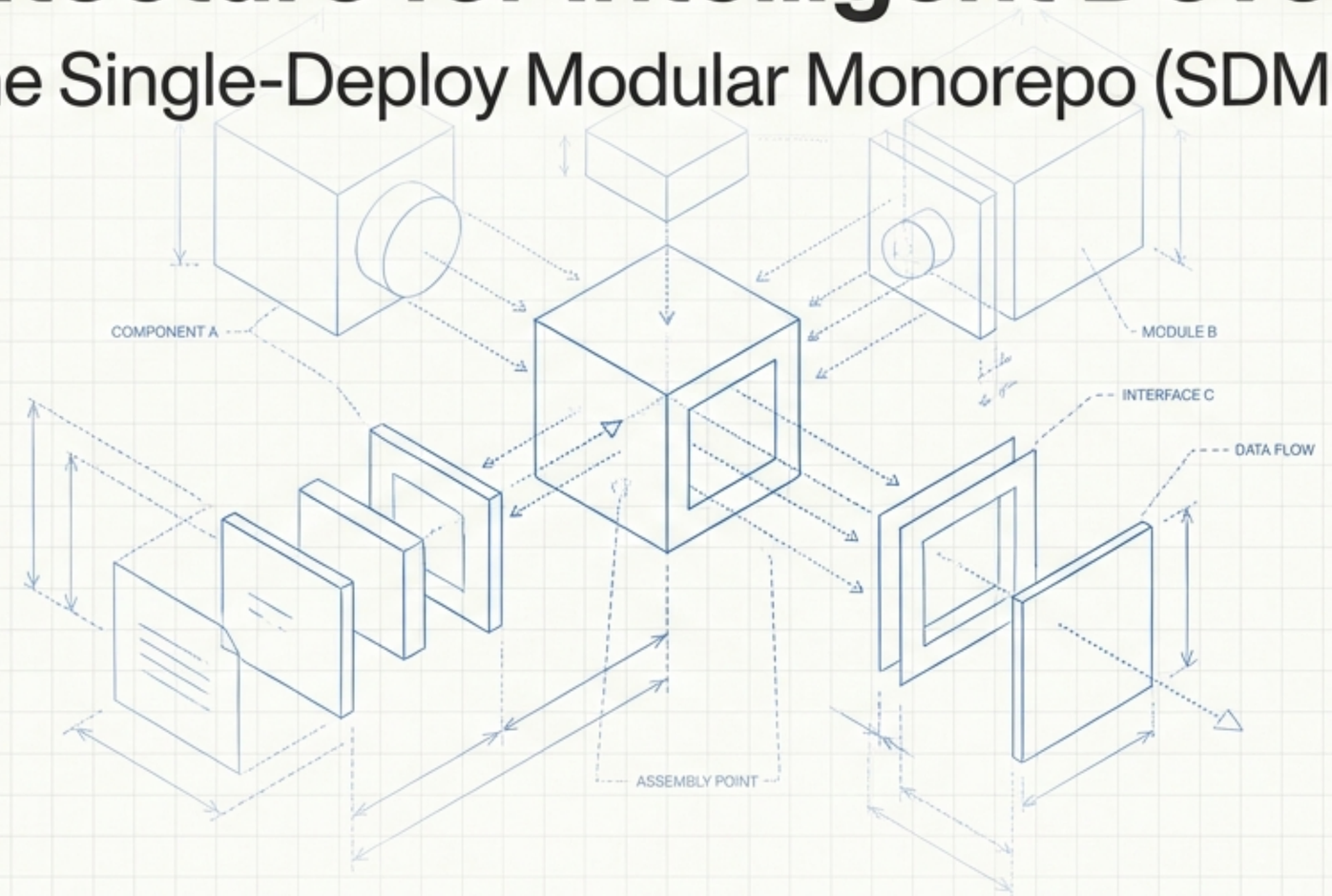
An Architecture for Intelligent Development

The Single-Deploy Modular Monorepo (SDMM)



