df++: A Secure JVM Language and Runtime with OOP, Functional, and Declarative Ideas

(Design Review & Implementation Proposal)

Independent Research
contact@example.org

***Abstract*—This document is a proposal. df++ is a small, statically typed language that combines (i) object-oriented *types* via interfaces and composition (no inheritance), (ii) *functional* iteration over `List<T>`, `Set<T>`, and `Map<K,V>` with fusible combinators and first-class *pattern matching*, and (iii) *declarative tasks* with contracts (`pre`/`pos`) and orchestration (`run('...').then('...')`, `parallel{ '...', '...' }`). Programs compile to JVM bytecode (JDK 21) via ANTLR+ASM and always execute inside dfVM, a mandatory userspace sandbox that gates I/O by capability, supports deterministic replay, and emits syscall traces. v1 includes a lightweight static contract solver (interval + cardinality + SAT) and basic local type inference. Tooling (CLI/LSP/test/packaging) and ecosystem elements are *also proposals* in this document.**

***Index Terms*—static typing, pattern matching, local type inference, multiple interfaces, functional collections, declarative tasks, contracts, capability sandbox, deterministic replay, JVM bytecode**

I. Motivation (Proposal)

One language, three styles. Many systems need OOP modularity, functional clarity, and declarative orchestration. df++ blends these with a deliberately small surface and predictable semantics.

Design goals.

- **Readable**: interface-based OOP (composition, no inheritance), uniform collections, and explicit tasks.
- **Safe by default**: capability-gated I/O, runtime contracts, reproducible runs.
- **Portable**: JVM bytecode, Java interop that feels native.
- **Auditable**: syscall traces + contracts explain failures and performance.
- **Implementable now**: ANTLR+ASM compiler and a practical sandbox (dfVM) without kernel tricks.

Why dfVM is mandatory. The sandbox delivers least privilege, determinism (virtual time/RNG; HTTP fixtures), resource quotas, and an effect ledger. It replaces many container use-cases with faster startup and lower operational weight while keeping JVM portability.

II. Language Model (v1, Proposal)

A. Surface

Eight keywords: `module`, `import`, `type`, `fn`, `task`, `const`, `let`, `mut`. Identifiers can be normal or back-ticked (e.g., `` `foo bar` ``); task names use single-quoted strings (e.g., `'fetch posts'`). Expression-oriented with ternary `c ? a : b`. No surface `for`/`while` loops; use functional iteration.

B. Types and Interfaces

Records: `{ field = expr, ... }`. **ADTs**: `type Option<T> = None | Some(T)`, `type Result<T,E> = Ok(T) | Err(E)` and user-defined sums. **Generics**: first-order (`List<T>`, `Set<T>`, `Map<K,V>`, `Result<T,E>`). **Interfaces**: named method sets; values can implement multiple interfaces. Favor composition; no inheritance.

C. Collections & Pipelines

List<T>: `map`, `flatMap`, `filter`, `fold`, `reduce`, `scan`, `zip`, `take`, `drop`, `groupBy`, `sortBy`, `forEach`, `size`, `isEmpty`, `nonEmpty`; `List.range(start,endExcl)`. **Set<T>**: `union`, `intersect`, `diff`, `contains`, `size`, `isEmpty`, `map`, `filter`, `toList`. **Map<K,V>**: `get`, `put`, `remove`, `containsKey`, `keys`, `values`, `entries`, `mapValues`, `filterKeys`, `size`, `isEmpty`. Pipelines over these are fusible where semantics allow.

D. Pattern Matching (v1)

A *match-expression* is first-class in v1:

```
::df++

type Option<T> = None | Some(T)

fn greet(opt: Option<String>): String =
  match (opt) {
    Some(s) -> "Hello, " + s
    None    -> "Hello"
  }
```

Patterns in v1 include ADT constructors with positional binders, literals (`true`/`false`/`null`, numbers, strings), record key subsets (`{k=v,...}` with fresh binders), and wildcard `_`. Matching is exhaustive-checked when the domain is finite and known (e.g., `Option`, small enums); otherwise a warning is issued and a runtime check inserted.

E. Tasks and Orchestration

A task is `task 'name' { pre P; act { ... }; pos Q }` with pure Boolean `pre`/`pos` and effectful `act`. Orchestration uses:

- `run('A').then('B')` to sequence.
- `parallel { 'A', 'B', 'C' }` to fan out on virtual threads and join.

