

A Normative Specification for a MoonBit Workflow and Parallel FFI Runtime

Part I defines the engineering contract for workflow graphs, capabilities, ownership, and separation-safe execution

Zhehao Zhu
Luna-Flow
GitHub: KCN-judu

Abstract

This document specifies a MoonBit workflow and parallel FFI runtime as a contract over workflow-graph representation, capability protocols, ownership transfer, and concurrent buffer safety. The semantic core is not a redefinition of `map` and `reduce` as elementary functions, but a precise engineering account of three boundary-critical questions: how MoonBit constructs a pure workflow graph, how that graph crosses the FFI boundary as an ABI-constrained descriptor family, and how the native runtime executes task, synchronization, and compute nodes without data races or lifetime violations. Part I defines a workflow ADT, a compute sub-IR, capability classes, layout isomorphism constraints, a linear ownership state machine, and separation-safe execution obligations. Part II states realization and conformance obligations for the native scheduler layer, the native compute lane, and the JavaScript compatibility surface. Part III refines those obligations into implementation-level conformance checks for exceptional control flow, byte-level layout validation, safe operator admissibility, and fail-closed validation scenarios. The English text is authoritative; the Chinese text is a strict mirror.

Keywords : MoonBit, workflow runtime, FFI, OpenMP, pthread, separation safety, linear ownership, ABI, engineering specification

Part I: Semantic Core

Normative Scope

This specification defines the version-1 contract of a MoonBit parallel library across three layers:

- a MoonBit facade that constructs pure workflow graphs;
- a backend-specific FFI boundary that lowers workflow graphs and compute subplans into ABI-constrained payloads and descriptors;
- a native runtime split into a workflow scheduler layer and a compute lane.

Normative force is limited to:

- the algebraic shape of the MoonBit-side workflow graph;
- the exact classes of values that may cross the FFI boundary;
- the layout isomorphism obligations required for zero-copy or borrowed views;
- the linear ownership state transitions between MoonBit and C-visible heaps;
- the separation-logic obligations required for race-free chunk scheduling and safe workflow capability use; and
- the conformance obligations for faithful native and JavaScript realizations.

Scheduler heuristics, queue policies, OpenMP tuning details, and local optimization strategy are non-normative unless they affect the observable ownership, layout, or race-safety contract.

Notation and Meta-Level Conventions

We use the following notation throughout:

- H_M for the MoonBit heap and H_C for the C/host-visible heap;
- Γ for the ownership-resource context carried across the FFI boundary;
- b for a buffer identity, v for a view, p for a plan, d for a runtime descriptor;
- mapsto for partial meta-level evaluation;
- $\text{Own}_M(b)$, $\text{Borrow}_C(b)$, $\text{Own}_C(b)$ for ownership states;
- $\Gamma \rightarrow \Gamma'$ for one ownership transition step;
- $P * Q$ for separation conjunction over disjoint heap fragments;
- $\text{Interval}(i, j)$ for half-open index intervals $[i, j)$;
- $\text{Chunks}(n, m)$ for a partition of $[0, n)$ into m subintervals;
- $\text{Den}(p)$ for plan denotation;
- $t \dashv p \mapsto q$ for lowering a MoonBit plan p on target t to payload q ;
- $\text{parse}(t, q) \mapsto d$ for wrapper parsing into runtime descriptor d ;
- $\text{exec}(d) = y$ for runtime execution of descriptor d .

Semantic Workflow Calculus

The facade is a pure workflow algebra. Its constructors carry enough information for the lowering and execution layers to interpret task, synchronization, and compute work without crossing a function-value or runtime-state boundary.

Workflow-first rule. Version 1 admits `Workflow` as the only top-level execution request. The historical `MapPlan`, `ReducePlan`, `ScanPlan`, and `MapReducePlan` remain normative only as a compute sub-IR that may appear inside `ComputeNode`.

Execution entry-point rule. Version 1 admits both asynchronous and blocking execution entry points for workflow graphs. A blocking submit is defined extensionally as `submit_async` followed by `wait` and resource destruction, and its return status is the final `WorkflowResult.status`.

Workflow capability classes. Version 1 admits the following first-class capability classes:

- `OwnedBuffer`
- `SharedReadView`
- `AtomicCell`
- `Mutex`
- `Condvar`
- `RwLock`
- `Semaphore`
- `Barrier`
- `Channel`
- `OpaqueCapability(name)`

The native realization in this version is required to execute only the subset `OwnedBuffer`, `SharedReadView`, `AtomicCell`, `Mutex`, `Condvar`, `Barrier`, and `Channel`. `RwLock`, `Semaphore`, and `OpaqueCapability(name)` remain descriptor-level surface classes whose runtime execution must fail closed with an explicit unsupported status.

Workflow node classes. Version 1 admits:

- `ComputeNode(plan)`

- SpawnNode
- JoinNode
- SendNode
- RecvNode
- LockNode
- UnlockNode
- WaitNode
- SignalNode
- BarrierNode
- ReadSharedNode
- WriteSharedNode

Workflow edge classes. Version 1 admits:

- DataDependency
- ControlDependency
- OwnershipTransfer
- SynchronizationDependency

Workflow well-formedness. Write $\text{WorkflowWF}(w)$ iff:

- w contains at least one node;
- every node id is unique;
- every referenced capability id is unique and resolvable;
- every edge endpoint references an existing node;
- no edge is a self-edge;
- the dependency graph is acyclic;
- every node that requires a capability names one;
- every node/capability pairing respects the protocol class of the capability;
- every compute node carries a compute subplan admitted by the compute sub-IR;
- every capability access mode is valid for its capability class.

Barrier group rule. Barrier groups are derived from workflow structure rather than from an explicit participant count field. In version 1, a barrier group is keyed by (barrier capability id, node depth). A derived barrier group whose participant count is less than 2 is invalid and must fail closed with `BARRIER_BROKEN`.

