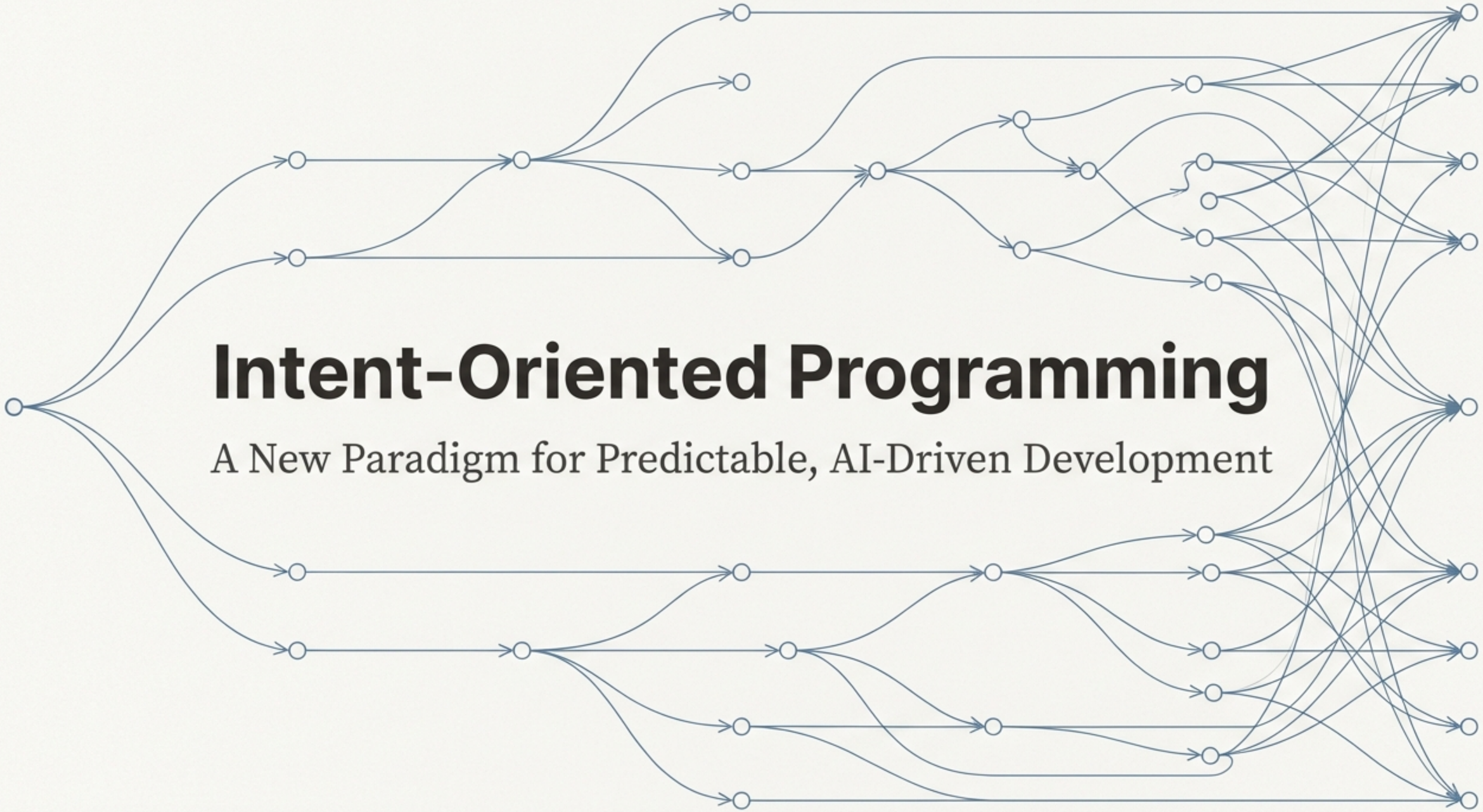