Testing keyword (tooling proposal): `test 'name' { arrange; act; assert }` recognized by tools; ignored by codegen.

F. Contracts, Invariants, and Static Solver (v1)

Contracts run at runtime and feed the trace. A static pass tries to discharge them ahead of time.

Contract language. Propositional formulas over:

- numeric constraints: `a <b`, `a` $\leq b, linear forms$
- `collection cardinalities: size(xs), isEmpty(xs), subset relations where statically known,`
- `equality/disequality over literals and finite ADTs.`

Static solver.

- **Front-end**: normalize to negation-normal form; constant fold; extract known sizes from literals and closed operators (`map, filter, take, drop, union, intersect, diff`).
- **Intervals**: compute interval abstractions for Int/Float; propagate through linear ops; detect contradictions.
- **Cardinality**: track `|List|, |Set|, |Map|` with flow-sensitive bounds.
- **SAT core**: CNF-encode residual propositional structure; attempt DPLL; learn simple clauses from interval/cardinality conflicts.
- **Results**: *Proved* (runtime check elided), *Refuted* (counterexample provided, optional compile error), *Unknown* (runtime check retained).

Roadmap: SMT backend (Z3) for quantified finite checks; LTL monitor synthesis for small temporal specs.

G. Type Inference

Local inference:

- `let x = expr` infers `x` from `expr`.
- Lambda parameters may omit types when context suffices.
- Top-level `fn` signatures recommended (required for exports).

H. Java Interop (Simplified, Proposal)

Direct, typed imports of JVM classes; usage looks like normal method calls:

```
::df++

module demo.jvm
import java: com.example.Http as Http        //
    proposal: direct JVM import
import std.io as IO

fn fetch(url: String): String = {
  // Static call; dfVM must grant 'net' capability and
     whitelist class.
  let bytes: Bytes = Http.get(url)           //
     signature read from JVM
  bytes.utf8()
}
```

Binding.

- The compiler queries classfiles on the classpath, maps JVM types to df++ (`int->Int, byte[]->Bytes, String->String`, generics erased with optional annotations), and generates call sites.
- dfVM validates manifests: each imported class/method must appear in `dfpkg.json` under `jvmAllowlist`.
- Reflection is denied; linkage is static; conversions are explicit and well-defined.

III. Compiler and Runtime (Proposal)

A. ANTLR → ASM Pipeline

Parsing. ANTLR4 builds an AST with source spans (Java 21 records).

Typing. Nominal types (records/ADTs), first-order generics, interface conformance, exhaustiveness hints for `match`, local inference, flow non-null.

Lowering. Expression IR; closure conversion; pipeline fusion; tail-call to loops; `match` desugars to decision DAG with guards and binds.

Bytecode emission (ASM).

- One public module class: `pkg/Module`, with static fields for globals.
- Each `fn` → `public static Object f(Object[])` (+ a typed wrapper when exporting to Java).
- Each task `t`: `t$pre():Z`, `t$act():Object`, `t$pos():Z`.
- `COMPUTE_FRAMES` and line/local tables for debugging.

Walkthrough (example).

```
::df++

// Source
fn add(a: Int, b: Int): Int = a + b
```

Lowering sketch:

1) AST: `Fn(name=add, params=[a:Int,b:Int], ret=Int, body=Plus(Var a, Var b))`
2) IR: `Load 0; Load 1; IAdd`
3) ASM:

```
/* Pseudo-ASM (Java) */
mv = cw.visitMethod(ACC_PUBLIC|ACC_STATIC, "add$O",
  "([Ljava/lang/Object;)Ljava/lang/Object;", null,
    null);
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);            // args
mv.visitInsn(ICONST_0);
mv.visitInsn(AALOAD);                 // args[0]
mv.visitTypeInsn(CHECKCAST, "java/lang/Integer");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "
    intValue", "()I", false);
mv.visitVarInsn(ALOAD, 0);            // args
mv.visitInsn(ICONST_1);
mv.visitInsn(AALOAD);                 // args[1]
mv.visitTypeInsn(CHECKCAST, "java/lang/Integer");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "
    intValue", "()I", false);
mv.visitInsn(IADD);                   // a + b
mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "
    valueOf", "(I)Ljava/lang/Integer;", false);
mv.visitInsn(ARETURN);
mv.visitMaxs(0,0);                    // COMPUTE_FRAMES
mv.visitEnd();
```

Match lowering. For

```
match (opt) { Some(x) -> f(x); None -> g() }
```

the compiler emits a typed tag-switch (synthetic ordinal per variant) plus casts/binds.

