Introduction to Functional Programming: Functions, Immutability, and Composition

- Organize programs with pure functions, immutable data, and composition
- FP in 2026: pragmatic response to concurrency, scaling costs, and complexity
- Easier reasoning, testability, and safer parallelism with fewer side effects
- Roadmap: functions as values, immutable state, map/filter/reduce, incremental adoption

Two Ways to Think: Imperative vs. Declarative

IMPERATIVE (HOW)

Step-by-step recipe

- Understand the problem
- Get the data
- Loop through items
- Apply logic
- Store intermediate results
- Return the final result

VS.

DECLARATIVE (WHAT)

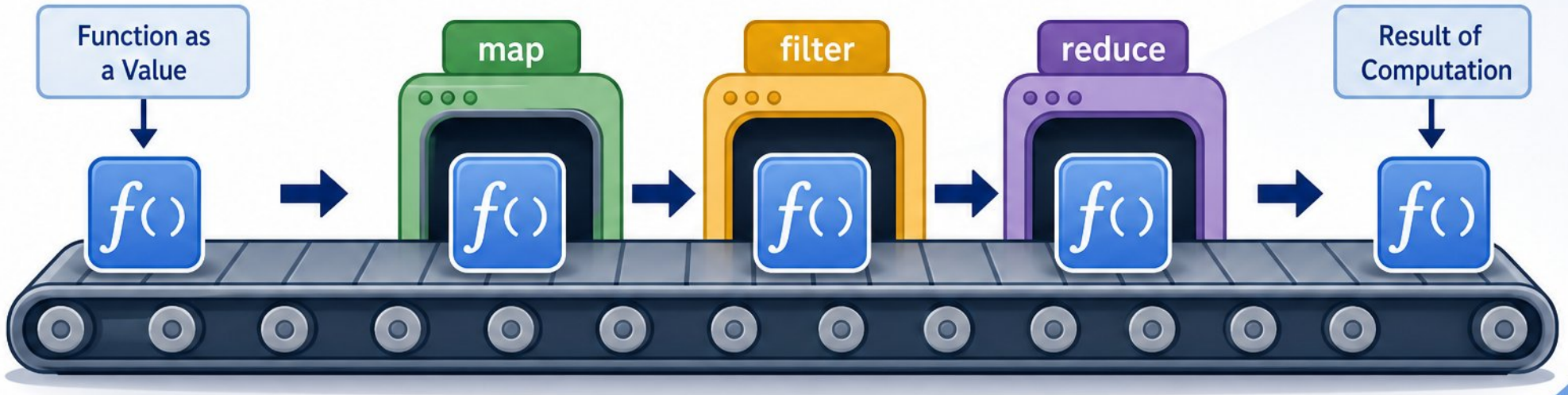
Describe the result

```
map( filter( data, condition ),
transform )
```

- Imperative: step-by-step recipes describing how to compute
- Declarative: descriptions of what result should be computed
- Loops (how) vs. map/filter (what) clarify intent
- Declarative code often reads closer to the problem statement
- Mental shift: think about data transformations, not control flow

