
A MUSIC NOTATION PLATFORM

Epiphany



Core Specification

A foundational specification for an open, Rust-native music notation platform: pitch and time primitives, the score graph, the layout intermediate representation, the file format, and the constraint-solver interface.

WORKING DRAFT · JUNE 17, 2026

This specification defines the foundational layer of the Epiphany platform: the data model, engraving engine, and file format on which all higher-level features are built. The user interface, audio engine, plugin runtime, and platform integrations are layered above the core and are addressed in separate specifications.

Status of This Document

This draft is **structurally stabilized**. Substantive review passes through the core type model, graph invariants, concurrent reduction semantics, file-format atomicity, solver conformance, and the determinism contract have been completed. The architecture is no longer under active redesign; further revisions are expected to be additive (companion specifications, normative algorithm choices for the open canonical algorithms, extension registry contents) rather than structural.

The remaining blockers to full conformance are companion specifications and the canonical algorithms identified in Appendix D Section D.11:

- Several companion specifications referenced throughout the document have not yet been delivered: Appendix A catalogs them and distinguishes them from accidentally missing content.
- Several algorithms affect canonical state and have not yet received one of the dispositions in Appendix D Section D.11: the spelling pre-pass, the notational-decomposition algorithm, the tempo curve integration algorithm, and the `wallclock_to_musical` root-finding algorithm. Cross-implementation byte equality of canonical score state **MUST NOT** be claimed for outputs derived from these algorithms until they are normatively specified or profile-declared (Appendix D Section D.2).
- Extension registry catalogs are referenced by typed identifier but not enumerated here; they are delivered as separate, versioned registry documents.

What conformance can be established now:

- The core data model can be structurally implemented and tested against the invariants in Chapter 5 Section 5.11.
- The operation framework (Chapter 6) can be implemented end-to-end against the canonical reduction algorithm.
- The file format (Chapter 8) can be implemented atomically and crash-recoverably against the fixed prelude and superblock protocol.
- Solver implementations can be developed against the interface contract (Chapter 9); reference-suite conformance becomes checkable when the Reference Suite companion is delivered.

The normative keywords **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, and **MAY** are to be interpreted as described in RFC 2119 when they appear in all capital letters.

Conventions

Code samples Rust syntax is used for type signatures, struct definitions, and algorithmic illustrations. Code samples are illustrative unless explicitly labeled normative.

Normative requirements Statements containing **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, or **MAY** are normative. Conforming implementations must respect them.

Non-normative content Discussion, rationale, examples, and diagrams are non-normative unless explicitly stated.

Forward references The specification builds bottom-up. Later chapters depend on earlier ones; forward references are minimized but occasionally unavoidable and are marked when they occur.

Contents

Status of This Document	i
Conventions	ii
Reading Guide	xi
1 Introduction and Scope	1
1.1 Purpose	1
1.2 Scope	1
1.3 Design Goals	2
1.4 Document Structure	2
2 Pitch	4
2.1 Design Principles	4
2.2 The <code>Pitch</code> Type	4
2.3 Scale Position	5
2.3.1 Octave Convention	6
2.3.2 The <code>CmnNominal</code> Type	7
2.4 Acoustic Realization	7
2.4.1 Cents as the Offset Unit	8
2.4.2 Tuning Reference and Inheritance	8
2.4.3 Reference Pitch	8
2.5 Spelling	8
2.5.1 The Spelling Attachment	9
2.5.2 Spelling Sources	11
2.5.3 Configurable Precedence	11
2.5.4 The Spelling Pre-Pass	12
2.5.5 Absent Accidentals	13
2.5.6 Analytical Layers	13
2.6 Pitch Identifiers	13
2.7 What Pitch Is Not	14
2.8 Equality and Comparison	14
2.9 Forward References	15
3 Time and Duration	16
3.1 Design Principles	16
3.2 The Rational Time Type	16
3.2.1 Recommended Implementation: Inline-or-Promoted	17

3.2.2	Promotion and Demotion	17
3.2.3	Equality and Ordering	18
3.2.4	Units	18
3.3	Position and Duration as Distinct Types	18
3.4	Wall-Clock Time	19
3.4.1	Wall-Clock and Musical Time Coexist	20
3.5	Time Anchors	20
3.5.1	Anchor Resolution	22
3.5.2	Anchor Stability Under Edits	22
3.6	Time Signatures and Meter	22
3.6.1	Irrational Meters	23
3.6.2	Polymetric and Polytemporal Scores	24
3.7	Tempo and the Tempo Map	24
3.7.1	Conversion	25
3.8	Sounding Duration and Notational Decomposition	26
3.8.1	The Notational Decomposition	26
3.8.2	The Decomposition Pre-Pass	27
3.9	Tuplets as Grouping Objects	28
3.9.1	Tuplet Nesting	28
3.9.2	Tuplet Consistency	29
3.10	Region Time Models	29
3.10.1	Metric Time	29
3.10.2	Proportional Time	30
3.10.3	Aleatoric Time	30
3.10.4	Concurrent Time Models	32
3.11	The Notion of “Now”	32
3.12	Forward References	32
4	Tuning Systems and Pitch Spaces	33
4.1	Design Principles	33
4.2	Pitch Spaces	34
4.2.1	Position Structure	34
4.2.2	Interval Algebra	35
4.2.3	Nominal Registry	36
4.2.4	Transposition Behavior	36
4.2.5	Spelling Rule Sets	37
4.3	Accidental Registries	37
4.3.1	Pitch Space Modifications	38
4.3.2	Accidental Engraving Metadata	38
4.3.3	Combination Behavior	39
4.3.4	Score-Local Extensions	39
4.4	Glyph References and SMuFL	39
4.4.1	SMuFL Versioning	40
4.5	Tuning Systems	40
4.5.1	Tuning Resolution	41
4.5.2	The Resolution Contract	42
4.5.3	Adaptive Tuning	42

4.6	Reference Pitch	42
4.6.1	Multiple Reference Pitches	43
4.7	Score Tuning Context and Hierarchical Resolution	43
4.7.1	Resolution Order	44
4.7.2	Compatibility Constraints	44
4.8	Built-in Catalog	45
4.8.1	Built-in Pitch Spaces	45
4.8.2	Built-in Tuning Systems	45
4.8.3	Default Score Configuration	46
4.9	Forward References	46
5	The Score Graph	47
5.1	Design Principles	47
5.2	Top-Level Score Structure	47
5.2.1	Score Metadata	49
5.3	Identifiers	49
5.3.1	Identifier Generation	51
5.3.2	System-Derived Identifier Namespace	52
5.3.3	System-Derived Counter Collisions	52
5.3.4	Identifier Stability	54
5.4	The Event Arena	54
5.4.1	The Event Type	55
5.5	The Canvas	60
5.5.1	Regions	61
5.5.2	Region Overlap and Concurrency	62
5.5.3	Staff-Based Content	62
5.6	Staves, Voices, and Instruments	65
5.6.1	Instruments	65
5.6.2	Staves: Identity Versus Instance	65
5.6.3	Voices	66
5.7	Cross-Cutting Structures	68
5.7.1	Slurs and Phrase Marks	69
5.7.2	Ties	69
5.7.3	Beams	71
5.7.4	Spanners	71
5.7.5	Markers	72
5.7.6	Repeat Structures	72
5.7.7	Analytical Annotations	73
5.7.8	Comments	73
5.7.9	Graphic Gestures	74
5.8	Graphic Content	74
5.8.1	Graphic Objects	75
5.8.2	Coordinates	76
5.8.3	Time Bindings	76
5.9	Parts	77
5.9.1	Parts Are Projections, Not Storage	78
5.10	Analysis Layers and Views	78

5.10.1	Analysis Layers	78
5.10.2	Views	78
5.11	Graph Invariants	79
5.12	Indexes and Auxiliary Structures	80
5.13	Forward References	81
6	Semantic Operations and Concurrent Reduction	82
6.1	Design Principles	82
6.2	The Operation Framework	83
6.2.1	Operation Identity and Stamps	83
6.2.2	Causal Context via Dotted Version Vectors	84
6.2.3	Operation Envelopes	84
6.2.4	The Operation Set as CRDT	86
6.2.5	OperationId Collisions and Replica Equivocation	86
6.2.6	Envelope Acceptance and Malformed-Envelope Behavior	88
6.2.7	Per-Replica HLC Monotonicity	89
6.3	The Canonical Reduction	91
6.3.1	Operation Lifecycle	91
6.3.2	The Reduction Algorithm	91
6.3.3	Canonical Reduction Order	92
6.3.4	Operation Effects	92
6.3.5	Cache and Snapshot Discipline	95
6.4	Object Existence and Tombstones	96
6.4.1	Add-Tombstone Semantics	96
6.4.2	Tombstone Retention	96
6.4.3	Reference Resolution Across Tombstones	97
6.5	Conflict Records	97
6.5.1	Conflict Record Type	97
6.5.2	The Conflict Registry	99
6.5.3	ConflictId Derivation	99
6.5.4	Conflict Resolution Operations	100
6.6	Transactions	101
6.6.1	Transaction Descriptor Causal Ordering	101
6.7	Re-Anchoring	103
6.7.1	The Re-Anchor Function	103
6.7.2	Total Ordering for "Nearest"	103
6.7.3	The Re-Anchoring Rule Table	104
6.8	Undo	107
6.8.1	UndoTransaction Payload	107
6.8.2	Undo Semantics	107
6.9	LWW Discipline	108
6.9.1	The LwwAdvisory Marker	108
6.9.2	Eligible Fields	109
6.10	Validation Modes	109
6.11	Per-Kind Reduction Summary	110
6.12	Representative Operations	111
6.12.1	InsertEvent	111

6.12.2	DeleteEvent	112
6.12.3	CreateCrossCutting (Slur)	114
6.12.4	RespellPitch	115
6.12.5	ChangeRegionTimeModel	116
6.12.6	Transpose	117
6.12.7	SetUserSystemBreak	117
6.13	Operation Catalog Conformance	118
6.14	Forward References	119
7	Layout Intermediate Representation	120
7.1	Design Principles	120
7.2	Spatial Primitives	121
7.2.1	Units	121
7.2.2	Geometric Types	121
7.3	Provenance	122
7.4	The Stage Pipeline	123
7.4.1	Pipeline Overview	123
7.4.2	Stage Contracts	124
7.5	LogicalLayoutIR	124
7.5.1	Top-Level Structure	124
7.5.2	Layout Regions	125
7.5.3	The Time Axis Trait	125
7.5.4	Layout Objects	127
7.5.5	Note Layout: A Worked Example	128
7.5.6	Engraving Decisions	129
7.6	Engraving Overrides	130
7.6.1	Override Definition	130
7.6.2	Override Resolution	131
7.7	ConstrainedLayoutIR	131
7.7.1	Glyph-Level Objects	132
7.7.2	Spring Slots	132
7.7.3	Vertical Bands	133
7.7.4	Constraints	133
7.8	ResolvedLayoutIR	134
7.9	RenderIR (Interface Only)	135
7.10	Incremental Layout and Caching	136
7.10.1	The Layout Cache	136
7.10.2	Invalidation Rules	137
7.10.3	Frame-Budget Requirements	137
7.11	Region Uniformity	137
7.12	Glyph Catalog Interface	138
7.12.1	Glyph Catalog Identity for Layout Conformance	139
7.13	Forward References	140
8	File Format	141
8.1	Design Principles	141
8.2	Durable Writes	142

8.3	The Bundle Layout	143
8.3.1	The Fixed Header	143
8.3.2	The Superblock Slots	144
8.3.3	Superblock Selection	146
8.4	The Manifest	146
8.4.1	Canonical and Non-Canonical Manifest Roots	148
8.4.2	File, Document, and Lineage Identity	148
8.4.3	Manifest Encoding	149
8.5	Content Hashing	150
8.5.1	Domain-Separated Preimages	150
8.6	Chunks	151
8.7	Operation Envelope Blocks	153
8.7.1	The Operation Index	154
8.8	Snapshots and Canonical Bases	154
8.8.1	The Canonical Document Identity	155
8.8.2	Pruning	156
8.9	Blobs	156
8.10	The Atomic Write Protocol	157
8.10.1	Crash Recovery	157
8.11	Garbage Collection and Retention	158
8.11.1	Retention Policy	158
8.11.2	Garbage Collection Rules	159
8.12	Forward Compatibility and Edit Barriers	159
8.12.1	Extension Declarations	159
8.12.2	Behavior Under Unknown Extensions	161
8.13	Text Projection	162
8.13.1	Use Cases	162
8.14	Schema Versioning	163
8.15	Format Profiles	164
8.16	Compression	164
8.17	Streaming Reads	165
8.17.1	Required Locatability	165
8.17.2	Cold Open Procedure	165
8.17.3	Memory Mapping	166
8.18	Binary Format Companion	166
8.19	Forward References	166
9	Constraint-Solver Interface	167
9.1	Design Principles	167
9.2	The Solver Interface	168
9.3	The Solver Report	169
9.4	Constraint Families	171
9.4.1	Strength Levels	171
9.4.2	Normative Constraint Families	172
9.5	Quality Metrics	172
9.5.1	Pareto Frontiers as Design Target	173
9.6	Conformance Tiers	174

9.6.1	Minimal Layout Solver	174
9.6.2	Standard Engraving Solver	175
9.6.3	Advanced / Extension-Aware Solver	175
9.6.4	Tier Declaration and Document Compatibility	175
9.7	Determinism	176
9.7.1	Within-Implementation Determinism	176
9.7.2	Cross-Implementation Conformance	176
9.8	Incremental Solving	177
9.8.1	Invalidation Scopes	177
9.8.2	Observational Equivalence	177
9.8.3	Propagation Documentation	178
9.9	Conformance: The Reference Suite	178
9.10	Reference Algorithm	179
9.11	Forward References	179
10	Performance Requirements	180
10.1	Core/Product Boundary	180
10.2	Design Principles	181
10.3	Reference Hardware Profile	181
10.4	Frame Budgets	182
10.4.1	Interactive Edit Latency	182
10.4.2	Multi-System Edits	182
10.4.3	Full Re-Layout	183
10.5	Cold Open	183
10.5.1	Cold Open Measurement	183
10.5.2	Streaming Read Discipline	183
10.6	Memory Budgets	184
10.6.1	Memory Categories	184
10.6.2	Per-Object Memory Discipline	184
10.7	Core Data Structures for Audio Thread Use	184
10.7.1	Snapshot API	185
10.8	Constraint-Solver Performance Targets	185
10.9	File Format Performance	186
10.10	Measurement Methodology	186
10.11	Regression Testing	186
10.12	Forward References	187
11	Extension Points	188
11.1	Design Principles	188
11.2	Notation Grammar Extensions	188
11.3	Tuning System Extensions	189
11.4	Glyph and Font Extensions	189
11.5	Score Graph Extensions	189
11.6	Layout IR Extensions	190
11.7	Constraint-Solver Extensions	190
11.8	File Format Extensions	190
11.9	Extension Registry Contract	190

11.9.1 Identifier Stability	191
11.9.2 Discoverability	191
11.10 Plugin Runtime Considerations	191
11.11 Conformance and Extensions	191
11.12 Forward References	192
A Intentionally Deferred Types and Specifications	193
A.1 Companion Specifications	193
A.2 Externally Provided Components	194
A.3 Open Canonical Algorithms	194
A.4 Extension Registry Catalogs	195
A.5 Support-Type Identity Semantics	195
A.6 Distinction from Open Questions	197
B Glossary	198
C Bibliography and References	203
D Determinism Contract	206
D.1 Thesis	206
D.2 Layers of Determinism	206
D.3 Floating-Point Values in Canonical State	207
D.3.1 Permitted Forms	208
D.3.2 Serialization	208
D.3.3 Equality	208
D.4 Rounding and CPU Behavior	208
D.5 Exact and Quantized Representations	209
D.5.1 Quantized Layout Coordinates	209
D.6 Tolerance Classes	210
D.7 Ordered Iteration over Sets and Maps	211
D.8 Text and Unicode	212
D.9 Randomness and Parallelism	213
D.9.1 Randomness	213
D.9.2 Parallelism	213
D.10 Compression and File Bytes	213
D.11 Open Algorithm Hooks	214
D.12 Conformance Statement	214
E Revision History	215

Reading Guide

This document is long. The following tables summarize the most load-bearing concepts and where they are defined. They are non-normative; the chapters they reference are authoritative.

Score Graph Identity Hierarchy

The score graph is organized as a containment hierarchy of identified objects. Each level introduces a distinct concept of identity.

Level	Identifier	Role
Score	—	Document root. Carries metadata, instruments, global staves, staff groups, tuning context, tempo map, canvas, conflict registry.
Canvas	—	Spatial root. Partitioned into regions.
Region	RegionId	Spatial-temporal container. Declares time model (metric/proportional/aleatoric) and content model (staff-based/free-graphic/hybrid).
Staff	StaffId	Global, abstract staff identity persisting across regions. Declared once at score level.
Staff instance	StaffInstanceId	Region-local manifestation of a staff. One Staff may have multiple instances across regions.
Voice	VoiceId	Polyphonic line within a staff instance. Carries VoiceOrigin: user-declared, imported, or system-promoted.
Event	EventId	Rhythmic atom: pitched, unpitched, rest, indeterminate, trajectory, graphic, cue. Lives in the score's event arena.
Identified pitch	PitchId	Stable identity for a pitch within an event. Enables spelling attachments, respelling, stable tie pairing through chord reordering.

Canonical versus Non-Canonical

Many file-format and runtime objects exist in two forms. Canonical objects define the document; non-canonical objects accelerate access but never affect canonical state.

Object	Status	Notes
Operation envelope set	Canonical	The replicated grow-only CRDT defining the document (Chapter 6).
Operation envelope block	Canonical (storage)	Soft 1MiB blocks on disk (Chapter 8).
Operation index	Non-canonical	Acceleration of operation-id lookup. Rebuildable.
Canonical-base snapshot	Canonical	The manifest's <code>canonical_base</code> : at most one active, with declared causal frontier and reduction algorithm version. Required for pruned documents.
Acceleration snapshot	Non-canonical	Listed in <code>acceleration_snapshots</code> . Cache only. Discardable.
Manifest	Canonical (index)	Table of roots; itself a chunk; referenced from the active superblock.
Materialized score graph	Derivative	Deterministic reduction of the operation set under the active reduction algorithm.
Layout cache, glyph cache, render artifacts	Non-canonical	Always discardable (Section 6.3.5).
Conflict registry	Canonical	Stable, addressable, user-visible records of operations that could not apply cleanly.

Operation Lifecycle

Every operation traverses four phases between user intent and its visible effect.

Phase	Description
Prepare	Local UI command validates enough to construct a well-formed envelope. Advisory only; failure here means the UI rejects the user's action. Does not bind canonical semantics.
Commit	The envelope is added to the replicated operation set. After commit, the operation is part of canonical state and MUST NOT be discarded outside the pruning protocol.
Reduce	During materialization, the operation is processed by the canonical reduction algorithm against working state. Deterministic: every replica with the same operation set produces the same output.

Phase	Description
Report	The reduction produces a deterministic <code>OperationEffect</code> : <code>Applied</code> , <code>AppliedWithRepair</code> , <code>Conflicted</code> , <code>TombstonedTarget</code> , or <code>NoOp</code> . Effects are visible graph facts.

Determinism Layers

The specification distinguishes five layers of determinism. Conflating them is the most common source of impossible requirements.

Layer	Contract
Canonical score determinism	Identical operation sets produce identical materialized score states across all conforming implementations, subject to all canonical algorithms having received the dispositions in Appendix D.
Canonical serialization	Same canonical state → same chunk hashes, same text projection, same canonical iteration order.
Layout determinism	Within one solver implementation at a fixed version: byte-equal output for identical input. Across implementations: reference-suite thresholds, not byte equal.
Conformance determinism	Reference-suite-based. Cross-implementation byte equality is not required for layout.
Non-canonical caches	May vary freely across runs, platforms, implementations. MUST NOT affect canonical state.

Solver Tiers

Solver implementations declare a conformance tier. Tiers are orthogonal to file-format profiles (Chapter 8).

Tier	Obligations
Minimal	Hard-constraint validity, within-implementation determinism, well-formed <code>SolveReport</code> , CMN + metric regions, Minimal reference-suite subset. Aesthetic quality may be mediocre.
Standard	Minimal + Standard-tier thresholds on the full reference suite, Measure-local invalidation under incremental solving, all Standard constraint families. Professional engraving quality for common-practice notation.
Advanced	Standard + proportional and aleatoric regions, microtonal accidentals, extension-contributed constraints, System-local invalidation under incremental solving.

Chapter Map

Ch.	Content
1	Scope, design principles, glossary of key terms used throughout.
2	Pitch as two intrinsic layers (scale position and acoustic), spelling subsystem with stacked accidentals, analytical layers.
3	Rational time, time anchors with <code>AnchorOffset</code> , three region time models, tempo map with explicit segment endpoints, sounding duration vs. notational decomposition.
4	Pitch spaces (including CMN and JI), tuning systems, score tuning context, accidental registries, glyph references.
5	Score graph, event arena, cross-cutting structures, regions, staff identity (Staff vs. StaffInstance), polymeter via per-staff metric grids, voice origin, conflict registry, graph invariants.
6	Operation framework, canonical reduction, DVV causal contexts, operation effects, tombstones, conflict records, transactions, re-anchoring, undo, LWW discipline.
7	The four-stage layout intermediate representation; provenance and incremental invalidation.
8	Fixed prelude, two superblock slots, atomic commit, content-addressed BLAKE3 chunks, operation-envelope blocks, snapshots as canonical bases, edit barriers, text projection.
9	Solver interface, SolveReport, deterministic budgets, NormalizedMetric, three conformance tiers, reference-suite conformance, six invalidation scopes.
10	Performance requirements: frame budgets, cold open, memory, audio-thread discipline, measurement.
11	Consolidated catalog of extension points by chapter of origin.
A	Intentionally deferred types: companion specifications, externally provided components, open canonical algorithms.
B	Glossary of defined terms with chapter cross-references.
C	Bibliography and references.
D	Determinism contract: the legal code for reproducibility.
E	Revision history.

Introduction and Scope

1.1 Purpose

This document specifies the core layer of *Epiphany*, a music notation platform. The core comprises the abstract musical data model, the engraving engine that transforms that data into a typeset layout, and the on-disk file format that serializes both. Everything else built atop the platform—editing UI, audio engine, plugin runtime, collaboration, export pipelines—depends on this core and must conform to its interfaces.

The name *Epiphany* evokes both the original sense of the word—a manifestation made visible—and the experience of music itself, which strikes the listener and the creator alike. Music notation is, in a literal sense, the act of making the inaudible visible: a body of work this central to that task deserves a name of corresponding weight.

