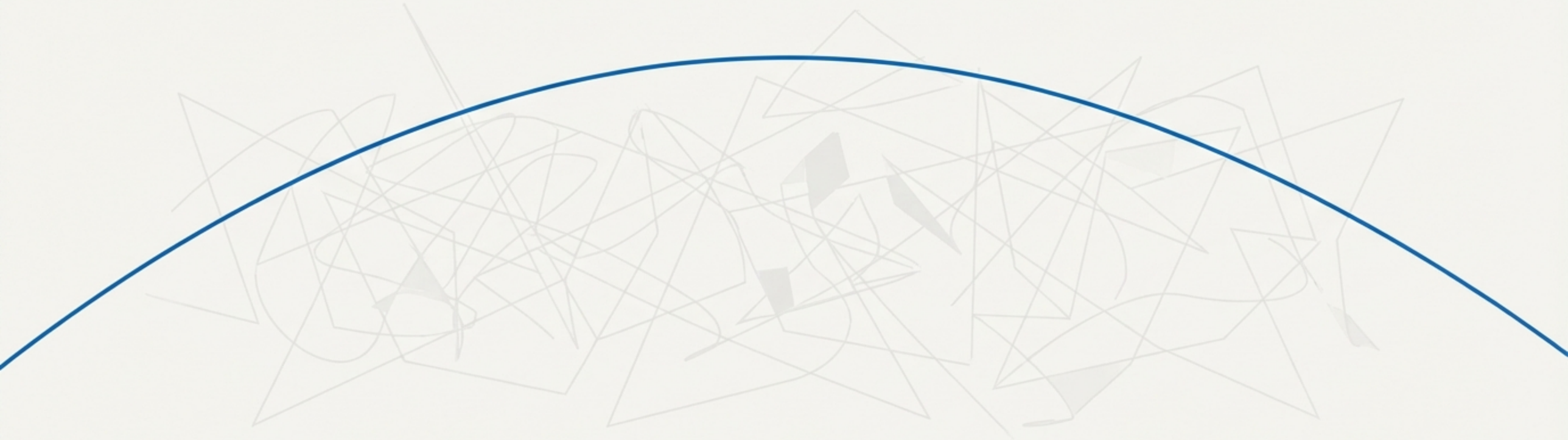
Iterative Flow Development

A Methodology for Rapid, Rigorous Web Development



The guiding principle: Preserving developer flow state while leveraging AI for code generation.

Modern development is a battle against context switching. IFD is designed to win it.

The Chaos of Context Switching

- Fighting framework boilerplate
- Managing complex dependencies
- Switching between code, UX, and mocked data
- Losing hours to setup and configuration

The Focus of Flow State

- You focus on UX and architecture
- The LLM handles boilerplate code
- Work in focused 2-3 hour sessions
- Minimise interruptions, maximise productivity

The Five Principles of Iterative Flow Development

Flow State Preservation

Minimise context switching by letting the developer focus on UX and the LLM on boilerplate.

UX-First Development

Define and iterate on the user experience before finalising the technical implementation.

Real Data From Day One

Eliminate integration surprises by testing against a live backend from version 0.1.0. No mocked data. No stubbed services.

Zero External Dependencies

Ensure longevity and simplicity by using only native web platform APIs (ES6+, Web Components).

Progressive Enhancement

Start with a simple, functional core and layer on complexity and features iteratively.