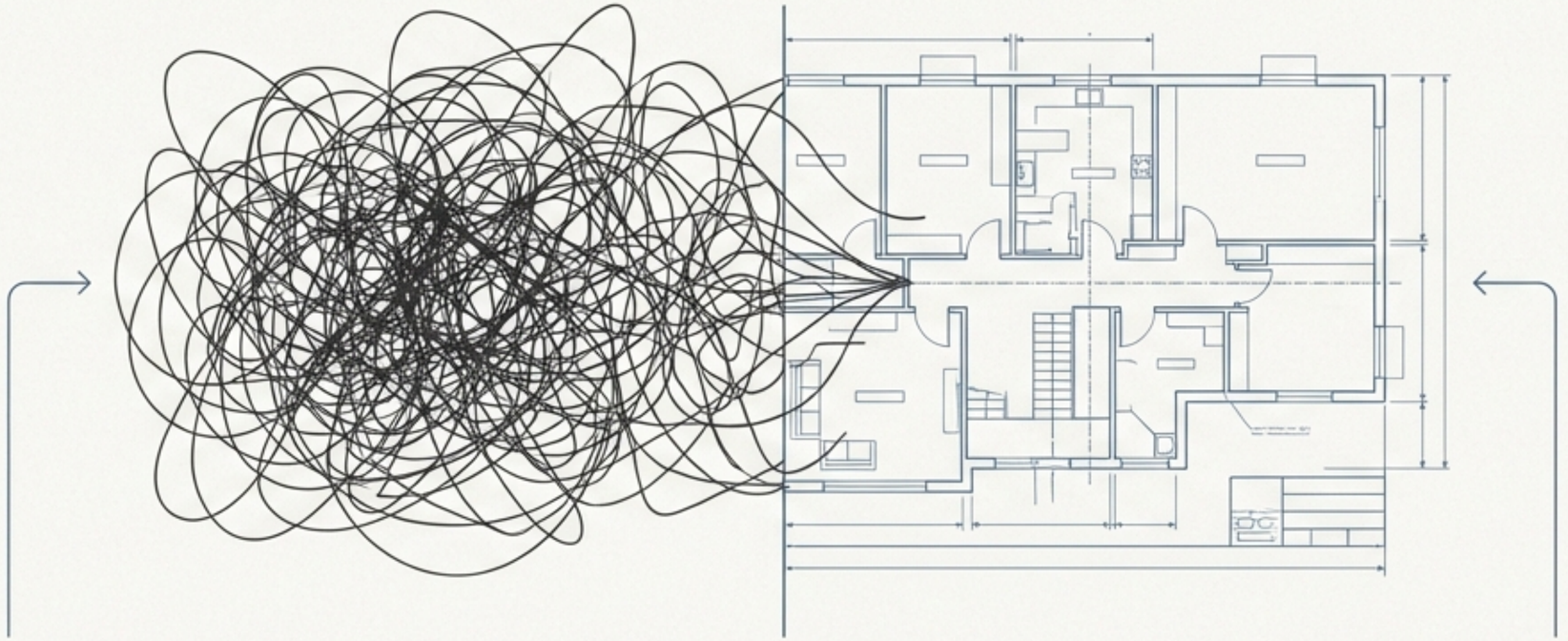