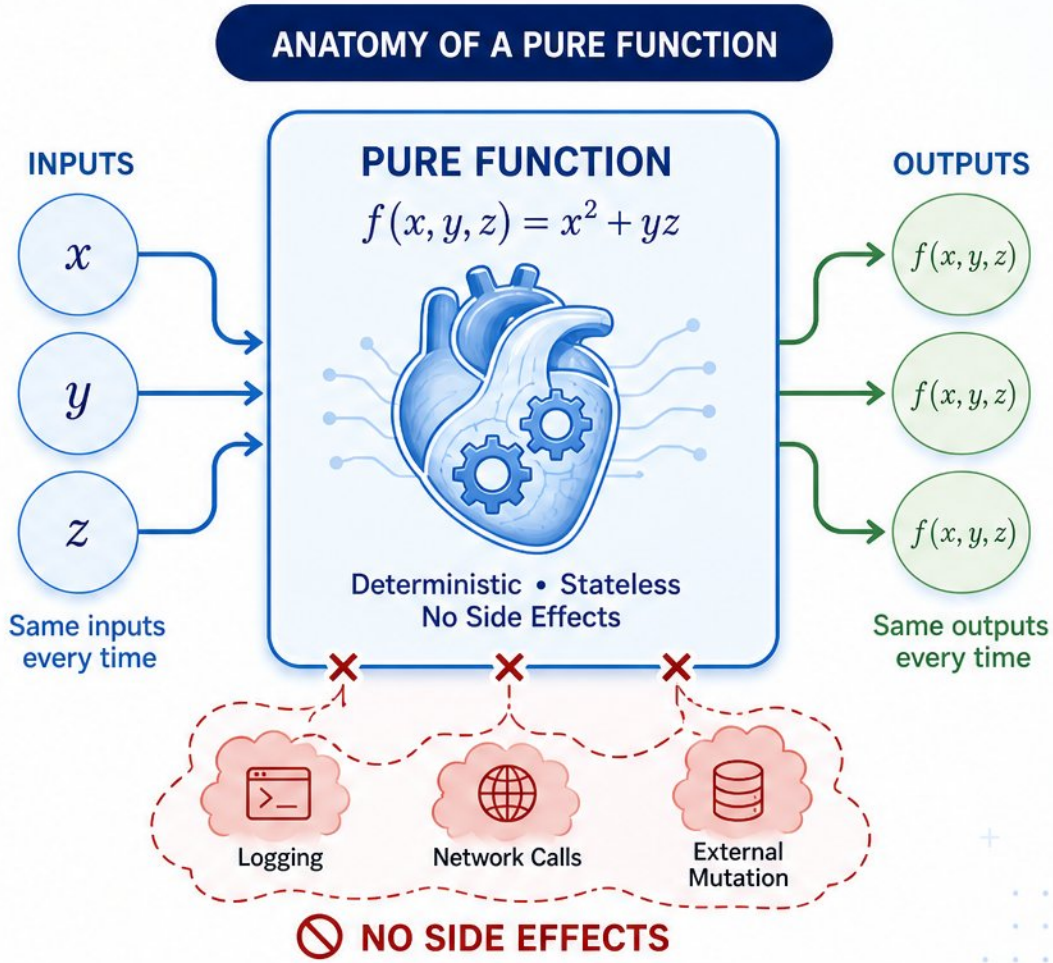
Functions as First-Class Citizens

- Functions can be assigned to variables, passed as arguments, and returned as results
- Higher-order functions (map, filter, reduce) abstract control flow patterns
- Lambda expressions create lightweight, inline behavior without naming
- Refactor loops into declarative chains: data.map(f).filter(g).reduce(h)



The Foundation of Predictability: Pure Functions

- Output depends only on inputs; no side effects (no logging, network calls, or external mutation).
- Referential transparency: replace a call with its result without changing behavior.
- Benefits: predictable code, trivial unit tests without mocks, safe caching, and parallelization.
- Impure: reads system clock internally. Pure: receives time as an explicit parameter.



IMPURE

Reads system clock internally.

```
function getCurrentTime() {
  return now(); // reads system clock
}
```