A Versioning System Built for Speed and Safety

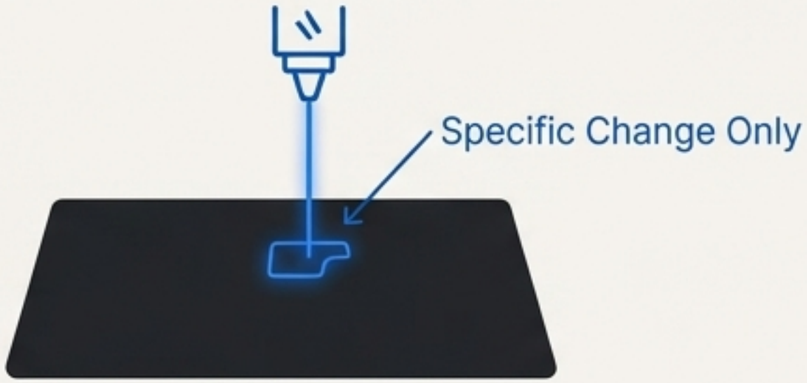
IFD uses three distinct version types, each with specific rules governing code sharing and purpose. This structure enables rapid experimentation in minor versions and guarantees stability in major versions.

Version Type	Pattern	Example	Code Sharing	Purpose
Release	`vX.0.0`	`v1.0.0`	None	Production deployment
Major	`vN.X.0`	`v0.2.0`	None (self-contained)	Consolidation checkpoint
Minor	`vN.N.X`	`v0.1.3`	Yes (link back + surgical override)	Active development

The Engine of IFD: Minor Versions use Surgical Overrides

The unit of change is **the change itself**, not the file. Minor versions (e.g., v0.1.1) contain *only* the specific code that has been added or changed.

Surgical Override (The IFD Way)



Traditional Change (The Old Way)

CSS Surgical Override Example

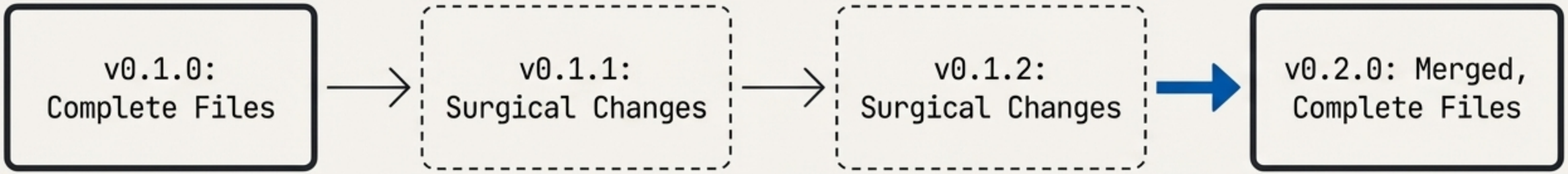
Instead of copying the entire `v0.1.0/styles.css` file, `v0.1.1/styles.css` contains only the new rule.

```
/* v0.1.1/styles.css */
/* Link back to base styles */
@import '../v0.1.0/styles.css';

/* Surgical override */
.button {
  background-color: var(--primary-color-dark);
}
```

This keeps changes small, isolated, and easy to review.

The Lifecycle: From Surgical Iteration to Consolidated Major Version



Side-by-Side Folder Structure Comparison

Minor Version - Surgical & Sparse

- v0.1.1/
  - component.js (contains one overridden method)
  - styles.css (contains one new CSS rule)

Major Version - Complete & Merged

- v0.2.0/
  - component.js (full file with all changes from v0.1.x merged)
  - styles.css (full file with all changes from v0.1.x merged)

Minor versions are for active development and contain only deltas. Major versions are consolidation checkpoints, merging all proven surgical overrides into a new, self-contained set of complete files.

Built on the Web Platform: A Zero-Dependency Architecture

We use native Web Components (custom elements) and browser APIs. This is not a “no-framework” stance for its own sake; it’s a deliberate choice for long-term stability and simplicity.

Key Characteristics

Technology: ES6+ JavaScript, Web Components, CustomEvents.

APIs: Native browser APIs only (e.g., Fetch, DOM).

Benefits

- **Longevity**: Code is not tied to a framework’s lifecycle.
- **Simplicity:** No build steps, no dependency management, no abstractions to learn.
- **Performance:** Closer to the metal, less overhead.

Designing Components for Surgical Iteration

Your component architecture must be designed to be easily and safely overridden.