1.2 Scope

In scope:

- Pitch, time, and duration primitives.
- The score graph: the in-memory representation of musical content.
- The layout intermediate representation produced by the engraving engine.
- The file format: the canonical on-disk projection of the score graph.
- The constraint-solver interface consumed by the engraving engine.
- Extension points for non-Common-Music-Notation systems, including microtonal pitch spaces and graphic-notation regions.

Out of scope for this document:

Non-Goal

The following are explicitly out of scope for this core specification. Each is addressed by a separate specification layered atop the core.

- User interface, input handling, and editing operations.
- Audio synthesis, playback, and plugin (VST3/AU/CLAP/LV2) hosting.

- The plugin runtime (WebAssembly capability-based extensions).
- Real-time collaboration transport and federation protocols.
- Import and export pipelines for foreign formats (MusicXML, MEI, LilyPond, MIDI, etc.), although the core defines the data sufficient to make lossless interchange feasible.
- Machine-learning components (transcription, OMR, engraving assistance).
- Platform-specific integrations (file pickers, sharing, accessibility APIs).

1.3 Design Goals

Correctness over convenience. The data model must represent music accurately, even when this is harder than a convenient approximation. MIDI-style pitch numbers, for example, are inadequate and rejected; see Chapter 2.

Extensibility without bolt-ons. Microtonality, graphic notation, historical notations, and future notational systems must be expressible within the core data model, not as escape hatches.

Determinism. Given identical input, the engraving engine must produce identical output. The on-disk format must serialize deterministically.

Incrementality. The data model and layout engine must support fine-grained incremental edits without full recomputation.

Reducibility. All editing operations on the score graph must have well-defined reduction rules under canonical deterministic reduction (Chapter 6), enabling collaboration and version control as natural consequences of the data model rather than bolt-on features.

Performance. Layout and rendering of a 100-page orchestral score must remain interactive on commodity hardware. See Chapter 10 for budgets.

Privacy. The core must operate entirely offline. No core functionality may depend on network access.

1.4 Document Structure

The remainder of this document is organized bottom-up:

1. Chapter 2 defines pitch: its layered structure, the spelling subsystem, scale positions, and acoustic realization.
2. Chapter 3 defines time and duration primitives.
3. Chapter 4 defines tuning systems, pitch spaces, and accidental registries.
4. Chapter 5 defines the score graph: how primitives compose into staves, voices, regions, and the overall score structure, including the canvas/region model that accommodates non-CMN notations.
5. Chapter 6 defines the semantic operations on the score graph: the canonical set of mutations and their reduction semantics under concurrent edits.

6. Chapter 7 defines the layout intermediate representation: the contract between the score graph (input) and the engraving engine (consumer).
7. Chapter 8 defines the on-disk file format, including serialization, chunking, the operation-envelope set and its deterministic reduction, and schema evolution.
8. Chapter 9 defines the constraint-solver interface consumed by the engraving engine.
9. Chapter 10 states performance requirements and measurement methodology.
10. Chapter 11 describes the extension points by which the core supports notation grammars beyond Common Music Notation.

Pitch

This chapter specifies the pitch primitive: the data type representing a single sounding (or to-be-sounded) note’s identity in the score graph. Pitch is the most consequential primitive in the core because nearly every higher-level operation—transposition, key analysis, accidental rendering, playback frequency, interchange, voice-leading analysis—is defined in terms of it. The definition is therefore deliberately verbose.

2.1 Design Principles

Separation of identity layers. Pitch comprises three independent semantic layers: scale position (analytical identity), acoustic realization (sounding frequency), and spelling (notational appearance). Each is a first-class concern and **MUST** be representable independently. Conflating them is a design error.

Spelling is attachment, not field. A pitch carries its scale position and acoustic identity intrinsically. Spelling is attached externally through a separate indexed subsystem (see Section 2.5). A pitch may have zero, one, or multiple spelling attachments.

Grammar neutrality. The pitch type **MUST** support Common Music Notation as a fast path while remaining capable of representing pitches in arbitrary user-defined notation grammars. CMN is a privileged default, not a hardcoded ceiling.

Microtonality is not an extension. Microtonal pitches are representable using the same primitive as 12-tone equal-tempered pitches. No separate type, no escape hatch.

Unpitched events are not pitches. Unpitched percussion and similar events **MUST NOT** be represented using the pitch type. See Section 2.7.

2.2 The Pitch Type

A pitch is a record of two intrinsic layers:

```
/// A pitch's intrinsic identity. Spellings are attached externally;
/// see Section~\ref{sec:pitch:spelling}.
pub struct Pitch {
    /// The analytical identity within a pitch space. Determines
    /// transposition, key analysis, and Roman-numeral behavior.
    pub scale_position: ScalePosition,
```

```

    /// The acoustic identity: what frequency this pitch produces when
    /// sounded, expressed relative to a tuning system.
    pub acoustic: AcousticPitch,
}

```

The two layers are independent. A given scale position can be acoustically realized in many ways depending on the active tuning system; a given acoustic realization can correspond to many scale positions in different analytical contexts. The pitch carries both; neither is derivable from the other in general.

RATIONALE

Storing both layers, rather than deriving one from the other, costs a small number of bytes per pitch and saves a large amount of complexity downstream. Derivation requires the renderer, the analyzer, the transcriber, and the importer to each agree on a derivation procedure; storage makes the pitch self-describing.

2.3 Scale Position

A scale position locates the pitch within a *pitch space*. A pitch space is the analytical universe in which scale-degree and interval relationships are defined.

```

pub struct ScalePosition {
    /// The pitch space this position is defined within. References an
    /// entry in the score's pitch-space registry.
    pub space: PitchSpaceId,

    /// The position within that space. Interpretation is space-defined.
    pub position: PitchSpacePosition,
}

```

The `PitchSpacePosition` type is a tagged union supporting the common cases with a fast representation, plus a registry-id escape hatch for arbitrary grammars:

```

pub enum PitchSpacePosition {
    /// Common Music Notation: a diatonic nominal plus chromatic
    /// alteration plus octave. The fast path for tonal Western music.
    Cmn {
        nominal: CmnNominal,          // C, D, E, F, G, A, B
        alteration: i8,                // -2..=+2 in semitones
        octave: i8,                    // scientific pitch notation
    },

    /// N-tone integer position. Used for 12-tone serial music, EDO
    /// systems, and any space where pitches are indexed as integers.
    Integer {
        space_size: u16,                // e.g., 12, 19, 31, 53
        index: i32,                    // absolute index; octave is implicit
    },
}

```

```

/// Just-intonation lattice position. Used for JI pitch spaces.
/// Each component is the integer power of the corresponding prime
/// in the space's prime basis. The pitch space declares the basis
/// ordering; by convention prime 2 is first, so the leading
/// component carries octave information.
JiVector {
    components: Vec<i32>,
},

/// A registered position whose interpretation is defined by the
/// pitch space's grammar plugin. Used for maqamat, gamelan,
/// historical modes, and other non-lattice, non-CMN grammars.
Registered(PositionRegistryId),
}

```

REQUIREMENT

For `PitchSpacePosition::JiVector`:

- `components.len()` **MUST** equal the number of primes in the enclosing pitch space's declared prime basis.
- The basis ordering is established by the pitch space; the built-in JI pitch spaces (`ji-5limit`, `ji-7limit`, `ji-11limit`) order primes ascending starting with 2.
- The vector $[a_0, a_1, a_2, \dots]$ denotes the ratio $p_0^{a_0} \cdot p_1^{a_1} \cdot p_2^{a_2} \dots$ where p_i is the i -th prime in the basis. With the conventional basis $\{2, 3, 5, \dots\}$, the first component holds octave information.
- Pitch spaces **MAY** declare octave-reduced equivalence, in which case the first component is normalized to a canonical range. Without such a declaration, full register is preserved.

REQUIREMENT

Every score **MUST** define at least one pitch space. The default pitch space for CMN scores is `cmn-12`, which is diatonic-over-chromatic in structure (seven diatonic nominals A–G over twelve chromatic positions), anchored to the score's reference pitch as resolved through the score tuning context (Chapter 4). Scores **MAY** define additional pitch spaces; pitches in different spaces cannot be directly compared and operations between them **MUST** go through an explicit space-conversion mechanism.

2.3.1 Octave Convention

For the `cmn` variant, the octave field uses Scientific Pitch Notation: middle C is C₄. This convention is normative throughout the specification.

REQUIREMENT

The octave field of `PitchSpacePosition::Cmn` **MUST** follow Scientific Pitch Notation. Middle C is C4. The lowest A on a standard piano is A0. C-1 is one octave below the lowest C on a standard piano.

RATIONALE

Scientific Pitch Notation is the dominant modern convention in English-language music theory and notation software. MIDI octave numbering, where middle C is sometimes C3 and sometimes C5 depending on vendor, is rejected as a source of foot-guns. Helmholtz notation is expressible at the display layer; the data model commits to Scientific.

2.3.2 The CmnNominal Type

```
#[repr(u8)]
pub enum CmnNominal { C = 0, D = 1, E = 2, F = 3, G = 4, A = 5, B = 6 }
```

The discriminants are normative: they define the diatonic step ordering used by transposition algorithms.

2.4 Acoustic Realization

The acoustic layer specifies the sounding frequency. It does not carry a frequency directly; instead it references a tuning system that, given the pitch's scale position, resolves to a frequency.

```
pub struct AcousticPitch {
    /// The tuning system governing this pitch's frequency.
    /// May be inherited; see Section~\ref{sec:pitch:tuning-inherit}.
    pub tuning: TuningReference,

    /// How the tuning system resolves to a frequency.
    pub realization: AcousticRealization,
}

pub enum AcousticRealization {
    /// Resolve through the tuning system using the scale position alone.
    /// This is the default case for ordinary CMN.
    Implicit,

    /// An explicit offset in cents from what the tuning system would
    /// otherwise produce. For inflections, expressive retuning, or
    /// just-intonation adjustments within a tempered piece.
    CentsOffset(f64),

    /// An explicit absolute frequency in Hertz, overriding the tuning
    /// system. For spectral music or fixed-frequency electronic parts.
    AbsoluteHz(f64),
}
```

```
}
```

2.4.1 Cents as the Offset Unit

Cents (one cent equals one hundredth of an equal-tempered semitone) are the normative offset unit. `f64` provides far more precision than the ear can resolve in any context.

2.4.2 Tuning Reference and Inheritance

A `TuningReference` is either an explicit tuning-system identifier or an inheritance marker:

```
pub enum TuningReference {  
    /// Inherit the tuning system from the enclosing scope.  
    Inherit,  
  
    /// Explicitly named tuning system.  
    Explicit(TuningSystemId),  
}
```

Resolution proceeds by walking outward: pitch, then voice, then staff, then region, then score. The score's tuning system **MUST** be explicit; `Inherit` at the score level is a malformed document. Tuning systems and their resolution rules are defined in detail in Chapter 4.

2.4.3 Reference Pitch

A pitch's `TuningReference` resolves, through the hierarchical walk, to a tuning system. The *reference pitch* (the position and frequency that anchor that tuning system's structure to absolute Hertz) is resolved separately through the same hierarchical walk, and is a property of the score's tuning context, not of the tuning system itself.

The pitch type does not own or imply a reference frequency. Two scores using the same tuning system (12-TET) but different references ($A_4 = 440$ Hz versus $A_4 = 415$ Hz) share their tuning structure entirely; only their reference pitch differs.

The full reference-pitch model, including the `ReferencePitch` type and the score tuning context that owns it, is specified in Chapter 4 (Section 4.6).

2.5 Spelling

Spelling determines what the performer sees on the page for a given pitch: the staff position, the accidental glyph (if any), and any ancillary notational marks (parenthesized accidentals, courtesy accidentals, microtonal accidentals).

Spelling **MUST NOT** be a field on the pitch type. Spellings are stored externally, indexed by pitch identifier, and managed by the spelling subsystem defined in this section.

RATIONALE

Spelling as a field on pitch causes destructive overwrites on respelling, prevents partial spelling (where only some pitches have user-chosen spellings and the rest are inferred), prevents multiple spellings per pitch (analytical layers), and prevents provenance tracking. Externalizing spelling resolves all four.

2.5.1 The Spelling Attachment

```
pub struct SpellingAttachment {
    /// What this attachment applies to: a single pitch or a scope.
    pub scope: SpellingScope,

    /// The directive: explicit spelling or a rule for inference.
    pub directive: SpellingDirective,

    /// Provenance of this attachment.
    pub source: SpellingSource,

    /// Tie-break priority among attachments. Higher wins. See
    /// Section~\ref{sec:pitch:precedence} for the configurable
    /// precedence model.
    pub priority: i32,

    /// Optional analytical layer. None means the engraved layer;
    /// Some(id) means an alternative analytical layer that does not
    /// affect the engraved score.
    pub layer: Option<AnalysisLayerId>,
}

pub enum SpellingScope {
    /// Applies to one specific pitch.
    Pitch(PitchId),

    /// Applies to all pitches matching the selector within a time range.
    Range {
        start: TimeAnchor,
        end: TimeAnchor,
        voices: VoiceSelector,
    },
}

pub enum SpellingDirective {
    /// An explicit spelling for a single pitch. Only valid with
    /// SpellingScope::Pitch.
    Explicit(PitchSpelling),

    /// A rule for inferring spellings within a scope.
    Rule(SpellingRule),
}

pub struct PitchSpelling {
```

```

/// The nominal: the staff position. For CMN, this is one of A-G.
/// For other grammars, an identifier into the grammar's nominal
/// registry.
pub nominal: SpellingNominal,

/// Stack of accidentals applied to the nominal, in stacking
/// order: innermost (closest to the notehead) first.
/// Empty vector means no accidental glyph is drawn, which does
/// not necessarily mean "natural"; see
/// Section~\ref{sec:pitch:absent-accidental}.
pub accidentals: Vec<AccidentalId>,

/// The octave designation.
pub octave: i8,

/// Optional rendering hints: parenthesized, cautionary, editorial,
/// small-print, etc. Non-normative for sounding pitch.
pub render_hints: SpellingRenderHints,
}

pub enum SpellingNominal {
/// CMN nominals: the fast path.
Cmn(CmnNominal),

/// Integer nominals for N-tone systems.
Integer(i32),

/// Registered nominals for grammar-specific systems.
Registered(NominalRegistryId),
}

```

Accidental Stack Semantics

REQUIREMENT

The `accidentals` vector **MUST** be interpreted as follows:

- The stored order is normative for engraving: index 0 is the innermost glyph (closest to the notehead), with subsequent indices placed further outward.
- For built-in additive accidental systems, the pitch-space modification contributed by the stack is the sum of the modifications of its members, and is independent of stored order. Reordering the stack changes engraving appearance but **MUST NOT** change sounding pitch.
- Custom accidental systems **MAY** declare non-additive combination semantics only through their registered `AccidentalCombination` (Section 4.3). Otherwise, accidentals are treated as commutative under their pitch effect.
- Two accidentals with the same `AccidentalId` in one stack constitute a malformed spelling, unless the accidental's registered combination semantics explicitly permits repetition. The conventional double-sharp and double-flat are represented as

single accidental definitions, not as repeated singles.

- An empty `accidentals` vector means no glyph is drawn; it is distinct from a vector containing only a natural sign (which explicitly cancels prior alterations and is drawn).

2.5.2 Spelling Sources

Every spelling attachment carries provenance:

```
pub enum SpellingSource {  
    /// The user explicitly chose this spelling.  
    UserChosen,  
  
    /// Inferred by the spelling pre-pass from key signature and  
    /// context. Lower precedence; may be revised by re-running the  
    /// pre-pass.  
    Inferred,  
  
    /// Imported from a foreign format.  
    Imported { format: ForeignFormatId },  
  
    /// Propagated from a transposition or other edit operation.  
    Propagated { from: PitchId },  
  
    /// An analytical spelling on a non-engraved layer.  
    Analytical,  
}
```

REQUIREMENT

Editing operations that respell a pitch **MUST** produce attachments with source `UserChosen`. Editing operations that move or transpose a pitch **MUST** produce attachments with source `Propagated`. The spelling pre-pass (Section 2.5.4) **MUST** produce attachments with source `Inferred`. Foreign-format importers **MUST** produce attachments with source `Imported`.

2.5.3 Configurable Precedence

When multiple attachments target the same pitch on the same analysis layer, the renderer resolves the conflict by precedence. The default precedence is:

1. `UserChosen` (highest)
2. `Imported`
3. `Propagated`
4. `Inferred` (lowest)

REQUIREMENT

Every score **MUST** carry a `SpellingPrecedence` configuration. The configuration **MUST** assign a total ordering over the variants of `SpellingSource`. The configuration **MAY** differ from the default; for example, a workflow that treats MusicXML imports as authoritative may rank `Imported` above `UserChosen`. Conflicts are broken first by precedence, then by the `priority` field of the attachment, then by attachment creation timestamp.

2.5.4 The Spelling Pre-Pass

When a score is loaded, edited, or about to be rendered, the spelling pre-pass produces `Inferred-source` attachments for every pitch that lacks a higher-precedence attachment. The pre-pass **MUST** be deterministic: given the same score graph and the same precedence configuration, it **MUST** produce identical attachments.

The pre-pass operates as follows:

1. Collect all pitches in the score, partitioned by voice and measure.
2. For each partition, in time order, resolve each pitch's spelling using:
 - (a) The active key signature for the staff.
 - (b) The accidental context of prior pitches in the same measure and voice.
 - (c) The melodic context (preceding interval).
 - (d) The harmonic context (concurrent pitches in other voices).
 - (e) Any active scope-level `Rule` attachments.
3. Write an `Inferred-source` attachment for each pitch.

The pre-pass **MUST** be incremental: an edit to a single pitch **MUST** re-run the pre-pass only on the affected measure and voice (and forward through measures whose context is altered, until the context stabilizes).

RATIONALE

Storing inferred spellings as attachments—rather than computing them at draw time—makes spellings inspectable in the UI, editable by promotion to `UserChosen`, and deterministic for serialization diffs. The cost is one attachment per unspelled pitch, which is negligible.

OPEN QUESTION

The melodic and harmonic context rules above are stated abstractly. The specific algorithm—whether based on Longuet-Higgins line-of-fifths distance, Temperley's preference rules, or another approach—is unresolved. The choice affects canonical state and must therefore receive one of the dispositions in Appendix D Section D.11: delivery as a normative algorithm in a Spelling Pre-Pass companion specification, profile-declared by versioned identifier, or explicitly marked non-canonical (in which case automatic

spellings are advisory and `UserChosen` attachments are the only canonical spellings).

2.5.5 Absent Accidentals

A `PitchSpelling` with an empty `accidentals` vector means no accidental glyph is drawn at this position. This is distinct from a vector containing an explicit natural sign. The distinction matters in two cases:

- A pitch on a line affected by the key signature carries no accidental and sounds the key-signature pitch; drawing a natural would be a respelling.
- A pitch that follows an in-measure accidental on the same line and octave inherits that accidental; an empty vector means inheritance applies, while a vector containing a natural cancels the inheritance.

REQUIREMENT

The renderer **MUST** distinguish between an empty `accidentals` vector (no glyph) and a vector containing only a natural (explicit natural glyph) when materializing a spelling. Promotion from empty to natural-only, or vice versa, is a respelling operation.

2.5.6 Analytical Layers

A pitch may carry multiple spelling attachments on different analytical layers. The engraved layer (`layer: None`) governs what the performer sees. Other layers (`layer: Some(AnalysisLayerId)`) record alternative spellings used by analytical views, theory tools, or pedagogical overlays.

The `AnalysisLayer` type itself is defined in Chapter 5 (Section 5.10); a layer identifier here refers to that authoritative definition.

REQUIREMENT

Resolution of a pitch's spelling for a given view **MUST** consider only attachments whose `layer` matches the view's active layer. The engraved view's active layer is `None`. Analytical views may set an explicit layer; if no attachment on that layer matches, the view **MUST** fall back to the engraved layer.

2.6 Pitch Identifiers

Every pitch in the score graph carries a stable identifier:

```
pub struct PitchId(pub u128);
```

REQUIREMENT

Pitch identifiers **MUST** be stable across edits: a pitch’s identifier is assigned at creation and never reassigned. Pitch identifiers **MUST** be unique within a score. Pitch identifiers **MUST** be collision-resistant across distributed editing scenarios; the recommended generation strategy is a 128-bit value combining a per-replica identifier and a monotonic counter, with the exact format defined in Chapter 6.

2.7 What Pitch Is Not

Several musical phenomena resemble pitches but are not represented by the pitch type:

Unpitched events. Snare drum hits, claves, cymbal strokes, and other unpitched percussion are sounding events with staff positions but no pitched identity. They are represented by a separate `UnpitchedEvent` type (see Chapter 5). They have no scale position and no acoustic pitch in the sense defined here.

Rests. A rest is a duration without a sounding event. Rests are represented by a separate `Rest` type. They carry timing and voice information but no pitch.

Indeterminate-pitch events. Events whose pitch is deliberately indeterminate (text instructions like “highest possible note,” aleatoric pitch material, certain graphic-notation events) are represented by an `IndeterminatePitchEvent` type, which may carry hints (range, contour, density) but no concrete pitch.

Pitch trajectories. Glissandi, portamenti, and continuous pitch bends are not single pitches but trajectories between or around pitches. They are represented as relationships between pitched events plus a trajectory shape, defined in Chapter 5.

RATIONALE

Forcing every sounding event into a pitch-shaped hole creates invariant-breaking edge cases throughout the codebase. Sibling types under a common `SoundingEvent` (or similar) abstraction keep each type coherent and let operations branch cleanly.

2.8 Equality and Comparison

Two pitches are *structurally equal* if and only if their scale positions and acoustic realizations are equal field-by-field. Structural equality is the default `Eq` implementation.

Three additional equivalence relations are defined:

Sounding equivalence. Two pitches are sounding-equivalent if they resolve to the same frequency under their respective tuning systems. Computed, not derived from the type.

Scale-position equivalence. Two pitches are scale-position-equivalent if their `ScalePosition` fields are equal, ignoring acoustic realization.

Enharmonic equivalence. Two pitches are enharmonically equivalent if they are sounding-equivalent under 12-tone equal temperament, regardless of their actual tuning system.

Used for foreign-format export and certain analytical operations.

REQUIREMENT

Implementations **MUST** provide functions for each of the three computed equivalences. Implementations **MUST NOT** conflate structural equality with sounding or enharmonic equivalence.

2.9 Forward References

- Tuning systems, pitch spaces, and accidental registries are defined in Chapter 4.
- The score graph, including how pitches are embedded in events, voices, and staves, is defined in Chapter 5.
- `TimeAnchor`, `VoiceSelector`, `PitchId` generation, and the reduction rules governing spelling attachments under concurrent edits are defined in Chapter 6.

Time and Duration

This chapter specifies the time and duration primitives: how positions and durations are represented, how time signatures and tempo maps are modelled, how notational rhythm is decomposed from sounding duration, how tuplets and other groupings work, and how the core supports metric, proportional, and aleatoric time models simultaneously.