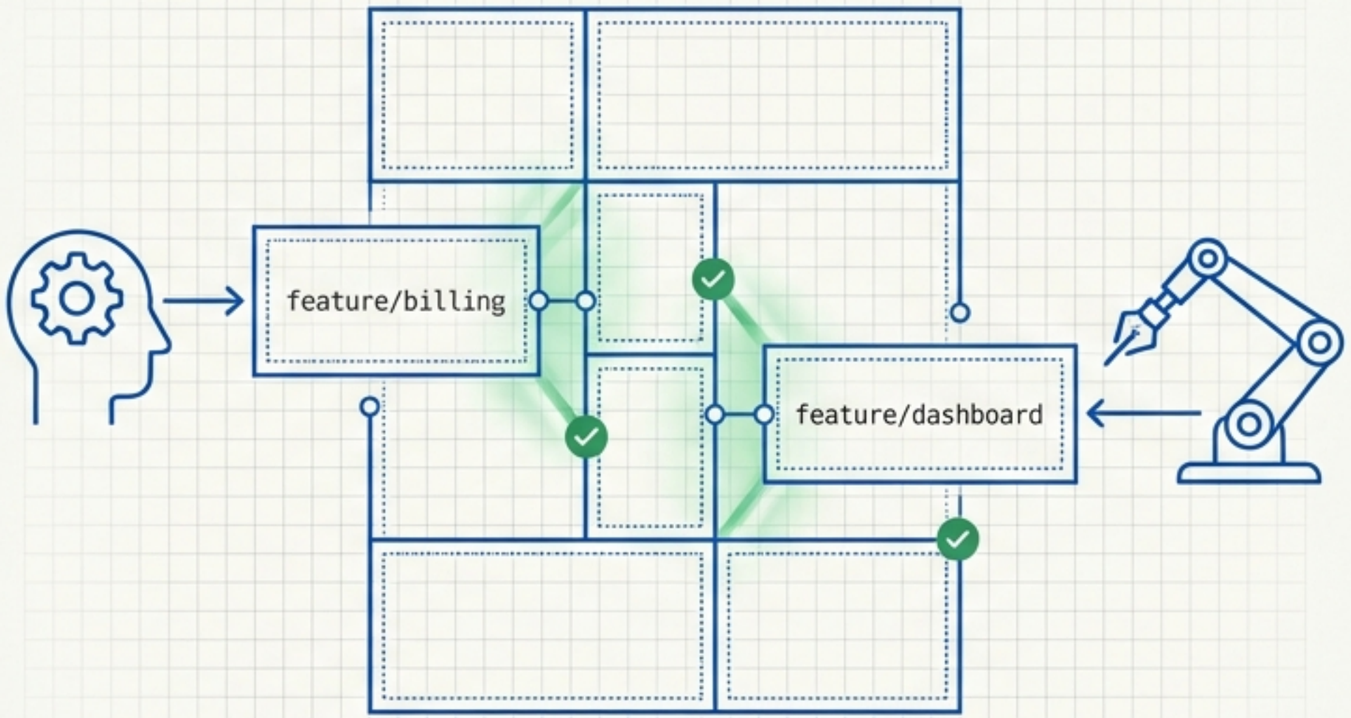
AzzCraft.com

What if your codebase was an asset, ready for AI?

Imagine a development process where AI agents can safely build, test, and refactor features. Where boundaries are so explicit that AI can implement isolated changes without breaking integration.

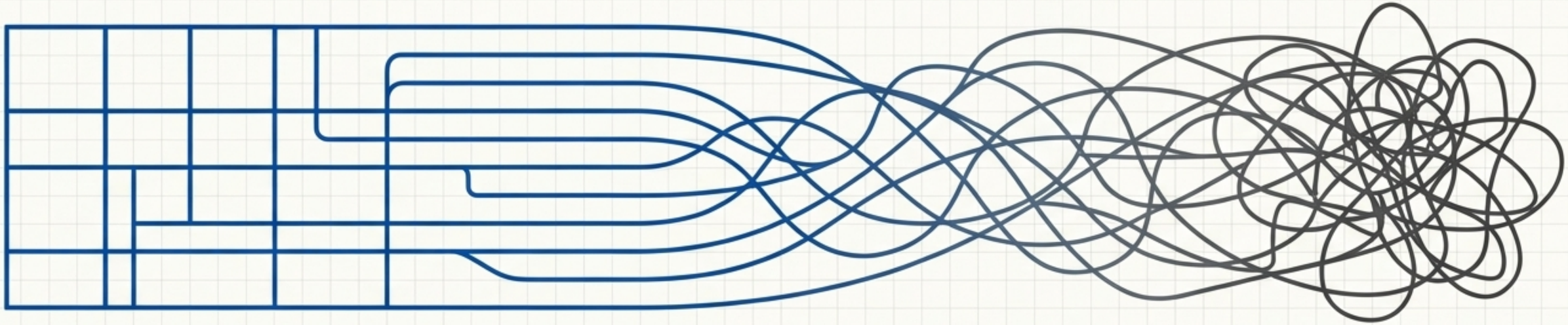
This isn’t a future fantasy; it’s a specific architectural goal.

*Make boundaries explicit so that **AI agents** can **implement** isolated changes safely without breaking integration.*



Today’s Codebases Create Chaos, Not Clarity

Most large frontends become a tangled mess over time, slowing down both human developers and future AI assistants. This isn't a failure of talent; it's a failure of architecture.



Accidental Coupling: Changing one thing unexpectedly breaks another. Regressions are common.

Inconsistent Behavior: Every feature handles errors, auth, and loading states differently, leading to a fragmented user experience.

Bundle Bloat: Unmanaged shared code and duplicate dependencies slow down the application for every user.