Compute sub-IR rule. The compute sub-IR remains the constructor family `MapPlan`, `ReducePlan`, `ScanPlan`, and `MapReducePlan`. In this version, compute plans are no longer top-level requests; they are payloads of `ComputeNode(plan)` and keep their prior denotation and split-combine laws.

```
ScalarKind ::= Int
             | UInt
             | Int64
             | UInt64
             | Float
             | Double
```

Figure 1: Scalar family.

```
Schedule ::= Static
           | Dynamic
           | Guided
```

Figure 2: Schedule class.

```

RuntimeConfig ::= {
    threads : Nat ,
    chunk : Nat ,
    schedule : Schedule
}

```

Figure 3: Runtime configuration record.

```

AccessMode ::= ReadOnly
             | WriteOnly
             | ReadWrite

```

Figure 4: Boundary access mode.

```

BufferRef ::= ( BufId , ScalarKind , Len , AccessMode , LayoutWitness )

```

Figure 5: Facade-level buffer reference.

```

OutRef ::= ( BufId , ScalarKind , Len , LayoutWitness )

```

Figure 6: Facade-level output reference.

```

LawClass ::= StrictAssoc
           | DeterministicTree

```

Figure 7: Reduction-law class.

```

MonoidLaw ::= ( LawClass , OpId , IdentityId , AssocWitness )

```

Figure 8: Monoid descriptor.

```

Plan ::= MapPlan ( RuntimeConfig , OpId , BufferRef , BufferRef )
       | ReducePlan ( RuntimeConfig , MonoidLaw , BufferRef , OutRef )
       | ScanPlan ( RuntimeConfig , MonoidLaw , BufferRef , BufferRef )
       | MapReducePlan ( RuntimeConfig , OpId , MonoidLaw , BufferRef , OutRef )

```

Figure 9: Parallel plan ADT.

Version 1 forbids arbitrary MoonBit function values and arbitrary JavaScript host functions from entering the plan.

Definition (Constructor Shape Obligations). Write $\text{ShapeOK}(p)$ iff:

- for $\text{MapPlan}(\text{cfg}, \text{op}, \text{src}, \text{dst})$, $\text{Len}(\text{src}) = \text{Len}(\text{dst})$ and dst is writable;
- for $\text{ReducePlan}(\text{cfg}, \text{mu}, \text{src}, \text{out})$, $\text{Len}(\text{out}) = 1$, $\text{ScalarKind}(\text{out}) = \text{ScalarKind}(\text{src})$, and out is the unique result location of the reduction;
- for $\text{ScanPlan}(\text{cfg}, \text{mu}, \text{src}, \text{dst})$, $\text{Len}(\text{src}) = \text{Len}(\text{dst})$, $\text{ScalarKind}(\text{src}) = \text{ScalarKind}(\text{dst})$, and dst is writable;
- for $\text{MapReducePlan}(\text{cfg}, \text{op}, \text{mu}, \text{src}, \text{out})$, $\text{Len}(\text{out}) = 1$, out is writable, and the mapped element type admitted by op matches the reduction element type required by mu .

Definition (Plan Denotation). Write $\text{Den}(p)$ for the unique observable result determined by the plan constructor:

- $\text{Den}(\text{MapPlan}(\text{cfg}, \text{op}, \text{src}, \text{dst}))$ is the buffer dst whose element at index i is $\text{Apply}(\text{op}, \text{src}[i])$;
- $\text{Den}(\text{ReducePlan}(\text{cfg}, \text{mu}, \text{src}, \text{out}))$ is the scalar result in $\text{out}[0]$ obtained by folding src with the law mu ;
- $\text{Den}(\text{ScanPlan}(\text{cfg}, \text{mu}, \text{src}, \text{dst}))$ is the inclusive prefix-result buffer dst whose element at index i equals the reduction of $\text{src}[0 .. i]$ under mu ;

- $\text{Den}(\text{MapReducePlan}(\text{cfg}, \text{op}, \text{mu}, \text{src}, \text{out}))$ is $\text{Den}(\text{ReducePlan}(\text{cfg}, \text{mu}, \text{upright}(\text{"MapTmp"})(\text{op}, \text{src}), \text{out}))$, where $\text{MapTmp}(\text{op}, \text{src})$ is the logical mapped sequence produced by applying op pointwise to src .

Definition (Plan Well-Formedness). Write $\text{PlanWF}(p)$ iff:

- $\text{threads} > 0$;
- $\text{chunk} > 0$ whenever chunked parallel execution is requested;
- empty-input `map` and `scan` plans are admitted only when their shape obligations still determine a unique empty result buffer;
- empty-input `reduce` plans are admitted only when `IdentityId` resolves to an audited identity element for the carried `MonoidLaw`, otherwise they are rejected before lowering;
- every referenced buffer has explicit scalar kind, length, access mode, and layout witness;
- every `OutRef` has explicit scalar kind, length, and layout witness;
- every operator id resolves to a trusted registered kernel descriptor;
- every monoid law descriptor carries an explicit law class, operator id, identity id, and associativity or deterministic-tree witness identifier;
- $\text{ShapeOK}(p)$ holds.

Definition (Reduction-Law Admissibility). Write $\text{LawOK}(\mu, \tau)$ iff:

- if $\text{ClassOf}(\mu) = \text{StrictAssoc}$, then $\text{AssocWitness}(\mu)$ denotes a strict associative law over element type τ ; and
- if $\text{ClassOf}(\mu) = \text{DeterministicTree}$, then τ in $\{\text{Float}, \text{Double}\}$ and $\text{AssocWitness}(\mu)$ denotes a fixed reduction-tree witness for the normalized descriptor.

Floating-point `reduce` and `scan` plans are admissible only through `DeterministicTree`; the specification does not treat IEEE 754 addition or multiplication as unrestricted algebraic monoids.

Definition (Floating-Point Conformance Profile). Write $\text{FPProfile}(t)$ for the target-local floating-point environment declared by target t , including rounding mode, contraction policy, NaN and infinity handling, subnormal handling, and any other condition required to interpret a `DeterministicTree` result.

ABI Layout Isomorphism

Zero-copy borrowing or representation-preserving transfer is permitted only when `MoonBit` and `C` layouts are isomorphic at the boundary.

Definition (Boundary View Layout). For a layout-bearing buffer view on the active backend target, define the boundary view record:

```
lv_view ::= {
    addr : UIntPtr ,
    len : UInt64 ,
    kind : UInt32 ,
    mode : UInt32 ,
    stride : UInt64
}
```

Figure 10: Canonical boundary view.

The canonical boundary ABI is not inferred from an unspecified machine word w ; it is selected by an audited target profile $\text{ABIPProfile}(t)$.

