Compile-time data contracts & Fiber-safe data pipelines

Let the compiler refuse broken migrations;
let fibers keep retries safe.

ScalaIO Paris 2025 | Vitthal Mirji | Staff Data Engineer @ Walmart

About Me

About Me

- Staff Data Engineer @ Walmart, Offices in India

About Me

- Staff Data Engineer @ Walmart, Offices in India
- From Mumbai, India

About Me

- Staff Data Engineer @ Walmart, Offices in India
- From Mumbai, India
- 10+ years building data platforms at scale (Deloitte → Walmart)

About Me

- Staff Data Engineer @ Walmart, Offices in India
- From Mumbai, India
- 10+ years building data platforms at scale (Deloitte → Walmart)
- I cook good Indian food 🍛

About Me

- Staff Data Engineer @ Walmart, Offices in India
- From Mumbai, India
- 10+ years building data platforms at scale (Deloitte → Walmart)
- I cook good Indian food 🍛
- Visit: https://vitthalmirji.com

About Walmart

About Walmart

- Global retail scale: diverse sources, strict SLAs/compliance, mixed batch & streaming workloads

About Walmart

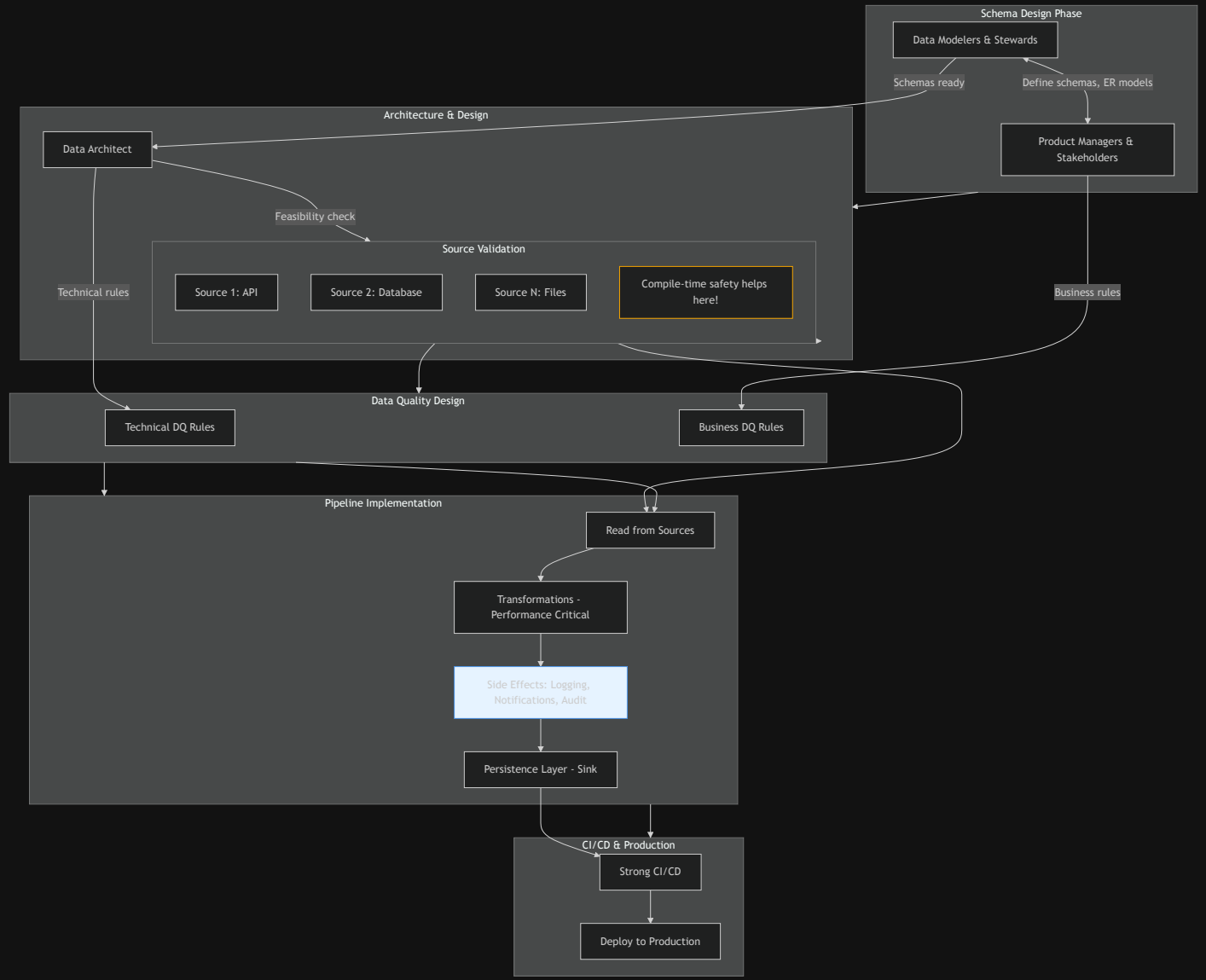
- Global retail scale: diverse sources, strict SLAs/compliance, mixed batch & streaming workloads
- Practical constraints: schema evolution under change, idempotency at edges, reproducible rollbacks

About Walmart

- Global retail scale: diverse sources, strict SLAs/compliance, mixed batch & streaming workloads
- Practical constraints: schema evolution under change, idempotency at edges, reproducible rollbacks
- These constraints motivated compile-time guarantees and fiber-safe execution