Unclear Ownership: When everything is connected, no one truly owns anything. Integration becomes a constant, painful negotiation.

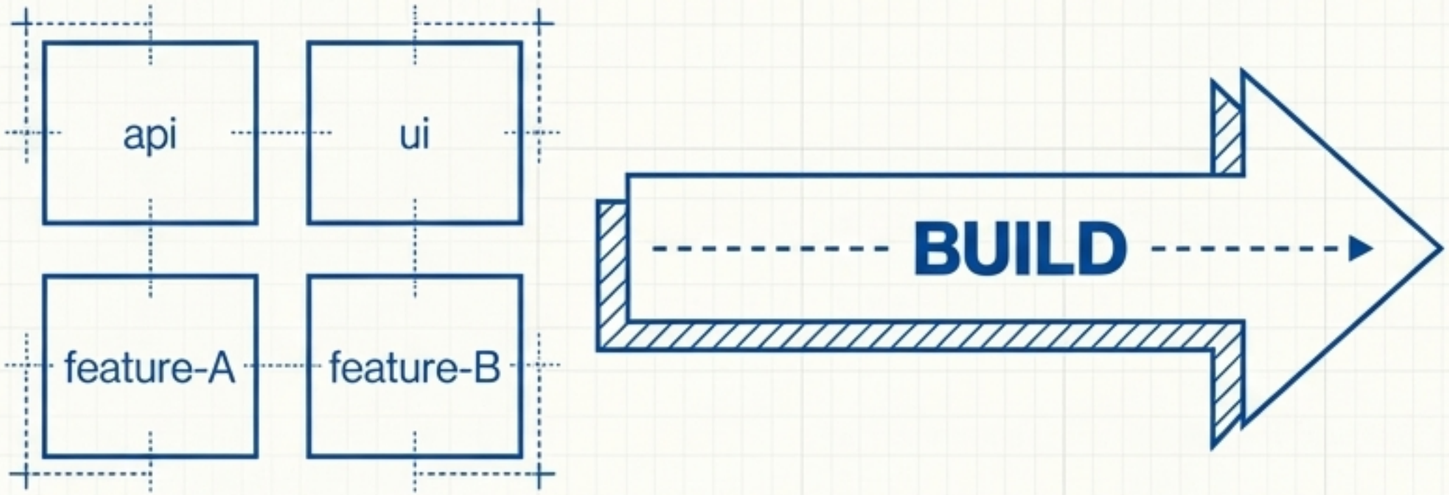
The Solution: Build-Time Modularity, Runtime Simplicity

What SDMM is:

- Split a codebase into small, independently editable packages.
- Deploy each product frontend as a **single application artifact**.
- Avoids the complexity and latency of micro-frontends at runtime.

This is not a "folder refactor."

"It is a boundary-and-contract refactor."