Intent-Oriented Programming

A New Paradigm for Predictable, AI-Driven Development

The Unpredictability of AI Co-Pilots

Traditional AI-assisted coding tools operate as autonomous agents that introduce chaos and unpredictability into the development process.



Large, Hard-to-Review Diffs

Agents produce monolithic changes that are difficult to validate, blurring authorship and responsibility.

Implicit, Brittle Decisions

Agents make unstated assumptions and implementation choices that can drift in behavior across runs, creating fragile code.

Conceptually Opaque Code

The resulting codebase may be functionally correct, but its underlying purpose and structure are obscured, making it hard to maintain.

Shift the Source of Truth from Code to Intent

In **Intent-Oriented** Programming, developers author structured **intent** as the canonical source of truth. Code becomes a derived build artifact, not the primary artifact.

The Key Shift

AI is no longer a collaborator.

AI becomes an invisible implementation mechanism. It is a reliable execution engine that operates on a pre-defined plan.

The Intent-Oriented Programming Workflow

1. AUTHOR INTENT

The developer defines the application's purpose, behavior, and structure in human-readable `.intent` files using Structured Natural Language.

2. ORCHESTRATOR BUILDS & DIFFS

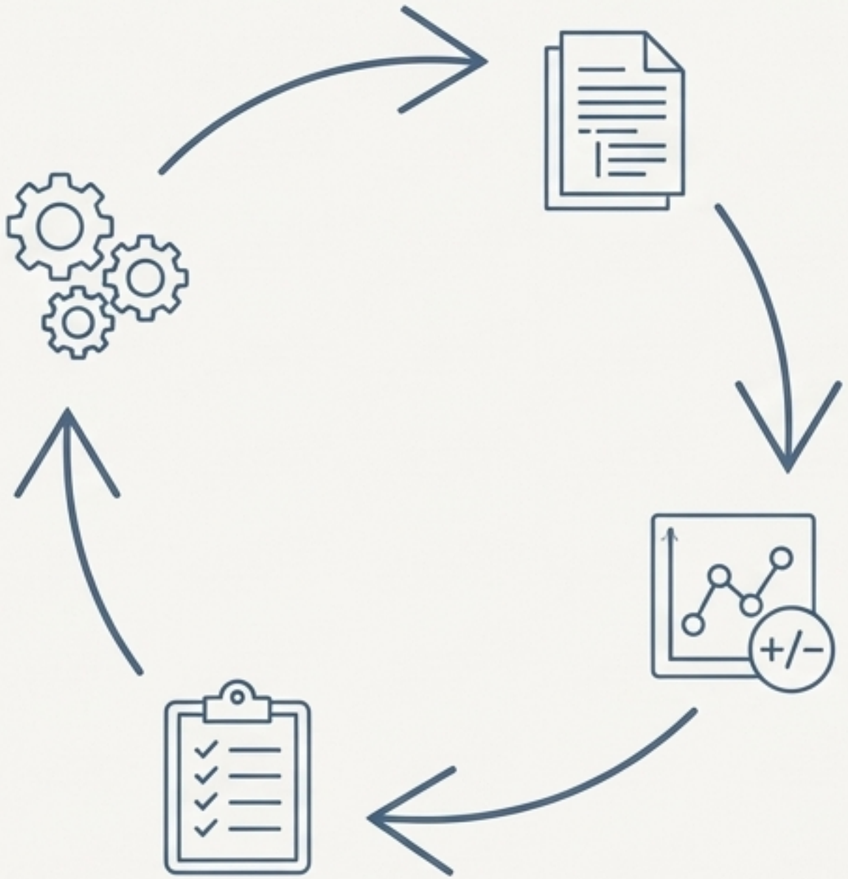
A deterministic compiler parses intents, builds a dependency graph, and computes a **semantic diff** against the previous state. The orchestrator is code, not an LLM.

3. GENERATE PATCH PLAN

Based on the semantic diff, the orchestrator produces a structured, ordered list of small, scoped change tasks. This plan is stable and reviewable.

4. AGENTS EXECUTE PATCHES

Invisible worker agents execute each task in the plan, applying small, targeted patches to the codebase. Agents cannot change scope or intent.



This deterministic process turns large, conceptual changes into a series of safe, incremental, and verifiable steps.

The Anatomy of an Intent

Header

The machine-addressable identity. The `id`, `kind`, and `version` allow the orchestrator to track and version the intent.