Data engineering reality

(Getting Up to Speed)



Where compile-time contracts & fiber-safe execution fit

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources
- **Quality:** Both technical & business DQ rules validated at compile + runtime

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources
- **Quality:** Both technical & business DQ rules validated at compile + runtime
- **Transform:** Pure functions for performance

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources
- **Quality:** Both technical & business DQ rules validated at compile + runtime
- **Transform:** Pure functions for performance
- **Effects:** Side effects (logging, notifications, audit) kept at edges—fiber-safe

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources
- **Quality:** Both technical & business DQ rules validated at compile + runtime
- **Transform:** Pure functions for performance
- **Effects:** Side effects (logging, notifications, audit) kept at edges—fiber-safe
- **Sink:** Strong persistence guarantees with idempotent retries

Where compile-time contracts & fiber-safe execution fit

- **Sources:** Compile-time prevents schema drift across many varied sources
- **Quality:** Both technical & business DQ rules validated at compile + runtime
- **Transform:** Pure functions for performance
- **Effects:** Side effects (logging, notifications, audit) kept at edges—fiber-safe
- **Sink:** Strong persistence guarantees with idempotent retries
- **CI/CD:** Compile-fail tests gate deployments

Scars, beliefs & stakes

Previous year, an upstream team renamed `amount` → `amt` in a microservice we consume & no one told us.

> **If the build had failed on that schema diff, none of that would have happened."**

Scars, beliefs & stakes

Previous year, an upstream team renamed `amount` → `amt` in a microservice we consume & no one told us.

> **If the build had failed on that schema diff, none of that would have happened."**

Promise: We will talk about to know how to move drift, effect leaks, and rollback risk to compile/build time-so you never present that postmortem.

Boundaries we must name

Critical distinctions:

1. **Compile-time vs Runtime**
   - Compile-time: structural contracts + builder typestate + illegal construction
   - Runtime: corrupt files, SLA breaches, retries, lineage, data quality

When to use which:

Problem Type	Compile-Time Solution	Runtime Solution
Schema drift within your repo	✅ Macro evidence	❌ Too late
External API changed schema	❌ Can't control their build	✅ Schema registry + alerts
Corrupt file (missing rows)	❌ Not a structural issue	✅ DQ checks
Forgot to wire a sink	✅ Type-indexing/Typestate builder	N/A

Boundaries we must name

Critical distinctions:

1. **Compile-time vs Runtime**
   - Compile-time: structural contracts + builder typestate + illegal construction
   - Runtime: corrupt files, SLA breaches, retries, lineage, data quality

When to use which:

Problem Type	Compile-Time Solution	Runtime Solution
Schema drift within your repo	✅ Macro evidence	❌ Too late
External API changed schema	❌ Can't control their build	✅ Schema registry + alerts
Corrupt file (missing rows)	❌ Not a structural issue	✅ DQ checks
Forgot to wire a sink	✅ Type-indexing/Typestate builder	N/A

"Compiler buys sleep; runtime guardrails deal with the world's messiness."

DX vs Process

- DX: fast local red→green loops
- Process: CI policy gates + compile-fail PR checks
- DX keeps individuals fast.
- Process keeps teams safe.
- Compile-fail policies bridge the two.

Boundary checkpoint

- **Compile-time** stops drift and illegal construction
- **Runtime** handles corrupt files, SLA breaches, retries, lineage
- **DX** = fast local red→green | **Process** = CI policy gates + compile-fail PR checks

Compile-time evidence pipeline

Compile-Time evidence flow:

Key insight

- Developer writes types →
- TypedSource/Sink require Evidence →
- SchemaConforms typeclass →
- Derivation compares schemas → match? Emit evidence. Diff? Abort with precise error.

Why data contracts come first

Policy Comparison:

Policy	Missing Fields	Extra Fields	Type Mismatch	Field Order	Name Case
Exact	❌ reject	❌ reject	❌ reject	As configured	As is
Backward	⚠ allow if optional/defaulted	✅ allow	❌ reject	Flexible	As is
Forward	✅ allow (consumer tolerates missing)	❌ reject	❌ reject	Flexible	As is
Full	✅ allow	✅ allow	✅ allow	Flexible	Flexible

Modifiers: Ordered, CI (Case-Insensitive), By-Position

Policy lattice

```scala
sealed trait SchemaPolicy
object SchemaPolicy {
  // Schemas must match exactly
  sealed trait Exact extends SchemaPolicy

  // Producer adds optional fields
  sealed trait Backward extends SchemaPolicy

  // Consumer ignores extras
  sealed trait Forward extends SchemaPolicy
}
```

Emphasize: Policies encode intent; compile proof is the go/no-go button.

```scala
// Contract expects: id: Long, email: String
case class Contract(id: Long, email: String)
// Producer missing email
case class Out(id: Long)

// Strict policy → compile-time error (Missing attributes: email:String) ❌
implicitly[Conforms[Out, Contract, Strict]]

// Relax policy for migration → compiles ✅
implicitly[Conforms[Out, Contract, Backward]]
```

What you write (user-facing API):

```
summon[SchemaConforms[Out, Contract, SchemaPolicy.Exact]]
```

What you write (user-facing API):

```
summon[SchemaConforms[Out, Contract, SchemaPolicy.Exact]]
```