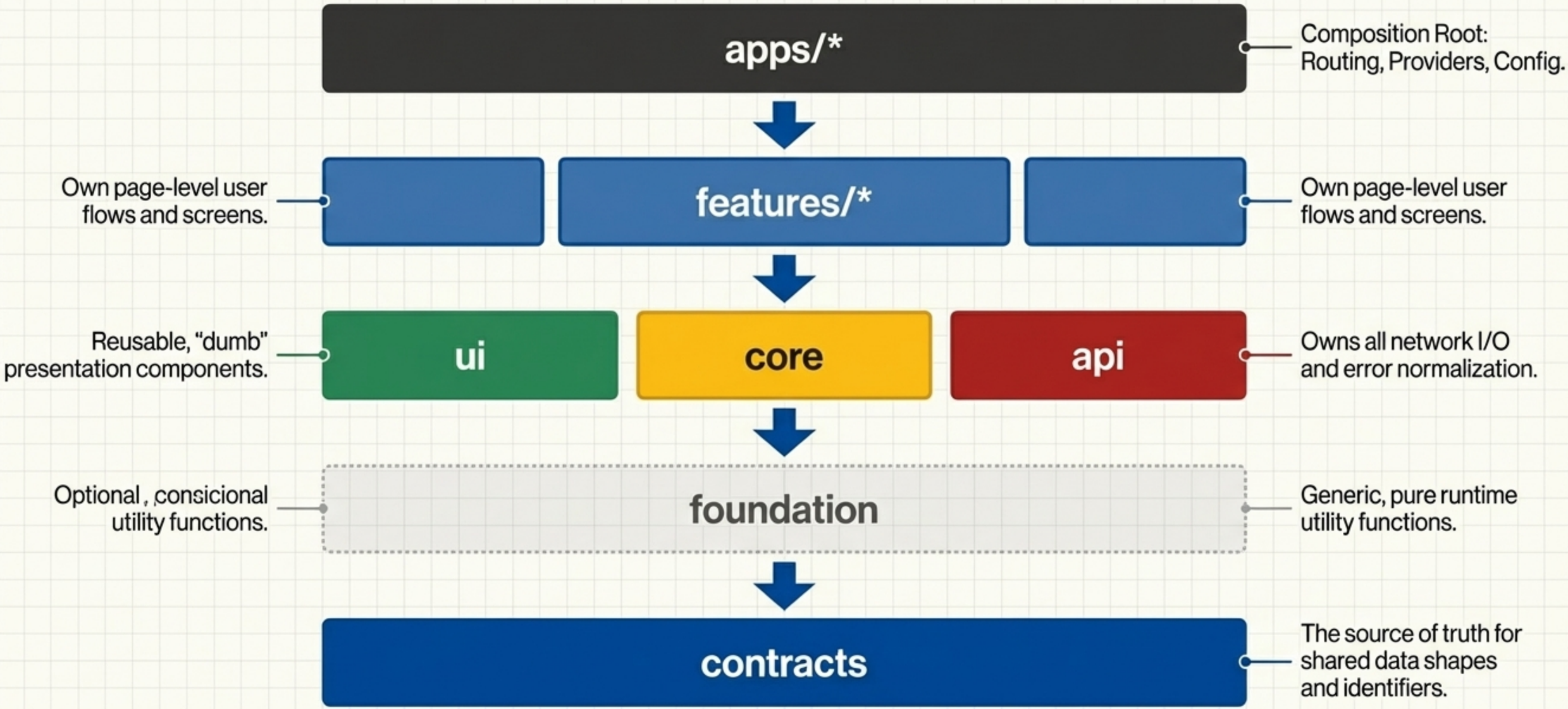


NOT a micro-frontend architecture (no separate deployments).

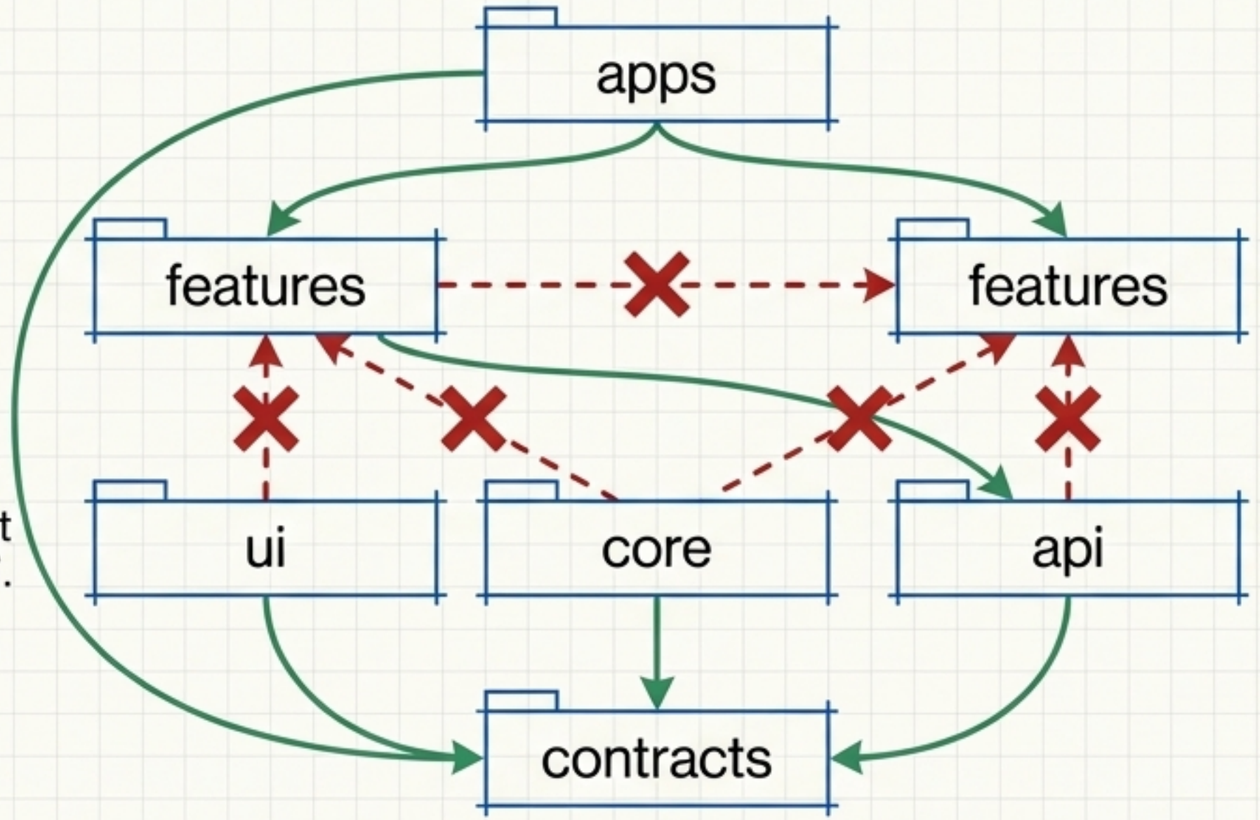
NOT a justification to duplicate domain logic.