3.1 Design Principles

Exact rational arithmetic. Musical positions and durations **MUST** be exact rationals. Floating-point time is rejected as a source of accumulated error; integer-tick time is rejected as incompatible with arbitrary tuplet nesting. Exactness is non-negotiable.

Two clocks. Musical time (measured in whole-note rationals) and wall-clock time (measured in fixed-point seconds) are distinct primitives. Conversion between them is a pure function of the tempo map.

Position and duration are distinct types. A position and a duration share an underlying representation but are not interchangeable. The algebra (position + duration $\rightarrow$ position; position – position $\rightarrow$ duration; position + position is undefined) is enforced at the type level.

Sounding duration is data; notational rhythm is attached. A note’s sounding duration is intrinsic; its notated decomposition (notehead values, ties, dots, tuplet membership) is produced by a deterministic pre-pass analogous to the spelling pre-pass and is overridable per-event.

Multiple time models coexist. A score **MUST** support metric, proportional, and aleatoric regions, possibly concurrent on different staves. The region’s declared time model determines the interpretation of positions within it.

Time anchors, not absolute positions. Stored references to points in time **MUST** anchor to identified objects (events, measures, regions) plus offsets. Absolute positions **MUST NOT** be stored where they could shift under edits.

3.2 The Rational Time Type

The fundamental time primitive is an exact rational number. The specification states the contract; implementations are free to choose representations meeting it.

REQUIREMENT

The rational time type **MUST** behave as an exact rational with at least 32-bit numerator and 32-bit denominator range in normalized form. Operations **MUST** produce exact results; on representation overflow, the implementation **MUST** either widen to an exact unbounded representation or return an error. Silent loss of precision is forbidden. Implementations **MAY** use any representation that meets this contract, including packed inline-or-heap-promoted designs.

3.2.1 Recommended Implementation: Inline-or-Promoted

The recommended implementation packs the common case (denominators under 2^{32}) into 8 bytes and promotes to a heap-allocated arbitrary-precision rational on overflow. This is normative for conformance only inasmuch as the observable behavior must match; it is illustrative as a reference design.

```
/// A musical time value: position or duration. Exact rational.
pub enum RationalTime {
    /// Inline case: fits in 8 bytes.
    Small(SmallRational),

    /// Promoted case: heap-allocated arbitrary-precision rational.
    /// Used only when arithmetic overflows the inline range.
    Large(Arc<BigRational>),
}

/// The inline rational: numerator i32, denominator NonZeroU32.
/// Always normalized: gcd(|numerator|, denominator) == 1.
#[derive(Copy, Clone, PartialEq, Eq, Hash)]
pub struct SmallRational {
    numerator: i32,
    denominator: NonZeroU32,
}
```

REQUIREMENT

All instances of the small rational **MUST** be normalized: the greatest common divisor of the absolute value of the numerator and the denominator **MUST** equal one, and the denominator **MUST** be strictly positive. The constructor **MUST** enforce normalization.

3.2.2 Promotion and Demotion

Arithmetic operations on two small rationals are performed in widened intermediate types (*i64* numerator, *u64* denominator) and the normalized result is examined. If the result fits in the small representation, it is returned as `Small`; otherwise it is returned as `Large`.

REQUIREMENT

Arithmetic that exceeds the small representation's range **MUST** silently promote to the large representation. The user of the type **MUST** observe no behavioral difference between small-only and small-or-large arithmetic; the contract is exact arithmetic on exact rationals.

Implementations **MAY** demote a `Large` value back to `Small` when its normalized representation fits, but **MUST NOT** make demotion observable (e.g., by changing equality or hash behavior across the variant boundary).

3.2.3 Equality and Ordering

Two rational time values are equal if and only if their normalized representations represent the same rational number. Equality between `Small` and `Large` variants **MUST** compare the underlying values, not the variants.

Ordering is the standard rational ordering. Comparison between rationals in the small representation is performed by cross-multiplication in widened intermediate types to avoid overflow.

3.2.4 Units

The unit of musical time is the *whole note*. A quarter note has duration $\frac{1}{4}$. A triplet eighth has duration $\frac{1}{12}$. A quintuplet sixteenth inside a duplet half (an unusual but legal construct) has duration $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{20}$.

REQUIREMENT

The whole note **MUST** be the unit of musical time. The duration of a whole note is the rational $\frac{1}{1}$. All other durations are expressed as rationals relative to the whole note.

RATIONALE

The whole note is the only unit under which durations form a multiplicative algebra with 1 as identity. Quarter-note units would make “the duration of a whole note” equal to 4, which is awkward; PPQ-tick units would force a choice of resolution. The whole-note unit also matches the etymology of note-value names: a half note is half of a whole note.

3.3 Position and Duration as Distinct Types

The rational time representation is shared between two distinct types: `MusicalPosition` and `MusicalDuration`. Both wrap a `RationalTime`. Their distinction is enforced at the type level to prevent algebraic errors.

```
/// A point in musical time, relative to the origin of a time region.
#[derive(Clone, PartialEq, Eq, Hash, PartialOrd, Ord)]
pub struct MusicalPosition(RationalTime);
```

```

/// A span of musical time.
#[derive(Clone, PartialEq, Eq, Hash, PartialOrd, Ord)]
pub struct MusicalDuration(RationalTime);

impl Add<MusicalDuration> for MusicalPosition {
    type Output = MusicalPosition;
    // ...
}

impl Add<MusicalDuration> for MusicalDuration {
    type Output = MusicalDuration;
    // ...
}

impl Sub<MusicalPosition> for MusicalPosition {
    type Output = MusicalDuration;
    // ...
}

// MusicalPosition + MusicalPosition is intentionally not implemented.

```

REQUIREMENT

Implementations **MUST** provide distinct types for position and duration. The type system **MUST** prevent adding two positions. The difference of two positions **MUST** yield a duration; the sum of a position and a duration **MUST** yield a position.

RATIONALE

The illustrative Rust signatures derive `Clone`, not `Copy`, because `RationalTime` contains an `Arc<BigRational>` variant that cannot satisfy `Copy`. Implementations **MAY** provide `Copy` semantics through a compact inline-or-pointer-tagged representation whose `Large` case is a non-owning interior pointer rather than an `Arc`; such implementations satisfy the contract here. Conformance does not require `MusicalPosition` or `MusicalDuration` to be `Copy`.

3.4 Wall-Clock Time

Wall-clock time is the time measured by the playback engine, by SMPTE timecode, and by external audio and video sources. It is distinct from musical time and is represented as a fixed-point integer count of a sub-second unit.

```

/// A point in wall-clock time, in nanoseconds from a region origin.
/// 64-bit signed: range +/- ~292 years.
#[derive(Copy, Clone, PartialEq, Eq, Hash, PartialOrd, Ord)]
pub struct WallClockTime(i64); // nanoseconds

#[derive(Copy, Clone, PartialEq, Eq, Hash, PartialOrd, Ord)]

```

```
pub struct WallClockDuration(i64); // nanoseconds
```

REQUIREMENT

Wall-clock time **MUST** be a fixed-point integer count of nanoseconds. Floating-point wall-clock time is forbidden in stored data. Implementations **MAY** convert to floating-point for transient computation provided no stored or serialized value loses precision.

RATIONALE

Nanosecond resolution provides over four orders of magnitude more precision than the human ear can resolve and aligns with the resolution conventions of professional audio interfaces and SMPTE-derived clocks. The 292-year range is sufficient for any conceivable musical work.

3.4.1 Wall-Clock and Musical Time Coexist

A score event may be anchored in either musical or wall-clock time. Musical-time anchors are the default for ordinary notation; wall-clock anchors are used for film hit points, audio reference markers, fixed electronic parts in scores with live ensemble, and similar sync-critical content. Both anchor types may coexist in the same score, possibly on the same staff.

Conversion between musical and wall-clock time is performed through the tempo map (Section 3.7).

3.5 Time Anchors

Stored references to points in time **MUST** be expressed as *time anchors*: references to identified objects plus offsets. Absolute positions **MUST NOT** be stored where they could be invalidated by edits to the score.

```
pub enum TimeAnchor {
    /// Anchored to a specific event. Survives edits that do not
    /// delete the event.
    Event { id: EventId, offset: AnchorOffset },

    /// Anchored to a measure boundary. Survives measure reordering
    /// (the anchor follows the measure).
    Measure {
        id: MeasureId,
        position: MeasurePosition,
        offset: AnchorOffset,
    },

    /// Anchored to the start or end of a region.
    Region { id: RegionId, edge: RegionEdge, offset: AnchorOffset },

    /// Anchored to absolute wall-clock time within the score.
    /// Used for film and audio sync.
```

```

WallClock { time: WallClockTime },
}

/// An offset applied to an anchor target. The variant kind is
/// constrained by the target's enclosing region's time model.
pub enum AnchorOffset {
    /// Offset in musical time. Valid for targets in metric regions
    /// and for aleatoric regions whose anchoring discipline
    /// admits musical coordinates.
    Musical(MusicalDuration),

    /// Offset in wall-clock time. Valid for targets in proportional
    /// regions and for aleatoric regions whose anchoring discipline
    /// admits wall-clock coordinates.
    WallClock(WallClockDuration),

    /// No offset; the anchor refers to the target's reference point
    /// exactly. Valid in any region.
    Zero,
}

pub enum MeasurePosition { Start, End }
pub enum RegionEdge { Start, End }

```

REQUIREMENT

Stored *references to external time points* (cross-cutting endpoints, marker locations, attachment anchors, anything pointing at an object outside the referencing object's own voice and region) **MUST** use the `TimeAnchor` type. Absolute musical positions **MUST NOT** appear as such references in stored data: a reference to “measure 47, beat 3” that does not survive a measure insertion in measure 12 is a broken reference.

An event's own position within its owning voice and region **MAY** be stored as an `EventPosition` (carrying a `MusicalPosition` for events in metric regions, or a `WallClockTime` for events in proportional regions, etc.). Event-local positions are part of the event's intrinsic data, not references to other objects. Operations that shift a voice or region update the contained events' positions directly; they do not need anchor indirection because the containment relationship makes the dependency explicit.

Implementations **MAY** compute and cache absolute positions from anchors for performance; such caches **MUST** be invalidated when their underlying anchored objects change.

REQUIREMENT

The variant of an `AnchorOffset` **MUST** agree with the time model of the anchor target's enclosing region:

- For targets in metric regions: `AnchorOffset::Musical` or `AnchorOffset::Zero`.
- For targets in proportional regions: `AnchorOffset::WallClock` or `AnchorOffset::Zero`.

- For targets in aleatoric regions: any offset variant consistent with the region's `AleatoricAnchoringDiscipline` (Section 3.10.3).

Implementations **MUST** reject anchors whose offset variant contradicts the target region's time model.

3.5.1 Anchor Resolution

Resolving a time anchor to an absolute position is a pure function of the score graph: locate the anchored object, query its position within its region, and add the offset. Resolution **MUST** be deterministic.

3.5.2 Anchor Stability Under Edits

The chosen anchor type determines what edits the anchor survives:

- An `Event` anchor survives any edit that does not delete the anchored event. Edits that move the event (insertions before it, tempo changes) move the anchor with it.
- A `Measure` anchor survives insertions and deletions of other measures, and survives reordering. It is invalidated only by deletion of the anchored measure.
- A `Region` anchor survives edits within the region. It is invalidated only by deletion of the region.
- A `WallClock` anchor is unaffected by musical edits but may correspond to different musical positions after tempo changes.

REQUIREMENT

When an anchored object is deleted, anchors targeting it **MUST** be either re-anchored to a surviving object or marked as orphaned. The semantic operations chapter (Chapter 6) specifies re-anchoring rules per operation.

3.6 Time Signatures and Meter

A time signature defines the metric organization of a passage: where bar lines fall, where beats group, and what accent pattern is implied. Time signatures are objects in the score graph; they are not merely display numerator/denominator pairs.

```
pub struct TimeSignature {
    /// Display form. Determines what the performer sees.
    pub display: TimeSignatureDisplay,

    /// The total duration of one measure under this signature.
    pub measure_duration: MusicalDuration,

    /// Beat groupings within the measure. Drives beaming, accent
    /// inference, and the notational-decomposition pre-pass.
    pub beat_groups: Vec<BeatGroup>,
}
```

```

pub enum TimeSignatureDisplay {
    /// Standard: a numerator over a power-of-two denominator.
    Standard { numerator: u16, denominator: PowerOfTwo },

    /// Compound display: e.g., 3+2+3/8.
    Compound { numerators: Vec<u16>, denominator: PowerOfTwo },

    /// Irrational meter: denominator is not a power of two.
    /// e.g., 4/6, 5/12. The denominator names a non-power-of-two
    /// note value.
    Irrational { numerator: u16, denominator: NonZeroU16 },

    /// Mixed denominators: e.g., 1/4 + 1/8 + 1/16 displayed additively.
    MixedDenominators { components: Vec<(u16, NonZeroU16)> },

    /// No visible time signature (free meter or proportional).
    None,

    /// Custom symbol (cut time, common time, etc.) or
    /// grammar-specific.
    Symbolic(SymbolId),
}

pub struct BeatGroup {
    /// Duration of this beat group.
    pub duration: MusicalDuration,

    /// Subdivision hint for beaming and accent.
    pub subdivision: Option<MusicalDuration>,

    /// Accent strength relative to other beat groups in the measure.
    /// Higher values indicate stronger accents.
    pub accent: u8,
}

```

REQUIREMENT

The sum of the durations of a time signature's beat groups **MUST** equal the measure duration. Implementations **MUST** reject time signatures whose beat groups do not sum to the measure duration.

3.6.1 Irrational Meters

Time signatures whose denominator is not a power of two (e.g., 4/6, 5/12) are first-class. Their measure duration is computed as a rational: a 4/6 measure has duration $\frac{4}{6} = \frac{2}{3}$ whole notes. No special case in the time model; only the display layer treats the unusual denominator.

3.6.2 Polymetric and Polytemporal Scores

A score **MAY** have different time signatures concurrent on different staves. Each staff carries its own meter sequence; bar lines align where the rational positions coincide, and may misalign otherwise. The score graph chapter (Chapter 5) defines how concurrent meters are represented; this chapter establishes only that the time model supports them.

3.7 Tempo and the Tempo Map

The tempo map is the function mapping musical positions to wall-clock positions. It is a piecewise definition over musical time:

```
pub struct TempoMap {
    /// Piecewise segments, in non-overlapping musical-time order.
    pub segments: Vec<TempoSegment>,

    /// Tempo before the first segment, if any. None means the first
    /// segment's start_tempo applies from time zero.
    pub initial: Option<Tempo>,
}

pub struct TempoSegment {
    /// Start of this segment, as a TimeAnchor.
    pub start: TimeAnchor,

    /// End of this segment. None means "valid until the next segment
    /// in segments order, or until the end of the score if this is
    /// the final segment." Implementations resolve None lazily.
    pub end: Option<TimeAnchor>,

    /// Tempo at the start of this segment.
    pub start_tempo: Tempo,

    /// Tempo at the end of this segment.
    /// Required if shape != Constant; ignored (and SHOULD be None)
    /// if shape == Constant. A Constant segment holds start_tempo
    /// throughout.
    pub end_tempo: Option<Tempo>,

    /// Shape of the tempo change over this segment.
    pub shape: TempoShape,
}

pub enum TempoShape {
    /// Constant tempo throughout: end_tempo MUST be None or equal to
    /// start_tempo.
    Constant,

    /// Linear interpolation from start_tempo to end_tempo.
    Linear,

    /// Exponential (continuous-rate) interpolation from start_tempo
```

```

    /// to end_tempo.
    Exponential,

    /// Arbitrary curve, parameterized by control points. The control
    /// points define intermediate tempos between start_tempo and
    /// end_tempo.
    Curve(TempoCurve),
}

pub struct Tempo {
    /// Beats per minute. The beat unit is part of the tempo
    /// specification: a quarter-note BPM and a dotted-quarter BPM
    /// at the same numeric value differ in actual rate.
    pub bpm: f64,

    /// Note value taken as one beat.
    pub beat_unit: MusicalDuration,
}

```

REQUIREMENT

The tempo map's segments **MUST** be non-overlapping when resolved to absolute musical positions, and **MUST** appear in monotonically-increasing start order. Two adjacent segments where the earlier segment's resolved end equals the later segment's resolved start form a continuous tempo curve; the later segment's `start_tempo` **SHOULD** equal the earlier segment's resolved end tempo when continuity is intended.

A segment with `end == None` extends until the next segment's resolved start or, if no later segment exists, until the end of the score. Implementations **MUST** resolve open-ended segments deterministically.

At musical positions not covered by any segment (gaps between segments, or before the first segment when `initial` is `None`), the tempo **MUST** be the most recent applicable segment's terminating tempo, or the next applicable segment's `start_tempo` if no earlier segment exists.

3.7.1 Conversion

Conversion between musical and wall-clock time integrates the tempo map. For constant segments, conversion is multiplication. For linear and exponential segments, conversion has a closed-form solution. For curve segments, conversion is numerical.

REQUIREMENT

The tempo map **MUST** expose two conversion functions: `musical_to_wallclock` and `wallclock_to_musical`. Both **MUST** be deterministic: given identical tempo maps and inputs, they **MUST** produce identical outputs across runs and platforms. Numerical integration and inversion, where used, **MUST** use a deterministic algorithm with documented tolerance and iteration bounds.

Tempo curves **MUST** be monotonically increasing in musical time (i.e., tempo may vary but musical time always advances) so that `wallclock_to_musical` is single-valued. Implementations **MUST** reject tempo segments whose shape produces a non-monotonic mapping (such curves are musically meaningless).

OPEN QUESTION

The specific numerical-integration algorithm for `TempoShape::Curve` (Romberg integration, adaptive Gauss-Kronrod, or a normative method) and the specific root-finding algorithm for `wallclock_to_musical` are unresolved. Both algorithms affect canonical state and must therefore receive one of the dispositions in Appendix D Section D.11: normatively specified in this document or a named companion, profile-declared by versioned identifier, or explicitly marked non-canonical. Until that disposition is made, cross-implementation byte equality of derived canonical state (notably tempo-driven audio scheduling) **MUST NOT** be claimed.

RATIONALE

Determinism of tempo conversion is necessary for reproducible audio output, click-track stability, and SMPTE sync. Non-determinism here would produce playback drift that compounds over long pieces.

3.8 Sounding Duration and Notational Decomposition

An event has a single *sounding duration* (an exact rational). Its *notational decomposition* is the sequence of notehead values, augmentation dots, and ties used to draw that duration on the staff.

The decomposition is computed from the sounding duration by a deterministic pre-pass and stored as an attachment on the event. The attachment is overridable by the user; the pre-pass produces an inferred decomposition that the user may accept or replace.

3.8.1 The Notational Decomposition

```
pub struct NotationalDecomposition {
    /// Sequence of notated components. Their durations sum to the
    /// event's sounding duration.
    pub components: Vec<NotatedComponent>,

    /// Provenance, analogous to SpellingSource.
    pub source: DecompositionSource,
}

pub struct NotatedComponent {
    /// Base note value: whole, half, quarter, eighth, etc.
    pub base_value: NoteValue,
```

```

    /// Number of augmentation dots (0..=4 typical).
    pub dots: u8,

    /// Tuplet membership, if any.
    pub tuplet: Option<TupletId>,

    /// Whether this component is tied to the next.
    pub tied_to_next: bool,
}

pub enum DecompositionSource {
    UserChosen,
    Inferred,
    Imported { format: ForeignFormatId },
    Propagated { from: EventId },
}

```

The structure deliberately mirrors the spelling attachment model (Section 2.5): same sources, same precedence machinery, same pre-pass discipline.

3.8.2 The Decomposition Pre-Pass

When a score is loaded or edited, the decomposition pre-pass produces inferred decompositions for events that lack a higher-precedence attachment. The algorithm is, in outline:

1. For each event in time order within a voice and measure, examine the sounding duration and the current beat-group context.
2. Decompose the duration into the longest leading note value that fits within the current beat group without crossing a stronger accent boundary.
3. If the duration is not exhausted, produce a tie and continue decomposition from the next beat-group boundary.
4. Augmentation dots are applied where they shorten the decomposition without crossing an accent boundary that the dot would obscure.
5. Tuplet membership is recorded for events that fall within a tuplet group (Section 3.9).

REQUIREMENT

The decomposition pre-pass **MUST** be deterministic: given identical score graph and identical configuration, it **MUST** produce identical decompositions. The pre-pass **MUST** be incremental: an edit to one event **MUST** re-run the pre-pass only on the affected voice and measure, with propagation only as far as the context is altered.

OPEN QUESTION

The specific algorithm for selecting decompositions when multiple are valid (e.g., $\frac{3}{8}$ in $\frac{4}{4}$ may be a dotted quarter or a quarter tied to an eighth depending on placement) involves style choices. The default rules must be specified; user-configurable alternate

rule sets are likely needed. The notational-decomposition algorithm affects canonical state and must therefore receive one of the dispositions in Appendix D Section D.11: delivery as a normative algorithm in the Notational Decomposition companion specification, or profile-declared by versioned identifier. Until that disposition is made, the default-decomposition output is non-canonical and cross-implementation byte equality **MUST NOT** be claimed for derived layouts.

3.9 Tuplets as Grouping Objects

A tuplet is a grouping object in the score graph that records notational intent: that a span of events forms a triplet, quintuplet, or other irregular grouping with a visible bracket and number. Tuplets do not modify the sounding durations of their members; sounding durations are always concrete rationals.

```
pub struct Tuplet {
    pub id: TupletId,

    /// The actual:notated ratio. A triplet is 3:2; a quintuplet is
    /// 5:4; an irregular grouping like 7:4 has ratio 7 over 4.
    pub ratio: TupletRatio,

    /// Range of events in this tuplet, by event ID.
    pub members: Vec<EventId>,

    /// Display: bracket style, number placement, parentheses, etc.
    pub display: TupletDisplay,

    /// Nesting: if this tuplet is itself inside another, the parent.
    pub parent: Option<TupletId>,
}

pub struct TupletRatio {
    pub actual: u32,
    pub notated: u32,
}
```

3.9.1 Tuplet Nesting

Tuplets **MAY** nest to arbitrary depth. The sounding duration of an event inside nested tuplets is computed directly from its rational duration, not from the tuplet ratios; the ratios are notational and analytical.

RATIONALE

Storing sounding durations directly (rather than as base-value times a stack of tuplet ratios) keeps the time model regular: every event has a single rational duration, full stop. Moving a tuplet, splitting it, or deleting it never alters the durations of its mem-

bers. Triplets thus become a pure annotation layer, which dramatically simplifies edit operations.

3.9.2 Triplet Consistency

REQUIREMENT

A triplet's notated ratio **MUST** be consistent with the sounding durations of its members: the sum of member durations **MUST** equal the duration that the triplet's notated value indicates when scaled by the ratio. For example, a 3:2 eighth-note triplet of three members has total sounding duration $\frac{1}{4}$; the sum of the three member durations **MUST** equal $\frac{1}{4}$.