B. Mandatory dfVM (Implementation Details)

Process model. A dedicated JVM process per run; module graph stripped to the minimum; custom ClassLoader; strict classpath.

Capability enforcement.

- *Gates*: env, fs (virtual root/overlay, path policies), net (host/port ACL + HTTP fixture layer), time (virtual clock), rng (seeded PRNG).
- *Mediation*: stdlib calls (env/net/fs/time/rng/conc) go through dfVM services; Java interop calls are checked against the `jvmAllowlist` and capability guards (e.g., `Http.get` requires `net`).
- *Deny*: reflection, `sun.misc.Unsafe`, raw sockets, `ProcessBuilder`, arbitrary file IO outside virtual root.

Determinism & replay.

- Virtual *clock* and *rng* (seeded).
- HTTP fixtures: record bodies, headers, status; keyed by method+URL+headers+body; replay exact responses.

Resource quotas. CPU wallclock, heap soft cap, egress bytes, open FDs, concurrent tasks. Policy: fail-fast with trace.

Tracing. Every effect creates a span: name, start/stop, attributes (sizes, durations), contract verdicts, and optional previews (first N bytes of bodies).

Packaging. *dfpkg* (zip): bytecode jar, `dfpkg.json` manifest (entrypoint, capabilities, `jvmAllowlist`, hashes), optional fixtures, signature file. Loader verifies signature, hashes, and policy.

IV. Conclusion (Proposal)

df++ aims to keep the surface small and semantics clear: interfaces over inheritance, functional collections over loops, explicit tasks with contracts, first-class pattern matching, and a mandatory sandbox. The ANTLR+ASM stack is straightforward; dfVM makes runs reproducible, auditable, and safe. The static solver catches simple issues early, and Java interop remains powerful while still capability-checked.

Appendix A: Mathematical Grammar (Abstract Syntax)

We write a minimal abstract grammar (omitting obvious lists):

$$
\begin{array}{lll}
M & ::= & \texttt{module}\ p\ ;\ I^*\ ;\ D^* \\
I & ::= & \texttt{import}\ q\ \texttt{as}\ x \\
D & ::= & \texttt{type}\ T\ =\ \Sigma \mid \texttt{fn}\ f(\overline{x:T}):T\ =\ e \mid \texttt{task}\ s\ \{C\} \mid \texttt{const}\ x:T=e \mid \texttt{let|mut}\ x:T=e \\
\Sigma & ::= & K(\overline{T}) \mid \Sigma \mid \ \ldots \\
e & ::= & c \mid x \mid e\ \mathrm{op}\ e \mid \lambda x{:}T.\ e \mid e(e) \mid \{\overline{x=e}\} \mid e.x \mid \mathrm{match}(e)\{\overline{\pi \to e}\} \\
\pi & ::= & K(\overline{x}) \mid \{\overline{x=y?}\} \mid c \mid \_ \\
C & ::= & \texttt{pre}\ \phi;\ \texttt{act}\ \{S\}\ ;\ \texttt{pos}\ \psi \\
S & ::= & \overline{\mathrm{stmt}};\ e? \\
\phi,\psi & ::= & \text{propositional formulas with } \texttt{size},\ \leq,\ =,\ \ldots
\end{array}
$$

Values and stores follow standard call-by-value with immutable `const`, single-assignment `let`, and mutable `mut`.

APPENDIX B: FORMAL SEMANTICS (TYPING & DYNAMICS)

Typing. We write $\Gamma \vdash e : T$.

$$\frac{}{\Gamma \vdash n : \mathrm{Int}} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e_1 : \mathrm{Int} \quad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 + e_2 : \mathrm{Int}}$$

Records: $\dfrac{\forall i.\ \Gamma \vdash e_i : T_i}{\Gamma \vdash \{x_i = e_i\}_i : \{x_i : T_i\}_i}$, field: $\dfrac{\Gamma \vdash e : \{x : T, \ldots\}}{\Gamma \vdash e.x : T}$.