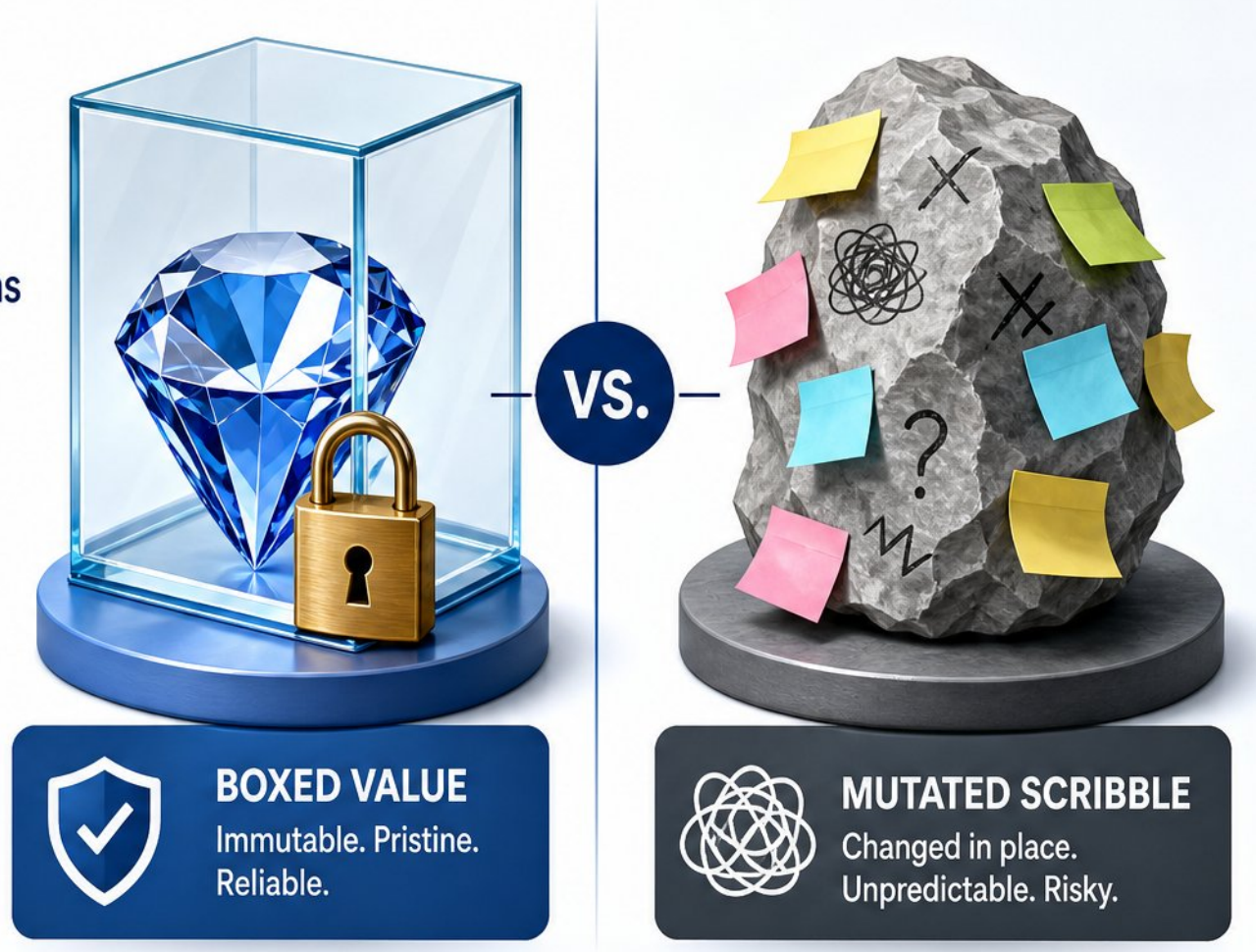
PURE

Receives time as an explicit parameter.

```
function formatTime(t) {
  return format(t);
}
```