References

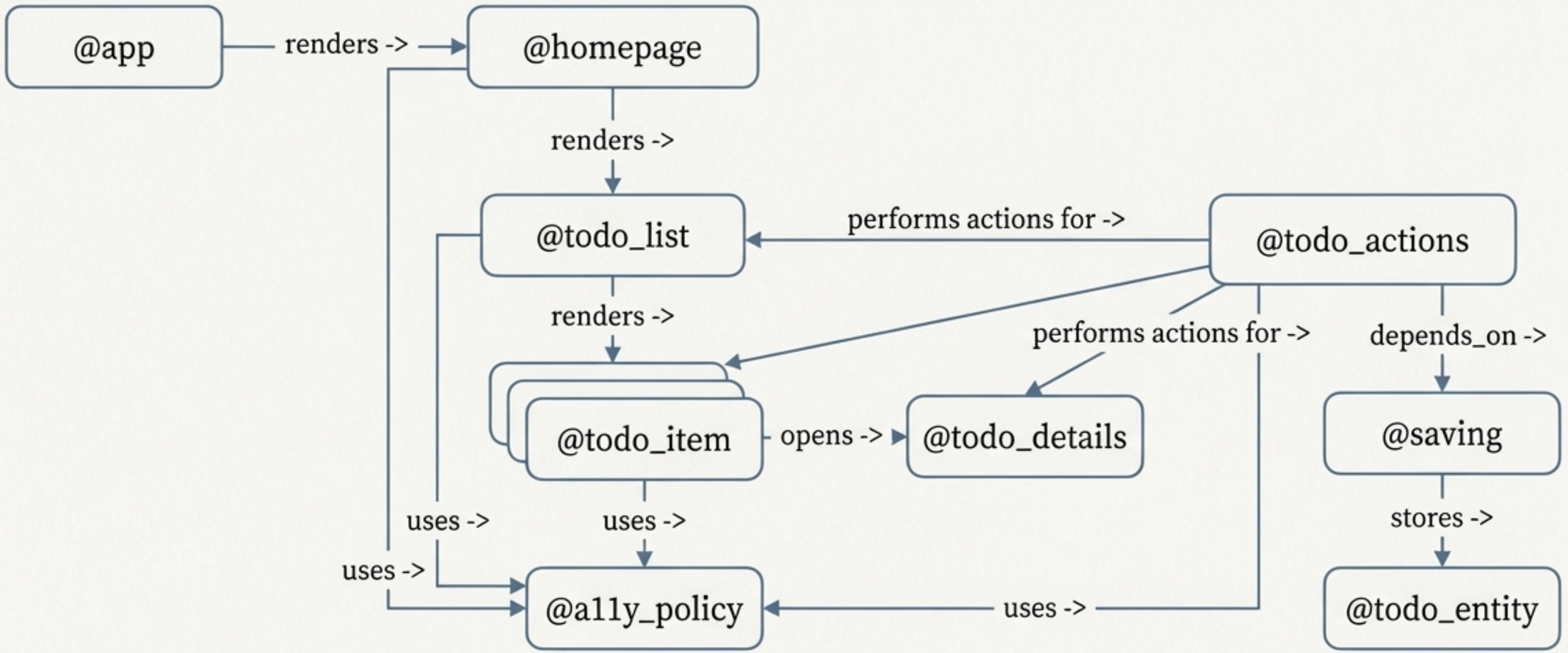
Declarative dependencies that form the Intent Graph. `renders`, `uses`, and `depends_on` explicitly define relationships without ambiguity.

```
// intents/features/homepage.intent

intent "homepage" {
  id      = "view.homepage.v1"
  kind    = "view"
  version = "1.0.0"

  route = "/"
  renders = ["@todo_list"]
  uses = ["@branding", "@a11y_policy"]

  purpose "p0" {
    text = <<PROMPT
    This is the first page users see...
    PROMPT
  }
  define "layout.shell" {
    text = <<PROMPT
    Render a header with the app name...
    PROMPT
  }
}
```

Atomic Blocks

The smallest unit of change. Natural language is contained within stable, ID-ed blocks (`purpose`, `define`) that describe *what* to build, not *how*.

An Application is a Graph of Composable Intents



The Intent Graph is the primary input to the orchestrator. It makes the entire application's architecture explicit, resolvable, and analyzable before any code is generated.

From Declarative Intent to Rendered UI

```
// intents/features/todo-item.intent

define "ui.layout" {
  text = <<PROMPT
  Show a drag handle on the left, then a
  checkbox, then the todo title. On the
  right, show a subtle indicator that
  details are available (e.g., chevron).
  PROMPT
}
```

Intents describe the desired outcome. The orchestrator and agents are responsible for the implementation details, translating human-readable purpose into production-ready components.

Making Change Safe, Part 1: The Semantic Diff

Let’s improve accessibility by adding a keyboard fallback for reordering items.