Sum intro: $\dfrac{\Gamma \vdash e : T}{\Gamma \vdash K(e) : K(T)}$. Match:

$$\frac{\Gamma \vdash e : \Sigma \quad \forall i.\ (\Gamma, \overline{x_i : T_i}) \vdash e_i : U \quad \mathrm{exh}(\{\pi_i\}, \Sigma)}{\Gamma \vdash \mathrm{match}(e)\{\pi_i \to e_i\}_i : U}$$

where *exh* checks exhaustiveness (exact for finite Σ, warning otherwise).

Tasks. A task τ evaluates as:

$$\frac{\Gamma \vdash \phi \Rightarrow \mathrm{true}}{\tau.\mathrm{pre} \Downarrow \mathrm{true}} \qquad \frac{\tau.\mathrm{pre} \Downarrow \mathrm{true} \quad \{S\} \Downarrow v \quad \tau.\mathrm{pos} \Downarrow \mathrm{true}}{\tau \Downarrow \mathrm{Ok}(v)}$$

Otherwise the result is $\mathrm{Err}(\mathrm{ContractFail})$. Orchestration is sequencing of these judgments; `parallel` evaluates in a product machine over Loom VTs and joins.

Contracts. Static solver reduces ϕ, ψ to $\{\mathrm{Proved}, \mathrm{Refuted}, \mathrm{Unknown}\}$; runtime always re-checks but may elide in *proved* mode.

Appendix C: ANTLR Grammar (Updated, v1 with match)

```
lexer grammar DfppLexer;

// keywords
MODULE:'module'; IMPORT:'import'; AS:'as';
TYPE:'type'; FN:'fn'; TASK:'task';
PRE:'pre'; ACT:'act'; POS:'pos';
RUN:'run'; PARALLEL:'parallel';
MATCH:'match';
CONST:'const'; LET:'let'; MUT:'mut';
TRUE:'true'; FALSE:'false'; NULL:'null';

// literals / identifiers
INTLIT:[0-9]+; DECLIT:[0-9]+'.'[0-9]+;
STRING: '"' ( ~["\\] | '\\' . )* '"'
      | '\'' ( ~['\\] | '\\' . )* '\'' ;
ID:[a-zA-Z_][a-zA-Z_0-9]*;
BTICKID:'`' ( ~[`\\] | '\\' . )+ '`';

// symbols
LP:'('; RP:')'; LB:'{'; RB:'}'; LSB:'['; RSB:']';
COMMA:','; DOT:'.'; COLON:':'; EQ:'='; QMARK:'?';
    ARROW:'->';
PLUS:'+'; MINUS:'-'; STAR:'*'; SLASH:'/'; PERC:'%';
AND:'&&'; OR:'||'; NOT:'!'; EE:'=='; NE:'!='; LT:'<';
    LE:'<='; GT:'>'; GE:'>=';

// trivia
WS:[ \t\r\n]+ -> skip;
LINE_COMMENT:'//' ~[\r\n]* -> skip;
BLOCK_COMMENT:'/*' .*? '*/' -> skip;
```

```
parser grammar DfppParser;
options { tokenVocab = DfppLexer; }

program   : moduleDecl? importDecl* topDecl* EOF ;
moduleDecl: MODULE qid ;
importDecl: IMPORT (qid | JAVA qid) AS ID ; //
    proposal: 'import java: ... as ...'
qid       : ident (DOT ident)* ;
ident     : ID | BTICKID ;

topDecl   : constDecl | letDecl | typeDecl | fnDecl |
    taskDecl /* | testDecl (tooling) */ ;

constDecl : CONST ident (COLON typeRef)? EQ expr ;
letDecl   : (LET|MUT) ident (COLON typeRef)? EQ expr ;

typeDecl  : TYPE ident EQ sumType ;
sumType   : variant ('|' variant)* ;
variant   : ident (LP typeRef (COMMA typeRef)* RP)? ;

// tasks use STRING names to allow spaces
taskDecl  : TASK STRING LB (PRE expr)? ACT block (POS
    expr)? RB ;

// blocks and statements
block     : LB stmt* expr? RB ;
stmt      : constDecl | letDecl | runStmt |
    parallelStmt | expr ;

runStmt   : runExpr ;
parallelStmt : PARALLEL LB stringList RB ;
stringList: STRING (COMMA STRING)* ;
```

```
// expressions
expr
 : expr OR expr
 | expr AND expr
 | expr EE expr | expr NE expr
 | expr LT expr | expr LE expr | expr GT expr | expr GE
    expr
 | expr PLUS expr | expr MINUS expr | expr STAR expr |
    expr SLASH expr | expr PERC expr
 | NOT expr
 | primary QMARK expr COLON expr
 | primary
 ;

primary
 : literal
 | ident
 | call
 | memberChain
 | LP expr RP
 | recordLit
 | arrayLit
 | matchExpr
 ;

literal   : INTLIT | DECLIT | STRING | TRUE | FALSE |
    NULL ;
call      : ident LP args? RP ;
args      : expr (COMMA expr)* ;

// run('A').then('B') as a chain
runExpr   : memberChain ;
memberChain
 : runCall (DOT ident LP args? RP )*
 ;
runCall   : RUN LP STRING RP ;

// pattern matching
matchExpr : MATCH LP expr RP LB matchArm+ RB ;
matchArm  : pattern ARROW expr ;
pattern   : ident (LP (ident (COMMA ident)*)? RP)?   //
    K(x,y) or ctor()
          | LB patField (COMMA patField)* RB         //
    {k = x, ...}
          | literal
          | '_' ;
patField  : ident EQ ident ;

recordLit : LB recField (COMMA recField)* RB ;
recField  : ident EQ expr ;
arrayLit  : LSB (expr (COMMA expr)*)? RSB ;

typeRef   : qid | LB typeField (COMMA typeField)* RB |
    typeRef ARROW typeRef ;
typeField : ident COLON typeRef ;
```

