

SHOULD: Propagate correlation IDs through:

- client logs
- API requests (headers)
- error reports

16.3 Secret handling

MUST: No secrets in repo source code.
Use environment variables and secret managers.

16.4 Untrusted inputs and boundary validation

SHOULD: Validate untrusted inputs at boundaries (API responses, query params, storage).
Use runtime schemas where the risk justifies the cost.

---

17 Testing strategy

17.1 What to test where

- `contracts`: schema/type-level tests and example fixtures
- `api`: client behavior tests (envelopes, error mapping, retry/timeout policy)
- `core`: policy logic tests (permission evaluation, session helpers)
- `ui`: component behavior tests (props/slots/accessibility)
- `features`: integration-ish tests of user flows
- app shell: smoke tests + routing/guard behavior

17.2 Contract tests

SHOULD: Add contract tests where failures are expensive:

- response decoding and error semantics
- streaming lifecycle correctness
- permission/route identifier stability

---

18 Tooling enforcement

Minimum enforcement set:

- path aliases for package root imports
- lint rules for boundary imports (no deep imports, no feature-to-feature)
- typecheck gate at workspace root
- build gate for each app shell
- (recommended) automated duplicate dependency/bundle check

---

19 AI-assisted development workflow

When delegating to AI, every task MUST include:

1) **Scope**: which packages may be modified
2) **Contracts**: which identifiers/shapes/semantics must remain stable
3) **Acceptance criteria**: build/typecheck/test expectations
4) **Non-goals**: what must not be refactored

AI agents perform best when boundaries, public APIs, and contracts are explicit.

---

20 Decision guide

20.1 S1 — Add a new screen/page

- implement in a feature package
- export route registration
- app shell composes routes

20.2 S2 — Add a new backend/API endpoint

- add typed wrapper in `api`
- export from `api` root
- features consume wrapper only

20.3 S3 — Add a shared UI component

- used in one feature -> keep local
- used across features -> promote to `ui`

20.4 S4 — Add shared logic/utility

- generic/pure -> `foundation`
- cross-feature policy -> `core`

20.5 S5 — Add a new permission or gated UI action

- define stable identifiers in `contracts`
- evaluate in `core` or app shell
- render gating in features/`ui` via props or directives/hooks

20.6 S6 — Add a streaming/realtime feature

- define event schemas in `contracts`
- implement lifecycle wrapper in `api`
- features handle presentation only

20.7 S7 — Add a long-running compute/generation operation

- implement request in `api` with explicit timeout/cancellation
- use idempotency keys if supported
- UI shows progress and prevents accidental duplicates

20.8 S8 — Two features need the same domain type

- if it crosses package boundaries, put it in `contracts`

20.9 S9 — Feature A needs data produced by Feature B

Prefer:

1) backend as source of truth
2) promote shared state to `core`
   Avoid direct feature-to-feature imports.

20.10 S10 — Change a contract (shape/semantics/identifier)

Treat as high-risk:

- update `contracts`
- update implementations (`api`/`core`)
- update consumers
- add/adjust boundary tests

20.11 S11 — A feature package becomes too large

Split by responsibility (sub-features) or promote reusable parts into `ui`/`core`/`foundation`.

20.12 S12 — You need to break a dependency rule

Only via an explicit exception process:

- document why
- define sunset plan
- add automated check to prevent spread

Appendix A. Migration playbook

1) Extract `contracts` (types/identifiers) without behavior change
2) Extract `api` (clients + wrappers) without behavior change
3) Extract `ui` shared components
4) Extract `core` policies and truly global stores
5) Split features one-by-one (routes + pages + feature stores)
6) Add enforcement tooling gates (lint/typecheck/build)
7) Add boundary/contract tests for critical paths
8) Add bundle duplication checks and code-splitting validation

---

21 SDMM checklist

- ☐ The product still builds and deploys as a **single application artifact** per app shell.
- ☐ No duplicate framework/runtime copies in the output.
- ☐ CI includes an automated duplicate dependency/bundle check (or a periodic enforced check) to catch drift early.
- ☐ Shared design-system config (tokens, Tailwind preset, etc.) is owned by `ui/design-tokens` and consumable by app shells and UI tooling without `ui -> app` imports.
- ☐ All network I/O is centralized in `api` (or explicitly named infra packages).
- ☐ No feature imports another feature.
- ☐ Shared UI lives in `ui`; shared pure helpers live in `foundation`; shared policies live in `core`; shared contracts live in `contracts`.
- ☐ Contracts are documented and encoded in types/schemas (validated where needed).
- ☐ Streaming connections and long-running operations have explicit lifecycle handling.
- ☐ Automated checks enforce boundary rules (lint/typecheck/build gates).