This is a structural invariant. Edit operations that would violate it **MUST** either adjust member durations or be rejected. The semantic operations chapter specifies which.

3.10 Region Time Models

A region of the score declares one of three time models. The declared model determines how positions within the region are interpreted, displayed, and played back.

```
pub enum RegionTimeModel {
    /// Metric time: measures, beats, exact rational positions.
    /// The dominant model for Western notated music.
    Metric(MetricTimeModel),

    /// Proportional time: events placed in space, where horizontal
    /// position is wall-clock time. No measures, no beats.
    Proportional(ProportionalTimeModel),

    /// Aleatoric time: events with partial or unspecified ordering.
    Aleatoric(AleatoricTimeModel),
}
```

3.10.1 Metric Time

```
pub struct MetricTimeModel {
    /// Sequence of meter changes within this region.
    pub meters: Vec<MeterChange>,

    /// Tempo map for this region. May inherit from the score.
    pub tempo: TempoMapReference,
}
```

Metric regions have measures, beats, and time signatures as described in Sections 3.6 and 3.7. Positions within them are `MusicalPosition` values.

3.10.2 Proportional Time

```
pub struct ProportionalTimeModel {
    /// Total wall-clock duration of the region.
    pub duration: WallClockDuration,

    /// Horizontal space per second. Drives the engraver's spacing.
    /// Stored as a layout hint; the actual spacing is decided by the
    /// constraint solver.
    pub space_per_second: SpaceUnit,

    /// Optional reference grid markings (vertical lines every N
    /// seconds, for example) for performer reference.
    pub grid: Option<ProportionalGrid>,
}
```

Events in proportional regions carry `WallClockTime` positions directly. There are no measures, no beats, no rational durations. Notational duration is computed from wall-clock duration by the notational decomposition pre-pass using a proportional-mode rule set, which generally produces unbeamed events with explicit duration notations or graphic-duration symbols.

REQUIREMENT

Events in a proportional region **MUST** use `WallClockTime` for positions and `WallClockDuration` for durations. The tempo map **MUST NOT** be applied to convert proportional-region positions; they are already in wall-clock time.

3.10.3 Aleatoric Time

```
pub struct AleatoricTimeModel {
    /// Ordering constraints among events: a directed acyclic graph
    /// where an edge a -> b means "a precedes b."
    pub ordering: EventOrderingDAG,

    /// Anchoring discipline: declares what event coordinate kinds
    /// this region permits and whether duration-bound kinds may
    /// be mixed. See Section~\ref{sec:time:aleatoric-anchoring}.
    pub anchoring: AleatoricAnchoringDiscipline,

    /// Optional per-event interval bounds. An event with start
    /// bounds [t1, t2] may begin at any time in that window.
    /// Bound kinds are constrained by the anchoring discipline.
    pub bounds: HashMap<EventId, EventBounds>,

    /// Approximate or maximum total duration, for layout.
    pub duration_hint: WallClockDuration,
}

pub enum AleatoricAnchoringDiscipline {
    /// Events in this region carry musical-time coordinates only.
    Musical,
```

```

    /// Events in this region carry wall-clock coordinates only.
    WallClock,

    /// Events may carry either kind, but each event's position and
    /// duration kinds must agree with each other. Mixed-kind
    /// duration bounds are not permitted.
    EitherPerEvent,

    /// Events may carry either kind freely, and duration bounds
    /// may mix kinds (one bound musical, the other wall-clock).
    /// Used for free-time scores synchronized to external media.
    FreelyMixed,
}

pub struct EventBounds {
    pub start: Option<TimeBounds>,
    pub end: Option<TimeBounds>,
}

pub enum TimeBounds {
    MusicalRange { min: MusicalPosition, max: MusicalPosition },
    WallClockRange { min: WallClockTime, max: WallClockTime },
    Unbounded,
}

```

Aleatoric regions support both unordered event collections (any permutation of events is valid) and time-bracket notation (events with specified start and end windows, à la Cage's later notation). The DAG encoding admits both as cases.

Anchoring Discipline

REQUIREMENT

Every aleatoric region **MUST** declare an `AleatoricAnchoringDiscipline`. Events within the region **MUST** carry coordinate kinds consistent with the discipline:

- Musical: events use `EventPosition::Musical` and `EventDuration::Musical` (or `EventDuration::Indeterminate` whose bounds are musical).
- WallClock: events use `EventPosition::WallClock` and `EventDuration::WallClock` (or `EventDuration::Indeterminate` whose bounds are wall-clock).
- EitherPerEvent: events choose per-event, but an event's position and duration kinds **MUST** agree, and `DurationBounds` **MUST** use a single `ConcreteDuration` variant.
- FreelyMixed: no agreement required; `DurationBounds` **MAY** mix kinds.

Implementations **MUST** reject events whose coordinate kinds violate the region's declared discipline.

REQUIREMENT

The ordering DAG **MUST** be acyclic. Cycles **MUST** be rejected at construction. Two events with no path between them in the DAG are unordered.

RATIONALE

A DAG with optional interval bounds expresses every aleatoric form the project intends to support: unordered collections (empty DAG), partial orderings (sparse DAG), strict orderings (chain DAG), and time-brackets (DAG plus bounds). Higher-level constructs (mobile-form notation, statistical scores) can be built atop this primitive in later specifications without altering the core.

3.10.4 Concurrent Time Models

Different staves in the same score **MAY** declare different time models. A metric solo line over a proportional aleatoric texture, for example, is expressible: each staff's region declares its own model. Synchronization between concurrent time models is by wall-clock time or by explicit cue points (see Chapter 5).

3.11 The Notion of “Now”

The core does not define playback state; it does not have a notion of “current position” or “transport time.” Such state belongs to the audio engine, which consumes the core's time model. The core defines only the static time structure of the score.

3.12 Forward References

- Event identifiers (`EventId`), measure identifiers (`MeasureId`), and region identifiers (`RegionId`) are defined in Chapter 5.
- The full event taxonomy (pitched, unpitched, rest, indeterminate, trajectory) is defined in Chapter 5.
- Re-anchoring rules for time anchors when their targets are deleted are defined per-operation in Chapter 6.
- `SpaceUnit`, `TupletDisplay`, and other purely display-oriented types are defined in Chapter 7.

Tuning Systems and Pitch Spaces

This chapter specifies the three intertwined registries that complete the primitives layer: *pitch spaces* (the analytical universes in which scale-position arithmetic lives), *accidental registries* (the catalogs of position-modifying glyphs), and *tuning systems* (the frequency-resolution mechanisms). It also specifies the hierarchical resolution rules that determine which tuning applies to each pitch in a score.

4.1 Design Principles

Pitch space and tuning system are distinct. A pitch space defines what pitches exist and how they relate analytically. A tuning system defines what frequencies those pitches produce. The two are independent: a single pitch space (e.g., CMN) admits many tuning systems (12-TET, meantone, well-temperaments, just intonation, Pythagorean); a single tuning structure may be anchored at different reference frequencies.

Accidentals belong to pitch spaces. An accidental defines a *position modification* within a pitch space. The tuning system handles the frequency consequences of that modification. The chromatic semitone implied by a sharp has different cent-values in 12-TET, quarter-comma meantone, and Pythagorean tuning, but it is the same accidental denoting the same position.

Reference pitch is score-level, not tuning-level. A score declares a reference pitch (position plus frequency); the tuning system defines structure relative to that reference. “12-TET at A=440” and “12-TET at A=415” share the same tuning system; the reference differs.

Resolution is hierarchical. Tuning, pitch space, and accidental registry are resolved by walking outward from the pitch: pitch, voice, staff, region, score. Inheritance markers permit any level to defer to the enclosing scope.

Built-in defaults, extensible by users. The core ships a baseline catalog of pitch spaces, tuning systems, and accidental registries sufficient for the common 95% case. Scores and grammar plugins extend these without forking.

SMuFL primary, custom fallback. Glyph references default to SMuFL codepoints. Custom glyphs are supported as fallback. SMuFL version is recorded per score.

4.2 Pitch Spaces

A pitch space is the analytical universe in which scale-degree and interval relationships are defined. It is the algebra; the tuning system is the physics.

```
pub struct PitchSpace {
    pub id: PitchSpaceId,

    /// Human-readable name (e.g., "CMN 12-tone chromatic").
    pub name: String,

    /// Optional description and provenance notes.
    pub description: Option<String>,

    /// What positions exist and how they are structured.
    pub positions: PositionStructure,

    /// How intervals between positions are defined.
    pub interval_algebra: IntervalAlgebra,

    /// The accidental registry valid in this pitch space.
    pub accidental_registry: AccidentalRegistryId,

    /// The nominal registry: named position-letters
    /// (e.g., A-G for CMN).
    pub nominal_registry: NominalRegistryId,

    /// Rules governing how pitches transpose within this space.
    pub transposition: TranspositionBehavior,

    /// Default rules for the spelling pre-pass when this pitch space
    /// is active.
    pub spelling_rules: SpellingRuleSet,
}
```

4.2.1 Position Structure

The `PositionStructure` declares the shape of the pitch space. Three families cover the vast majority of useful cases; a fourth is the escape hatch for grammars defined entirely by plugin.

```
pub enum PositionStructure {
    /// Chromatic: a fixed number of equally-numbered positions per
    /// octave, with no hierarchical diatonic substructure.
    /// Includes EDOs and serial 12-tone.
    Chromatic { positions_per_octave: u16 },

    /// Diatonic over chromatic: a smaller set of named nominals
    /// (e.g., A-G for CMN) plus accidental modifications reaching
    /// the full chromatic range. CMN's structure.
    DiatonicOverChromatic {
        nominals_per_octave: u16,
        chromatic_positions_per_octave: u16,
    }
}
```

```

    /// Mapping from each nominal to its position in the
    /// chromatic layer (e.g., C=0, D=2, E=4, F=5, G=7, A=9, B=11
    /// for CMN).
    nominal_to_chromatic: Vec<u16>,
},

/// Just intonation lattice: positions defined by exact rational
/// ratios from a tonic, organized along prime-axis dimensions
/// (3-limit, 5-limit, 7-limit, etc.).
JiLattice {
    limit: u8, // prime limit
    generators: Vec<JiRatio>, // one per prime dimension
},

/// Grammar-defined: structure is opaque to the core and resolved
/// by a grammar plugin. Used for maqam, gamelan, raga frameworks,
/// and other systems whose position structure is not flat or
/// hierarchical in the above senses.
Registered(PositionStructureRegistryId),
}

```

REQUIREMENT

The `DiatonicOverChromatic` variant **MUST** have a `nominal_to_chromatic` mapping of length equal to `nominals_per_octave`, with each entry strictly less than `chromatic_positions_per_octave`. The mapping **MUST** be strictly increasing.

4.2.2 Interval Algebra

An interval algebra defines how distances between positions are computed and how transposition is realized. For chromatic spaces the algebra is integer arithmetic modulo the position count; for diatonic-over-chromatic the algebra distinguishes diatonic from chromatic intervals (a diatonic third may correspond to three or four chromatic semitones depending on quality).

```

pub enum IntervalAlgebra {
    /// Single-axis integer arithmetic. Two positions are subtracted
    /// to yield a single integer interval.
    Chromatic,

    /// Two-axis arithmetic distinguishing diatonic and chromatic
    /// components. CMN's algebra: a major third is (3 diatonic steps,
    /// 4 chromatic steps); a diminished fourth is (3 diatonic steps,
    /// 4 chromatic steps) -- distinguishable from the major third only
    /// by spelling intent.
    DiatonicChromatic,

    /// Multi-axis JI lattice arithmetic.
    JiVector { dimensions: u8 },

    /// Grammar-defined algebra.
    Registered(IntervalAlgebraRegistryId),
}

```

```
}
```

4.2.3 Nominal Registry

A nominal registry catalogs the named position-letters of a pitch space. For CMN this is the seven-element set $\{C, D, E, F, G, A, B\}$. For 12-tone serial music the nominal registry may simply be the integers 0 through 11. For maqam the nominals are the names of maqam degrees.

```
pub struct NominalRegistry {
  pub id: NominalRegistryId,
  pub nominals: Vec<NominalDefinition>,
}

pub struct NominalDefinition {
  pub id: NominalId,

  /// Canonical name.
  pub name: String,

  /// Display glyph, typically a letter or symbol.
  pub display: GlyphReference,

  /// Default position in the pitch space's chromatic layer, if
  /// applicable.
  pub default_position: Option<u16>,
}
```

4.2.4 Transposition Behavior

Transposition behavior governs how scale positions move under interval operations and how spellings follow.

```
pub enum TranspositionBehavior {
  /// Diatonic transposition: shift by N diatonic steps within a
  /// chosen key. Accidentals adjust to fit the destination key.
  Diatonic,

  /// Chromatic transposition: shift by N chromatic steps. Spellings
  /// follow rules of the SpellingRuleSet.
  Chromatic,

  /// Compound: support both, parameterized at the operation site.
  Compound,

  /// Grammar-defined.
  Registered(NominalRegistryId),
}
```

4.2.5 Spelling Rule Sets

A pitch space carries a default spelling rule set that the spelling pre-pass (Section 2.5.4) consults when inferring spellings for unspelled pitches.

```
pub struct SpellingRuleSet {
    pub id: SpellingRuleSetId,
    pub name: String,

    /// Algorithmic family this rule set belongs to. The specific
    /// algorithm is referenced by id and resolved against the
    /// implementation's registered spelling algorithms.
    pub algorithm: SpellingAlgorithmId,

    /// Algorithm-specific parameters.
    pub parameters: SpellingParameters,
}
```

OPEN QUESTION

The catalog of registered spelling algorithms is the subject of the open question parked in Section 2.5.4. Candidates include Longuet-Higgins line-of-fifths distance, Temperley preference rules, Cambouropoulos pitch-spelling, and Meredith PS13. The chosen default and the parameter schemas are normative once decided.

4.3 Accidental Registries

An accidental registry catalogs the position-modifying glyphs valid within a pitch space. Each accidental defines a glyph, a position modification, and engraving metadata.

```
pub struct AccidentalRegistry {
    pub id: AccidentalRegistryId,
    pub name: String,
    pub accidentals: Vec<AccidentalDefinition>,

    /// Whether scores may extend this registry with score-local
    /// accidentals. Most registries permit this; some (e.g., a
    /// closed conformance profile) may not.
    pub extensible: bool,
}

pub struct AccidentalDefinition {
    pub id: AccidentalId,

    /// Canonical name. Used for serialization, accessibility, and
    /// foreign-format export.
    pub name: String,

    /// Glyph reference, typically SMuFL.
    pub glyph: GlyphReference,
```

```

    /// The position modification this accidental applies.
    pub modification: PitchSpaceModification,

    /// Engraving metadata.
    pub engraving: AccidentalEngraving,

    /// Combination behavior with other accidentals.
    pub combination: AccidentalCombination,
}

```

4.3.1 Pitch Space Modifications

```

pub enum PitchSpaceModification {
    /// CMN-style integer chromatic alteration. -2 = double flat,
    /// -1 = flat, +1 = sharp, +2 = double sharp, etc.
    CmnChromatic(i8),

    /// Integer step offset in an EDO position space.
    EdoSteps(i16),

    /// Modification expressed as an exact rational ratio. Used for
    /// JI accidentals such as the HEJI syntonic-comma symbols
    /// (81/80) and septimal-comma symbols (64/63).
    JiRatio { numerator: i32, denominator: NonZeroU32 },

    /// Modification expressed in cents. Used for Sagittal-style
    /// precise microtonal accidentals.
    Cents(f64),

    /// Modification defined by a grammar plugin.
    Registered(ModificationRegistryId),
}

```

REQUIREMENT

An accidental’s modification **MUST** be expressible in the interval algebra of every pitch space that references its registry. A `CmnChromatic` modification is valid only in spaces with `DiatonicChromatic` or compatible algebra; an `EdoSteps` modification is valid only in `Chromatic` or `Registered` spaces with appropriate compatibility. Implementations **MUST** reject scores referencing an accidental in a space whose algebra does not admit the modification.

4.3.2 Accidental Engraving Metadata

```

pub struct AccidentalEngraving {
    /// Bounding box in staff-space units, relative to the glyph’s
    /// anchor point.
    pub bounding_box: BoundingBox,

    /// Where the glyph attaches to the note: typically the geometric

```

```

/// center or a custom anchor for compound glyphs.
pub anchor: AnchorPoint,

/// Advance width for horizontal spacing computations.
pub advance_width: SpaceUnit,

/// Stacking order when multiple accidentals attach to one note.
/// Lower values are placed closer to the notehead.
pub stacking_order: i32,

/// Whether this glyph should be drawn with parentheses by default
/// (e.g., editorial or cautionary accidentals).
pub default_parenthesized: bool,
}

```

4.3.3 Combination Behavior

Most accidentals do not combine; a note carries one accidental at a time. Some systems (HEJI, certain microtonal notations) permit stacking multiple accidental glyphs on a single note to express compound modifications.

```

pub enum AccidentalCombination {
    /// Stands alone; replaces any prior accidental on the same note.
    Solitary,

    /// May stack with members of the listed compatibility groups.
    /// Stacking order is determined by the engraving metadata.
    Stacking { compatible_groups: Vec<AccidentalGroupId> },
}

```

4.3.4 Score-Local Extensions

A score **MAY** extend a referenced accidental registry with additional accidental definitions, provided the registry's `extensible` flag is true. Extensions are stored on the score and override or augment the base registry during resolution.

```

pub struct ScoreAccidentalExtensions {
    pub base: AccidentalRegistryId,
    pub additions: Vec<AccidentalDefinition>,
    pub overrides: Vec<AccidentalDefinition>,
}

```

4.4 Glyph References and SMuFL

The Standard Music Font Layout (SMuFL) is the normative source for glyph references in the core. Every standard accidental, notehead, articulation, dynamic, and ornament has a SMuFL codepoint. Non-standard glyphs are supported via custom glyph references.

```

pub enum GlyphReference {

```

```

    /// SMuFL codepoint, resolved against the active SMuFL font.
    Smufl(u32),

    /// Custom glyph defined by the score or a plugin.
    Custom(CustomGlyphId),

    /// Composite glyph: multiple glyph references rendered as a
    /// single accidental. Used for compound HEJI symbols and similar.
    Composite(Vec<GlyphReference>),
}

```

4.4.1 SMuFL Versioning

REQUIREMENT

Every score **MUST** declare the SMuFL version it targets. Resolving a SMuFL glyph reference whose codepoint is not present in the active font’s SMuFL version **MUST** produce a deterministic fallback (the reference SMuFL substitution glyph, or a clearly-marked missing-glyph indicator); it **MUST NOT** silently fail.

```

pub struct SmuflVersionRequirement {
    /// Minimum SMuFL version required by this score.
    pub minimum: SmuflVersion,

    /// SMuFL version this score was authored against. Used for
    /// detecting whether newer glyphs are in use.
    pub authored_against: SmuflVersion,
}

```

4.5 Tuning Systems

A tuning system maps positions in a pitch space to frequencies. Given a reference pitch (position plus frequency) and the system’s resolution rules, every position in the space resolves to a determinate frequency.

```

pub struct TuningSystem {
    pub id: TuningSystemId,

    /// Human-readable name.
    pub name: String,

    /// The pitch space whose positions this tuning resolves.
    pub pitch_space: PitchSpaceId,

    /// How the tuning resolves positions to frequencies, given a
    /// reference pitch. The reference itself is supplied separately
    /// (see Section~\ref{sec:tuning:reference}).
    pub resolution: TuningResolution,
}

```

```

    /// Optional historical or provenance notes.
    pub description: Option<String>,
}

```

4.5.1 Tuning Resolution

```

pub enum TuningResolution {
    /// N-tone equal temperament: each step is the Nth root of the
    /// octave ratio. Includes 12-TET when N=12.
    EqualTemperament { divisions_per_octave: u16 },

    /// Explicit per-position ratios. Each entry is a ratio relative
    /// to the reference position.
    PerPositionRatios(Vec<PositionRatio>),

    /// Procedural definition: a registered tuning function that
    /// computes frequencies from a reference. Historical temperaments
    /// (Werckmeister, Vallotti, meantone variants) live here.
    Function {
        function: TuningFunctionId,
        parameters: TuningParameters,
    },

    /// A base resolution with per-position overrides. Used for
    /// split-accidental keyboards and similar irregular tunings.
    Overlay {
        base: Box<TuningResolution>,
        overrides: Vec<PositionOverride>,
    },

    /// Imported from a tuning file (Scala .scl, MIDI Tuning Standard,
    /// etc.). The data is captured at import; the original file is
    /// referenced for provenance but is not required at runtime.
    Imported {
        format: TuningFileFormat,
        data: Arc<ImportedTuningData>,
    },

    /// Adaptive tuning: resolution depends on the harmonic context
    /// at the moment of sounding. Used for adaptive JI and similar
    /// systems.
    Adaptive {
        function: AdaptiveTuningFunctionId,
        parameters: AdaptiveTuningParameters,
    },
}

```

4.5.2 The Resolution Contract

REQUIREMENT

Every tuning resolution **MUST** expose a deterministic function from pitch-space position (plus, where applicable, harmonic context) to frequency in Hertz, given a reference pitch. Determinism **MUST** hold across runs and platforms; numerical computations **MUST NOT** depend on floating-point rounding modes.

4.5.3 Adaptive Tuning

Adaptive tuning systems compute frequency from position *plus harmonic context*. In a 5-limit adaptive JI system, for example, the frequency of a given E depends on whether it is the major third of a C chord, the perfect fifth of an A chord, or a passing tone between other harmonic interpretations.

```
pub struct HarmonicContext {
    /// Concurrently sounding pitches at the moment of resolution.
    pub concurrent: Vec<PitchId>,

    /// Recently sounding pitches, with decay weighting.
    pub recent: Vec<(PitchId, f64)>,

    /// The active key or modal center, if known.
    pub key_context: Option<KeyContext>,

    /// User-supplied context hints attached to the score.
    pub hints: Vec<ContextHint>,
}
```

The harmonic context is constructed by the audio engine (or by analysis tooling) and passed to the tuning resolution function. Static tuning systems ignore it; adaptive tuning systems consume it.

REQUIREMENT

Adaptive tuning resolution **MUST** be a pure function of position and harmonic context. Implementations **MAY** cache resolution results, but **MUST** invalidate caches when the harmonic context changes.

4.6 Reference Pitch

The reference pitch is the anchor that converts the tuning system's structural ratios into absolute frequencies. It is a property of the score, not of the tuning system.

```
pub struct ReferencePitch {
    /// The pitch-space position chosen as the reference.
    /// Conventionally A4 in CMN, but configurable.
    pub position: PitchSpacePosition,
```

```

    /// The frequency in Hertz assigned to that position.
    /// Conventional values: 440 Hz (modern standard), 415 Hz (Baroque),
    /// 430 Hz (early Classical), 442 Hz (some modern orchestras).
    pub frequency_hz: f64,
}

```

REQUIREMENT

Every score **MUST** declare a reference pitch. The reference pitch **MUST** be expressible as a valid position within the score’s default pitch space. The frequency **MUST** be positive and finite.