Definition (Boundary ABI Profile). $\text{ABIPProfile}(t)$ must declare the unique width, alignment, and endian contract used by boundary records on target t , including the mapping of UIntPtr , UInt64 , and UInt32 .

with required layout equations under $\text{ABIPProfile}(t)$:

$$\text{sizeof}(\text{lv_view}) = \text{LVSize}(\text{ABIPProfile}(t)) \wedge \text{alignof}(\text{lv_view}) = \text{LVAlign}(\text{ABIPProfile}(t))$$

$$\text{offset}(\text{addr}) = 0 \wedge \text{offset}(\text{len}) = \text{OffLen}(\text{ABIPProfile}(t)) \wedge \text{offset}(\text{kind}) = \text{OffKind}(\text{ABIPProfile}(t))$$

$$\text{offset}(\text{mode}) = \text{OffMode}(\text{ABIPProfile}(t)) \wedge \text{offset}(\text{stride}) = \text{OffStride}(\text{ABIPProfile}(t))$$

Definition (Layout Record). For a scalar or buffer element type τ , define:

$$\text{Layout}(\tau) = (\text{size}(\tau), \text{align}(\tau), \text{stride}(\tau), \text{endian}(\tau))$$

Definition (Layout Isomorphism). Write $\text{Iso}(\tau_M, \tau_C)$ iff:

$$\text{Layout}(\tau_M) = \text{Layout}(\tau_C)$$

and both sides agree on scalar interpretation, element width, and contiguous traversal semantics.

Constraint (Boundary Layout Gate).

$$\text{Iso} \frac{\tau_M, \tau_C}{\text{LayoutOK}}(\tau_M, \tau_C)$$

If LayoutOK fails, the implementation must either copy into a conforming layout or reject the request. It may not silently reinterpret one layout as another.

LayoutOK additionally requires that the active target admit one audited $\text{ABIPProfile}(t)$ for the path that carries the view. A realization may not substitute a different record layout based on accidental host ABI compatibility.

Linear Ownership State Machine

The boundary contract is a small transition system over ownership-resource contexts.

Ownership states. $\text{Own_M}(b)$, $\text{Borrow_MR}(b)$, $\text{Borrow_C}(b)$, $\text{Own_C}(b)$, and $\text{Returned_M}(b)$ are the only admissible ownership states for a buffer identity b .

$\text{Returned_M}(b)$ is an observable return-complete intermediate state: C-visible authority has been relinquished, but MoonBit mutable authority is not restored until the rehydration step succeeds.

Transition rules.

$$\begin{array}{c}
\frac{\Gamma \vdash \text{Own_M}(b) \wedge \text{LayoutOK}(\tau_M, \tau_C) \wedge \text{Live}(b) \wedge \text{ReadOnly}(q)}{\Gamma \rightarrow (\Gamma - \{\text{Own_M}(b)\}) \cup \{\text{Borrow_MR}(b), \text{Borrow_C}(b)\}} \quad (\text{Borrow-Step}) \\
\frac{\Gamma \vdash \text{Own_M}(b) \wedge \text{LayoutOK}(\tau_M, \tau_C) \wedge \text{Exclusive}(b) \wedge \text{TransferOK}(q)}{\Gamma \rightarrow (\Gamma - \{\text{Own_M}(b)\}) \cup \{\text{Own_C}(b)\}} \quad (\text{Transfer-Step}) \\
\frac{\Gamma \vdash \text{Own_C}(b) \wedge \text{Received}(b) \wedge \text{Rebind}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Own_C}(b)\}) \cup \{\text{Returned_M}(b)\}} \quad (\text{Return-Step}) \\
\frac{\Gamma \vdash \text{Returned_M}(b) \wedge \text{RehydrateOK}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Returned_M}(b)\}) \cup \{\text{Own_M}(b)\}} \quad (\text{Rehydrate-Step}) \\
\frac{\Gamma \vdash \text{Borrow_MR}(b) \wedge \text{Borrow_C}(b) \wedge \text{BorrowDone}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Borrow_MR}(b), \text{Borrow_C}(b)\}) \cup \{\text{Own_M}(b)\}} \quad (\text{Discharge-Step})
\end{array}$$

Figure 11: Linear ownership transitions.

Linearity condition. No reachable context may contain both $\text{Own_M}(b)$ and $\text{Own_C}(b)$ for the same b , and any active $\text{Borrow_C}(b)$ blocks MoonBit-side mutation and free until the borrow token is discharged by Discharge-Step.

Version 1 Borrow means read-only borrow only. Any writable boundary view must use Transfer-Step; this specification defines no writable-borrow state transition.

Boundary Payloads

The FFI boundary transports plans and layout-checked views, not arbitrary object graphs.

Payload skeleton by constructor.

```

typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; OpBits op_bits; } Payload_Map;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; LawBits law_bits; } Payload_Reduce;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; LawBits law_bits; } Payload_Scan;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; OpBits op_bits; LawBits law_bits; }
Payload_MapReduce;

```

`Payload_Map` carries no `LawBits`. `Payload_Reduce` and `Payload_Scan` carry no `OpBits`. `Payload_MapReduce` carries both. No constructor may rely on absent fields, target-defined sentinels, or ignored payload bytes.

where each view entry carries:

- buffer identity or boundary view handle;
- scalar-kind tag;
- length;
- access mode;
- layout witness or layout class tag;
- ownership mode (Borrow Or Transfer).

In version 1, ownership mode Borrow is admissible only for read-only views. Writable boundary outputs must be lowered with ownership mode Transfer.

Boundary admissibility. $\text{Adm}(t, q)$ holds only when:

- the payload tag belongs to the target-specific payload grammar;
- every carried view satisfies `LayoutOK`;
- every requested ownership transition is legal under the linear state machine;
- every carried Borrow view is `ReadOnly`;
- every operator or monoid id resolves to a target-admitted trusted descriptor or law for t .

Wrapper Parsing and Descriptor Formation

The C wrapper or JS/N-API boundary is the interpreter from payload syntax to runtime descriptors.

Runtime descriptor by constructor.

```
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; } Descriptor_Map;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_Reduce;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_Scan;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_MapReduce;
```

Descriptor_Map carries no LawHandle. Descriptor_Reduce, Descriptor_Scan, and Descriptor_MapReduce each carry one resolved LawHandle.

Kernel descriptor and registry.

```
typedef struct {
    KernelId kernel_id;
    FnPtr fn_ptr;
    PurityWitness purity_witness;
    FootprintWitness footprint_witness;
    LayoutContract layout_contract;
    ArityMeta arity_meta;
    LawMeta law_meta;
    TargetMeta target_meta;
} KernelDesc;
```

KernelRegistry(t) is a static audited mapping from KernelId to KernelDesc for target t. KernelId is the only kernel-level identifier admissible in plan payloads; FnPtr is implementation-only and never crosses the MoonBit plan boundary.