NOT a replacement for contract and E2E testing.

The Reference Architecture: A Blueprint for Responsibility



The Golden Rule: Dependencies Flow in One Direction



can import from anywhere.

can import from `ui`, `core`, `api`, `foundation`, `contracts`.

`core` MUST NOT import from `ui` or `features`.

`api` MUST NOT import from `core`, `ui`, or `features`.

imports from nothing internal.

`ui` MUST NOT import from `core` or `features`.

***MUST:** No feature-to-feature imports. Shared code is explicitly promoted to a lower layer (`ui`, `core`, etc.).

Principle 1: Contracts First, Implementation Second

Before writing any implementation that crosses a package boundary, the agreement between those packages must be made explicit. This prevents ambiguity and reduces integration bugs.

What is a Contract?

Data: Request/ Response shapes, error payloads.

Identifiers: Route IDs, permission codes, feature flags.

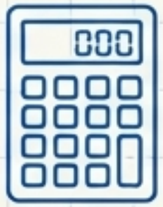
Semantics: Auth expectations, retry policies, lifecycle rules.

The Process

1. **Encode:** Define types, schemas, and constants in a `contracts` package.
2. **Document:** Clarify semantics, edge cases, and invariants.
3. **Test:** Add at least one boundary test to verify the contract.
4. **Implement:** Only then should feature work begin.

Principle 2: Separate Pure Logic from Side Effects

To ensure code is testable, predictable, and safely shareable, we enforce a strict separation between deterministic computation and interactions with the outside world.



Pure Logic (Deterministic)

Given the same inputs, always returns the same output.

Formatting helpers, permission evaluation, domain calculations

MAY be promoted to shared layers like `foundation` or `core`.

Side Effects (Non-deterministic)

Interacts with the environment (network, disk, DOM, timers).

API calls, writing to `localStorage`, DOM manipulation

MUST NOT be promoted into `foundation` / `core` / `ui`. External I/O belongs in `api`.

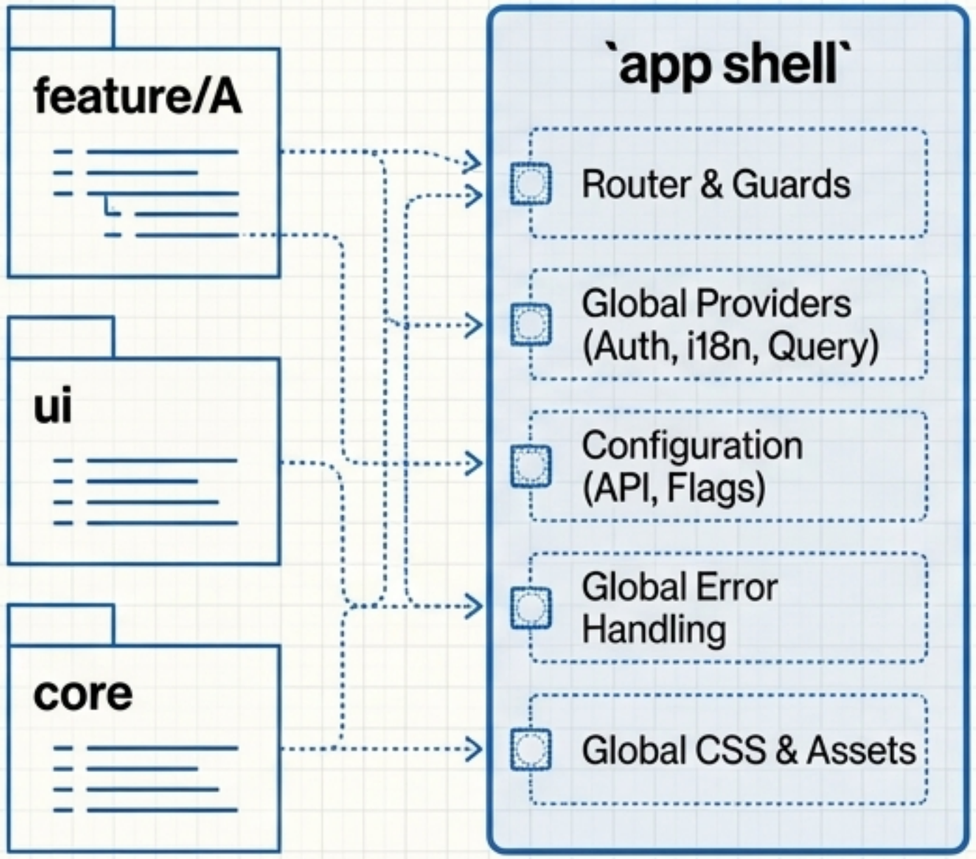
If code mixes pure logic with effects, split it. Keep the pure part in a lower layer and the effectful adapter in a higher one.