What you write (user-facing API):

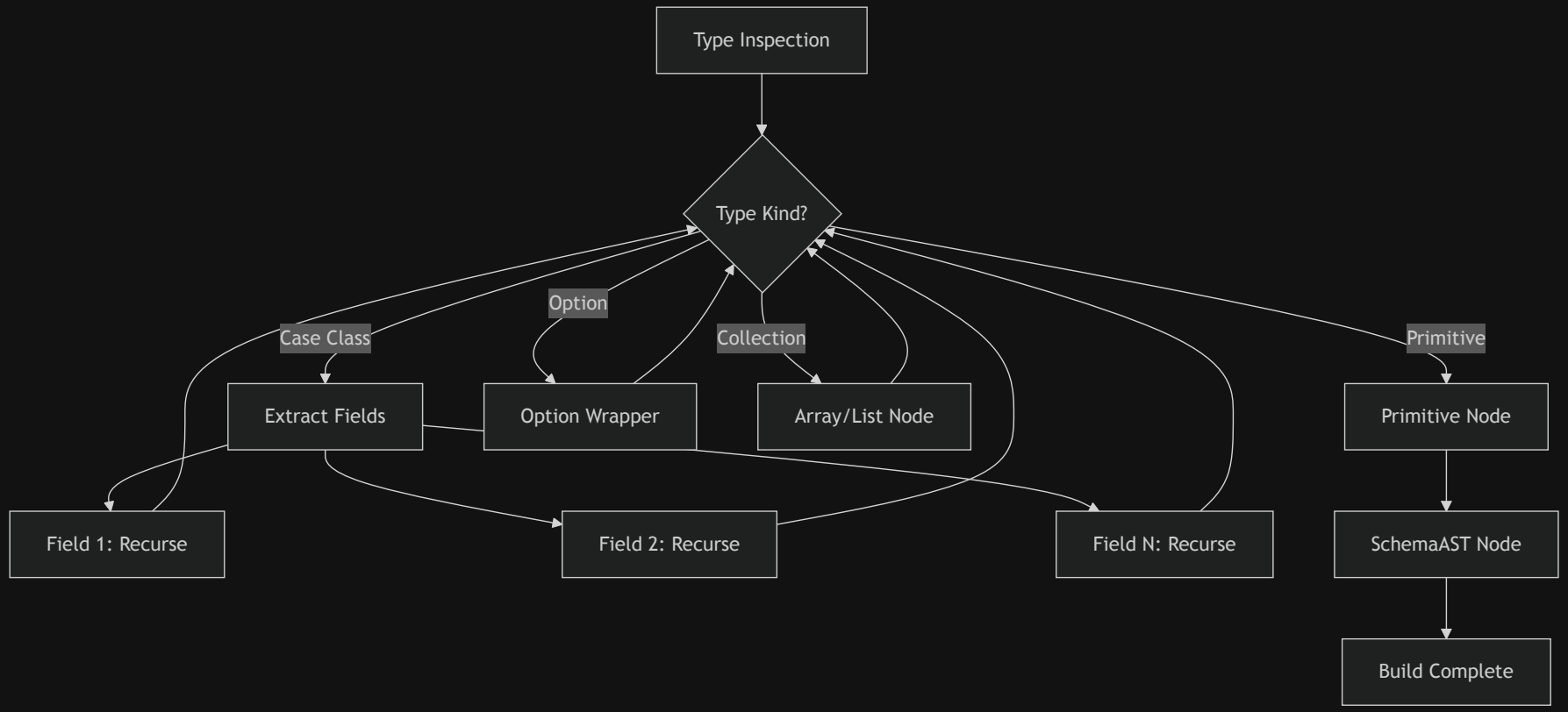
```
summon[SchemaConforms[Out, Contract, SchemaPolicy.Exact]]
```

Example compile error:

```
[error] Compile-time contract drift (policy: SchemaPolicy.Exact)
[error] Out: CustomerProducer vs Contract: CustomerContract
[error] Extra: segment
[error] Missing:
[error] Mismatched:
```

Key insight: compiler errors pinpoint the exact field/path that drifted-no grepping logs at 2 AM.

Macro TypeRepr recursion:



Red → Green migration

"This is how we stay away from War rooms early."

Setup - Define types with drift:

```scala
// Contract expects: id, email
case class Contract(id: Long, email: String)

// Producer has EXTRA field: age
case class Producer(
  id: Long,
  email: String,
  age: Int
)
```

Red → Green migration

"This is how we stay away from War rooms early."

Setup - Define types with drift:

```scala
// Contract expects: id, email
case class Contract(id: Long, email: String)

// Producer has EXTRA field: age
case class Producer(
  id: Long,
  email: String,
  age: Int
)
```

❌ RED: Exact policy REFUSES the drift

```scala
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Exact]]

// Compiler error:
// [error] Compile-time contract drift (policy: SchemaPolicy.Exact)
// [error] Out: Producer vs Contract: Contract
// [error] Extra: age: Int
// [error] Missing: (none)
```

Compiler refuses to proceed!

Red → Green migration

"This is how we stay away from War rooms early."