Parsing relation.

Write ParseCoreMap(t, q, p, c, V, k) iff:

- DecodePlan(q) = p ;
- NormCfg(q) = c ;
- NormViews(q) = V ;
- ResolveKernel(KernelRegistry(t), q) = k ; and
- plan_tag(p) = MapPlan.

Write ParseCoreLaw(t, q, p, c, V, k, μ) iff:

- DecodePlan(q) = p ;
- NormCfg(q) = c ;
- NormViews(q) = V ;
- ResolveKernel(KernelRegistry(t), q) = k ; and
- ResolveLaw(q) = μ .

Write ViewsOK(V) iff every v in V satisfies both LayoutOK(v) and OwnerOK(v).

Write ParseReadyMap(t, p, V, k) iff KernelOK(k, p, V , emptyset, t).

Write ParseReadyLaw(t, p, V, k, μ) iff KernelOK(k, p, V, μ, t).

$$\begin{array}{c}
\frac{\text{ParseCoreMap}(t, q, p, c, V, k) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyMap}(t, p, V, k)}{\text{parse}(t, q) \mapsto \text{Descriptor_Map}(p, c, \text{ChunkPolicy}(c), V, k)} \\
\frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan_tag}(p) = \text{ReducePlan}}{\text{parse}(t, q) \mapsto \text{Descriptor_Reduce}(p, c, \text{ChunkPolicy}(c), V, k, \mu)} \\
\frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan_tag}(p) = \text{ScanPlan}}{\text{parse}(t, q) \mapsto \text{Descriptor_Scan}(p, c, \text{ChunkPolicy}(c), V, k, \mu)} \\
\frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan_tag}(p) = \text{MapReducePlan}}{\text{parse}(t, q) \mapsto \text{Descriptor_MapReduce}(p, c, \text{ChunkPolicy}(c), V, k, \mu)}
\end{array}$$

Figure 12: Wrapper parsing relation.

Descriptor fidelity rule. The produced descriptor must preserve:

- the constructor class of the plan;
- the scalar-kind and layout class of each carried buffer;
- the ownership mode attached to each boundary view;
- the `KernelId` selected by the plan and the audited descriptor resolved from the trusted registry;
- the shape obligations required by the plan constructor; and
- the denotation-defining choice of scan prefix semantics and map-reduce lowering.

Additionally:

- `Descriptor_Map` preserves no `LawHandle`;
- `Descriptor_Reduce` and `Descriptor_MapReduce` preserve one reduction law handle; and
- `Descriptor_Scan` preserves one scan law handle and the sequential join contract used for carry correctness.

Fork-Join Operational Semantics

We define a labelled small-step semantics over configurations $\text{Conf}(C, H, S)$, where H is the host heap and S is the worker-local store. Part III refines this abstract machine with explicit fail-closed error publication obligations for physical C/OpenMP realizations.

Worker syntax.

```

Worker ::= Read ( Addr )
        | Write ( Addr , Val )
        | MapStep ( Op , Addr , Addr )
        | ScanStep ( Law , Addr , Addr , State )
        | Seq ( Worker , Worker )
        | Done ( Val )

```

Figure 13: Worker syntax.

Read rule.

$$\frac{H(a) = v}{\text{Conf}(\text{Read}(a), H, S) \rightarrow \text{Conf}(\text{Done}(v), H, S)}$$

Figure 14: Read step.

Write rule.

$$\frac{\text{Writable}(a, S)}{\text{Conf}(\text{Write}(a, v), H, S) \rightarrow \text{Conf}(\text{Done}(v), H[a := v], S)}$$

Figure 15: Write step.

Map rule.

$$\frac{H(a_s) = x \wedge \text{Writable}(a_d, S) \wedge \text{Apply}(\text{op}, x) = y}{\text{Conf}(\text{MapStep}(\text{op}, a_s, a_d), H, S) \rightarrow \text{Conf}(\text{Done}(y), H[a_d := y], S)}$$

Listing 1: Map step.

Scan rule.

$$\frac{H(a_s) = x \wedge \text{Writable}(a_d, S) \wedge \text{Combine}(\mu, s, x) = s'}{\text{Conf}(\text{ScanStep}(\mu, a_s, a_d, s), H, S) \rightarrow \text{Conf}(\text{Done}(s'), H[a_d := s'], S)}$$

Figure 16: Scan step.

Sequence rules.

$$\frac{\text{Conf}(C_1, H, S) \rightarrow \text{Conf}(C_1', H', S')}{\text{Conf}(\text{Seq}(C_1, C_2), H, S) \rightarrow \text{Conf}(\text{Seq}(C_1', C_2), H', S')}$$

$$\frac{\text{Conf}(C_2, H, S) \rightarrow \text{Conf}(C_2', H', S')}{\text{Conf}(\text{Seq}(\text{Done}(v), C_2), H, S) \rightarrow \text{Conf}(C_2', H', S')}$$

Figure 17: Sequence steps.

Footprint contract. $R(C)$ is the set of addresses read by one step of C , $W(C)$ is the set of addresses written by one step of C , and $FP(C) = R(C) \cup W(C)$.

Descriptor metadata region. For every parsed descriptor d , define $\text{MetaOf}(d)$ as the read-only metadata region that stores the normalized runtime configuration, operator handle and parameters, law handle, and any per-plan constants required by spawned workers.

Chunk partition contract. $\text{Chunks}(n, k) = [I_0, \dots, I_{(k-1)}]$ only when the intervals cover $[0, n)$ and are pairwise disjoint.

Worker instantiation contract. If worker C_i is instantiated for slice I_i , then $FP(C_i)$ must stay within $\text{AddrOf}(I_i) \cup \text{MetaOf}(d)$, and $\text{MetaOf}(d)$ must remain read-only.

Sequential join-state relation. Write $\text{JoinState}(d, \sigma)$ for the constructor-specific sequential state used after worker completion. JoinState is empty for MapPlan , carries one fold accumulator for ReducePlan and MapReducePlan , and carries left-to-right chunk summaries and prefix carries for ScanPlan .