4.6.1 Multiple Reference Pitches

A score **MAY** declare additional reference pitches at the staff or region level, overriding the score default. This supports mixed-instrumentation scores in which different instruments use different concert-pitch references (e.g., a piece for period and modern instruments with different A frequencies).

4.7 Score Tuning Context and Hierarchical Resolution

The *score tuning context* is the top-level structure declaring the tuning environment for a score. Per-scope overrides extend it.

```

pub struct ScoreTuningContext {
    /// Default pitch space for the score.
    pub default_pitch_space: PitchSpaceId,

    /// Default tuning system. Its pitch_space must match (or be
    /// compatible with) the default_pitch_space.
    pub default_tuning_system: TuningSystemId,

    /// The reference pitch for the score.
    pub reference: ReferencePitch,

    /// Score-local accidental registry extensions.
    pub accidental_extensions: Vec<ScoreAccidentalExtensions>,

    /// SMuFL version targeting.
    pub smufl: SmuflVersionRequirement,

    /// Per-scope overrides, in the order they apply.
    pub overrides: Vec<TuningOverride>,
}

pub struct TuningOverride {
    pub scope: TuningScope,
    pub pitch_space: Option<PitchSpaceId>,
    pub tuning_system: Option<TuningSystemId>,
}

```

```

    pub reference: Option<ReferencePitch>,
  }

  pub enum TuningScope {
    Voice(VoiceId),
    Staff(StaffId),
    Region(RegionId),
    Range { start: TimeAnchor, end: TimeAnchor, voices: VoiceSelector },
  }

```

4.7.1 Resolution Order

For a given pitch, resolution proceeds by walking outward through scopes from most specific to most general until each component (pitch space, tuning system, reference) is determined.

REQUIREMENT

Resolution **MUST** proceed in the following order, halting at the first scope that supplies a non-inherited value for each component:

1. The pitch's own `AcousticPitch` (an explicit `TuningReference::Explicit` or `AcousticRealization::AbsoluteHz` short-circuits the walk).
2. The voice containing the pitch.
3. The staff containing the voice.
4. Each region enclosing the pitch, from innermost to outermost.
5. The score's default tuning context.

Each of the three components (pitch space, tuning system, reference) is resolved independently along this chain.

RATIONALE

Independent per-component resolution allows a passage to override only the parts that differ. A period-instrument inset in a modern orchestra score may override only the reference frequency (415 Hz instead of 440 Hz) while inheriting the pitch space and tuning structure from the parent context. Conversely, a microtonal passage in an otherwise 12-TET score may override the pitch space and tuning system but inherit the reference.

4.7.2 Compatibility Constraints

REQUIREMENT

When a tuning override is applied, the resolved tuning system's declared `pitch_space` **MUST** either equal the resolved pitch space or be declared compatible with it via a registered compatibility mapping. Implementations **MUST** reject configurations in which the resolved pitch space and the resolved tuning system's pitch space disagree without a compatibility mapping.

4.8 Built-in Catalog

The core ships with a baseline catalog of pitch spaces, tuning systems, and accidental registries sufficient for the common case. The catalog is normative: a conforming implementation **MUST** provide these identifiers with the specified semantics.

4.8.1 Built-in Pitch Spaces

Identifier	Description
cmn-12	Common Music Notation, 7 diatonic nominals (A–G) over 12 chromatic positions. Standard sharps, flats, double-sharps, and double-flats. The default for Western tonal and post-tonal music.
cmn-24	CMN extended with 24-EDO quarter-tone accidentals. Used for Arabic-derived microtonal practice in CMN-compatible notation.
edo-19	19-tone equal division of the octave. Notable for its closer-to-just major thirds.
edo-22	22-tone equal division. Distinguishes harmonic intervals collapsed in 12-TET.
edo-31	31-tone equal division. Approximates quarter-comma meantone almost exactly.
edo-53	53-tone equal division. Excellent approximation of 5-limit JI.
edo-72	72-tone equal division. Common in research and contemporary microtonal practice.
ji-5limit	5-limit just intonation lattice with HEJI accidentals. Two-dimensional (prime axes 3, 5).
ji-7limit	7-limit JI lattice with HEJI accidentals. Three-dimensional.
ji-11limit	11-limit JI lattice with HEJI accidentals. Four-dimensional.
maqam-base	Skeletal maqam framework with quarter-flat and quarter-sharp accidentals. Deep coverage of specific maqamat is the province of grammar plugins.
gamelan-slendro	Five-tone slendro framework.
gamelan-pelog	Seven-tone pelog framework.

4.8.2 Built-in Tuning Systems

Identifier	Description
tet-12	12-tone equal temperament. The default.
tet-19	19-tone equal temperament.
tet-22	22-tone equal temperament.
tet-31	31-tone equal temperament.
tet-53	53-tone equal temperament.
tet-72	72-tone equal temperament.

Identifier	Description
<code>pythagorean</code>	Pure-fifth (3:2) Pythagorean tuning.
<code>meantone-1/4-comma</code>	Quarter-comma meantone (pure major thirds).
<code>meantone-1/6-comma</code>	Sixth-comma meantone.
<code>meantone-1/5-comma</code>	Fifth-comma meantone.
<code>werckmeister-iii</code>	Werckmeister III well temperament.
<code>werckmeister-iv</code>	Werckmeister IV well temperament.
<code>vallotti</code>	Vallotti well temperament.
<code>kirnberger-ii</code>	Kirnberger II well temperament.
<code>kirnberger-iii</code>	Kirnberger III well temperament.
<code>young-ii</code>	Thomas Young’s second temperament.
<code>ji-static-5limit-C</code>	Static 5-limit JI anchored to C tonic.
<code>ji-static-5limit-G</code>	Static 5-limit JI anchored to G.
<code>ji-static-5limit-D</code>	Static 5-limit JI anchored to D.
<code>ji-adaptive-5limit</code>	Adaptive 5-limit JI; resolves based on harmonic context.

REQUIREMENT

All built-in catalog identifiers **MUST** resolve in a conforming implementation. The semantics specified above are normative. Implementations **MAY** provide additional built-ins; they **MUST NOT** redefine the semantics of those listed.

4.8.3 Default Score Configuration

A newly-created score with no explicit tuning context **MUST** default to:

- Pitch space: `cmn-12`.
- Tuning system: `tet-12`.
- Reference pitch: $A_4 = 440$ Hz.
- SMuFL version: the highest SMuFL version supported by the implementation.

4.9 Forward References

- `VoiceId`, `StaffId`, `RegionId`, `VoiceSelector`, and the score graph’s hierarchical organization are defined in Chapter 5.
- `KeyContext` and the harmonic-context construction machinery are partially defined in Chapter 5 and completed in the audio engine specification (out of scope for this document).
- The specific spelling algorithms referenced by `SpellingAlgorithmId` remain an open question; see the related open question in Section 2.5.4.

The Score Graph

This chapter specifies the score graph: the in-memory representation of all musical content in a score. The graph is the canonical truth about the music; layout, serialization, and editing operations are downstream projections and consumers of it. This chapter is the largest in the specification because it threads together every primitive defined in Chapters 2–4 into a coherent whole.

5.1 Design Principles

Hybrid topology. The graph is a tree of containment overlaid with cross-cutting structures that hold references. The tree determines existence: deleting a parent deletes its descendants. Cross-cutting structures hold references that may become dangling and require re-anchoring per the rules defined in this chapter and in Chapter 6.

Canvas before staff. The spatial root of a score is the canvas, a 2D space partitioned into regions. Staves exist inside regions, not as a top-level concept. This permits regions of free graphic content to coexist with regions of staff-based CMN without forcing one paradigm to subsume the other.

Arena storage. Events are stored in a flat arena owned by the score. Voices, cross-cutting structures, and analysis layers hold event identifiers, not events. Arena storage enables fast bulk iteration, simple cross-cutting reference, and clean CRDT semantics.

Parts are projections. A part is a view onto the score defined by a part-extraction specification. Parts do not live in the canvas; they are separate objects that select and reshape canvas content for printed extraction.

Typed identifiers. Every named object kind has its own identifier type. Cross-kind confusion is a compile-time error.

Identifiers are stable and deterministic. Identifiers are assigned at creation, never reassigned, and generated by a scheme that is deterministic per replica.

5.2 Top-Level Score Structure

The score is the root object. All other objects in the specification exist within or reference into a score.

```
pub struct Score {
```

```

/// Bibliographic and authorship metadata.
pub metadata: ScoreMetadata,

/// The spatial root: regions and their staff/graphic content.
pub canvas: Canvas,

/// Abstract instrument definitions used by staves.
pub instruments: Vec<Instrument>,

/// Globally-identified staves of the score. Each region's
/// staff instances reference these by StaffId.
pub staves: Vec<Staff>,

/// Optional staff groupings (grand staves, choral groups,
/// bracketed orchestral sections). Used by engraving and parts.
pub staff_groups: Vec<StaffGroup>,

/// Per-part view definitions for extraction.
pub parts: Vec<PartDefinition>,

/// Cross-cutting structures: slurs, ties, hairpins, markers,
/// graphic gestures, comments, and analytical annotations.
pub cross_cutting: CrossCuttingRegistry,

/// Conflict records produced by the canonical reduction
/// (Chapter~\ref{ch:semops}). Conflict records persist until
/// resolved or dismissed by explicit operations.
pub conflicts: ConflictRegistry,

/// Tuning systems, pitch spaces, and reference pitch.
pub tuning_context: ScoreTuningContext,

/// The score-level tempo map. Regions may declare local tempo
/// maps that override this.
pub tempo_map: TempoMap,

/// Flat arena of all events in the score.
pub events: EventArena,

/// Spelling attachments index (see Chapter 2).
pub spelling_attachments: SpellingAttachmentIndex,

/// Notational decomposition attachments index (see Chapter 3).
pub decomposition_attachments: DecompositionAttachmentIndex,

/// Analysis layers and view definitions.
pub analysis_layers: Vec<AnalysisLayer>,
pub views: Vec<ViewDefinition>,

/// Replica identifier and identifier generation state.
pub identity: IdentityContext,
}

```

```

pub struct StaffGroup {
    pub id: StaffGroupId,
    pub name: Option<String>,
    pub kind: StaffGroupKind,
    pub members: Vec<StaffId>,
}

pub enum StaffGroupKind {
    /// Grand staff: typically piano, organ, harp.
    GrandStaff,
    /// Bracketed group: orchestral sections.
    Bracket,
    /// Sub-bracket: divisi within a section.
    SubBracket,
    /// Choral group: SATB and similar.
    Choral,
    /// Custom group defined by a layout extension.
    Registered(StaffGroupKindRegistryId),
}

```

5.2.1 Score Metadata

```

pub struct ScoreMetadata {
    pub title: Option<String>,
    pub subtitle: Option<String>,
    pub composer: Option<String>,
    pub lyricist: Option<String>,
    pub arranger: Option<String>,
    pub copyright: Option<String>,
    pub creation_timestamp: Timestamp,
    pub modification_timestamp: Timestamp,
    pub additional: Vec<MetadataEntry>,
}

pub struct MetadataEntry {
    pub key: String,
    pub value: MetadataValue,
}

```

The metadata structure is deliberately small. Extended bibliographic, catalog, and editorial-apparatus metadata is the province of foreign-format mappings and analytical layers, not the core.

5.3 Identifiers

Every named object in the graph carries a typed 128-bit identifier.

```

pub struct EventId(u128);
pub struct VoiceId(u128);
pub struct StaffId(u128);
pub struct StaffInstanceId(u128);

```

```

pub struct StaffGroupId(u128);
pub struct RegionId(u128);
pub struct InstrumentId(u128);
pub struct PartDefinitionId(u128);
pub struct MeasureId(u128);
pub struct BarlineAlignmentGroupId(u128);
pub struct SlurId(u128);
pub struct TieId(u128);
pub struct BeamId(u128);
pub struct SpannerId(u128);
pub struct MarkerId(u128);
pub struct AnalyticalAnnotationId(u128);
pub struct CommentId(u128);
pub struct GraphicObjectId(u128);
pub struct GraphicGestureId(u128);
pub struct TimeSignatureId(u128);
pub struct ConflictId(u128);
pub struct TransactionId(u128);
// ... and so on for every named object kind

/// A tagged identifier over every typed identifier kind in the
/// score graph. Used wherever an object is referenced generically
/// (cross-cutting endpoints, conflict affected_objects, repair
/// records, edit barriers). The variant tag is part of canonical
/// content: distinct variants with the same underlying u128 are
/// distinct TypedObjectIds.
pub enum TypedObjectId {
    Event(EventId),
    Pitch(PitchId),
    Voice(VoiceId),
    Staff(StaffId),
    StaffInstance(StaffInstanceId),
    StaffGroup(StaffGroupId),
    Region(RegionId),
    Instrument(InstrumentId),
    PartDefinition(PartDefinitionId),
    Measure(MeasureId),
    BarlineAlignmentGroup(BarlineAlignmentGroupId),
    Slur(SlurId),
    Tie(TieId),
    Beam(BeamId),
    Spanner(SpannerId),
    Marker(MarkerId),
    AnalyticalAnnotation(AnalyticalAnnotationId),
    Comment(CommentId),
    GraphicObject(GraphicObjectId),
    GraphicGesture(GraphicGestureId),
    TimeSignature(TimeSignatureId),
    AnalysisLayer(AnalysisLayerId),
    /// Extension-defined object kind, identified by registry id
    /// plus the extension's own 128-bit identifier.
    Registered(ObjectKindRegistryId, u128),
}

```

```
impl TypedObjectId {
    /// Canonical byte form for hashing, ordering, and equality.
    /// Variant tag is encoded as a 16-bit big-endian discriminant
    /// followed by the variant payload's canonical bytes.
    pub fn canonical_bytes(&self) -> Vec<u8> { /* ... */ }
}
```

5.3.1 Identifier Generation

REQUIREMENT

Identifiers **MUST** be 128-bit values composed of a 64-bit replica identifier and a 64-bit monotonic counter local to that replica. The replica identifier **MUST** be unique among all replicas of the score; collision resistance is achieved by random initialization of the replica identifier at creation time with at least 64 bits of entropy from a cryptographically appropriate source. The counter **MUST** be monotonically increasing within a replica and **MUST NOT** be reused, even if the corresponding object is deleted.

```
pub struct IdentityContext {
    /// This replica's identifier. Generated at score creation.
    pub replica_id: ReplicaId,

    /// Monotonic counter for new identifiers. Per-kind counters
    /// optional; a single counter suffices.
    pub next_counter: u64,
}

pub struct ReplicaId(u64);

impl ReplicaId {
    /// Reserved replica identifier for deterministically-derived
    /// system identifiers (system-promoted voices, content-derived
    /// conflict IDs, future deterministic synthetic identifiers).
    /// User-authored replicas MUST NOT use this value.
    pub const SYSTEM_DERIVED: ReplicaId = ReplicaId(0xffff_ffff_ffff_ffff);
}
```

RATIONALE

Replica-plus-counter identifiers are deterministic given a replica and a creation order, which simplifies reasoning about CRDT merge semantics and reproducible builds of generated scores. UUIDv7 was considered and rejected: the timestamp embedding leaks ordering information that the CRDT layer would have to reconstruct anyway, and the random portion gives less collision resistance per bit than a properly initialized replica identifier.

5.3.2 System-Derived Identifier Namespace

Some identifiers are not author-authored: they arise during deterministic reduction (system-promoted voices, content-derived conflict identifiers, attachment-deduplication identifiers). To prevent collisions with user-authored identifiers, the spec reserves a dedicated replica namespace.

REQUIREMENT

The replica identifier `ReplicaId::SYSTEM_DERIVED` (`0xffff_ffff_ffff_ffff`) is reserved for deterministically-derived system identifiers. User-authored replicas **MUST NOT** use this value as their `ReplicaId`. Implementations creating a new score **MUST** reject this value when generating the local replica identifier and **MUST** regenerate the random portion until a non-reserved value is obtained.

System-derived identifiers within the `ReplicaId::SYSTEM_DERIVED` namespace **MUST** have their 64-bit counter portion derived deterministically by BLAKE3 truncation:

```
fn derive_system_counter(
    domain_tag: &[u8; 8],
    canonical_inputs: &[u8],
) -> u64 {
    let mut preimage = Vec::new();
    preimage.extend_from_slice(domain_tag);
    preimage.extend_from_slice(canonical_inputs);
    let hash = blake3(&preimage);
    u64::from_be_bytes(hash[0..8].try_into().unwrap())
}
```

Domain tags for system-derived identifiers:

- "MUSCSVCE" for system-promoted voices.
- "MUSCSPCH" for system-derived pitches (rare; only when promotion logic introduces a synthetic pitch).
- Additional domain tags introduced by registered extensions **MUST** begin with "MUSCS" and have length exactly 8 bytes.

Two replicas reducing the same operation set **MUST** derive identical system identifiers from identical canonical inputs.

5.3.3 System-Derived Counter Collisions

The 64-bit counter portion of a system-derived identifier is derived by BLAKE3 truncation. Truncation to 64 bits offers sufficient collision resistance against accidental collision (roughly 2^{32} system-derived identifiers of the same kind before a 50% chance of any pair colliding), but is weaker than the rest of the identifier story and is potentially vulnerable to deliberate construction of colliding canonical inputs. A collision means the identity layer has failed: canonical state derived from a colliding identifier cannot be trusted.

```
/// An integrity anomaly that takes the document out of ordinary
/// canonical operation. Distinct from ConflictRecord (an
/// ordinary canonical-state fact): integrity anomalies indicate
```

```

/// that the structural assumptions underlying canonical state
/// have failed.
pub struct IntegrityAnomaly {
    pub id: IntegrityAnomalyId,
    pub kind: IntegrityAnomalyKind,
}

pub enum IntegrityAnomalyKind {
    /// A system-derived identifier counter collision.
    SystemIdentifierCollision {
        kind: ObjectKind,
        colliding_counter: u64,
        input_set_a: SerializedCanonicalInputs,
        input_set_b: SerializedCanonicalInputs,
    },
    /// An OperationSlot is Equivocated. The slot's candidates
    /// are referenced; reduction has excluded the operation.
    OperationSlotEquivocated {
        operation_id: OperationId,
    },
    /// A replica's stream has been quarantined per the HLC
    /// monotonicity rule. The AnomalousReplicaSegment is
    /// referenced.
    ReplicaStreamQuarantined {
        replica: ReplicaId,
        first_bad_counter: u64,
    },
    /// A registered anomaly defined by an extension or
    /// transport.
    Registered(IntegrityAnomalyRegistryId),
}

```

REQUIREMENT

Implementations **MUST** perform a collision check during reduction whenever a new system-derived identifier is minted. If a newly derived 64-bit counter, within the same typed identifier kind and within the `ReplicaId::SYSTEM_DERIVED` namespace, collides with a previously-derived counter from *different* canonical inputs:

- An `IntegrityAnomaly` of kind `SystemIdentifierCollision` is recorded in the document's diagnostic integrity-anomaly register. This register is distinct from the canonical conflict registry: conflict records are ordinary canonical-state facts; integrity anomalies indicate the structural assumptions underlying canonical state have failed.
- Canonical reduction **MUST NOT** continue past the collision. The colliding object's identifier is held pending recovery: neither input set is allowed to occupy the collided counter in canonical state.
- Readers **MUST** open such a bundle in diagnostic recovery mode, surfacing the collision prominently and prohibiting ordinary editing of the affected portion of canonical state.

- Recovery is by external action: a profile-declared deterministic policy, a transport-level credential revocation, or an explicit user reconciliation operation. No automatic in-spec resolution is defined.

Authoring implementations **SHOULD** detect predictable collisions before authoring (e.g., by canonicalizing inputs with additional disambiguating data such as the causally-greatest operation visible at authoring time) and **SHOULD NOT** rely on collision frequency being negligible in adversarial scenarios.

RATIONALE

Moving system-derived collisions out of `ConflictKind` reflects the difference in kind between the two. A `ConflictRecord` is an ordinary canonical-state fact: the user has two valid musical edits that conflict, and the resolution is a normal editing action. A system-derived identifier collision is not a musical conflict; it is a structural failure of the identity layer. Treating it as a conflict would let canonical state grow under the collision, which is precisely the wrong behavior for an integrity failure. Treating it as a diagnostic anomaly halts canonical growth until external recovery resolves the structural issue.

5.3.4 Identifier Stability

REQUIREMENT

An identifier, once assigned, **MUST** never be reassigned, even after deletion of the corresponding object. Stale identifiers encountered in stored references **MUST** be detected and reported by re-anchoring rules (see Chapter 6); they **MUST NOT** silently match a different object.

5.4 The Event Arena

Events are the rhythmic atoms of the score: pitched events, unpitched events, rests, indeterminate events, trajectories, graphic events, and cues. All events of all kinds in a score are stored in a single arena.

```
pub struct EventArena {
    /// Storage backing for events. Implementation-defined; the
    /// observable contract is  $O(1)$  lookup by EventId and stable
    /// addresses for the lifetime of the event.
    storage: ArenaStorage<Event>,

    /// Auxiliary index for fast time-range queries.
    time_index: EventTimeIndex,
}
```

REQUIREMENT

Event lookup by `EventId` **MUST** be $O(1)$ amortized. Implementations **MAY** use any storage layout meeting this contract; vector-of-events with stable indices, slot-allocator schemes, and generational arenas are all acceptable.

5.4.1 The Event Type

```
pub enum Event {
    Pitched(PitchedEvent),
    Unpitched(UnpitchedEvent),
    Rest(Rest),
    Indeterminate(IndeterminateEvent),
    Trajectory(TrajectoryEvent),
    Graphic(GraphicEvent),
    Cue(CueEvent),
}
```

Every event variant carries:

- Its own `EventId`.
- Its *voice membership*: a `VoiceId` identifying the unique voice that owns it.
- Its *position* within that voice (either a `MusicalPosition` or `WallClockTime`, depending on the enclosing region's time model).

Voice membership and position are stored on the event so that operations addressing an event can resolve its context without traversing parent containers.