APPENDIX D: STANDARD LIBRARY INTERFACES (V1, FULL TYPES)

Implementations live behind dfVM; these are the signatures.

std.core

- `type Option<T> = None | Some(T)`
- `type Result<T,E> = Ok(T) | Err(E)`
- `trait Future<T> { map<U>((T)->U): Future<U>; then<U>((T)->Future<U>): Future<U>; await(): T }`
- `List<T>`:
  - `map<U>((T)->U): List<U>, flatMap<U>((T)->List<U>): List<U>, filter((T)->Bool): List<T>`
  - `fold<A>(A, (A,T)->A): A, reduce((T,T)->T): T, scan<A>(A,(A,T)->A): List<A>`
  - `zip<U>(List<U>): List<(T,U)>, take(Int): List<T>, drop(Int): List<T>`
  - `groupBy<K>((T)->K): Map<K,List<T», sortBy<K:Comparable>((T)->K): List<T>`
  - `forEach((T)->Unit): Unit, size(): Int, isEmpty(): Bool, nonEmpty(): Bool`
  - `static range(Int, Int): List<Int>`
- `Set<T>`:
  - `union(Set<T>): Set<T>, intersect(Set<T>): Set<T>, diff(Set<T>): Set<T>`
  - `contains(T): Bool, size(): Int, isEmpty(): Bool`
  - `map<U>((T)->U): Set<U>, filter((T)->Bool): Set<T>, toList(): List<T>`
- `Map<K,V>`:
  - `get(K): Option<V>, put(K,V): Map<K,V>, remove(K): Map<K,V>`
  - `containsKey(K): Bool, keys(): Set<K>, values(): List<V>, entries(): List<(K,V)>`
  - `mapValues<U>((V)->U): Map<K,U>, filterKeys((K)->Bool): Map<K,V>`
  - `size(): Int, isEmpty(): Bool`
- `Option.match<U>(some:(T)->U, none:()->U): U`
- `Result.match<U>(ok:(T)->U, err:(E)->U): U`
- `getOrElse(T): T` on `Option` and `Result<T,E>`
- Base: all base types expose `str(): String`

std.json

- `type Json = ...` (opaque)
- `parse(String): Result<Json,String`
- `stringify(Json): String`
- `encode<T>(T): Json` (derived for records/ADTs)
- `decode<T>(Json): Result<T,String`

std.io — `print(String): Unit`

std.sys.env/time/rng/net/conc (via dfVM)

- env: `env(String): Option<String>`
- time: `nowMs(): Long`, `sleep(Long): Unit`
- rng: `randBytes(Int): Bytes`
- net:
  - `type HttpResp = { status: Int, headers: Map<String,String>, body: Bytes }`
  - `httpGet(String, Map<String,String>): HttpResp`
  - `httpPost(String, Map<String,String>, Bytes): HttpResp`
- conc: `spawn(()->Unit): Future<Unit>`

std.flow.orch

- `type TaskChain = { then(String): TaskChain, await(): Result<Unit,String> }`
- `run(String): TaskChain`
- `parallel(List<String>): Result<Unit,String>`

std.contracts — `assert(Bool): Unit`; pipelines accept `.inv(Bool)` (debug-mode check).