Fork and join rules.

$$\begin{array}{c}
\frac{\text{Chunks}(n, k) = [I_0, \dots, I_{k-1}] \wedge \forall i. \text{Spawn}(d, I_i) = C_i}{\text{Conf}(\text{Fork}(d), H, S) \rightarrow \text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S)} \\
\frac{\text{plan_tag}(d) = \text{MapPlan} \wedge \forall i. C_i = \text{Done}(y_i)}{\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{MapResult}(d)), H, S)} \\
\frac{\text{plan_tag}(d) \in \{\text{ReducePlan}, \text{MapReducePlan}\} \wedge \forall i. C_i = \text{Done}(y_i)}{\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{Fold}(\mu, \text{TreeOf}(d), [y_0, \dots, y_{k-1}])), H, S)} \\
\frac{\text{plan_tag}(d) = \text{ScanPlan} \wedge \forall i. C_i = \text{Done}(y_i) \wedge \text{JoinScan}(d, [y_0, \dots, y_{k-1}]) = \sigma}{\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{ScanResult}(d, \sigma)), H, S)}
\end{array}$$

Figure 18: Fork and join steps.

Scan and map-reduce realization rule. If $\text{plan_tag}(d) = \text{ScanPlan}$, then $\text{Spawn}(d, I_i)$ must emit workers that produce only chunk-local scan outputs and one chunk summary each. $\text{JoinScan}(d, \dots)$ must combine those summaries left-to-right and derive the inclusive prefix result required by $\text{Den}(\text{ScanPlan}(\dots))$. If $\text{plan_tag}(d) = \text{MapReducePlan}$, then descriptor normalization must preserve the logical lowering to map-then-reduce used by $\text{Den}(\text{MapReducePlan}(\dots))$; implementations may fuse the physical execution only when the observable result remains the same as that denotation.

For ScanPlan , carry correctness depends on JoinScan as a sequential join contract; it is not encoded as writable metadata and does not authorize an audited scratch region in version 1.

Floating-point combine rule. If $\text{ClassOf}(\mu) = \text{DeterministicTree}$, then $\text{TreeOf}(d)$ is part of descriptor normalization and uniquely determines the split-combine order. Reordering by worker scheduling does not change the combine tree selected by d . This rule guarantees a stable result for repeated executions under one target-local $\text{FPProfile}(t)$; cross-target equality is required only when the compared targets declare compatible audited floating-point profiles.

Parallel step side condition. If $\text{Conf}(C_i, H, S) \rightarrow \text{Conf}(C_i', H', S')$, then the lifted step on $\text{Par}([C_0, \dots, C_i, \dots, C_{k-1}])$ is admissible only when $W(C_i) \cap \text{FP}(C_j) = \text{emptyset}$ for every $j \neq i$.

This side condition does not apply to JoinState because JoinState is evaluated only after the worker-parallel phase completes.

JS ArrayBuffer Boundary Semantics

The JavaScript boundary is modeled as a target-specific transition system on host views.

JS view syntax.

$$\text{JSView} ::= (\text{ArrayBufferKind} , \text{Offset} , \text{Len} , \text{ScalarKind} , \text{AccessMode} , \text{Share})$$

Figure 19: JavaScript boundary views.

Boundary rules.

$$\begin{array}{c}
\frac{\text{Kind}(v) = \text{SharedArrayBuffer} \wedge \text{Access}(v) = \text{ReadOnly} \wedge \text{LayoutOK}(v) \wedge \text{AtomicOK}(v)}{(\text{Own_M}(b), v) \rightarrow (\text{Borrow_MR}(b) * \text{Borrow_C}(b), v)} \\
\frac{\text{Kind}(v) = \text{ArrayBuffer} \wedge \text{CopyBytes}(b) = b' \wedge \text{CopyOwner}(b') = \text{HostRuntime}}{(\text{Own_M}(b), v) \rightarrow (\text{Own_M}(b) * \text{Own_C}(b'), v)} \\
\frac{\text{TransferOK}(v) \wedge \text{Tombstone}(v)}{(\text{Own_M}(b), v) \rightarrow (\text{Own_C}(b), v)}
\end{array}$$

Figure 20: Borrow, copy, and transfer at the JS boundary.

Locking side condition. Every derivation of the shared-borrow rule carries a host-side lock obligation: no write on the shared region is admissible unless the matching atomic discipline is present in `Share`.

Shared snapshot obligation. An admitted `SharedArrayBuffer` borrow is interpreted against one logical snapshot of the read-only input region. The implementation may realize that snapshot by copying, freezing, or an observationally equivalent mechanism, but the accepted execution must preserve `Den(p)` for the snapped input rather than for later host-side writes.

If snapshotting is realized by copying, that copy is unobservable at the specification layer: it does not allocate a new normative buffer identity, does not alter ownership fidelity, and does not reclassify the admitted path from borrow to copy.

JS lifecycle obligation. For the copy rule, the host runtime owns `b` and must dispose it exactly once by explicit release or finalizer. For the transfer rule, the host object becomes a tombstone immediately after transfer admission; explicit return has priority over finalizer cleanup, and finalizer cleanup has priority over leak-on-fault behavior. If runtime admission or worker execution fails closed after transfer, the target must publish one deterministic cleanup path for the still-owned C-visible buffer.

Separation and Data-Race Freedom

Spatial separation predicate.

$$\text{SepChunks}(b, [I_0, \dots, I_{k-1}]) = \forall i \neq j. \text{AddrOf}(I_i) \cap \text{AddrOf}(I_j) = \emptyset$$

Parallel admissibility.

$$\text{ParOK}([C_0, \dots, C_{k-1}]) = \forall i \neq j. W(C_i) \cap \text{FP}(C_j) = \emptyset$$

Theorem (Data-Race Freedom). If:

1. `KernelSafe(d)` holds, meaning the resolved audited kernel descriptor satisfies `PurityWitness`, `FootprintWitness`, `LayoutContract`, and `LawMeta`,
2. `SepChunks(b, [I_0, ..., I_{(k-1)}])`,
3. each worker `C_i` is spawned from `I_i`,
4. `FP(C_i) subset.eq AddrOf(I_i) union MetaOf(d)` for every `i`, and
5. `MetaOf(d)` is read-only,

then every reachable configuration `Conf(Par([C_0, ..., C_{(k-1)}]), H, S)` satisfies `ParOK([C_0, ..., C_{(k-1)}])`.