Event Position and Duration

An event's temporal coordinates use union types matching its enclosing region's time model. Position is unioned over musical and wall-clock forms; duration is unioned over musical, wall-clock, and indeterminate forms.

```
pub enum EventPosition {
    Musical(MusicalPosition),
    WallClock(WallClockTime),
}

pub enum EventDuration {
    Musical(MusicalDuration),
    WallClock(WallClockDuration),
    Indeterminate(DurationBounds),
}

/// Bounded interval expressing an indeterminate duration. Bounds are
/// concrete (not themselves indeterminate); this prevents recursive
/// indeterminacy in the type system.
pub struct DurationBounds {
    pub lower: Option<ConcreteDuration>,
}
```

```

pub upper: Option<ConcreteDuration>,
}

pub enum ConcreteDuration {
    Musical (MusicalDuration),
    WallClock (WallClockDuration),
}

```

REQUIREMENT

Event position and duration variants **MUST** agree with the time model of the enclosing region:

- Metric regions accept `EventPosition::Musical` and `EventDuration::Musical` only.
- Proportional regions accept `EventPosition::WallClock` and `EventDuration::WallClock` only.
- Aleatoric regions **MAY** accept any combination consistent with the region's declared anchoring discipline (see Section 3.10.3).

Implementations **MUST** reject events whose coordinate variants contradict their enclosing region's time model.

REQUIREMENT

If both bounds of a `DurationBounds` are present, they **MUST** use the same `ConcreteDuration` variant, unless the enclosing aleatoric region's anchoring discipline explicitly permits mixed-kind bounds.

Identified Pitches

Pitches embedded in events are wrapped in `IdentifiedPitch` records that pair each pitch with a stable `PitchId` (Section 2.6). The identifier enables spelling attachments, respelling operations, pitch-level reduction rules under concurrent edits, and stable tie pairing through chord reordering.

```

pub struct IdentifiedPitch {
    pub id: PitchId,
    pub pitch: Pitch,
}

```

REQUIREMENT

Every pitch embedded in an event **MUST** be wrapped in an `IdentifiedPitch`. The contained `PitchId` **MUST** be unique within the score's pitch-identity index. Pitch identifiers **MUST** be minted atomically with their containing event: the operation creating an event mints `EventId` and the `PitchIds` of all embedded pitches in a single causal step.

Pitched Events

```
pub struct PitchedEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// One or more identified pitches. A single pitch is a
    /// single-note event; multiple pitches are a chord.
    pub pitches: Vec<IdentifiedPitch>,

    /// Articulations attached directly to the event.
    pub articulations: Vec<ArticulationMark>,

    /// Single-event dynamic marking (e.g., sfz, fp). Spanning
    /// dynamics are cross-cutting structures, not event fields.
    pub dynamic: Option<DynamicMark>,

    /// Ornaments attached to the event. Ornament resolutions (the
    /// realized notes of a trill, mordent, etc.) are cross-cutting
    /// structures.
    pub ornaments: Vec<OrnamentMark>,

    /// Stem configuration: direction, length adjustment, hidden.
    pub stem: StemConfiguration,

    /// Whether this event is a grace note, and if so its kind.
    pub grace: Option<GraceKind>,
}

pub enum GraceKind {
    Acciaccatura,
    Appoggiatura,
    Unmeasured,
    MeasuredFraction(MusicalDuration),
}
```

REQUIREMENT

A `PitchedEvent` **MUST** have at least one pitch. Empty pitch lists are forbidden; use `Rest` for the no-pitch case.

Unpitched Events

```
pub struct UnpitchedEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,
```

```

    /// Staff line position for the unpitched event (e.g., snare
    /// drum at the middle line of a 5-line staff, or position 0
    /// for a single-line staff).
    pub staff_position: StaffPosition,

    /// The unpitched instrument identifier, for sound mapping.
    pub instrument_member: UnpitchedMemberId,

    pub articulations: Vec<ArticulationMark>,
    pub dynamic: Option<DynamicMark>,
    pub stem: StemConfiguration,
    pub grace: Option<GraceKind>,
}

```

Rests

```

pub struct Rest {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// Optional explicit vertical positioning. If None, the engraver
    /// chooses based on voice and meter context.
    pub vertical_position: Option<StaffPosition>,

    /// Whether this rest is visible. Invisible rests preserve timing
    /// but do not render.
    pub visible: bool,
}

```

Indeterminate Events

```

pub struct IndeterminateEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// What aspect of the event is indeterminate.
    pub indeterminacy: IndeterminacyKind,

    /// Optional hints constraining the indeterminacy.
    pub hints: IndeterminacyHints,
}

pub enum IndeterminacyKind {
    /// "Any pitch in this range/contour."
    Pitch,

    /// "Any duration within these bounds." (Realized via

```

```

    /// EventDuration::Indeterminate on the event itself.
    Duration,

    /// "Any of these alternatives."
    Choice,

    /// Combination of multiple indeterminacies.
    Compound(Vec<IndeterminacyKind>),
}

pub struct IndeterminacyHints {
    pub pitch_range: Option<PitchRange>,
    pub pitch_contour: Option<ContourHint>,
    pub density: Option<DensityHint>,
    pub textual_instruction: Option<String>,
    pub duration_bounds: Option<DurationBounds>,
    pub alternatives: Vec<EventId>,
}

```

Trajectory Events

A trajectory event represents a continuous pitch motion: a glissando, portamento, or pitch bend.

```

pub struct TrajectoryEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// Start and end pitch endpoints. Either may reference another
    /// event's pitch by id, or may carry an explicit identified
    /// pitch local to the trajectory.
    pub start: TrajectoryEndpoint,
    pub end: TrajectoryEndpoint,

    /// Shape of the trajectory between endpoints.
    pub shape: TrajectoryShape,

    /// Visual representation.
    pub display: TrajectoryDisplay,
}

pub enum TrajectoryEndpoint {
    /// Reference to a pitch belonging to another event.
    EventPitch(PitchId),

    /// An explicit identified pitch local to this trajectory.
    /// Such pitches participate in spelling, identity, and editing
    /// the same way embedded pitches do.
    ExplicitPitch(IdentifiedPitch),
}

```

```
pub enum TrajectoryShape {
    Linear,
    Exponential,
    Curve(Vec<TrajectoryControlPoint>),
    Stepwise(Vec<IdentifiedPitch>),
}
```

Graphic Events

A graphic event is an event whose primary content is graphic (drawn or constructed) rather than notated through the standard staff system. Used within hybrid and free-graphic regions for events that nevertheless occupy a rhythmic slot in a voice.

```
pub struct GraphicEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// Reference to the graphic object(s) realizing this event.
    pub graphics: Vec<GraphicObjectId>,

    /// Optional playback parameter bindings (see
    /// Section~\ref{sec:graph:timebinding}).
    pub playback_bindings: Vec<PlaybackBinding>,
}
```

Cue Events

A cue is a small-print rendering of music from another voice or instrument, displayed as a guide.

```
pub struct CueEvent {
    pub id: EventId,
    pub voice: VoiceId,
    pub position: EventPosition,
    pub duration: EventDuration,

    /// The source events being cued.
    pub source: Vec<EventId>,

    /// Rendering hints: how the cue should be drawn (size, label,
    /// abbreviation of the source instrument's name, etc.).
    pub rendering: CueRendering,
}
```

5.5 The Canvas

The canvas is the spatial root of the score. It contains regions.

```

pub struct Canvas {
    /// Regions in this canvas. Each region declares its time and
    /// content models and occupies a time and staff extent.
    pub regions: Vec<Region>,

    /// Canvas-level configuration: paper size, margins, default
    /// system layout, page-break behavior.
    pub layout_defaults: CanvasLayoutDefaults,
}

```

5.5.1 Regions

```

pub struct Region {
    pub id: RegionId,

    /// Time model: metric, proportional, or aleatoric.
    pub time_model: RegionTimeModel,

    /// Content model: staff-based, free graphic, or hybrid.
    pub content: RegionContent,

    /// Range in time that this region occupies.
    pub time_extent: TimeExtent,

    /// Range in vertical staff space that this region occupies.
    /// A region may cover all staves (full-score region) or a subset
    /// (a free-graphic gesture occupying only one staff's vertical
    /// band, for example).
    pub staff_extent: StaffExtent,

    /// Optional tempo and tuning overrides scoped to this region.
    pub local_tempo_map: Option<TempoMap>,
    pub local_tuning_overrides: Vec<TuningOverride>,
}

pub enum RegionContent {
    /// Staff-based notation: one or more staves carrying voices and
    /// events.
    StaffBased(StaffBasedContent),

    /// Free graphic content: no staves, only graphic objects placed
    /// in the region's coordinate space.
    FreeGraphic(GraphicContent),

    /// Hybrid: staves with overlaid graphic content. The graphic
    /// layer is rendered above (or below, configurably) the staff
    /// layer.
    Hybrid {
        staves: StaffBasedContent,
        overlay: GraphicContent,
        overlay_below_staves: bool,
    },
}

```

```

}

pub struct TimeExtent {
    pub start: TimeAnchor,
    pub end: TimeAnchor,
}

pub struct StaffExtent {
    /// Identifiers of the globally-identified staves whose content
    /// this region carries. A region carries one StaffInstance per
    /// listed StaffId. The same StaffId may appear in the
    /// staff_extent of adjacent regions; this is the mechanism by
    /// which a staff's identity persists across regions.
    pub staves: Vec<StaffId>,
}

```

5.5.2 Region Overlap and Concurrency

REQUIREMENT

Regions **MAY** overlap in time if their staff extents are disjoint. Regions **MUST NOT** overlap in both time and staff extent. The graph's construction **MUST** reject configurations violating this constraint.

RATIONALE

Concurrent regions on disjoint staves are essential for the hybrid notation feature: a metric staff and a proportional staff sounding simultaneously are two regions with overlapping time extents but disjoint staff extents. Overlapping time *and* staff would mean two regions claim the same content, which is ambiguous.

5.5.3 Staff-Based Content

A staff-based region carries one or more staff *instances*, each of which manifests a globally-identified staff for the duration of the region. The score's metric organization is per-staff (admitting polymeter); barline alignment between staves is recorded as an explicit relationship when meters coincide.

```

pub struct StaffBasedContent {
    /// Staff instances appearing in this region. Each references a
    /// globally-identified Staff and provides region-local content.
    pub staff_instances: Vec<StaffInstance>,

    /// Default metric grid for this region. Staves without their own
    /// local_metric_grid inherit this grid. None means the region
    /// has no default metric organization (acceptable for transitional
    /// regions or for regions where every staff declares its own).
    pub default_metric_grid: Option<MetricGrid>,
}

```

```

    /// Explicit barline alignment groups: places where measures
    /// across multiple staves are declared to coincide. Used for
    /// engraving barlines that span multiple staves under polymeter
    /// and for analytical purposes. Implementations MAY derive
    /// some alignment groups automatically from coincident measure
    /// starts; explicit groups are authoritative.
    pub barline_alignment_groups: Vec<BarlineAlignmentGroup>,

    /// System breaks declared by the user. The engraver may
    /// override these if region settings permit (see Chapter 6).
    pub user_system_breaks: Vec<TimeAnchor>,

    /// Page breaks declared by the user. Same override rule.
    pub user_page_breaks: Vec<TimeAnchor>,
}

/// A region-local manifestation of a globally-identified Staff.
pub struct StaffInstance {
    pub id: StaffInstanceId,

    /// The globally-identified staff this instance manifests.
    pub staff: StaffId,

    /// Voices appearing on this staff instance.
    pub voices: Vec<Voice>,

    /// Clef changes within this instance.
    pub clef_sequence: Vec<ClefChange>,

    /// Key signature changes within this instance.
    pub key_sequence: Vec<KeySignatureChange>,

    /// Optional local metric grid: if Some, overrides the region's
    /// default_metric_grid for this staff (polymetric case).
    /// If None, the staff inherits the region's default_metric_grid.
    pub local_metric_grid: Option<MetricGrid>,

    /// Measures belonging to this staff instance, in order. Measures
    /// are per-staff (admitting polymeter); alignment across staves
    /// is recorded in barline_alignment_groups.
    pub measures: Vec<Measure>,

    /// Optional per-instance instrument override. None means inherit
    /// from the global Staff's declared instrument.
    pub instrument_override: Option<InstrumentId>,

    /// Optional per-instance staff line configuration override.
    /// None means inherit from the global Staff's default.
    pub staff_lines_override: Option<StaffLineConfiguration>,

    /// Whether this instance is visible. Hidden instances preserve
    /// content but are not rendered (used for empty-staff suppression).
    pub visible: bool,

```

```

}

/// A per-staff (or per-region-default) metric organization.
pub struct MetricGrid {
    /// Sequence of meter changes within the scope of this grid.
    pub meter_sequence: Vec<MeterChange>,
}

pub struct MeterChange {
    /// Where this meter takes effect, expressed as an anchor within
    /// the enclosing staff or region.
    pub anchor: TimeAnchor,

    /// The active time signature from this point until the next
    /// MeterChange or the end of the grid's scope.
    pub time_signature: TimeSignatureId,
}

/// A declaration that the measure boundaries of two or more staff
/// instances coincide at a particular point.
pub struct BarlineAlignmentGroup {
    pub id: BarlineAlignmentGroupId,

    /// The staff instances participating in this alignment, paired
    /// with the measures whose boundaries coincide.
    pub members: Vec<BarlineAlignmentMember>,
}

pub struct BarlineAlignmentMember {
    pub staff_instance: StaffInstanceId,
    pub measure: MeasureId,
    pub position: MeasurePosition,
}

pub struct Measure {
    pub id: MeasureId,
    pub start: TimeAnchor,
    pub time_signature: Option<TimeSignatureId>,
    pub explicit_number: Option<u32>,
    pub number_visibility: MeasureNumberVisibility,
}

```

REQUIREMENT

Measures **MUST** belong to a single `StaffInstance`, not to the enclosing region. This admits polymeter: different staves in the same region **MAY** carry different meter sequences and consequently different measure boundaries. Barline coincidences across staves **MUST** be expressed via `BarlineAlignmentGroup` entries when explicit alignment is authoritative; otherwise alignment is derived at engraving time from coincident measure-start positions.

5.6 Staves, Voices, and Instruments

5.6.1 Instruments

An instrument is an abstract definition of a sounding entity.

```
pub struct Instrument {
    pub id: InstrumentId,
    pub name: String,
    pub abbreviation: Option<String>,

    /// MIDI program, sound library mapping, or unpitched instrument
    /// definitions. The structure is detailed in the audio engine
    /// specification (out of scope here); the core stores the
    /// configuration opaquely.
    pub sound_config: SoundConfiguration,

    /// Default transposition (e.g., B-flat clarinet sounds a major
    /// second below written).
    pub transposition: Option<TranspositionInterval>,

    /// Default playable range. Engraving may flag out-of-range
    /// pitches; not a hard constraint.
    pub range: Option<PitchRange>,

    /// Default clef and staff line count for staves of this
    /// instrument.
    pub default_clef: ClefId,
    pub default_staff_lines: u8,

    /// For unpitched instruments: definitions of each playable
    /// member (snare, kick, ride bell, etc.) and their staff
    /// positions.
    pub unpitched_members: Vec<UnpitchedMember>,
}
```

5.6.2 Staves: Identity Versus Instance

The specification distinguishes two staff concepts:

Staff is the *global, abstract* staff identity persisting across the entire score. A staff is the thing a performer is reading: the “Flute 1” staff, the “Piano upper staff.” A staff is declared once in the score and is referenced by staff instances in any region where it appears.

StaffInstance is the *region-local manifestation* of a staff: the voices, measures, clef changes, key changes, and metric grid that belong to that staff for the duration of one region. A single Staff may have multiple StaffInstances across the score; each instance belongs to exactly one region.

```
pub struct Staff {
    pub id: StaffId,
```

```

    /// Human-readable name (e.g., "Flute 1", "Violin II").
    pub name: String,

    /// Optional abbreviated name for subsequent systems.
    pub abbreviation: Option<String>,

    /// The instrument this staff renders by default. May be
    /// overridden per-instance.
    pub instrument: InstrumentId,

    /// Default clef for new instances of this staff.
    pub default_clef: ClefId,

    /// Default staff line configuration.
    pub default_staff_lines: StaffLineConfiguration,

    /// Visual grouping: which staff group (if any) this staff
    /// belongs to (e.g., piano grand staff, choral group).
    pub group: Option<StaffGroupId>,
}

pub struct StaffLineConfiguration {
    pub line_count: u8,
    pub line_spacing: SpaceUnit,
    pub line_style: LineStyle,
    pub bracket: Option<StaffBracketKind>,
}

```

The score's top-level structure carries a list of `Staff` objects alongside its list of `Instrument` objects; each region's staff instances reference these by `StaffId`.

REQUIREMENT

Every `StaffInstance.staff` **MUST** resolve to a `Staff` declared at the score level. A single `StaffId` **MAY** be referenced by multiple `StaffInstances` in different regions: this is how a staff maintains continuous identity across the score's regions. A `StaffInstance` **MUST** belong to exactly one region. The region's `staff_instances` list **MUST** contain the instance, and the instance's content is owned by that region. Continuous attachments (e.g., key signatures, clef changes, metric grids) **MAY** appear on multiple staff instances of the same `Staff` when those instances are adjacent in time; the engraving pipeline reconstructs continuous staff appearance from per-instance content.

5.6.3 Voices

A voice is a polyphonic line within a staff instance. Voices hold ordered references to events; the events themselves live in the score's arena.

```

pub struct Voice {
    pub id: VoiceId,
}

```

```

    /// Ordered references to events in this voice, sorted by
    /// position. Each event has voice == this voice's id.
    pub events: Vec<EventId>,

    /// Default stem direction for this voice. Common conventions:
    /// voice 1 stems up, voice 2 stems down on the same staff.
    pub default_stem_direction: Option<StemDirection>,

    /// Whether this voice is the staff's primary voice (for purposes
    /// of rest placement, beam direction defaults, and similar
    /// engraving decisions).
    pub is_primary: bool,

    /// Provenance of this voice: user-declared, imported, or
    /// system-promoted by concurrent-edit conflict resolution.
    pub origin: VoiceOrigin,
}

pub enum VoiceOrigin {
    /// Created by an explicit user action.
    UserDeclared,

    /// Imported from a foreign format.
    Imported { format: ForeignFormatId },

    /// System-promoted to resolve a concurrent-edit collision.
    /// The cause operation is the InsertEvent (or similar) whose
    /// reduction required voice allocation. The original_voice is
    /// the voice into which the user had intended to insert; the
    /// promoted voice carries the event that lost the collision.
    SystemPromoted {
        cause: OperationId,
        original_voice: VoiceId,
    },
}

```

REQUIREMENT

The number of voices per staff instance **MUST** be unbounded in the data model. Engraving heuristics (Chapter 7 and beyond) may flag visual issues with high voice counts; the graph admits any number.

REQUIREMENT

Every event **MUST** belong to exactly one voice. Voice membership is stored on the event (the `voice` field) and **MUST** agree with the membership of the voice's `events` list: for any event e in voice v 's `events` list, $e.voice$ **MUST** equal $v.id$.

System-Promoted Voices

When concurrent operations cause an `InsertEvent` collision that cannot be resolved by ordering alone (because both events have positive duration and would overlap within the target voice), the deterministically-losing event is placed in a newly allocated voice with `origin: VoiceOrigin::SystemPromoted`. This preserves the non-overlap invariant without rejecting user intent.

REQUIREMENT

The identifier of a system-promoted voice **MUST** be derived deterministically from a fixed function of:

1. The enclosing staff instance's `StaffInstanceId`.
2. The original target voice's `VoiceId`.
3. The winning operation's `OperationId`.
4. The losing operation's `OperationId`.

The exact derivation function (a hash of the concatenated identifiers, normalized to the replica-plus-counter format reserved for system-derived identifiers) is specified in the semantic-operations companion document. Determinism is required so that all replicas independently arrive at the same promoted voice identity.

REQUIREMENT

System-promoted voices are first-class graph objects: they appear in the staff instance's `voices` list, they participate in cross-cutting structures, and they are visible to users. They are *not* hidden or transient. The layout pipeline (Chapter 7) **SHOULD** provide a visual indication that a voice was system-promoted until the user normalizes it (via a later `NormalizePromotedVoice` or `MergeVoices` operation, specified in the operation catalog).

Promoted voices **MUST NOT** silently become user-authored voices; doing so would convert a CRDT conflict resolution into musical authorship without the user's consent.

REQUIREMENT

Within a voice, events **MUST** be sorted by position and **MUST NOT** overlap in time. Concurrent material on the same staff **MUST** be expressed via multiple voices.

5.7 Cross-Cutting Structures

Cross-cutting structures hold references into the tree (typically event identifiers) and represent musical phenomena that span multiple events. The registry is partitioned by type for both type safety and fast indexed lookup.

```
pub struct CrossCuttingRegistry {
    pub slurs: Vec<Slur>,
    pub ties: Vec<Tie>,
```

```

pub beams: Vec<Beam>,
pub triplets: Vec<Triplet>,
pub spanners: Vec<Spanner>,
pub markers: Vec<Marker>,
pub repeats: Vec<RepeatStructure>,
pub analytical: Vec<AnalyticalAnnotation>,
pub comments: Vec<Comment>,
pub graphic_gestures: Vec<GraphicGesture>,
pub lyrics: Vec<LyricLine>,
pub chord_symbols: Vec<ChordSymbol>,
}

```

5.7.1 Slurs and Phrase Marks

```

pub struct Slur {
    pub id: SlurId,

    /// First and last events under the slur.
    pub start_event: EventId,
    pub end_event: EventId,

    /// Slur kind: legato slur, phrase mark, articulation slur,
    /// editorial.
    pub kind: SlurKind,

    /// Optional curvature override. The engraver computes default
    /// curvature; this field overrides it.
    pub curvature_override: Option<CurvatureOverride>,

    /// Visual style: solid, dashed, dotted; line thickness curve.
    pub style: SlurStyle,
}

```

5.7.2 Ties

```

pub struct Tie {
    pub id: TieId,

    /// The two events being tied. The end event's pitch(es) must
    /// match the corresponding pitches in the start event.
    pub start_event: EventId,
    pub end_event: EventId,

    /// For chord ties: which pitches in the start event are tied
    /// to which in the end event, expressed by stable PitchId pairs.
    /// None means all pitches are tied by enharmonic matching in
    /// pitch-id-ascending order.
    pub pitch_pairing: Option<Vec<(PitchId, PitchId)>>,

    /// Tie class: determines the validation profile applied to this
    /// tie. The default Standard class requires same-voice immediate

```

```

    /// adjacency; other classes relax the rule for editorial,
    /// cross-voice, or notation-specific use.
    pub class: TieClass,

    pub style: TieStyle,
}

pub enum TieClass {
    /// Standard CMN tie: same voice, immediate adjacency in voice
    /// order, pitches enharmonically equivalent.
    Standard,

    /// Editorial tie: connects pitches the editor considers
    /// continuous, spanning intervening rests or invisible events.
    /// Drawn with editorial style (dashed or bracketed) by default.
    Editorial,

    /// Cross-voice tie: connects pitches across voice boundaries
    /// within the same staff instance. Used for keyboard divisi,
    /// re-voicing across hands, and similar constructions.
    CrossVoice,

    /// Laissez-vibrer tie: a tie with no defined end event, drawn
    /// trailing from the start event. The end_event field is
    /// permitted to equal the start_event for this class, with
    /// pitch_pairing self-referential.
    LaissezVibrer,

    /// Registered class with custom validation behavior, defined
    /// by a notation grammar plugin.
    Registered(TieClassRegistryId),
}

```

RATIONALE

Pairing by stable `PitchId` rather than by positional index preserves tie identity through chord-reordering edits and through CRDT-style concurrent modification of the chord's pitch list. Index-based pairing would silently retarget when pitches were inserted, deleted, or reordered within a chord.