✓ DO

- Use **prototype methods**: They are easily overridable in subsequent minor versions.
- Use **CSS custom properties**: Allows for simple theme and style tweaks without touching selectors.
- Keep methods **small and focused**: Smaller targets are easier to override surgically.
- Use **CustomEvents** for communication: Decouples components, preventing brittle direct method calls.

✗ DON'T

- Use **closures** that hide methods: Private methods cannot be overridden.
- Use **direct method calls** between components: Creates tight coupling that breaks when one component is changed.
- Forget `disconnectedCallback` cleanup: Essential for preventing memory leaks in single-page applications.

Testing is Co-Located, Surgical, and Version-Scoped

In IFD, testing mirrors development. Tests are as surgical as the code changes they verify, ensuring focus and relevance.

Co-Located Tests

Test files live directly next to the source files they test.

```
component.js
component.test.js
```

Surgical Tests Match Surgical Changes

If you override two methods in `v0.1.2` (comment), you write tests for *only those two methods* in `v0.1.2/component.test.js`.

Version-Scoped

Each minor version tests its own changes. The full, consolidated test suite runs against major versions.

A Flexible Testing Infrastructure for Any Environment

Tests are written once and run everywhere, from real-time feedback in your IDE to your CI pipeline.

Configuration Files

`test-paths.js`

A centralised configuration file that defines paths to source and test files for each version.

`test-utils.js`

A library of shared helper functions for tests (e.g., creating elements, dispatching events).

The Three Test Environments

Environment	Tool	Use Case
 Browser	`tests/index.html`	Local development and production QA
 CLI	`Karma + ChromeHeadless`	CI pipelines
 IDE	`Wallaby.js`	Real-time TDD and in-editor feedback

The Development Rhythm: From UX Idea to Consolidated Feature

Feature Iteration Process

Start with UX: Describe the desired user experience in plain language to the LLM.

LLM Generates: The LLM generates the initial code for a new minor version.

Human Integrates: In an air-gapped environment, the developer reviews, refines, and integrates the code.

Test with Real Backend: The developer immediately tests the new feature against live APIs.

Iterate: Create subsequent minor versions (v0.1.1, v0.1.2...) to surgically polish UI and add details based on real interaction.

Version-by-Version Focus

A typical feature rollout

Version	Focus
`v0.1.0`	Core MVP only – embarrassingly simple.
`v0.1.1`	First UI improvements.
`v0.1.2`	UI polish.
`v0.1.3`	Data enhancements.
`v0.1.4`	Monitoring/logging.
`v0.2.0`	Consolidation – merge proven features.