Proof. By induction on the length of the reduction derivation. The base case follows from the chunk-spawn invariant and the definition of `SepChunks`. Premise (1) closes the theorem under the audited-kernel admissibility assumptions needed to derive worker footprints from plan data alone. For the inductive step, only one worker `C_i` can take a small step. By premise (4), any write of `C_i` lies in `AddrOf(I_i)`. For every `j != i`, premise (4) gives `FP(C_j) subset.eq AddrOf(I_j) union MetaOf(d)`. Premise (2) makes `AddrOf(I_i)` disjoint from `AddrOf(I_j)`, and premise (5) forbids writes into `MetaOf(d)`. Therefore `W(C_i) inter FP(C_j) = emptyset` is preserved for all `j != i`, so the parallel configuration remains `ParOK`.

Corollary (Ownership Safety). If `Borrow_C(b)` is active, `MoonBit` cannot derive a step that frees or mutates `b`; if `Own_M(b) -> Own_C(b)` has fired, `MoonBit` cannot derive mutable authority for `b` again without an explicit `Return-Step` followed by `Rehydrate-Step`; if `Borrow_MR(b) *` `Borrow_C(b)` is active, `Own_M(b)` is restored only by `Discharge-Step`.

Part II: Realization and Conformance

Native Realization

The native realization is:

MoonBit facade -> workflow descriptor -> pthread scheduler runtime -> compute lane -> OpenMP compute runtime

A conforming native realization must:

- lower only workflow graphs admitted by Part I;
- preserve one audited `ABIPProfile(t)` for each admitted boundary path, or perform explicit copying when layout is not isomorphic;
- implement read-only borrow and transfer edges according to the linear state machine;
- schedule writable chunks only under the separation contract of Part I;
- maintain a workflow scheduler layer for task, synchronization, and capability-state transitions;
- route compute nodes through a dedicated compute lane rather than opening OpenMP teams inside ordinary scheduler workers;
- expose a blocking workflow submit whose status is definitionally the final status of the underlying async workflow result.

JavaScript Realization

The JavaScript realization is:

MoonBit facade -> MoonBit JS FFI -> JS wrapper -> N-API addon -> runtime descriptor -> C runtime

A conforming JavaScript realization must:

- lower workflow graphs and typed-array-compatible compute views into payload views with explicit element-kind tags;
- reject host objects that do not satisfy layout and ownership obligations;
- preserve the same workflow and compute denotation contracts as the native realization, and require stronger cross-target equality only when both sides declare compatible `FPPProfile`;
- realize admitted `SharedArrayBuffer` borrows as logical snapshots for denotation purposes;
- define owner, tombstone, finalizer, and explicit-return behavior for copy and transfer paths before runtime execution starts;
- enforce borrow or transfer discipline before runtime execution starts.

Facade Obligations

A conforming facade must:

- expose only pure workflow constructors, compute-subplan constructors, and explicit execution entry points;
- refuse to encode arbitrary function-value environments or arbitrary host functions into workflow graphs or compute subplans;
- attach explicit ownership mode and access mode to every exported boundary view;
- reject any construction path that attempts writable lowering through ownership mode `Borrow`;
- surface capability queries that are extensionally equal to actual target support.

Runtime Obligations

A conforming runtime must:

- interpret descriptors only through the workflow and compute constructor classes defined in Part I;
- reject any descriptor that combines writable access with ownership mode `Borrow`;
- reject any descriptor whose chunk schedule fails separation or side-condition checks;

- preserve the ownership mode attached to each carried view;
- lift failures into explicit status classes rather than partial success;
- prohibit exception escape, panic propagation, or stack unwinding across the FFI boundary;
- implement deterministic layout fallback or rejection for non-isomorphic views before worker execution starts;
- treat temporary synchronization unavailability as a blocking state rather than as an immediate failure;
- fail closed when encountering runtime classes that are descriptor-valid but not yet executable in this version, including `RwLock`, `Semaphore`, and `OpaqueCapability`.

Unsupported Surface

The following are outside the version-1 contract:

- arbitrary MoonBit function-value execution on foreign worker threads;
- arbitrary JavaScript host-function execution inside native worker execution;
- implicit layout reinterpretation across non-isomorphic ABI records or arrays;
- implicit ownership transfer without a boundary state transition;
- chunk schedules that cannot be justified by separation and split-combine laws;
- running compute nodes by recursively opening OpenMP teams inside ordinary workflow scheduler workers;
- silently accepting `RwLock`, `Semaphore`, or `OpaqueCapability` as executable native workflow primitives.

Conformance

An implementation may claim conformance only if:

- its MoonBit-facing API constructs only the workflow graph and compute sub-IR admitted by Part I;
- its boundary payloads preserve explicit layout and ownership witnesses;
- its wrappers satisfy parsing fidelity and ownership fidelity;
- its scheduler/runtime satisfies workflow capability safety, chunk orthogonality, and split-combine soundness;
- invalid, non-isomorphic, or lifetime-unsafe requests fail closed before execution.

Conformance Transfer

Faithful realization. An implementation is faithful only if:

- every accepted payload is the lowering of a well-formed workflow graph;
- every borrow or transfer edge follows the linear ownership state machine;
- every writable chunk schedule satisfies `SepChunks`;
- every reduction or scan combine path is justified by a valid monoid descriptor; and
- every constructor-specific payload, workflow descriptor, compute descriptor, and join path preserves the constructor class admitted by Part I.

Conformance rule. If a faithful implementation accepts a workflow w , then the observable result and ownership trace of executing w must stay within Part I. If it rejects w , that rejection must be a spec-valid fail-closed outcome. For floating-point compute nodes, “the observable result” means stable repeated results under one admitted target profile, and cross-target equality only under explicitly compatible audited floating-point profiles.