The tie class permits common editorial and contemporary-notation practices (editorial dashed ties spanning rests, cross-voice ties in keyboard scores, laissez-vibrer ties without a notated end) to be expressed without relaxing the structural invariants of the standard tie.

REQUIREMENT

Tie validation is class-specific:

- `TieClass::Standard`: the start and end events **MUST** be in the same voice; the start event **MUST** immediately precede the end event in that voice's event se-

quence; the paired pitches **MUST** be enharmonically equivalent under the active tuning system.

- `TieClass::Editorial`: the start and end events **MUST** be in the same voice and the start **MUST** precede the end in voice order, but intervening events are permitted. Paired pitches **MUST** be enharmonically equivalent.
- `TieClass::CrossVoice`: the start and end events **MAY** be in different voices on the same staff instance. The start's resolved position **MUST** be less than or equal to the end's. Paired pitches **MUST** be enharmonically equivalent.
- `TieClass::LaissezVibrer`: the end event **MAY** equal the start event, in which case the tie has no notated destination. No adjacency requirement applies.
- `TieClass::Registered`: validation is defined by the registered class's contract; the registry **MUST** specify adjacency and pitch-equivalence requirements for the class.

5.7.3 Beams

```
pub struct Beam {
    pub id: BeamId,
    pub events: Vec<EventId>,
    pub level: u8,                // primary beam level

    /// Sub-beam structure for inner subdivisions. Each entry
    /// describes a beam segment at a deeper level.
    pub sub_beams: Vec<SubBeam>,

    /// Beam angle and stem-length overrides.
    pub geometry_override: Option<BeamGeometryOverride>,
}
```

5.7.4 Spanners

A spanner is a generic spanning mark: hairpins, octave lines, pedal lines, trill extensions, and similar.

```
pub struct Spanner {
    pub id: SpannerId,
    pub kind: SpannerKind,
    pub start: TimeAnchor,
    pub end: TimeAnchor,

    /// Which staff or staves this spanner attaches to. Most spanners
    /// target a single staff; some (system-level dynamics, pedal
    /// lines spanning grand staff) target multiple.
    pub staves: Vec<StaffId>,

    pub style: SpannerStyle,
}

pub enum SpannerKind {
```

```

    Hairpin(HairpinDirection),
    OctaveLine(OctaveOffset),
    PedalLine(PedalKind),
    TrillExtension,
    Glissando,
    Portamento,
    TextLine(TextLineDefinition),
    Bracket(BracketKind),
}

```

5.7.5 Markers

A marker is a point annotation: rehearsal marks, section labels, tempo text, fermatas (which technically attach to events but may also be region-level), and similar.

```

pub struct Marker {
    pub id: MarkerId,
    pub anchor: TimeAnchor,
    pub kind: MarkerKind,
    pub display: MarkerDisplay,
}

pub enum MarkerKind {
    Rehearsal(RehearsalMark),
    Section(SectionLabel),
    Tempo(TempoMarking),
    Coda,
    Segno,
    DalSegno,
    DaCapo,
    Fine,
    Custom(CustomMarkerId),
}

```

5.7.6 Repeat Structures

```

pub struct RepeatStructure {
    pub id: RepeatStructureId,
    pub kind: RepeatKind,
    pub start: TimeAnchor,
    pub end: TimeAnchor,

    /// For voltas: which endings apply on which passes.
    pub voltas: Vec<Volta>,
}

pub enum RepeatKind {
    SimpleRepeat { count: u32 },
    DaCapo { end_target: TimeAnchor },
    DalSegno { segno: TimeAnchor, end_target: TimeAnchor },
    Volta,
}

```

```
}

```

5.7.7 Analytical Annotations

Analytical annotations record analyses of the music: Roman numerals, form labels, motivic brackets, set-theory analyses, Schenkerian graphs. They are first-class and may appear on dedicated analysis layers.

```
pub struct AnalyticalAnnotation {
    pub id: AnalyticalAnnotationId,
    pub kind: AnalyticalKind,
    pub anchor: AnnotationAnchor,

    /// Optional analysis layer. None means the engraved layer;
    /// Some(id) targets a specific analysis layer.
    pub layer: Option<AnalysisLayerId>,

    pub content: AnalyticalContent,
}

pub enum AnalyticalKind {
    RomanNumeral,
    FunctionalLabel,
    SetTheoryLabel,
    SchenkerianGraph,
    FormSection,
    MotivicBracket,
    Custom(CustomAnalyticalKindId),
}

pub enum AnnotationAnchor {
    Event(EventId),
    Range { start: TimeAnchor, end: TimeAnchor },
    Region(RegionId),
}
```

5.7.8 Comments

Comments are review-mode annotations: discussion threads anchored to points or ranges in the score.

```
pub struct Comment {
    pub id: CommentId,
    pub anchor: AnnotationAnchor,
    pub thread: Vec<CommentMessage>,
    pub resolved: bool,
}

pub struct CommentMessage {
    pub author: AuthorId,
    pub timestamp: Timestamp,
```

```
pub body: String,
}
```

5.7.9 Graphic Gestures

A graphic gesture is a drawn or constructed graphic element that spans events, staves, or regions and is not contained within a single graphic region. Used for performance-direction sweeps, multi-staff curved gestures, and similar.

```
pub struct GraphicGesture {
    pub id: GraphicGestureId,

    /// The graphic objects composing this gesture. Stored in the
    /// canvas's graphic-object storage.
    pub objects: Vec<GraphicObjectId>,

    /// Anchoring: how this gesture is positioned relative to score
    /// content.
    pub anchoring: GestureAnchoring,

    /// Optional playback parameter bindings.
    pub playback_bindings: Vec<PlaybackBinding>,
}

pub enum GestureAnchoring {
    /// Anchored to events: the gesture moves with them.
    Events(Vec<EventId>),

    /// Anchored to a time and staff range.
    Range {
        start: TimeAnchor,
        end: TimeAnchor,
        staves: Vec<StaffId>,
    },

    /// Free canvas coordinates: does not follow score edits.
    Free(CanvasCoordinates),
}
```

5.8 Graphic Content

Free-graphic and hybrid regions carry graphic content: vector primitives, drawn strokes, text, and images. Graphic content is expressed in the region's local coordinate space.

```
pub struct GraphicContent {
    pub objects: Vec<GraphicObject>,

    /// The region's local coordinate system. Region transforms
    /// map this to canvas coordinates.
    pub coordinate_system: LocalCoordinateSystem,
}
```

5.8.1 Graphic Objects

```

pub struct GraphicObject {
  pub id: GraphicObjectId,

  /// The object's geometric kind.
  pub kind: GraphicObjectKind,

  /// Transform applied within the region's coordinate space.
  pub transform: Transform2D,

  /// Layer ordering: higher values draw on top of lower.
  pub layer: i32,

  /// Visual style: stroke color, fill, line weight, opacity.
  pub style: GraphicStyle,

  /// Optional time binding. See Section~\ref{sec:graph:timebinding}.
  pub time_binding: Option<TimeBinding>,
}

pub enum GraphicObjectKind {
  Path(PathData),
  Shape(ShapePrimitive),
  Text(TextData),
  Image(ImageData),
  Group(Vec<GraphicObjectId>),
  Stroke(StrokeData),
}

pub enum ShapePrimitive {
  Line { start: Point2D, end: Point2D },
  Rectangle { origin: Point2D, size: Size2D },
  Ellipse { center: Point2D, radii: Size2D },
  Polygon(Vec<Point2D>),
  Bezier(Vec<BezierSegment>),
}

pub struct StrokeData {
  /// Pressure-sensitive stylus stroke samples: position, pressure,
  /// tilt, and time per sample.
  pub samples: Vec<StrokeSample>,
}

pub struct StrokeSample {
  pub position: Point2D,
  pub pressure: f32,           // 0.0..=1.0
  pub tilt_xy: (f32, f32),    // radians
  pub timestamp: WallClockTime,
}

```

5.8.2 Coordinates

REQUIREMENT

Graphic objects **MUST** be stored in region-local coordinates. The region's transform maps local coordinates to canvas coordinates. This permits regions to be moved, resized, or otherwise transformed without rewriting the coordinates of every contained object.

5.8.3 Time Bindings

A graphic object **MAY** carry an optional time binding, which assigns a temporal meaning to the object beyond its visual appearance. Time bindings are the mechanism by which a drawn curve becomes a performance instruction (a filter sweep, an amplitude envelope, a density trajectory).

```
pub struct TimeBinding {
    /// What musical or wall-clock range this graphic spans.
    pub time_extent: TimeExtent,

    /// What parameter, if any, this graphic drives. None means the
    /// graphic is decorative; Some specifies a named parameter.
    pub parameter: Option<ParameterBinding>,

    /// How the graphic's geometry maps to parameter values.
    pub mapping: ParameterMapping,
}

pub struct ParameterBinding {
    pub target: ParameterTarget,
    pub parameter_name: String,
    pub value_range: ParameterRange,
}

pub enum ParameterTarget {
    /// A specific event's parameter (e.g., this event's velocity).
    Event(EventId),

    /// A voice-level parameter (e.g., voice expression CC).
    Voice(VoiceId),

    /// A staff-level parameter.
    Staff(StaffId),

    /// A region-level parameter.
    Region(RegionId),

    /// A score-global parameter (e.g., master gain).
    Score,
}

pub enum ParameterMapping {
    /// Vertical position in region coordinates maps linearly to
```

```

    /// parameter value.
    VerticalLinear,

    /// Vertical position with exponential mapping.
    VerticalExponential,

    /// Stroke pressure maps to parameter value.
    StrokePressure,

    /// Stroke velocity maps to parameter value.
    StrokeVelocity,

    /// Custom mapping defined by a plugin.
    Registered(MappingRegistryId),
}

```

RATIONALE

Time bindings make graphic-notation playback expressible in the core data model without committing to specific audio engine semantics. The core stores what parameter a graphic drives and how its geometry maps to values; the audio engine spec consumes this binding and applies it. Decorative graphics (binding is `None`) remain pure visual content.

5.9 Parts

A part is a per-instrument or per-section view onto the score, intended for printed extraction or focused performer use. Parts live separately from the canvas; they are projections defined by a part-extraction specification.

```

pub struct PartDefinition {
    pub id: PartDefinitionId,

    /// Human-readable name (e.g., "Flute 1", "Violins II").
    pub name: String,

    /// Which staves are included in this part, in order from top
    /// to bottom.
    pub staves: Vec<StaffId>,

    /// Part-specific layout overrides: page size, system count,
    /// cue insertions, multirest consolidation, page-turn placement.
    pub layout_overrides: PartLayoutOverrides,

    /// Per-event visibility overrides specific to this part (e.g.,
    /// hide a cue in the full score but show it in the part).
    pub visibility_overrides: Vec<VisibilityOverride>,

    /// Auto-cue insertions: source staves to draw cues from when
    /// the part has long rests.
}

```

```
pub auto_cue_sources: Vec<StaffId>,
}
```

5.9.1 Parts Are Projections, Not Storage

REQUIREMENT

A part **MUST NOT** store musical content directly; it **MUST** consist only of references to score content and overrides. Edits to score content **MUST** propagate to all parts that reference that content.

5.10 Analysis Layers and Views

5.10.1 Analysis Layers

Analysis layers, introduced for spelling in Section 2.5.6 and used by analytical annotations, are first-class objects on the score.

```
pub struct AnalysisLayer {
  pub id: AnalysisLayerId,
  pub name: String,
  pub description: Option<String>,

  /// Which views display this layer.
  pub visible_in_views: Vec<ViewId>,
}
```

5.10.2 Views

A view is a recipe for displaying a configuration of the score: which analysis layers are active, which parts are rendered, what overrides apply. Views are how the system supports condensed scores, analytical views, study scores, and lead-sheet projections.

```
pub struct ViewDefinition {
  pub id: ViewId,
  pub name: String,
  pub kind: ViewKind,

  /// Which analysis layers are active in this view.
  pub active_layers: Vec<AnalysisLayerId>,

  /// View-specific overrides.
  pub overrides: ViewOverrides,
}

pub enum ViewKind {
  FullScore,
  Part(PartDefinitionId),
  CondensedScore(CondensationSpec),
}
```

```

LeadSheet (LeadSheetSpec),
Analysis (AnalysisViewSpec),
Custom (CustomViewKindId),
}

```

5.11 Graph Invariants

The score graph maintains a set of structural invariants. Implementations **MUST** preserve these invariants across all edits.

REQUIREMENT

The following invariants hold over every well-formed score graph:

1. Every event in the arena has a valid `voice` field pointing to a voice that lists the event in its `events` vector.
2. Every event in a voice's events list has its `voice` field pointing back to that voice.
3. Events within a voice are sorted by position and do not overlap in time.
4. Every event's `EventPosition` and `EventDuration` variants agree with the time model of its enclosing region (Section 5.4.1). For aleatoric regions, agreement is governed by the region's declared `AleatoricAnchoringDiscipline`.
5. Every voice belongs to exactly one `StaffInstance`; every `StaffInstance` belongs to exactly one region; every region belongs to the canvas.
6. Every `StaffInstance.staff` resolves to a `Staff` declared at the score level. A single `StaffId` **MAY** be referenced by multiple `StaffInstances` in different regions (this is how continuous staff identity is expressed across regions); a `StaffId` **MUST NOT** be referenced by two `StaffInstances` in the same region.
7. Region time and staff extents do not simultaneously overlap (Section 5.5). A region's `staff_extent` lists exactly the `StaffIds` manifested by the region's `staff_instances`, with no duplicates.
8. Measures belong to exactly one `StaffInstance`. A region's metric organization is per-staff (admitting polymeter); barline alignment across staves is recorded via `BarlineAlignmentGroup` entries when explicit alignment is authoritative.
9. Every `TimeAnchor`'s `AnchorOffset` variant agrees with the time model of the target object's enclosing region (Section 3.5).
10. Every cross-cutting structure's references resolve to extant objects in the graph, except where explicit re-anchoring rules permit transient dangling states during edits (see Chapter 6).
11. Every identifier in the graph is unique within its kind. Identifiers reserved for system-derived objects (notably `VoiceIds` of system-promoted voices) **MUST** be derived from the deterministic function specified for their kind and **MUST NOT** collide with user-authored identifiers.
12. Every `IdentifiedPitch` in the arena has a `PitchId` that is unique within the score's pitch-identity index.

13. Every `SpellingScope::Pitch(PitchId)` resolves to a live or tombstoned `PitchId` in the pitch-identity index. A live target resolves to exactly one `IdentifiedPitch`; a tombstoned target is preserved for operation-log replay, undo, conflict reporting, and historical annotation, and **MUST NOT** be silently re-matched to a different live pitch.
14. Every notational decomposition attachment's target `EventId` resolves to a live or tombstoned event; attachments to tombstoned events follow the same preservation discipline as for tombstoned pitches.
15. For live decomposition attachments, the targeted event's duration matches the decomposition's component sum.
16. Every tuplet's member event durations sum to the tuplet's structurally-required total (Section 3.9). Operations that would break this invariant **MUST** include compensating changes (Section 6.12.2).
17. Every `Tie`'s `pitch_pairing` entries reference `PitchIds` that resolve to identified pitches belonging to the start and end events respectively. The class-specific adjacency, voice-membership, and pitch-equivalence rules of Section 5.7.2 hold for the tie's declared `TieClass`.
18. Every `Voice` declares an `origin` consistent with how it was created: `UserDeclared` for explicit user actions, `Imported` for foreign-format imports, `SystemPromoted` for voices allocated by concurrent-edit conflict resolution. The `VoiceId` of a `SystemPromoted` voice **MUST** be the deterministic derivation specified in Section 5.6.3.
19. Every `BarlineAlignmentGroup`'s members reference `StaffInstances` within the same region and `Measures` belonging to those instances. Group members **MUST NOT** reference staff instances or measures across region boundaries.

Implementations **MUST** reject graph configurations that violate any of the above invariants. Edits proposed in Chapter 6 must either preserve invariants directly or include compensating changes that restore them within the same operation.

5.12 Indexes and Auxiliary Structures

Performance-critical operations require indexed access. The specification states the required indexes; implementations **MAY** construct additional indexes provided they are kept consistent.

REQUIREMENT

Implementations **MUST** maintain at least the following indexes:

Event time index. Maps (region, voice) to a sorted view of event positions, supporting $O(\log n)$ time-range queries.

Cross-cutting reference index. Maps each object identifier to the cross-cutting structures referencing it, supporting $O(1)$ amortized lookup for re-anchoring.

Measure index. Maps measure number (and measure id) to `MeasureId` for fast naviga-

tion.

Spelling attachment index. Maps each `PitchId` to its attachments, partitioned by analysis layer.

Indexes **MUST** be invalidated and rebuilt (or incrementally updated) on the corresponding edits. The semantic operations chapter specifies which operations invalidate which indexes.

5.13 Forward References

- The complete catalog of semantic operations, their preconditions, postconditions, and re-anchoring rules is in Chapter 6.
- The mapping from score graph to layout intermediate representation is defined in Chapter 7.
- The on-disk serialization of every type introduced in this chapter is defined in Chapter 8.
- Engraving-specific types (`StemConfiguration`, `ClefId`, `ArticulationMark`, `ClefChange`, `KeySignatureChange`, `LineStyle`, and similar) are introduced informally here and fully defined in Chapter 7 where their interaction with the engraver is specified.

Semantic Operations and Concurrent Reduction

This chapter specifies the semantic operations on the score graph: the mutations through which the data model becomes a live model. It defines the operation framework, the canonical reduction procedure by which a set of operations becomes a materialized score state, the object existence model with tombstones, the conflict-record machinery, the re-anchoring rules, transactions, and undo. The chapter ends with representative operation specifications.

The collaboration mechanics specified here are the foundation on which Chapter 8's synchronization protocols build. The materialized score is the deterministic reduction of an operation set; the file format stores that operation set together with optional acceleration snapshots.

6.1 Design Principles

The operation set is canonical. A score's state is defined by the set of operations that have been committed to it. Any materialized graph is a deterministic reduction of that set; caches, snapshots, and partial reductions are acceleration structures, never the source of truth.

The replicated operation set is a CRDT; the materialized graph is not. The grow-only set of operation envelopes with compact causal metadata is a true CRDT: replicas independently accumulate envelopes and converge on the same set. The materialized score graph is not itself claimed to be a CRDT; it is the deterministic reduction of that set, defined by the reduction algorithm specified here.

Pairwise operation merge is rejected. Musical operations do not, in general, commute in intention: transpose-then-respell differs musically from respell-then-transpose; delete-then-tie differs from tie-then-delete. The framework therefore does not attempt to make every pair of operations algebraically commute. Instead, operations are reduced in a canonical order, and that order determines the outcome.

Conflicts, no-ops, and rejections are replicated facts. No operation silently disappears because it could not apply. Every operation has a deterministic `OperationEffect` recorded as part of the materialized state; every replica reducing the same operation set records the same effects with the same reasons.

Identifier-based targeting. Operations target objects by identifier, never by structural path.

Identifiers are stable under concurrent edits; paths are not.

Tombstones preserve identity. Deleted objects retain their identifiers as tombstones. References to tombstoned identifiers are not silently invalidated; they enter a tombstoned-target state unless the operation kind specifies deterministic repair.

Transactions materialize atomically. Primitive members of a transaction **MUST NOT** be independently visible as committed score states. Either the whole transaction reduces successfully, or the transaction transitions to a conflict effect with a recorded conflict.

Undo is forward, not backward. Undo is a new operation that computes a compensating edit against the current materialized state; it is not literal time travel. History remains append-only.

Two validation modes for advisory checking. Operations are subject to strict precondition checking in *authoring mode* (interactive edits) and lenient checking in *replay mode* (loading historical operations or applying remote operations). Validation modes apply only to advisory preconditions; invariant preconditions are enforced unconditionally.

6.2 The Operation Framework

6.2.1 Operation Identity and Stamps

Operations carry two related but distinct concepts: *identity* (who authored the operation, with what counter) and *stamp* (when the operation was committed, for ordering purposes).

```
/// Stable identity of an operation. Determined at authoring time.
pub struct OperationId {
    pub replica: ReplicaId,
    pub counter: u64,
}

/// Ordering metadata for an operation. Used to derive the canonical
/// reduction order; not part of the operation's identity.
pub struct OperationStamp {
    pub hlc: HybridLogicalClock,
    pub id: OperationId,
}

/// A hybrid logical clock combines wall-clock and logical components.
pub struct HybridLogicalClock {
    /// Physical time component, in canonical units (see Chapter 8).
    pub physical_time: WallClockTime,
    /// Logical counter, advanced when physical time does not.
    pub logical_counter: u32,
}
```

REQUIREMENT

OperationId **MUST** be stable: an operation's identity is fixed at the moment of authoring and **MUST NOT** change with reordering, retransmission, or merging into operation

sets at other replicas.

OperationStamp **MUST** be set by the authoring replica using a hybrid logical clock that respects causal order: for any operation *A* that is in operation *B*'s causal predecessor set, *A*'s stamp **MUST** be strictly less than *B*'s under the canonical comparison (Section 6.3.3).

6.2.2 Causal Context via Dotted Version Vectors

Operations carry a compact causal context, not an exhaustive predecessor list. This is essential: exhaustive predecessor lists scale linearly with history and become untenable.

```
/// A compact causal context: for each replica known to the authoring
/// replica, the highest contiguous counter observed, plus any
/// individual operation identifiers known but not contiguous (the
/// "dots").
pub struct CausalContext {
    /// Contiguous counter highest known per replica.
    pub vector: BTreeMap<ReplicaId, u64>,

    /// Individual operations known but not yet contiguous in the
    /// vector. Used when an authoring replica has observed an
    /// operation whose causal predecessors include operations not
    /// yet present.
    pub dots: BTreeSet<OperationId>,
}
```

REQUIREMENT

Causal contexts **MUST** be expressed as dotted version vectors as defined above. Exhaustive predecessor lists **MUST NOT** be used as the canonical causal representation; they may appear in debugging output or in the operation log's expanded form, but the wire and storage representation of causal context is the DVV.

Interval tree clocks and other compact causal representations **MAY** appear in future revisions as performance optimizations but are non-normative in this specification.

RATIONALE

Dotted version vectors are well-understood, well-tested, and the right complexity trade-off for the human-scale collaboration model Epiphany targets. Their normative use here closes the high-priority review item about exhaustive predecessor lists scaling badly with history.