Setup - Define types with drift:

```scala
// Contract expects: id, email
case class Contract(id: Long, email: String)

// Producer has EXTRA field: age
case class Producer(
  id: Long,
  email: String,
  age: Int
)
```

❌ RED: Exact policy REFUSES the drift

```scala
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Exact]]

// Compiler error:
// [error] Compile-time contract drift (policy: SchemaPolicy.Exact)
// [error] Out: Producer vs Contract: Contract
// [error] Extra: age: Int
// [error] Missing: (none)
```

Compiler refuses to proceed!

Red → Green migration

"This is how we stay away from War rooms early."

Setup - Define types with drift:

```scala
// Contract expects: id, email
case class Contract(id: Long, email: String)

// Producer has EXTRA field: age
case class Producer(
  id: Long,
  email: String,
  age: Int
)
```

❌ RED: Exact policy REFUSES the drift

```scala
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Exact]]

// Compiler error:
// [error] Compile-time contract drift (policy: SchemaPolicy.Exact)
// [error] Out: Producer vs Contract: Contract
// [error] Extra: age: Int
// [error] Missing: (none)
```

Compiler refuses to proceed!

Red → Green migration (contd.)

Red → Green migration (contd.)

✅ GREEN: Backward policy ALLOWS the migration

```scala
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Backward]]
// Compiles successfully! ✅
```

Core principle: Paralyze the API

- ShapeT → SchemaAST
- Macro compares → ASTOut vs ASTContract under Policy
- `SchemaConforms[Out, Contract, Policy]` exists or compilation abort
- Drift become compile error ❌

```scala
// Compile-time policy gate
final case class Consumer(id: Long, email: String)
final case class Producer(id: Long, email: String, age: Int)

// Exact: must match fields exactly  → should FAIL
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Exact]] // ❌ compile error

// Backward: producer may add fields, consumer tolerates missing → OK
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Backward]] // ✅ compiles
```

Red → Green migration (contd.)

✅ GREEN: Backward policy ALLOWS the migration

```scala
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Backward]]
// Compiles successfully! ✅
```

Core principle: Paralyze the API

- ShapeT → SchemaAST
- Macro compares → ASTOut vs ASTContract under Policy
- `SchemaConforms[Out, Contract, Policy]` exists or compilation abort
- Drift become compile error ❌

```scala
// Compile-time policy gate
final case class Consumer(id: Long, email: String)
final case class Producer(id: Long, email: String, age: Int)

// Exact: must match fields exactly  → should FAIL
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Exact]] // ❌ compile error

// Backward: producer may add fields, consumer tolerates missing → OK
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Backward]] // ✅ compiles
```

Red → Green migration (contd.)

✅ GREEN: Backward policy ALLOWS the migration

```
val _ = implicitly[SchemaConforms[Producer, Contract, SchemaPolicy.Backward]]
// Compiles successfully! ✅
```

Core principle: Paralyze the API

- ShapeT → SchemaAST
- Macro compares → ASTOut vs ASTContract under Policy
- `SchemaConforms[Out, Contract, Policy]` exists or compilation abort
- Drift become compile error ❌

```
// Compile-time policy gate
final case class Consumer(id: Long, email: String)
final case class Producer(id: Long, email: String, age: Int)

// Exact: must match fields exactly  → should FAIL
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Exact]] // ❌ compile error

// Backward: producer may add fields, consumer tolerates missing → OK
implicitly[SchemaConforms[Producer, Consumer, SchemaPolicy.Backward]] // ✅ compiles
```

Key takeaway: The compiler pinpoints the exact field (`age: Int`) that drifted-no log grepping at 2 AM. You choose when to relax the policy for safe migrations.

Templates & golden path + Typestate builder

Scaffold output (`sbt new flowforge.g8`):

```
val pipeline = PipelineBuilder[IO]("example")
  .addTypedSource[Producer, Contract, SchemaPolicy.Backward](source, reader)
  .addTransform[Contract](transform)
  .addTypedSink[Contract, SchemaPolicy.Exact](sink, writer)
  .build  // ✅ Only compiles when Complete
```

Runbook snippet:

```
val builder = PipelineBuilder("demo")
  .addSource(typedSource)
  .noTransform

builder.build // ❌ type error: requires Complete

val ok = builder
  .addSink(typedSink)
  .build // ✅
```

Typestate prevents incomplete pipelines:

Templates & golden path + Typestate builder

Scaffold output (`sbt new flowforge.g8`):

```
val pipeline = PipelineBuilder[IO]("example")
  .addTypedSource[Producer, Contract, SchemaPolicy.Backward](source, reader)
  .addTransform[Contract](transform)
  .addTypedSink[Contract, SchemaPolicy.Exact](sink, writer)
  .build  // ✅ Only compiles when Complete
```

Runbook snippet:

```
val builder = PipelineBuilder("demo")
  .addSource(typedSource)
  .noTransform

builder.build // ❌ type error: requires Complete

val ok = builder
  .addSink(typedSink)
  .build // ✅
```

❌ **This WON'T compile:**

```
val broken = PipelineBuilder[IO]("incomplete")
  .addTypedSource[Producer, Contract, SchemaPolicy.Backward](source, reader)
  .addTransform[Contract](transform)
  // Missing .addSink
  .build  // ❌ Compile error: required BuilderState.Complete
```

Typestate prevents incomplete pipelines:

Templates & golden path + Typestate builder

Scaffold output (`sbt new flowforge.g8`):

```
val pipeline = PipelineBuilder[IO]("example")
  .addTypedSource[Producer, Contract, SchemaPolicy.Backward](source, reader)
  .addTransform[Contract](transform)
  .addTypedSink[Contract, SchemaPolicy.Exact](sink, writer)
  .build  // ✅ Only compiles when Complete
```

Runbook snippet:

```
val builder = PipelineBuilder("demo")
  .addSource(typedSource)
  .noTransform

builder.build // ❌ type error: requires Complete

val ok = builder
  .addSink(typedSink)
  .build // ✅
```

❌ **This WON'T compile:**

```
val broken = PipelineBuilder[IO]("incomplete")
  .addTypedSource[Producer, Contract, SchemaPolicy.Backward](source, reader)
  .addTransform[Contract](transform)
  // Missing .addSink
  .build  // ❌ Compile error: required BuilderState.Complete
```

Typestate prevents incomplete pipelines:

Configuration safety

What teams feel:

- Single failure surface for config errors.
- Structured diffs pointing at bad keys.
- Compile + CLI guardrails before promotion.

```
config/application.conf
✗ storage.bucket missing
✗ retries not within 1-5
```

Configuration safety

- Typed config decoder pattern using `ValidatedNel`

What teams feel:

- Single failure surface for config errors.
- Structured diffs pointing at bad keys.
- Compile + CLI guardrails before promotion.

```
config/application.conf
✗ storage.bucket missing
✗ retries not within 1-5
```

Configuration safety

- Typed config decoder pattern using `ValidatedNel`
- Refined keys with `eu.timepit.refined`

What teams feel:

- Single failure surface for config errors.
- Structured diffs pointing at bad keys.
- Compile + CLI guardrails before promotion.

```
config/application.conf
✗ storage.bucket missing
✗ retries not within 1-5
```

Configuration safety

- Typed config decoder pattern using `ValidatedNel`
- Refined keys with `eu.timepit.refined`
- `ff-validate-schema` CLI catches Spark/Delta/Hive/Avro/Protobuf drift pre-deploy

What teams feel:

- Single failure surface for config errors.
- Structured diffs pointing at bad keys.
- Compile + CLI guardrails before promotion.

```
config/application.conf
✗ storage.bucket missing
✗ retries not within 1-5
```

Configuration safety

- Typed config decoder pattern using `ValidatedNel`
- Refined keys with `eu.timepit.refined`
- `ff-validate-schema` CLI catches Spark/Delta/Hive/Avro/Protobuf drift pre-deploy
- **Message:** Config fails fast—no more "missing S3 bucket" at runtime.

What teams feel:

- Single failure surface for config errors.
- Structured diffs pointing at bad keys.
- Compile + CLI guardrails before promotion.

```
config/application.conf
✗ storage.bucket missing
✗ retries not within 1-5
```

Functional validation & Data quality

Example signal:

```
DQ failed (customer-import):
 - email: must be valid RFC-5322
 - amount: expected ⩾ 0
 - region: missing (required)
```

- Same structure emitted to alerts & dashboards. - Compile-fail tests cover schema contracts; runtime catches dirty data.

Functional validation & Data quality

- Cats `Validated` → `ValidatedNel` / `ValidatedNes` aggregates multi-rule DQ errors

Example signal:

```
DQ failed (customer-import):
 - email: must be valid RFC-5322
 - amount: expected ⩾ 0
 - region: missing (required)
```

- Same structure emitted to alerts & dashboards. - Compile-fail tests cover schema contracts; runtime catches dirty data.

Functional validation & Data quality

- Cats `Validated` → `ValidatedNel` / `ValidatedNes` aggregates multi-rule DQ errors
- Optional Deequ adapter for statistical checks

Example signal:

```
DQ failed (customer-import):
 - email: must be valid RFC-5322
 - amount: expected ⩾ 0
 - region: missing (required)
```

- Same structure emitted to alerts & dashboards. - Compile-fail tests cover schema contracts; runtime catches dirty data.

Functional validation & Data quality

- Cats `Validated` → `ValidatedNel` / `ValidatedNes` aggregates multi-rule DQ errors
- Optional Deequ adapter for statistical checks
- Promise: meaningful error accumulation instead of single BOOLEAN

Example signal:

```
DQ failed (customer-import):
 - email: must be valid RFC-5322
 - amount: expected ⩾ 0
 - region: missing (required)
```

- Same structure emitted to alerts & dashboards. - Compile-fail tests cover schema contracts; runtime catches dirty data.

Functional validation & Data quality

- Cats `Validated` → `ValidatedNel` / `ValidatedNes` aggregates multi-rule DQ errors
- Optional Deequ adapter for statistical checks
- Promise: meaningful error accumulation instead of single BOOLEAN
- Business rules live as runtime checks or compile-fail property tests

Example signal:

```
DQ failed (customer-import):
 - email: must be valid RFC-5322
 - amount: expected ⩾ 0
 - region: missing (required)
```

- Same structure emitted to alerts & dashboards. - Compile-fail tests cover schema contracts; runtime catches dirty data.

Effect boundary

(Pure Inside, Effects at Edges)

Connection to contracts:

> "Contracts prove the schema *before* you run. Now let's talk about keeping pipelines *composable* and *testable* while you run. Both are type-level guarantees-one for structure, one for effects."