Validation Scenarios

1. Submit a workflow whose compute node carries source and destination layouts that are not isomorphic. Expected result: explicit copy into a conforming layout, or fail-closed rejection.
2. Submit a workflow containing a read-only borrow path and attempt MoonBit-side mutation before borrow discharge. Expected result: mutation is rejected or blocked by the borrow discipline.
3. Submit a workflow whose compute node requires a writable chunk schedule with overlapping intervals. Expected result: descriptor formation or runtime admission fails before execution.
4. Submit a workflow containing a reduce or scan compute node with an operator lacking a valid monoid descriptor. Expected result: the compute node is rejected before fork-join execution.
5. Submit a transfer workflow and attempt MoonBit-side free through an old alias. Expected result: mutable alias permission is absent; the operation is rejected.
6. Execute equivalent floating-point compute nodes on native and JavaScript targets with layout-compatible views but incompatible floating-point profiles. Expected result: each target is internally deterministic, but cross-target equality is not claimed.
7. Execute equivalent floating-point compute nodes on targets that declare compatible floating-point profiles. Expected result: both targets preserve the same declared observable result and ownership discipline.
8. Submit a transfer-return-reuse workflow and attempt reuse before Rehydrate-Step. Expected result: reuse is rejected until rehydration succeeds.
9. Submit a JS copy or transfer workflow and trigger explicit return, finalizer cleanup, and fail-closed runtime admission on separate runs. Expected result: the documented lifecycle priority and single-disposal obligations hold.
10. Submit a workflow whose nodes form a cycle. Expected result: facade validation or native admission rejects the workflow before execution.
11. Submit a workflow that binds LockNode to Channel. Expected result: facade validation or native admission rejects the capability mismatch.
12. Submit a workflow that uses RwLock, Semaphore, or OpaqueCapability. Expected result: descriptor admission may succeed structurally, but native runtime fails closed with an explicit unsupported-runtime status before execution.
13. Submit a workflow with exactly one BarrierNode in one derived barrier group. Expected result: the final workflow status is BARRIER_BROKEN.

Closure Checklist

The specification is not closed unless all of the following hold:

1. Every exported execution request is representable by the workflow ADT, with compute requests represented as compute subnodes.
2. Every boundary view carries explicit layout and ownership witnesses.
3. Every zero-copy path satisfies layout isomorphism.
4. Every borrow path freezes MoonBit mutation until discharge.
5. Every transfer path consumes MoonBit mutable authority and restores it only through explicit return plus rehydration.
6. Every writable chunk schedule is justified by separation.
7. Every reduction or scan combine path is justified by a valid monoid law.
8. No exception, panic, or abort path escapes across the FFI boundary.
9. Every zero-copy admission path discharges byte-level layout and alignment checks.
10. Every JS copy and transfer path has one closed cleanup story, including finalizer and fail-closed cases.
11. Every witness obligation maps to an auditable artifact format.
12. Every invalid input vector fails by explicit status rather than crash or undefined behavior.
13. Every referenced guarantee is stated in the document.

Part III: Implementation Conformance Verification

Exceptional Control Flow and Fail-Closed Semantics

Part I models worker execution under successful local steps. A faithful physical realization must additionally close all exceptional control-flow paths that may arise in MoonBit, the C wrapper, or the native runtime.

Status taxonomy. Every implementation-visible failure must belong to a closed status family:

```
RejectReason ::= ParseReject
               | LayoutReject
               | OwnershipReject
               | DescriptorReject
               | OperatorReject
               | KernelLookupReject
               | KernelWitnessReject
               | KernelLayoutReject
               | KernelArityReject
               | KernelLawReject
               | BorrowLifecycleReject
               | TransferLifecycleReject
               | JSLifecycleReject
               | DeterministicTreeReject
               | TargetSupportReject
```

Figure 21: Closed reject-reason family.

```
Status ::= Ok
         | Reject ( RejectReason )
         | Fault ( RuntimeFault )
         | Worker ( WorkerFault )
```

Figure 22: Tagged conformance status algebra.

Boundary result carrier. Every MoonBit-exposed native kernel and wrapper entry used by runtime execution must return an ABI-safe result envelope:

```
typedef struct {
    Status status;
    Payload payload;
} FFIResult_Payload;
```

or an observationally equivalent target-specific record. No stack unwinding, panic propagation, or host exception propagation may cross the FFI boundary.

Phase mapping obligation. Every non-Ok status must identify one of the following conformance phases:

- facade construction rejection;
- wrapper parse rejection;
- descriptor admission rejection;
- pre-parallel runtime rejection; or
- in-region worker fail-closed publication.

This phase mapping is part of the observable contract and must be recoverable by MoonBit-side pattern matching or by an observationally equivalent target-facing API.

Shared worker status. Within a native parallel region, workers communicate failure only through a shared atomic status word initialized to Ok.

```
typedef struct {  
    _Atomic uint32_t code;  
} AtomicStatusWord;
```

Worker error publication rule. If a worker detects an error status $e \neq \text{Ok}$, then it must:

1. publish e to AtomicStatusWord with release ordering unless a prior non-Ok status is already visible;
2. stop issuing further writes outside thread-local cleanup;
3. take only cleanup or early-exit steps until the parallel region joins.

All other workers must observe the shared status word with acquire semantics at admission or loop checkpoints and degrade to deterministic early exit once a non-Ok value is seen.

CFG closure obligation. For every path reachable inside `#pragma omp parallel`, the implementation must prove by static analysis, code audit, or equivalent argument that control terminates in exactly one of the following ways:

- normal worker completion with Ok;
- explicit publication of a non-Ok status followed by local cleanup;
- early exit caused by observing a previously published non-Ok status.

No path may terminate by `abort()`, uncaught exception escape, cross-boundary unwinding, or undefined control transfer.

Layout Isomorphism Realization and Safe Degradation

Part I defines `LayoutOK` abstractly. A conforming implementation must realize it through byte-level static assertions and dynamic admission checks.

C boundary record obligations. For `lv_view`, the C realization must preserve the canonical size, alignment, and field offset equations of Part I through implementation-defined layout controls such as attributes, packing directives, or equivalent ABI-stabilizing mechanisms.

The implementation must bind those checks to one declared `ABIPProfile(t)` rather than to an inferred host word size.

Compile-time layout discharge. The native wrapper must discharge at compile time:

- `sizeof(lv_view)` and `alignof(lv_view)`;
- field offsets for `addr`, `len`, `kind`, `mode`, and `stride`;
- width checks for `UIntPtr`, `UInt64`, and `UInt32` in the active target profile;
- scalar-width and endian assumptions required by the active target contract.

These checks must be expressed by `static_assert` or an observationally equivalent compile-time mechanism.

Dynamic zero-copy gate. Before a boundary view is admitted as borrowed or transferred without copying, descriptor formation must check:

- base-pointer modulo alignment is zero for the required element alignment;
- stride equals the declared contiguous traversal rule for the scalar kind;
- total byte span is in range and free of arithmetic overflow;
- source and destination layout witnesses agree on scalar interpretation and width.

If any check fails, the implementation must not reinterpret the view in place.

Deterministic degradation rule. For every target, the wrapper must document one of the following outcomes for non-isomorphic views:

- CopyConforming: copy bytes into a conforming representation and continue; or
- RejectNonIsomorphic: reject before runtime admission.

That choice must be stable per target-path pair and may not depend on accidental host ABI compatibility.