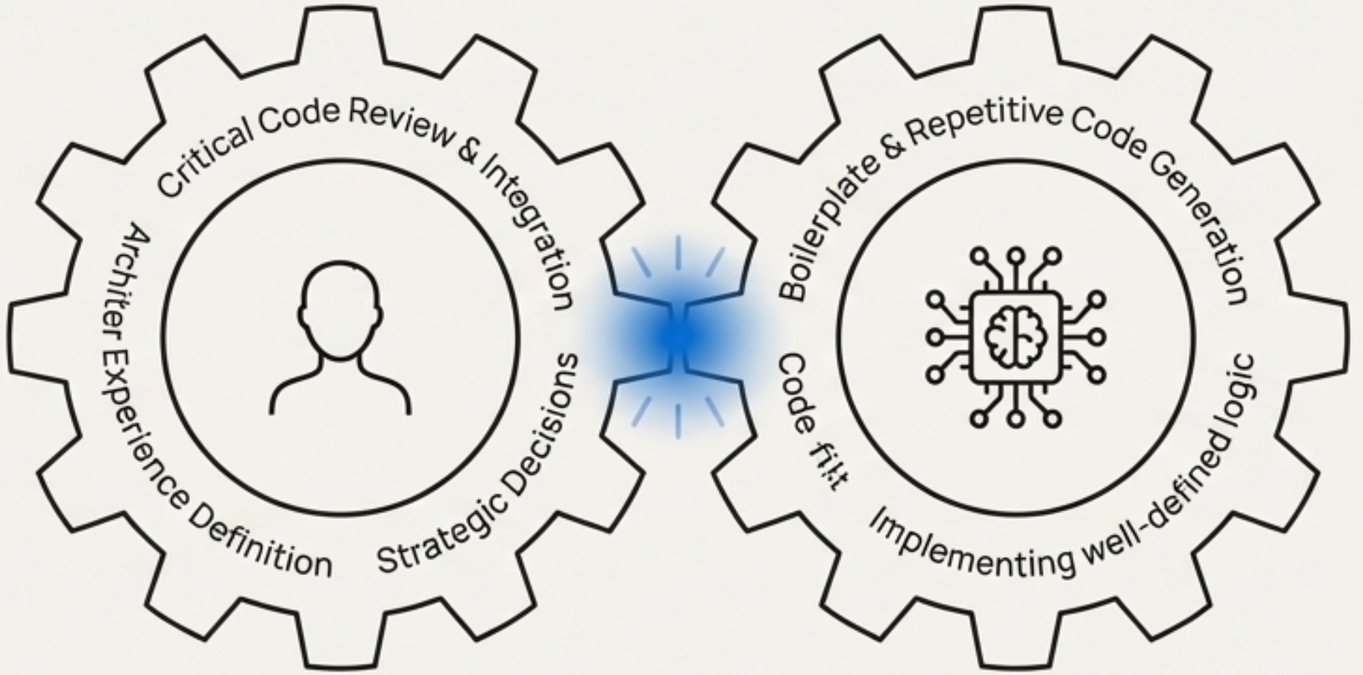
The Human and the LLM: A Partnership in Code

In IFD, the LLM is a powerful code generator, but the human is always the architect and final arbiter.

Human Role

Architecture & System Design

User Experience Definition



AI Role

Boilerplate & Repetitive Code Generation

Code Refinement & Suggestion

Example in Practice: LLM Prompt for a Surgical Bug Fix

```
I am working on version v0.4.5, which is based on v0.4.4.
In the file v0.4.4/components/user-profile.js, the method 'updateAvatar' has a bug where it doesn't handle null image URLs.

Create a new file at v0.4.5/components/user-profile.js.
This new file should contain ONLY the overridden 'updateAvatar' method with added null-checking logic.
```

Critical Anti-Patterns to Avoid

- ❌ **Don’t** copy entire files to change one method — use surgical overrides.
- ❌ **Don’t** mock data or stub APIs — use real data from day one.
- ❌ **Don’t** use external JavaScript frameworks/libraries — stick to the web platform.
- ❌ **Don’t** create shared code between Major versions — they must be self-contained.
- ❌ **Don’t** use direct method calls between components — use events.
- ❌ **Don’t** bundle unrelated changes in one minor version — keep them focused.
- ❌ **Don’t** create full test files for surgical overrides — test only what changed.

Iterative Flow Development at a Glance

Aspect	IFD Approach
Dependencies	None (native web only)
Components	Web Components (custom elements)
Communication	CustomEvents (event-driven)
Minor Versions	Surgical overrides, link back
Major Versions	Self-contained, merged files
API	Real from v0.1.0, no mocks
Testing	Co-located, surgical
AI Role	Code generation + refinement
Human Role	Architecture + UX + review + decisions

The Freedom of Flow, The Confidence of Rigour

IFD is about maintaining flow state. The methodology keeps you focused on solving user problems, not fighting tools.

Version independence means freedom. Experiment without fear. If v0.4.5 is a dead end, you simply build v0.4.6 from v0.4.4.

Real data from day one means confidence. You know v1.0.0 works because it has been tested with the real API since v0.1.0.

What stakeholders test IS what ships. A release version (v1.0.0) is an exact copy of the last major version, with only version strings changed. No last-minute code changes.

This is IFD. Simple, fast, maintainable, and built for flow.