Principle 3: The App Shell is the Composition Root

Individual packages must not create application-global behaviors. All integration, configuration, and orchestration is explicitly owned and executed by the `app shell' at bootstrap.

App Shell Responsibilities

- ☑ Wires together the router and route guards.
- ☑ Composes all global providers (Auth, i18n, Query Client).
- ☑ Injects configuration (API URLs, feature flags).
- ☑ Sets up global error handling and telemetry.
- ☑ Manages global CSS, fonts, and assets.



A Non-Negotiable Rule

- ☒ **MUST NOT**: Packages create global side effects at import time (e.g., registering routes, installing plugins, starting timers).
- ☑ **MUST**: If a package needs global registration, it MUST expose an `install(app)` function for the app shell to call.

The Payoff for Developers: Velocity and Clarity

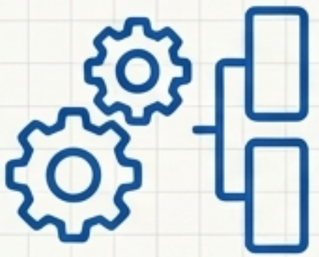
This isn't just about rules; it's about creating a better development experience.

Clear Ownership

You know exactly where code for a specific concern lives (e.g., all network logic is in `api`).

Safer Changes

Explicit boundaries and contracts dramatically reduce the risk of accidental regressions.



High Cohesion, Low Coupling

Packages are small, focused, and replaceable.

Consistent Patterns

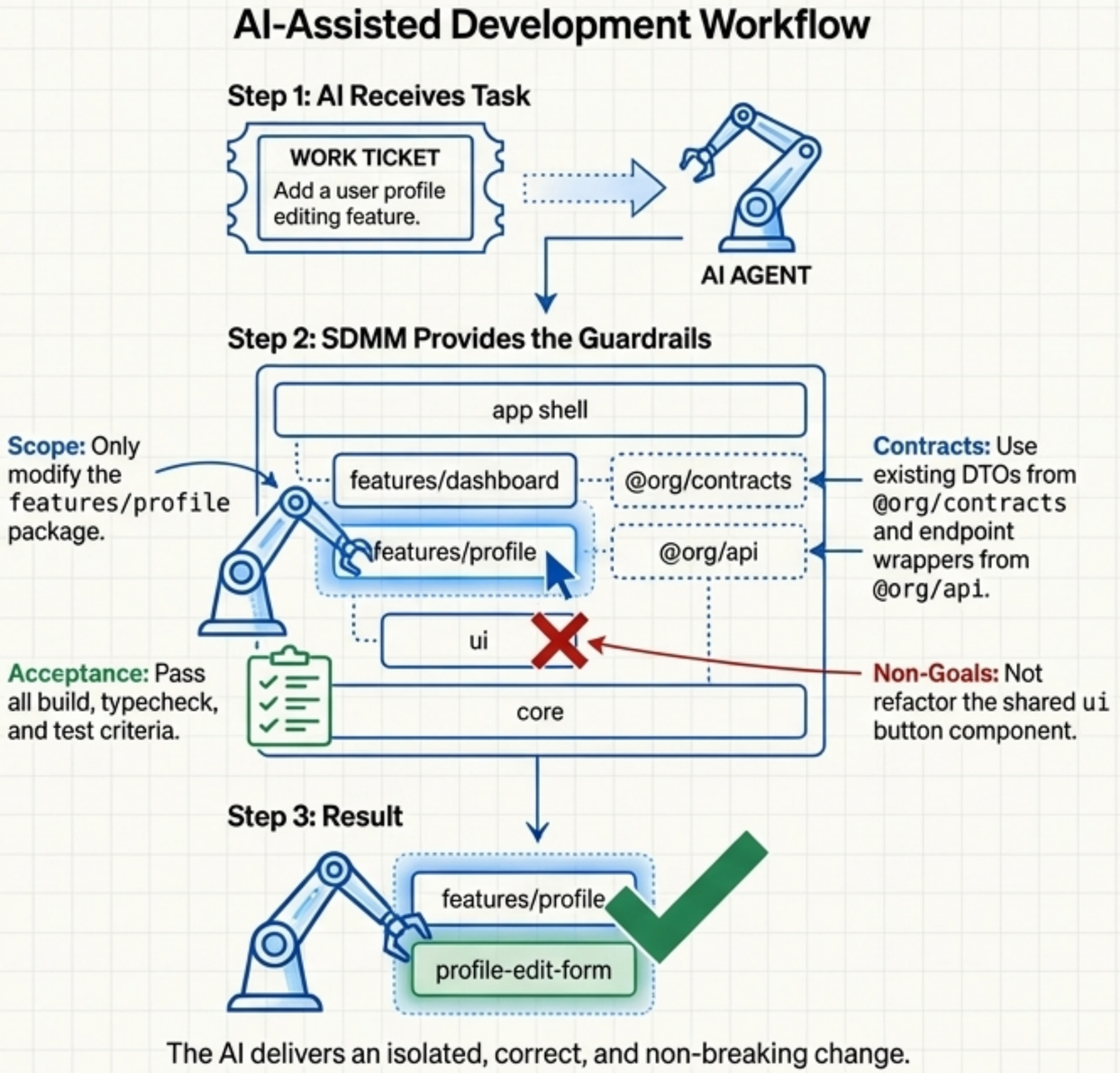
A single way to handle cross-cutting concerns like auth and error handling.