Core principle:

Effect boundary

(Pure Inside, Effects at Edges)

Connection to contracts:

> "Contracts prove the schema *before* you run. Now let's talk about keeping pipelines *composable* and *testable* while you run. Both are type-level guarantees-one for structure, one for effects."

Core principle:

- **Pure inside:** Transforms are `A ⇒ B` (map/filter)-no side effects

Effect boundary

(Pure Inside, Effects at Edges)

Connection to contracts:

> "Contracts prove the schema *before* you run. Now let's talk about keeping pipelines *composable* and *testable* while you run. Both are type-level guarantees-one for structure, one for effects."

Core principle:

- **Pure inside:** Transforms are `A ⇒ B` (map/filter)-no side effects
- If we want to touch outside world: `F[_]`

Effect boundary

(Pure Inside, Effects at Edges)

Connection to contracts:

> "Contracts prove the schema *before* you run. Now let's talk about keeping pipelines *composable* and *testable* while you run. Both are type-level guarantees-one for structure, one for effects."

Core principle:

- **Pure inside:** Transforms are `A ⇒ B` (map/filter)-no side effects
- If we want to touch outside world: `F[_]`
- If we want to transform records: `pure`

Effect boundary

(Pure Inside, Effects at Edges)

Connection to contracts:

> "Contracts prove the schema *before* you run. Now let's talk about keeping pipelines *composable* and *testable* while you run. Both are type-level guarantees-one for structure, one for effects."

Core principle:

- **Pure inside:** Transforms are `A ⇒ B` (map/filter)-no side effects
- If we want to touch outside world: `F[_]`
- If we want to transform records: `pure`
- **Effects at edges:** Read/write/stream are `A ⇒ F[B]` where `F[_]` captures IO/Task/Pending

Why it matters: Retries are safe; tests don't need real databases; logic is portable.

Pipelines composition

(Effectful Glue)

The problem: `A ⇒ F[B]` functions don't compose like regular functions.

The solution:

- Named step: give me an `A`, I will work in box `F` to give you `B`.
- `Kleisli[F, A, B]` wraps `A ⇒ F[B]` and makes it composable.

```
val read:      Kleisli[IO, Unit, In]
val transform: Kleisli[IO, In, Out]
val write:     Kleisli[IO, Out, Unit]

val pipeline = read andThen transform andThen write
```

Effect-polymorphic design: Use `F[_]: Async` so the same code runs on Cats Effect or ZIO.

Key insight: Kleisli turns `A ⇒ F[B]` into composable pipelines-write once, run on any effect system.

Pipelines composition

(Effectful Glue)

The problem: `A ⇒ F[B]` functions don't compose like regular functions.

The solution:

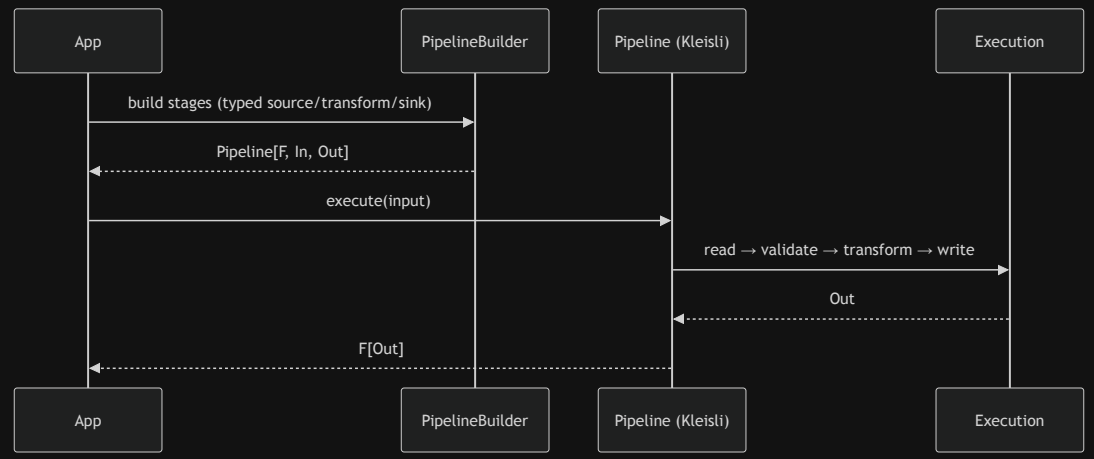
- Named step: give me an `A`, I will work in box `F` to give you `B`.
- `Kleisli[F, A, B]` wraps `A ⇒ F[B]` and makes it composable.

```
val read:      Kleisli[IO, Unit, In]
val transform: Kleisli[IO, In, Out]
val write:     Kleisli[IO, Out, Unit]

val pipeline = read andThen transform andThen write
```

Effect-polymorphic design: Use `F[_]: Async` so the same code runs on Cats Effect or ZIO.

Key insight: Kleisli turns `A ⇒ F[B]` into composable pipelines-write once, run on any effect system.



Fiber-safe execution & effect choices

Translate "fiber" for data engineers:

> "Fibers are lightweight threads. When a pipeline stage fails, you want **retries to be safe** (no duplicate writes) and **cancellation to be clean** (no leaked resources). Cats Effect and ZIO give you this via structured concurrency."

Dual effect support:

Fiber-safe execution & effect choices

Translate "fiber" for data engineers:

> "Fibers are lightweight threads. When a pipeline stage fails, you want **retries to be safe** (no duplicate writes) and **cancellation to be clean** (no leaked resources). Cats Effect and ZIO give you this via structured concurrency."