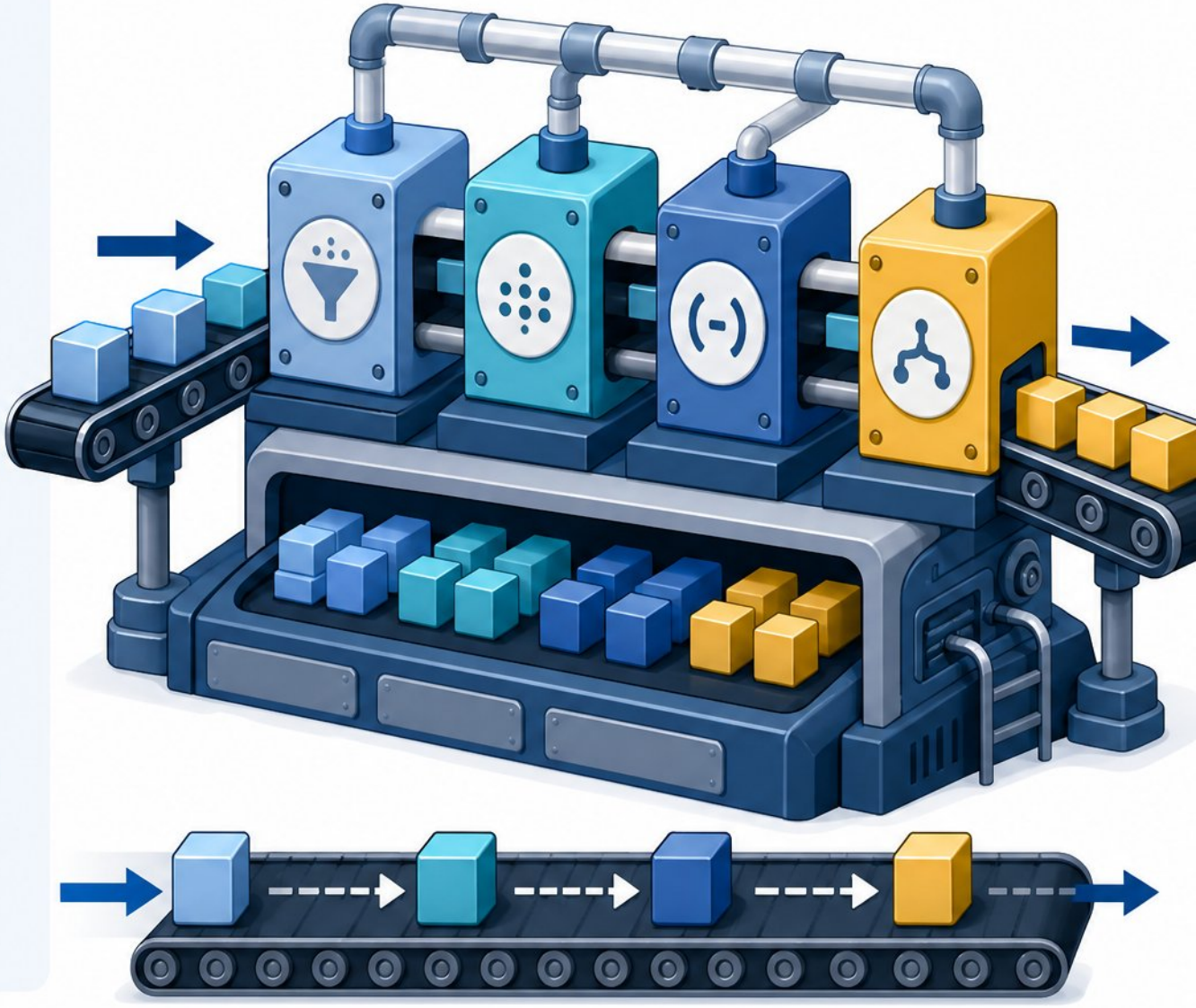
Immutability: The Key to Safer State

- Immutable data: values that never change; 'updates' always produce a new copy
- Eliminates unexpected mutations and the need for defensive copying completely
- Persistent data structures use structural sharing to make immutability efficient
- Mutable: `array.push(x)` vs. Immutable: `[...array, x]` (spread/concat)



Composition: Building Complexity from Simple Parts

- Build complex logic by combining small, focused pure functions
- Thread data through pipelines using pipe, compose, or native operators
- Favor composition over inheritance to organize logic flexibly
- Compose manually: `f(g(x))`; with helpers; or in real data pipelines