Improved Testability

Separating pure logic from side effects makes unit testing simpler and more reliable.

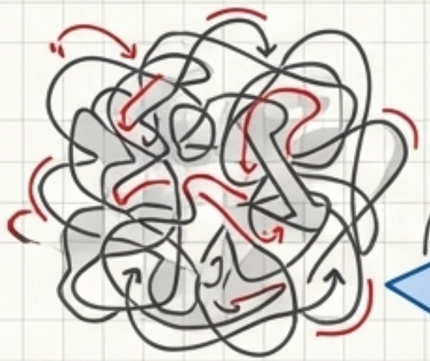
The Ultimate Payoff: A Codebase Built for AI Collaboration

The structure and discipline of SDMM are precisely what AI agents require to be effective and safe. A well-defined architecture turns the codebase into a structured environment where AI can perform complex tasks reliably.



The Path to Adoption is Incremental and Low-Risk

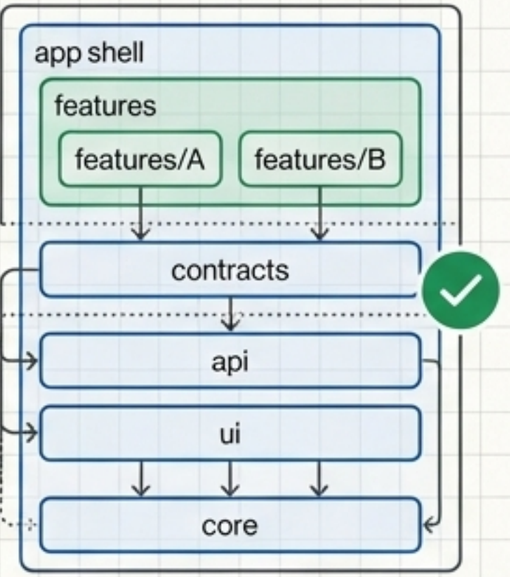
You can migrate an existing monolith to SDMM without a “big bang” rewrite. The process is a series of safe, incremental refactorings, each delivering value.



Monolith

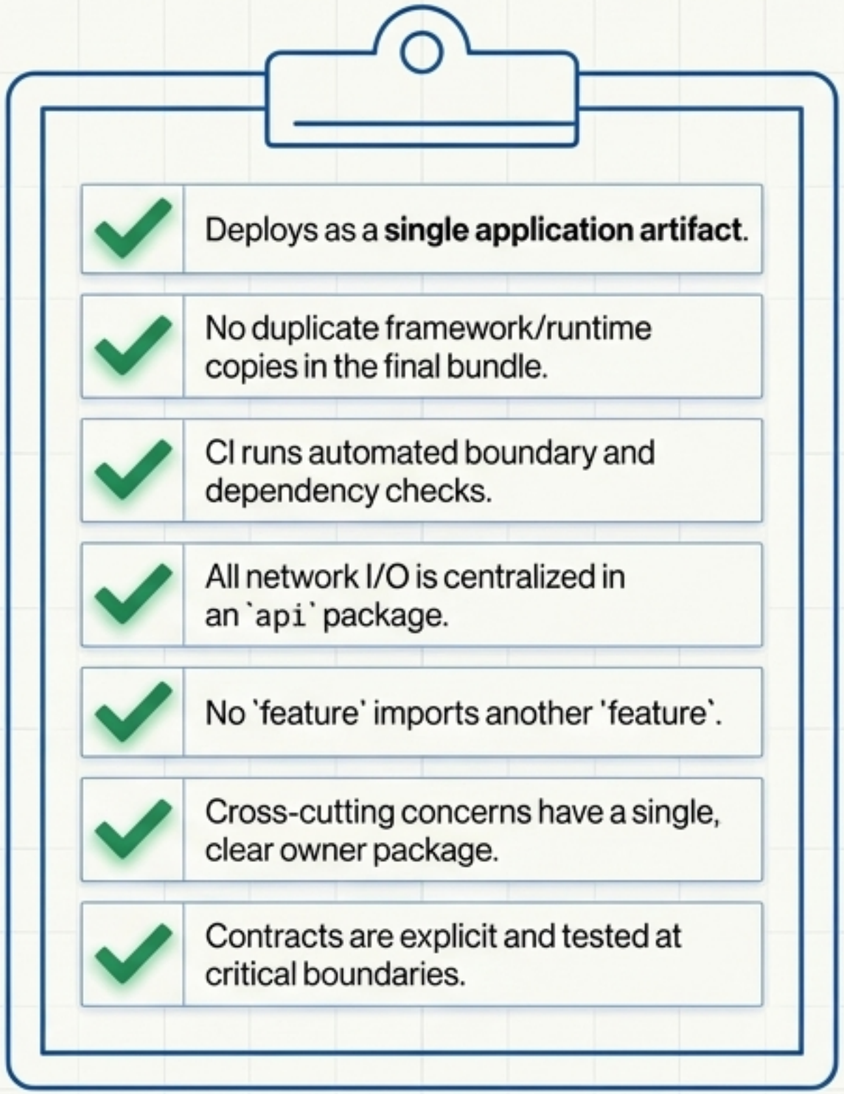
1. Extract `contracts`
2. Extract `api`
3. Extract `ui`
4. Extract `core`
5. Split `features`
6. Add Enforcement
7. Add Boundary Tests
8. Validate Bundles