std.jvm (only if the direct Java import shortcut is not used)

- `callStatic(class:String, method:String, paramSig:String, args:List<Any>): Any`
- `new(class:String, paramSig:String, args:List<Any>): Any`

Appendix E: Worked Examples

Listing 1. ::df++ — Pattern matching with lists and options

```
::df++

module demo.match

type Option<T> = None | Some(T)

fn head<T>(xs: List<T>): Option<T> =
  match (xs.size()) {
    0 -> None
    _ -> Some(xs.take(1).head)          // assume head on non-empty
  }

fn greet(opt: Option<String>): String =
  match (opt) {
    Some(s) -> "Hello, " + s
    None    -> "Hello"
  }
```

Listing 2. ::df++ — Tasks, sequencing, parallelism

```
::df++

module demo.tasks
import std.json as Json
import std.sys.net as Net
import std.io as IO

type Post = { id: Int, title: String }

mut posts: List<Post> = []
mut titles: List<String> = []

task 'fetch posts' {
  act {
    let r = Net.httpGet("https://example.org/posts", {})
    posts = Json.parse(r.body.utf8())
              .then(j -> Json.decode<List<Post>>(j))
              .getOrElse([])
  }
  pos posts.size() >= 0
}

task 'process titles' {
  pre posts.nonEmpty()
  act {
    titles = posts.filter(p -> p.title.toLower().contains("ai"))
                  .map(p -> p.title)
  }
  pos titles.size() <= posts.size()
}

fn main(): Unit = {
  run('fetch posts').then('process titles')
  parallel { 'fetch posts', 'process titles' }
  IO.print("titles: " + titles.size().str())
}
```

APPENDIX F: TOOLING (PROPOSAL: CLI, LSP, TEST, PACKAGING)

CLI. `dfpp build` → signed `.dfpkg`; `dfpp run`; `dfpp test` (discovers `test '...'` blocks); `dfpp trace`; `dfpp replay`.

LSP. Hover types, go-to-def, rename, contract diagnostics (static solver verdict), quick-fixes for missing types/capabilities.

Debugger. Step tasks, inspect contract verdicts, see dfVM caps, tail spans.

dfpkg format. Zip with: bytecode JAR, `dfpkg.json` (entrypoint, capabilities, `jvmAllowlist`, hashes), fixtures/, `bundle.sig`. Loader verifies signature + policy.

Tests. `test '...'` blocks run under dfVM with deterministic time/RNG; JUnit XML output for CI.

APPENDIX G: CASE STUDY — FLOWTOMIC DEPLOYMENT (EXPANDED)

Mapping nodes to df++.

- Each node → `task 'node name'`; validation → `pre`; success criteria → `pos`.
- Edges → `run('A').then('B')` or `parallel{ 'A','B' }`.
- Shared transforms → pure `fns`; data reshaping uses `List/Set/Map`.

External code (TS/Python). For v1, Flowtomic "code actions" run in sidecar services (Node/Python). df++ tasks call them via `std.sys.net.http*` (record/replay supported), keeping dfVM deterministic and safe. (A GraalVM plugin is future work.)

dfpkg manifest (example).

```
{
  "entry": "example.flow.Main.main",
  "capabilities": {
    "net": { "hosts": ["api.example.org:443"] },
    "env": ["API_KEY"],
    "time": { "deterministic": true },
    "rng": { "seed": 42 },
    "fs": { "root": "/work", "writable": ["/work/out"] }
  },
  "jvmAllowlist": [
    "
     com.vendor.sdk.Client#get(Ljava/lang/String;)Ljava/lang/String;"
  ],
  "fixtures": "fixtures/",
  "hashes": { "jar": "sha256:...", "fixtures": "
     sha256:..." },
  "signature": "ed25519:..."
}
```

Operational flow.

1) Author designs graph, exports to df++.
2) `dfpp build` produces a signed `.dfpkg`.
3) Runtime loads under dfVM, which enforces capabilities, traces effects, and optionally records fixtures.
4) CI uses `dfpp replay` to reproduce runs bit-for-bit (virtual time/RNG, HTTP fixtures).

Why this pairing works. Flowtomic's graph abstractions align directly with tasks+orchestration; data wrangling uses functional collections; dfVM provides operational guarantees (least-privilege, determinism, audit) crucial for deployed models.