Taming the Real World: Managing Side Effects

- Real programs need I/O, databases, and randomness; pure functions alone aren't enough

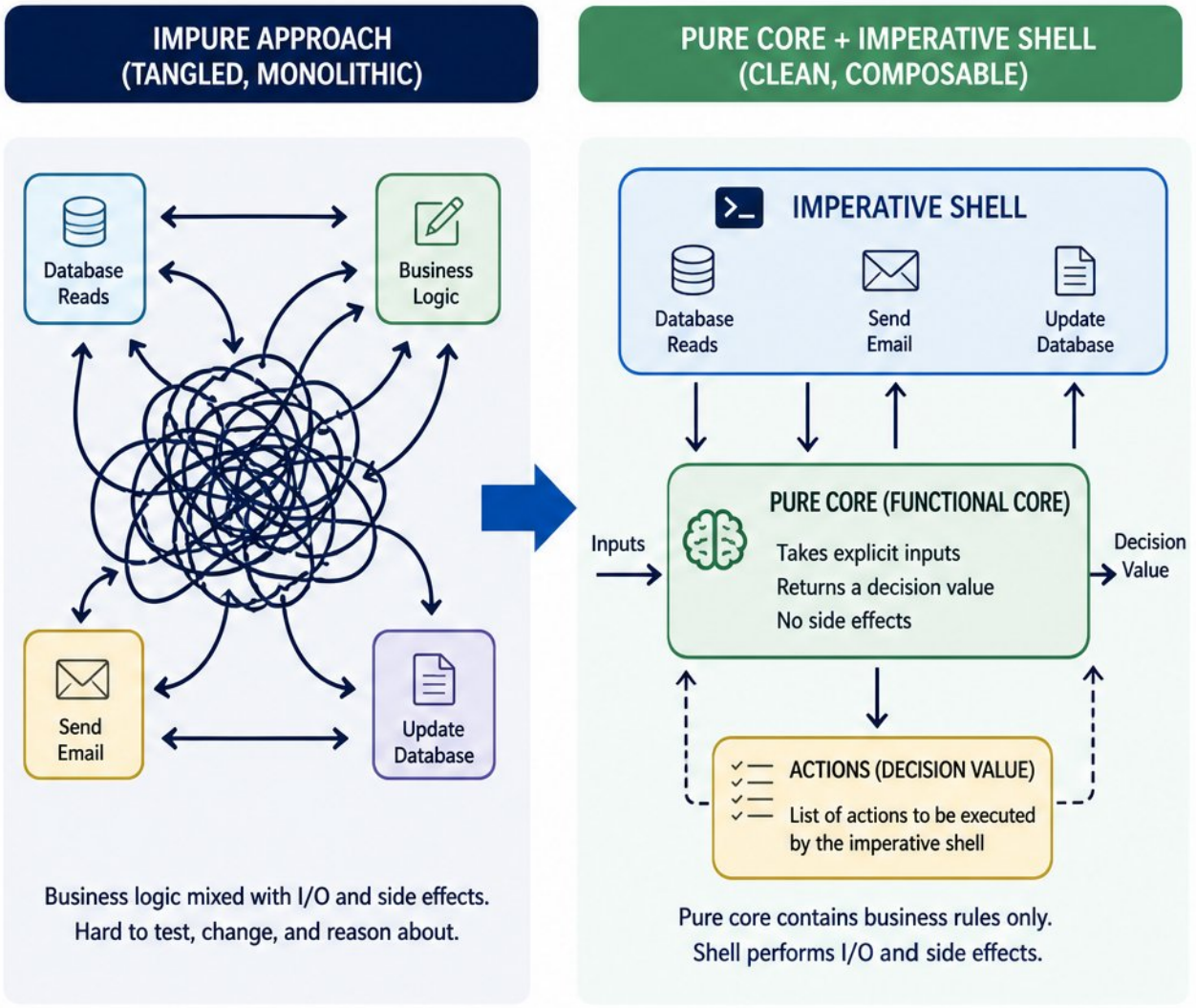
- Functional Core, Imperative Shell: push side effects to the edges, keep core logic pure