Low-risk, high-value steps



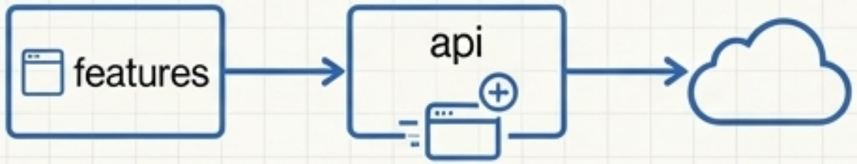
SDMM Architecture

The SDMM Checklist: A Definition of ‘Done’



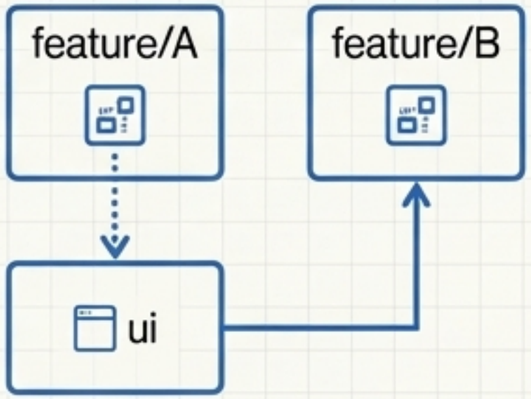
Common Scenarios, Solved with SDMM

I need to add a new API endpoint.



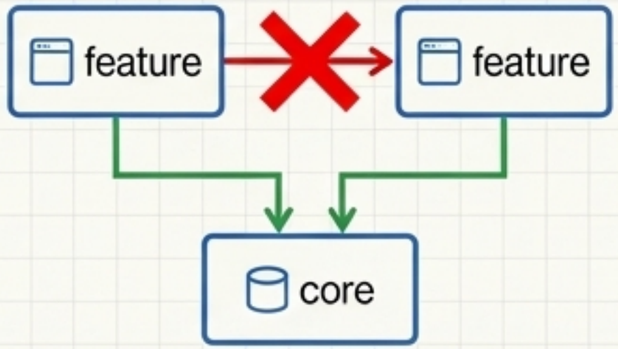
Add a typed wrapper in the `api` package. Export it. Features consume the new wrapper.

This UI component is used in two features.



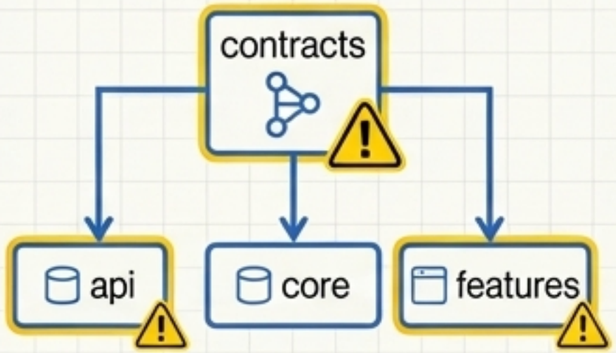
Promote the component from the feature package into the `ui` package, making it available to all features.

Two features need to share state.



Avoid direct feature-to-feature communication. Either use the backend as the source of truth or promote the shared state into the `core` package.

I need to change a core data shape.



Treat this as a high-risk change. Update the type in `contracts`, then update the implementation (`api`/`core`) and all consumers in the same atomic change set.

The Guiding Philosophy

Build-time Modularity, Runtime Simplicity.

Get the benefits of modularity without the costs of micro-frontends.

Boundaries and Contracts over Folders and Conventions.

Make rules explicit and enforceable, not just suggested.

A Codebase Designed for Humans *and* AI.

Structure your code today for the collaborators of tomorrow.

AzzCraft.com