Dual effect support:

- `BYOE` : Bring your own Effect system - Cats Effect / ZIO / Kyo
- Capability algebra → interpreter per effect system
- Minimal effect interface:

Fiber-safe execution & effect choices

Translate "fiber" for data engineers:

> "Fibers are lightweight threads. When a pipeline stage fails, you want **retries to be safe** (no duplicate writes) and **cancellation to be clean** (no leaked resources). Cats Effect and ZIO give you this via structured concurrency."

Dual effect support:

- `BYOE` : Bring your own Effect system - Cats Effect / ZIO / Kyo
- Capability algebra → interpreter per effect system
- Minimal effect interface:

```
trait EffectSystem[F[_]] extends MonadError[F, Throwable] {
  trait Fiber[F[_], A] { def cancel: F[Unit]; def join: F[A] }
  def start[A](fa: F[A]): F[Fiber[F, A]]
  def parTraverse[A,B](as: List[A])(f: A ⇒ F[B]): F[List[B]]
  def bracket[A,B](acq: F[A])(use: A ⇒ F[B])(rel: A ⇒ F[Unit]): F[B]
  def timeout[A](fa: F[A], d: FiniteDuration): F[A]
  // ... async/fromFuture/map/flatMap/handleErrorWith
}
```

Fiber-aware orchestration:

- Cancellation propagates cleanly
- Resource safety (files, connections) guaranteed
- Retries don’t duplicate side effects

Engine portability seam

Wait, does this only work on Spark?

> "The algebra separates *what* you do (read/transform/write) from *how* you run it. Swap `SparkRunner` for `FlinkRunner` and the same pipeline code runs on Flink. The appendix has the proof if you're curious."

Key insight: Pipeline code stays the same; choose the interpreter at wiring time.

Code references:

- `modules/engines-spark`
- `modules/engines-flink`

Deep-dive:

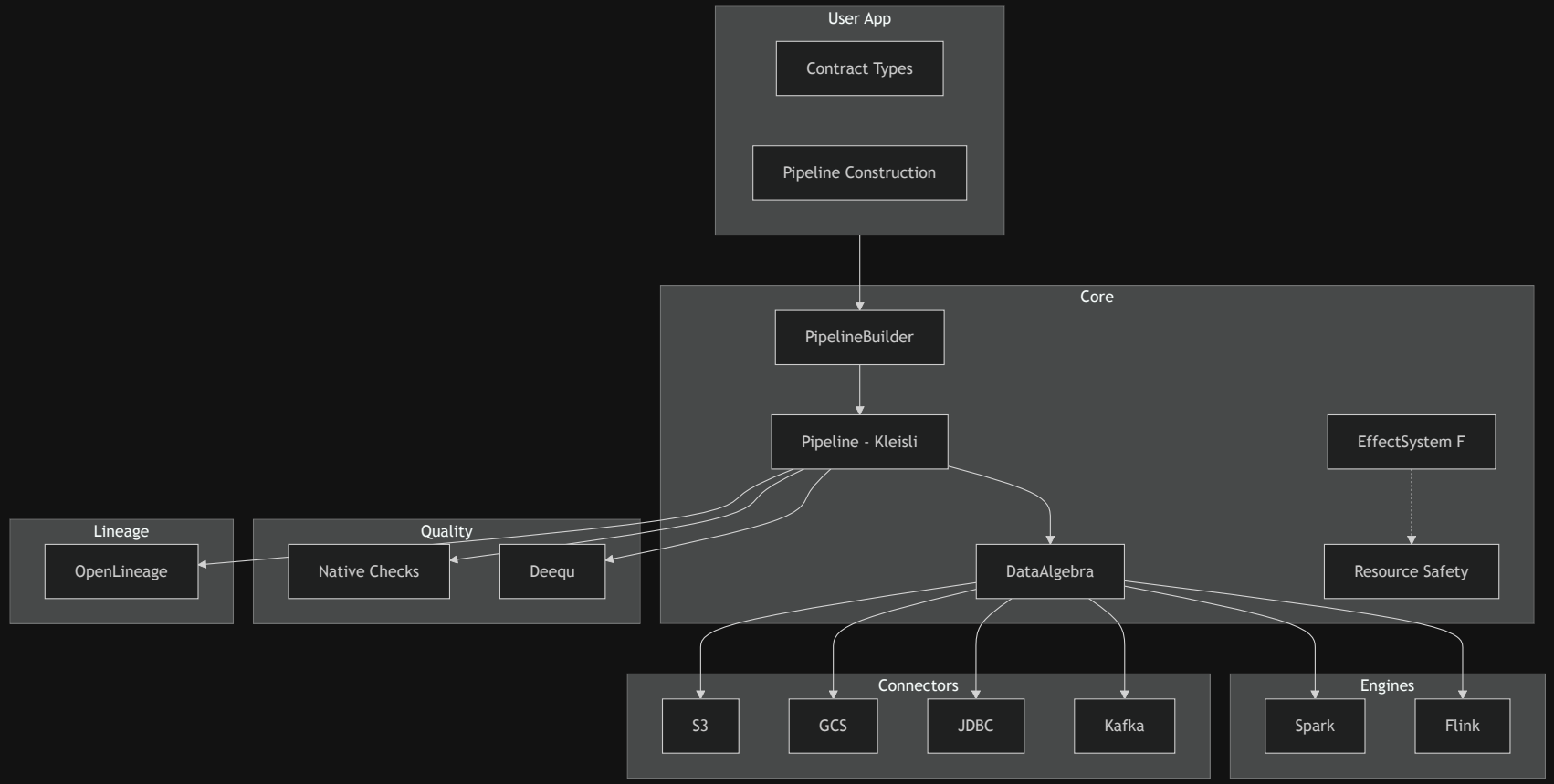
```scala
trait EngineRunner[F[_]] {
  def run[A](pipeline: Pipeline[F, Unit, A]): F[A]
}

final class SparkRunner[F[_]: Async](session: SparkSession) extends EngineRunner[F] { /* ... */ }
final class FlinkRunner[F[_]: Async](env: StreamExecutionEnvironment) extends EngineRunner[F] { /* ... */ }

// Pipeline code stays unchanged
val job: Pipeline[F, Unit, Output] = customerPipeline.build

// Choose engine at the edge
val sparkResult = new SparkRunner[F](spark).run(job)
val flinkResult = new FlinkRunner[F](env).run(job)
```