- Describe effects as data to delay execution and improve testability

- Core returns decisions; thin shell executes I/O based on those decisions

The Functional Core, Imperative Shell in Practice

- Walkthrough of an 'Update Customer Profile' scenario, contrasting impure vs. pure core implementations
- Pure core uses explicit inputs and returns a decision value; no side effects inside the core logic
- Imperative shell handles database reads, calls the pure core, and executes the returned actions
- The pattern naturally aligns with Hexagonal Architecture, Ports & Adapters, and Clean Architecture
- Result: a thin, simple shell and an easy-to-test, predictable pure logic core



Functional Tools in Your Everyday Language

Imperative Loop (Before)

```
let result = [];
for (let item of items) {
  if (item.active) {
    let value = transform(item);
    result.push(value);
  }
}
return result;
```

Functional Pipeline (After)

```
const result = items
  .filter(item => item.active)
  .map(item => transform(item))
  .reduce((acc, value) => {
    acc.push(value);
    return acc;
  }, []);
```

- Map/filter/reduce chains replace loops in Python, JS, Java Streams, C# LINQ
- `const` over `let`, spread operators, and frozen data classes for immutability
- Refactor imperative loops to declarative pipelines for clearer intent
- Start where you are: use your current language's functional features first

Modeling Data, Not Nulls: Optionals and Pattern Matching

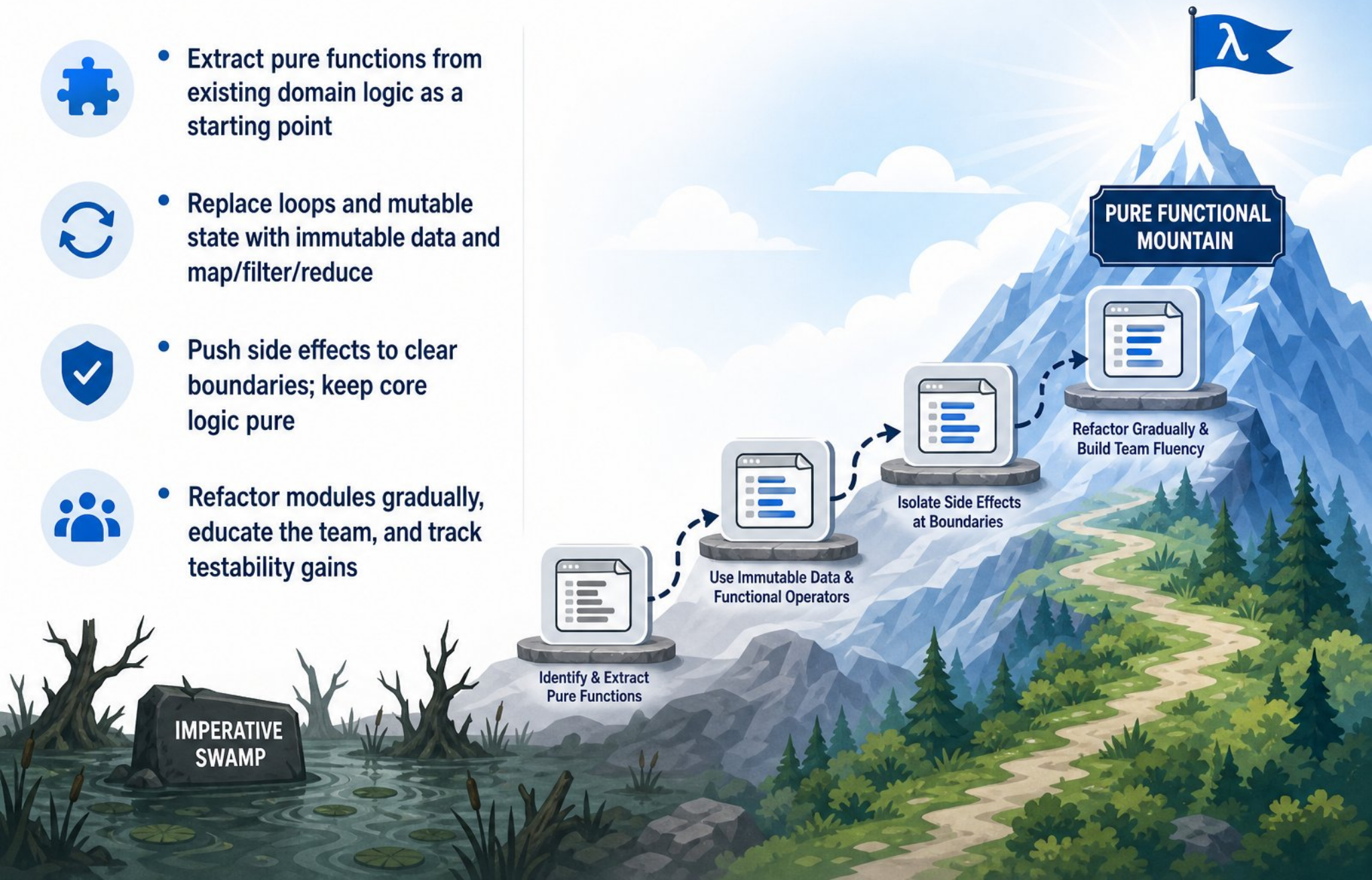
- Optional/Maybe types explicitly model value absence, avoiding null-reference errors
- Java Optional, TypeScript strict null checks, and library types like fp-ts Option
- Pattern matching enables declarative, safe branching on object shape and structure
- Now available in Python 3.10+, Rust, Kotlin, Swift, Ruby 4.1, and Java (JEP 441)