For every admitted target path, the implementation must document the `ABIProfile(t)` and the chosen degradation rule as one auditable boundary contract.

Operator Purity and Plan Admissibility

The data-race-freedom theorem depends on worker footprints being derivable from plan data alone. Therefore the facade and lowering boundary must reject executable content that smuggles unverifiable effects across the FFI boundary.

Kernel registry surface. Version 1 executable plans may carry only `KernelId` references that resolve in the static audited `KernelRegistry(t)` of the active target. Arbitrary MoonBit closures, arbitrary JavaScript host functions, uncatalogued native function pointers, and captured mutable environments are not admissible plan payloads.

Each trusted `KernelDesc` must carry:

- `KernelId`, the only kernel-level identifier visible to the plan;
- hidden native `FnPtr`, which never crosses the MoonBit plan boundary;
- `PurityWitness`;
- `FootprintWitness`;
- `LayoutContract`;
- `ArityMeta`;
- `LawMeta`; and
- `TargetMeta`.

Dynamic arbitrary kernel registration is outside the contract. Conformance claims may rely only on descriptors present in the static audited registry.

For floating-point `reduce` and `scan`, admissibility additionally requires a `DeterministicTree` law witness so that the runtime descriptor fixes one canonical combine tree for the accepted execution.

Safe abstraction obligation. A conforming MoonBit-facing API must ensure that any type-checked and admitted execution request lowers to a plan whose worker footprint is determined by:

- the plan constructor;
- explicit buffer views and layout witnesses;
- the trusted `KernelId` and resolved `KernelDesc`; and
- runtime configuration fields admitted by `PlanWF`.

It must reject any construction path that would allow hidden writable aliases, escaped mutable references, or effectful operator bodies that cannot be justified against `SepChunks`, ownership linearity, and the boundary access mode.

Witness obligations. For every admitted descriptor `k = KernelDesc(...)`:

- `PurityWitness(k)` certifies that execution has no side effects outside the declared input/output views and `MetaOf(d)`;
- `FootprintWitness(k)` certifies that worker writes stay within the assigned chunk and that any read outside the chunk is limited to declared read-only inputs or `MetaOf(d)`;
- `LayoutContract(k)` must be discharged against all carried boundary views before runtime execution;
- `ArityMeta(k)` must match the constructor class and access modes of the plan; and
- `LawMeta(k)` must satisfy the reduction-law obligations required by `mu`, including the deterministic-tree requirement for floating-point reductions.

Failure of these obligations must reject execution through the dedicated reject branches of `Status`.

Auditable witness format. Every witness-bearing registry entry must expose an artifact bundle that is independently reviewable. The minimum acceptable bundle contains:

- a stable kernel identifier and version/hash binding for the audited implementation;
- a machine-checkable or tabulated summary of purity, footprint, layout, arity, and law claims;
- the admitted target or `ABIPProfile(t)` and floating-point profile assumptions under which the claims hold;
- the audit method used for each claim: proof artifact, static analysis result, test certificate, or manual review record; and
- the audit verdict recorded in a form that lets a third party distinguish “missing”, “rejected”, and “accepted” evidence.

The semantic obligation is normative; the artifact format is the minimum acceptable evidence by which conformance claims become reviewable rather than self-asserted.

Contract-Based Validation Matrix

Conformance claims must be backed by an executable validation matrix that covers acceptance, rejection, and fail-closed execution outcomes.

Validation dimensions.

1. Positive realization: equivalent `map`, `reduce`, `scan`, and `map-reduce` plans preserve plan denotation and ownership traces across the admitted native and JavaScript targets.
2. Exceptional control flow: injected operator faults, allocation failures, malformed payloads, and runtime admission failures surface as explicit status codes rather than process termination.
3. Layout and byte-level safety: bad alignment, inconsistent stride, non-isomorphic scalar kind, and overflowing byte-span calculations trigger `CopyConforming` or explicit rejection according to the target rule.
4. Ownership and chunk safety: overlapping writable chunks, stale aliases after transfer, mutation during active borrow, and double-borrow attempts fail before unsound execution.
5. Operator admissibility: closure-carrying, host-function-carrying, uncatalogued `KernelId`, or otherwise unverifiable plans are rejected at facade or wrapper admission time.
6. Floating-point deterministic reduction: repeated executions of the same admitted floating-point `reduce` or `scan` descriptor produce the same combine tree and the same observable result despite differences in worker scheduling under one target profile; cross-target equality is tested only for targets whose audited floating-point profiles are declared compatible.
7. Borrow lifecycle: successful discharge restores `Own_M(b)`, while premature discharge, duplicate discharge, or discharge without an active borrow is rejected.
8. Transfer-return lifecycle: successful `Return-Step` followed by `Rehydrate-Step` restores `Own_M(b)`, while reuse from `Returned_M(b)`, duplicate return, or return without active `Own_C(b)` is rejected through `TransferLifecycleReject`.

9. JS lifecycle: copied buffers have one owner and one disposal path; transfer tombstones prevent stale host reuse; finalizer and explicit-return order follows the documented target rule; a missing snapshot obligation or lifecycle violation rejects through `JSLifecycleReject`.
10. Borrow discipline: any view lowered with ownership mode `Borrow` is `ReadOnly`; writable borrow attempts reject before execution.
11. Metadata discipline: workers may read only `MetaOf(d)` outside their chunk footprint, and any writable or malformed metadata region is rejected before unsound execution.
12. Registry resolution: known `KernelId` values resolve to audited descriptors, while unknown ids reject through `KernelLookupReject`.
13. Witness validation: missing purity, footprint, layout, arity, law, profile, or target-support evidence rejects through the corresponding dedicated reject branch.
14. Audit artifact completeness: every registry entry used by the validation matrix identifies its evidence bundle and audit verdict.

Rejection-phase contract. Every negative validation case must specify:

- the violated precondition;
- the phase of rejection or failure publication: facade construction, wrapper parse, descriptor admission, pre-parallel runtime check, or in-region worker fail-closed publication;
- the expected tagged `Status` branch and reject subreason when applicable.

In particular, borrow discharge violations map to `BorrowLifecycleReject`, transfer-return violations map to `TransferLifecycleReject`, and JS snapshot / copy / transfer lifecycle violations map to `JSLifecycleReject`.

Coverage obligation. An implementation is conformance-complete only if every invalid input vector in the validation matrix resolves to an explicit spec-defined status and no invalid vector can produce segmentation fault, silent data corruption, or undefined behavior.