Runtime guardrails



Runtime capabilities:

- Infrastructure: resource safety, typed config, metrics, tracing
- Data Quality: native checks + optional Deequ integration
- Lineage: OpenLineage emitter (sync / async / noop)

Boundary checkpoint (Recap)

Pop quiz for the audience:

> "Does compile-time catch corrupt CSV files?"

Boundary checkpoint (Recap)

Pop quiz for the audience:

> "Does compile-time catch corrupt CSV files?"

Answer: "No-that’s runtime."

Boundary checkpoint (Recap)

Pop quiz for the audience:

> "Does compile-time catch corrupt CSV files?"

Answer: "No-that's runtime."

> "Does it catch schema drift in your repo?"

Answer: "Yes-that's compile-time."

Repeat the boundary:

- **Compile-time stops drift and illegal construction**
- **Runtime handles corrupt files, SLA breaches, retries, lineage**
- **DX = fast local red→green | Process = CI policy gates**

Outcome takeaways

Lead with feelings, then mechanisms:

Outcome takeaways

Lead with feelings, then mechanisms:

1. **Stop getting paged at 2 AM** because schema drift dies before deploy

    How: Drift fails at compile time, not in production

Outcome takeaways

Lead with feelings, then mechanisms:

1. **Stop getting paged at 2 AM** because schema drift dies before deploy

   How: Drift fails at compile time, not in production

2. **Go from hours of log-grepping to seconds of compiler errors**

   How: Compiler errors point at the exact field/path that drifted

Outcome takeaways

Lead with feelings, then mechanisms:

1. **Stop getting paged at 2 AM** because schema drift dies before deploy

   How: Drift fails at compile time, not in production

2. **Go from hours of log-grepping to seconds of compiler errors**

   How: Compiler errors point at the exact field/path that drifted

3. **Keep traders, analysts, and finance teams out of war rooms**

   How: Retries are fiber-safe and idempotent at the edges; SLAs protected

Remember WHY → HOW → WHAT?



WHY: Protect Sleep



HOW: Contracts



WHAT: Quality

- "We try to keep traders, analysts, and on-call engineers out of war rooms, early"
- "Compile gates buy confidence; runtime guardrails keep reality in check."
- "Join us if you refuse to debug schema drift at 02:00 ever again."

Remember WHY → HOW → WHAT?



WHY: Protect Sleep



HOW: Contracts



WHAT: Quality

- "We try to keep traders, analysts, and on-call engineers out of war rooms, early"
- "Compile gates buy confidence; runtime guardrails keep reality in check."
- "Join us if you refuse to debug schema drift at 02:00 ever again."

Think about your worst schema incident-this is how we stop repeating it.

Invitation (Not a Pitch)

> We built flowforge to prove these ideas work at scale. The repo is public-try `sbt new flowforge.g8` , run the red→green demo, and share your stories. **The ideas belong to you; the code is just a reference.**

Invitation (Not a Pitch)

> We built flowforge to prove these ideas work at scale. The repo is public-try `sbt new flowforge.g8` , run the red→green demo, and share your stories. **The ideas belong to you; the code is just a reference.**

Incremental adoption:

- Start with one pipeline
- Add evidence at the sink
- Run compile-fail tests in CI
- You don’t need to convert everything at once

Thank you!

Questions?

https://vitthalmirji.com

github.com/vim89

Appendix: Additional Resources

Documentation:

- Engine portability: `docs/talks/appendix-engine-portability.md`
- Migration timeline (4-week schema evolution playbook): Week 1 (Backward policy), Week 4 (tighten to Exact), Week 6 (sunset old schema)
- Blog posts: `compile-time-data-contracts` (2025-09-30), `kleisli-data-engineering` (2025-10-08)

Code References:

- Compile-time evidence: `modules/core/ ... /SchemaConformsMacro.scala`
- Typestate builder: `modules/core/ ... /PipelineBuilder.scala`
- Effect system: `modules/core/algebra/EffectSystem.scala`
- Examples: `modules/examples/ ... /CompileConformanceDemo.scala`

Experimental:

- Kyo/Caprese explorations: `docs/archive/brainstorming/flowforge/kyo-caprese.md`