- TC39 proposal advancing; ts-pattern (~900K weekly downloads) fills the gap for TypeScript today

The 2026 FP Landscape: More Than Just Niche Languages

- **FP adoption:** a pragmatic response to complex distributed systems and rising maintenance costs
- **Surveying FP families:** Haskell (high-assurance), Gleam/Rust (pragmatic systems), Elixir/Erlang (fault-tolerant), OCaml/F# (.NET/infrastructure)
- **The dominant trend:** "Functional but pragmatic." FP principles absorbed into TypeScript, Kotlin, Rust, Swift

Incremental Adoption: Where Your Codebase Wins

- Extract pure functions from existing domain logic as a starting point
- Replace loops and mutable state with immutable data and map/filter/reduce
- Push side effects to clear boundaries; keep core logic pure
- Refactor modules gradually, educate the team, and track testability gains



Testing and Reasoning with Confidence

- Pure functions & immutable data mean no complex mocks
- Property-based testing finds edge cases via random inputs
- Functional core, imperative shell shrinks flaky integration tests
- Same inputs always replay for isolated debugging

Where to Go Next: Languages and Communities

- Explore Haskell, Scala, Clojure, F#, Elixir, or Gleam
- Start with Elm/Elixir or deepen FP within TypeScript/Python
- Use "Mostly Adequate Guide," Haskell MOOC, or Exercism.io
- TypeScript learners: try the browser-based fp-ts playground

Practical Toolkit: Your First Steps and Key Takeaways

- Prioritize pure functions; pass dependencies explicitly
- Default to immutable data: use const, spread, and new returns
- Replace simple loops with map, filter, and reduce for clarity
- Push I/O to the edges; keep your core logic pure and testable
- Focus on ideas first (purity, immutability, composition) over language

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/66589eb4/public