Before

```
// todo-list.intent
define "interaction.drag_drop" {
  text = <<PROMPT
  Items can be drag-and-dropped to reorder.
  Drag handle is on the left with a grip icon.
  PROMPT
}
```

After

```
// todo-list.intent
define "interaction.drag_drop" {
  text = <<PROMPT
  Items can be drag-and-dropped to reorder.
  Drag handle is on the left with a grip icon.
  PROMPT
  acceptance = [
    "Keyboard can reorder via fallback controls"
  ]
}
```

Unlike a text diff, the orchestrator understands this isn’t just a text change. It’s a **semantic change**: `MODIFY_BLOCK(component.todo_list.v1::define.interaction.drag_drop)`. It has modified the acceptance criteria, which triggers a specific, targeted plan.

Making Change Safe, Part 2: The Generated Patch Plan

The semantic diff is used to generate a structured, reviewable **Patch Plan.** Agents will execute this plan, and only this plan.

Patch Plan: `20240521-add-keyboard-reorder.json`

Task 1	Task 2	Task 3
Implement keyboard reorder controls (e.g., up/down arrows) in the Todo List component.	Ensure the `reorder` action from @todo_actions is invoked by the new keyboard controls.	Add integration tests to verify keyboard reordering and persistence.
Scope: component.todo_list.v1 | **Constraints:** Must adhere to @a11y_policy	**Scope:** component.todo_list.v1, actions.todo.v1 | **Constraints:** Must not alter existing drag-and-drop functionality.	**Scope:** Test suite for component.todo_list.v1 | **Acceptance:** Order persists after refresh

A small change in intent produces a safe, auditable execution plan. There are no surprises, no scope creep, and no opaque changes.

The IOP Advantage

1. Predictability

Deterministic compilation and semantic diffs mean that the same intent change always produces the same plan. No surprises.

2. Clarity & Maintainability

The codebase's purpose is explicit and human-readable in the intent files. The architecture is always in synce with the source of truth.

3. Safety & Control

Changes are applied as small, incremental, reviewable patches. The developer remains the final authority, approving the plan before execution.

4. Evolvability

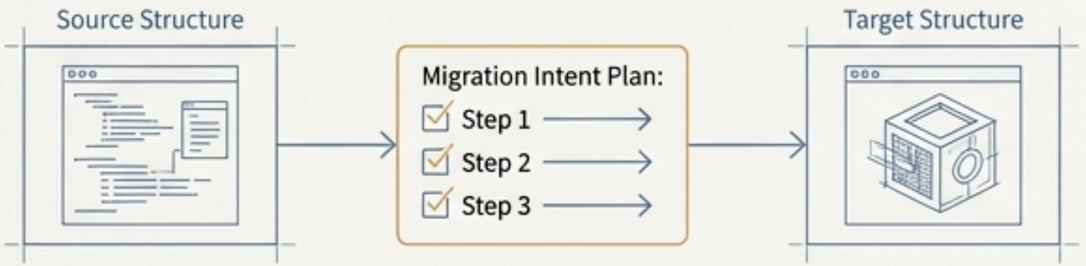
Refactoring and decommissioning are first-class, structured concepts, not risky, monolithic pull requests. The system is designed for long-term health.

A System Designed for Long-Term Evolution

Refactoring as Planned Migration

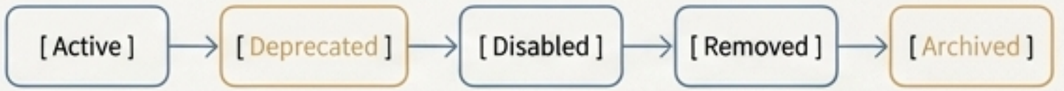
Large refactors are not giant, risky PRs. They are handled via **Migration Intents**.

A migration intent is a declarative, multi-step plan that defines the evolution from a source structure to a target structure. The compiler enforces invariants at each step, applying the change as multiple, small patch batches.



Deletion as Safe Decommissioning

Features are never “hard deleted.” Instead, intents move through a managed lifecycle.



This allows for gradual cleanup, provides compiler warnings on deprecated intents, and enables safe, automated garbage collection of unreachable code artifacts.

Programming is Evolving

The paradigm shifts from authoring *code* to authoring *intent*.

Intent-Oriented Programming treats AI as a powerful, invisible compilation backend, not an unpredictable collaborator.

It allows us to build codebases that remain consistent, understandable, and evolvable as they grow in complexity.

It decomposes large, ambitious changes into a series of safe, progressive, and predictable steps.