6.2.3 Operation Envelopes

An operation is transmitted, stored, and reduced as an *envelope*: the operation's payload together with its identity, ordering stamp, causal context, and optional transaction grouping.

```
pub struct OperationEnvelope {
    /// Stable identifier of this operation.
```

```

pub id: OperationId,

/// Author of this operation (may differ from replica in shared
/// authoring sessions where multiple authors share a replica).
pub author: AuthorId,

/// Ordering stamp. Used for canonical reduction order; never
/// for identity.
pub stamp: OperationStamp,

/// Compact causal context: what operations were visible to the
/// authoring replica when this operation was created.
pub causal_context: CausalContext,

/// Optional transaction grouping. Multiple operations sharing a
/// TransactionId are reduced atomically.
pub transaction: Option<TransactionId>,

/// The mutation itself.
pub payload: OperationPayload,
}

pub enum OperationPayload {
    Primitive(OperationKind),

    /// A meta-operation that explicitly resolves a previously-recorded
    /// conflict (see Section~\ref{sec:semops:conflicts}).
    ResolveConflict(ResolveConflictPayload),

    /// A meta-operation that compensates for a previously-committed
    /// transaction; the realization of "undo" (see
    /// Section~\ref{sec:semops:undo}).
    UndoTransaction(UndoTransactionPayload),
}

/// The catalog of primitive operation kinds. The variants listed
/// here are normative for the core; additional kinds are introduced
/// by registered extensions via the Registered variant. The full
/// catalog including detailed payload schemas is the Operation
/// Catalog companion specification
/// (Appendix~\ref{app:deferred}).
pub enum OperationKind {
    // Event operations
    InsertEvent(InsertEventOp),
    DeleteEvent(DeleteEventOp),
    ModifyEvent(ModifyEventOp),

    // Pitch and tuning operations
    RespellPitch(RespellPitchOp),
    Transpose(TransposeOp),

    // Cross-cutting operations
    CreateCrossCutting(CreateCrossCuttingPayload),

```

```

DeleteCrossCutting(DeleteCrossCuttingPayload),
ModifyCrossCutting(ModifyCrossCuttingPayload),

// Region and structure operations
ChangeRegionTimeModel(ChangeRegionTimeModelOp),
InsertRegion(InsertRegionOp),
DeleteRegion(DeleteRegionOp),
InsertStaffInstance(InsertStaffInstanceOp),
DeleteStaffInstance(DeleteStaffInstanceOp),

// Layout-semantic operations
SetUserSystemBreak(SetUserSystemBreakOp),
SetUserPageBreak(SetUserPageBreakOp),

// Transaction declaration. Carries metadata; member primitives
// reference the transaction by id in their envelopes.
DeclareTransaction(TransactionDescriptor),

// Extension-defined primitive operation.
Registered(OperationKindRegistryId, SerializedRegisteredOp),
}

```

6.2.4 The Operation Set as CRDT

REQUIREMENT

The replicated operation set is a grow-only CRDT: replicas accumulate envelopes by gossip, broadcast, or any other delivery mechanism, and converge on the same set when they have observed the same envelopes. The set is set-valued, not multiset-valued: an envelope is a member of the set or it is not, with no count.

A replica **MUST NOT** discard envelopes from the operation set except by an explicit pruning operation (Chapter 8), which preserves canonical state through retained base snapshots and the post-pruning operation set.

6.2.5 OperationId Collisions and Replica Equivocation

Two received envelopes may carry the same `OperationId`. Acceptance must not depend on arrival order: if it did, two replicas observing the same envelopes in different orders could diverge on which one is canonical. The operation set therefore maps each `OperationId` to an `OperationSlot` rather than directly to an envelope.

```

pub enum OperationSlot {
    /// Exactly one canonical envelope is known for this
    /// OperationId. The envelope reduces normally.
    Single(OperationEnvelope),

    /// Two or more distinct canonical envelopes have been
    /// observed for this OperationId. The slot is equivocated.
    /// The operation produces no canonical effect until external
    /// recovery resolves the equivocation.
}

```

```

Equivocated {
    operation_id: OperationId,
    /// Every distinct canonical envelope observed for this
    /// OperationId. Identified by content hash to avoid
    /// duplicating large payloads. Sorted lexicographically by
    /// hash for deterministic enumeration.
    candidates: BTreeSet<EnvelopeHash>,
},
}

/// BLAKE3-256 hash of a canonical envelope serialization with
/// domain tag "MUSCENVH".
pub struct EnvelopeHash(pub [u8; 32]);

```

REQUIREMENT

Slot transitions are determined as follows, independently of arrival order:

- If no slot exists for the incoming envelope's `OperationId`: the incoming envelope is well-formedness checked (Section 6.2.6) and, if well-formed, a `Single` slot is created.
- If a `Single` slot exists and the incoming envelope is byte-identical (canonical serialization) to the slot's envelope: the duplicate is dropped silently; the slot is unchanged.
- If a `Single` slot exists and the incoming envelope's canonical bytes differ from the slot's envelope: the slot transitions to `Equivocated` with both envelopes' hashes in its `candidates` set. The previously-canonical envelope is retained in a diagnostic candidate store keyed by its hash; the incoming envelope is likewise retained. Neither envelope contributes to canonical reduction.
- If an `Equivocated` slot exists: the incoming envelope's hash is added to its `candidates` set if not already present; otherwise the duplicate is dropped. The envelope is retained in the candidate store. The slot remains `Equivocated`.

An equivocated slot **MUST NOT** contribute to canonical reduction. Reduction proceeds as if the operation does not exist: any operation that causally depends on the equivocated `OperationId` is reduced under the same rule as a missing causal predecessor (the dependent operation is retained but produces no effect until the equivocation is resolved).

Resolution is by explicit recovery action, not by arrival order or content. Three resolution mechanisms are permitted:

- *Transport-level credential revocation*: the collaboration transport revokes the equivocating replica's credentials and the user accepts one envelope timeline as canonical via an explicit reconciliation operation.
- *Local user choice*: the user (or an authorized administrator) issues an explicit `ResolveEquivocation` operation naming the `OperationId` and the chosen `EnvelopeHash`, promoting the slot back to `Single`.
- *Profile-declared deterministic policy*: a conformance profile **MAY** declare a deterministic selection function (e.g., "lowest canonical envelope hash wins"). This is a policy choice, not a default. A bundle reduced under a profile with a declared res-

olution policy is canonical under that profile; the same bundle reduced under a profile without such a policy remains anomalous.

If a bundle on disk contains an equivocated slot without a resolution policy or pending resolution operation, the bundle is anomalous. Readers **MUST** open it, if at all, in a diagnostic recovery mode that surfaces the equivocation prominently and that prohibits ordinary editing of the affected portion of canonical state.

RATIONALE

Replica equivocation (the same replica producing two distinct envelopes with the same `OperationId`) is a real failure mode: replica-key compromise, software bug, replay attack, or filesystem corruption can all produce it. The `OperationSlot` model makes equivocation a representable, order-independent state: two replicas that have observed the same set of envelopes converge on the same slot states regardless of the order of observation. Treating the first arrival as canonical would let a malicious early-arriver pin canonical state at one replica while a late-arriver pins different state at another replica; the slot model avoids that asymmetry by refusing to elevate either candidate without an explicit resolution action.

Recovery is a policy choice rather than a built-in default because the right answer depends on context: a collaborating team with credential revocation will resolve differently from a single-user backup-restore scenario, which differs again from an adversarial-document forensic scenario.

6.2.6 Envelope Acceptance and Malformed-Envelope Behavior

Operation envelopes arrive from local authoring and from remote replicas. Acceptance rules determine when an envelope enters the operation set.

REQUIREMENT

An incoming envelope is well-formed if and only if all of the following hold:

- Its `OperationId` consists of a 64-bit `ReplicaId` that is not `ReplicaId::SYSTEM_DERIVED` (operations are not authored by the system namespace) and a 64-bit counter.
- Its `stamp.id` is byte-identical to its top-level `id` field: an envelope's stamp **MUST** address the same operation as the envelope itself. This eliminates an ambiguity where an authoring replica could otherwise stamp an envelope as belonging to a different operation than its identity declares.
- Its `stamp.hlc.physical_time` is a finite, non-negative integer value, expressed in the canonical time unit (Chapter 8).
- Its causal context's DVV references only well-formed replica identifiers and non-negative integer counter values; dots are well-formed `OperationIds`.
- Its payload deserializes successfully against the profile's declared operation catalog.

An envelope that is not well-formed **MUST** be rejected at reception. Rejection is recorded in the local diagnostic log but does *not* enter the canonical operation set. Remote replicas **MAY** retransmit; persistent rejection indicates a non-conforming peer.

Acceptance **MUST NOT** require trust in the envelope's `stamp.hlc` content: a malicious or misconfigured peer may emit envelopes with implausible future physical times or zero/extreme logical counters. The canonical reduction order (Section 6.3.3) consumes the stamp as ordering metadata; it does not validate plausibility. Implementations **MAY** surface clock-skew warnings in diagnostics but **MUST NOT** reorder envelopes for plausibility reasons that would deviate from the canonical reduction order.

6.2.7 Per-Replica HLC Monotonicity

Per-replica monotonicity is a property of the *set* of accepted envelopes, not of arrival order. Out-of-order delivery is normal in gossip systems and is not a violation.

```
/// A range of envelopes from a single replica that have been
/// identified as monotonicity-violating and excluded from
/// ordinary canonical reduction. Retained for diagnostic and
/// recovery purposes.
pub struct AnomalousReplicaSegment {
    pub replica: ReplicaId,
    /// The smallest counter at or after which the replica's
    /// stream is anomalous. All envelopes from this replica
    /// with counter >= first_bad_counter are excluded from
    /// canonical reduction.
    pub first_bad_counter: u64,
    /// The detected anomaly reason.
    pub reason: ReplicaAnomalyReason,
    /// Operation ids in the excluded segment, retained for
    /// diagnostic recovery. Sorted by counter.
    pub excluded: Vec<OperationId>,
}

pub enum ReplicaAnomalyReason {
    /// The replica produced an envelope at counter c2 with a
    /// stamp tuple strictly less than a previously-known
    /// envelope at counter c1 < c2.
    HlcMonotonicityViolation {
        violating_pair: (OperationId, OperationId),
    },
    /// A registered anomaly reason defined by an extension or
    /// transport.
    Registered(ReplicaAnomalyRegistryId),
}
```

REQUIREMENT

For any two envelopes accepted into the operation set that share the same authoring `ReplicaId`, with operation- counter values $c_1 < c_2$:

The stamp tuple `(stamp.hlc.physical_time, stamp.hlc.logical_counter, id.counter)` of the envelope with counter c_1 **MUST** be lexicographically less than or equal to the corresponding tuple of the envelope with counter c_2 .

Arrival order **MUST NOT** be used to determine whether this property holds. If a replica receives counter 10 before counter 9, that is a property of delivery, not of monotonicity. If two envelopes from the same replica violate this property (their stamp tuples are inconsistent with their counter order), the document carries a **replica equivocation anomaly**. An `AnomalousReplicaSegment` is created (or extended) for the offending replica. The segment's `first_bad_counter` is the smaller of the two counters in the violating pair.

Effects on canonical reduction:

- Every envelope from the offending replica with counter $\geq \text{first_bad_counter}$ is excluded from canonical reduction. Such envelopes are retained on disk and in memory (the CRDT property does not allow silent removal) but are held in the `AnomalousReplicaSegment`'s excluded list, not in the canonical operation set.
- Envelopes from the offending replica with counter $< \text{first_bad_counter}$ remain in the canonical operation set and reduce normally.
- Any operation in the canonical set that causally depends on an excluded envelope reduces under the missing-causal-predecessor rule: the dependent operation is retained, produces no canonical effect, and is held pending resolution.
- The document is marked as containing a replica- equivocation anomaly. Ordinary editing **MAY** continue on the unaffected portion of canonical state; collaborative synchronization with the equivocating replica **MUST** be suspended until external resolution.

Resolution paths mirror the `OperationSlot` equivocation rules (Section 6.2.5): transport-level credential revocation followed by an explicit reconciliation operation, an explicit local recovery action, or a profile-declared deterministic policy.

Authentication of replica identities and revocation of compromised replicas are the responsibility of the collaboration transport (Appendix A). This chapter specifies only local detection, segment formation, exclusion from canonical reduction, and the structural shape of resolution.

RATIONALE

Excluding the violating segment from canonical reduction rather than letting it reduce-but-mark-tainted is the safer choice: a tainted reduction still produces canonical-looking state derived from operations the spec has flagged as untrustworthy. Excluding the segment keeps canonical state derived only from operations that satisfy the monotonicity invariant, at the cost of holding pending operations that causally depended on the excluded segment.

6.3 The Canonical Reduction

The materialized score graph is the deterministic reduction of the operation set. This section specifies how that reduction is performed.

6.3.1 Operation Lifecycle

Every operation traverses four phases:

Prepare. The local UI command validates enough to construct a well-formed operation envelope. *Prepare is advisory validation only*: failure here means the UI rejects the user’s action and never authors the operation. Prepare does not bind semantics; an operation that successfully prepares at one replica **MAY** be reduced to a conflict effect at another.

Commit. The operation envelope is added to the replicated operation set. After commit, the operation is part of canonical state and **MUST NOT** be discarded outside the pruning protocol.

Reduce. During materialization, the operation is processed by the canonical reduction algorithm against the current materialized state. Reduction is deterministic: every replica with the same operation set produces the same materialized state.

Report. The reduction produces a deterministic `OperationEffect` for each operation. Effects are visible graph facts: users can inspect what each operation did, and subsequent operations can address conflicts and no-ops directly.

6.3.2 The Reduction Algorithm

REQUIREMENT

The materialized score state is computed from the operation set as follows:

1. Sort the operation set into the canonical reduction order (Section 6.3.3).
2. Group consecutive operations that share a transaction id into transaction blocks.
3. For each non-transaction operation, apply the operation to the working state, computing its `OperationEffect`.
4. For each transaction block, apply all member operations atomically: either all succeed and the transaction reduces to `Applied` (with each member’s individual effect recorded), or some member fails its invariant preconditions, in which case the whole transaction reduces to a `Conflicted` effect with a recorded conflict.
5. Record every effect in the materialized state’s effect log (which is part of canonical state, not implementation detail).

Given the same operation set, conformant implementations **MUST** produce score states with byte-identical structural content and byte-identical effect logs. Differences in implementation language, storage layout, threading model, or hardware **MUST NOT** produce divergent reductions.

6.3.3 Canonical Reduction Order

REQUIREMENT

The canonical reduction order is the lexicographic ordering by:

1. Causal order: for operations A and B , if A is in B 's causal context (transitively, via DVV closure), then A precedes B .
2. For operations not ordered by causal order (i.e., concurrent operations), lexicographic order by:
 - (a) `stamp.hlc.physical_time` (ascending)
 - (b) `stamp.hlc.logical_counter` (ascending)
 - (c) `stamp.id.replica` (ascending, lexicographic on the replica identifier's canonical byte form)
 - (d) `stamp.id.counter` (ascending)

Causal order strictly dominates HLC order: wall-clock skew may affect the ordering of concurrent operations but **MUST NOT** cause a causal predecessor to sort after a causal successor. The HLC authoring rule (Section 6.2) guarantees this by construction.

6.3.4 Operation Effects

```

/// The deterministic effect of an operation under canonical
/// reduction. Recorded as part of materialized state.
pub enum OperationEffect {
    /// The operation applied cleanly with no compensating changes.
    Applied,

    /// The operation applied with deterministic compensating changes
    /// (e.g., re-anchoring records, attachment migration).
    AppliedWithRepair { repairs: Vec<RepairRecord> },

    /// The operation could not apply cleanly; a conflict record was
    /// created in the conflict registry.
    Conflicted { conflict: ConflictId },

    /// The operation's target was tombstoned. The operation is
    /// preserved in the operation set but has no effect on the
    /// materialized graph beyond the recorded effect.
    TombstonedTarget { target: TypedObjectId },

    /// The operation reduces to no effect. The reason is recorded
    /// and is part of canonical state.
    NoOp { reason: NoOpReason },
}

pub enum NoOpReason {
    /// The operation's target was tombstoned by a causally-prior
    /// operation, and the target operation has no compensating
    /// repair rule.
    TargetTombstoned,
```

```

    /// The operation duplicates a causally-prior operation's effect.
    AlreadyApplied,

    /// A later operation in the canonical order subsumed this
    /// operation's effect (e.g., a later DeleteEvent on the same
    /// target supersedes an earlier RespellPitch).
    SupersededByLaterOperation { superseder: OperationId },

    /// An invariant precondition that was satisfied at authoring
    /// time fails when the operation is reduced under concurrent
    /// edits. Distinguished from Conflicted because the operation's
    /// intent is not preserved: it simply has no applicable effect.
    /// The failure reason is a typed enum, not a free-form string:
    /// canonical effects MUST NOT contain free-form text, since two
    /// implementations could otherwise produce divergent canonical
    /// state while agreeing semantically.
    PreconditionFailedUnderReduction {
        reason: PreconditionFailureReason,
    },

    /// The operation belonged to a transaction whose other members
    /// failed; the transaction was conflicted, and this operation
    /// produced no individual effect.
    TransactionConflict,
}

pub enum PreconditionFailureReason {
    /// The target object identifier did not exist in the working
    /// state at the moment of reduction.
    TargetMissing,

    /// The target object was tombstoned by a causally-prior
    /// operation. Distinguished from NoOpReason::TargetTombstoned
    /// in that the kind's rules permitted an attempt at
    /// repair-on-tombstone, but the repair itself failed precondition.
    TargetTombstoned,

    /// The operation requires a region time model (e.g., metric)
    /// different from the one currently in effect at the target
    /// position.
    WrongRegionTimeModel,

    /// A tuple compensation declared by the operation is invalid
    /// against the current tuple structure at the target.
    TupleCompensationInvalid,

    /// An event duration the operation specifies is invalid in
    /// the target voice or region (e.g., overlapping a tied
    /// chord-onset boundary in an incompatible way).
    EventDurationInvalid,

    /// The operation's target position falls outside the region

```

```

    /// declared by its envelope, or addresses a region that does
    /// not exist.
    PositionOutsideRegion,

    /// A pitch-space or tuning-context precondition failed: e.g.,
    /// the operation declared a pitch in a pitch space that is not
    /// active at the target region.
    PitchSpaceMismatch,

    /// A voice precondition failed: the operation targeted a
    /// voice that does not exist or has been tombstoned.
    VoiceMissing,

    /// An extension-declared precondition failed. The extension
    /// is identified; the extension's own catalog enumerates the
    /// specific precondition codes.
    ExtensionPrecondition(ExtensionPreconditionId),

    /// Registered precondition code defined by a versioned
    /// registry.
    Registered(PreconditionFailureRegistryId),
}

/// A deterministic compensating change made during reduction. Each
/// RepairRecord describes one repair the reduction performed to
/// preserve graph invariants in the presence of a tombstoning or
/// migrating operation. Recorded as part of canonical state.
pub struct RepairRecord {
    /// The kind of repair performed.
    pub kind: RepairKind,

    /// The object affected by this repair.
    pub target: TypedObjectId,
}

pub enum RepairKind {
    /// A reference was re-anchored to a surviving target per the
    /// re-anchoring rule table (Section~\ref{sec:semops:reanchor}).
    Reanchored {
        from: TypedObjectId,
        to: TypedObjectId,
        reason: ReanchorReason,
    },

    /// A spanner or beam had members removed but enough remained
    /// for the structure to survive.
    SpannerTruncated {
        removed_members: Vec<TypedObjectId>,
    },

    /// A user-content reference was orphaned (target lost; reference
    /// preserved for diagnostic purposes).
    Orphaned,

```

```

    /// A reference whose existence required its target was
    /// cascade-deleted.
    CascadeDeleted,

    /// An attachment transitioned to tombstoned-target state.
    AttachmentTombstoned,

    /// A voice was promoted to a system-derived voice identity due
    /// to concurrent insertion collision.
    VoicePromoted {
        from: VoiceId,
        to: VoiceId,
    },

    /// A tuple was compensated according to the operation's
    /// declared TupleCompensation.
    TupleCompensated {
        compensation_kind: TupleCompensationKind,
    },

    /// A registered extension-defined repair kind.
    Registered(RepairKindRegistryId),
}

```

REQUIREMENT

Every operation in the operation set **MUST** produce exactly one `OperationEffect` under reduction. Effects **MUST** be deterministic: every replica reducing the same operation set in the canonical order produces the same effect for each operation, including identical `NoOpReason` values where applicable.

6.3.5 Cache and Snapshot Discipline

REQUIREMENT

Caches, snapshots, and partial reductions are acceleration structures only. They **MUST NOT** affect the canonical materialized state, which is defined solely by the operation set, the active conformance profile, and the normative reduction algorithm. Implementations **MUST** invalidate or recompute caches whenever any of the following change:

- The visible operation set (additions, or pruning operations under the file-format protocol).
- The causal order derived from any envelope's causal context.
- The reduction algorithm version declared by the active conformance profile.
- The active conformance profile itself.

Cache hits **MUST** return data that is bit-identical to the data a fresh reduction would

produce. Caches that cannot guarantee this property **MUST** be invalidated.

6.4 Object Existence and Tombstones

6.4.1 Add-Tombstone Semantics

Object lifetimes in the materialized graph follow add-tombstone semantics: an operation creates an object by minting a stable identifier, and an operation deletes an object by tombstoning that identifier. Tombstoned identifiers are retained; their resolution yields a deletion record rather than the original object.

REQUIREMENT

Identifiers **MUST** have one of three states in the materialized graph:

- **Live.** The identifier resolves to a current object.
- **Tombstoned.** The identifier resolves to a deletion record carrying the operation that performed the deletion and the operation that originally minted the identifier.
- **Unknown.** The identifier has not been seen by this replica. This state is observable in collaborative scenarios where envelopes arrive out of order; it resolves to either Live or Tombstoned once causal predecessors arrive.

Implementations **MUST NOT** resurrect tombstoned identifiers (recreating an object with a previously-tombstoned id) except through an explicit inverse-of-delete operation in the operation set. Independent re-creation is forbidden.

6.4.2 Tombstone Retention

REQUIREMENT

Tombstones **MUST** be retained while any of the following hold:

- Pending replication: operations referencing the tombstoned identifier may not yet have been observed by all replicas.
- Pending undo: a transaction containing the deletion is within undo range.
- Active conflict resolution: a conflict record references the tombstoned identifier and has not been resolved.
- Attachment resolution: a tombstoned-target attachment remains in the score for historical or diagnostic purposes.

Tombstone garbage collection is governed by the pruning protocol in Chapter 8. Tombstones **MUST NOT** be discarded outside that protocol.

6.4.3 Reference Resolution Across Tombstones

REQUIREMENT

A reference to a tombstoned identifier resolves to a `TombstonedTarget` state rather than being silently invalidated. Operations targeting tombstoned identifiers reduce to `TombstonedTarget` or `NoOp` effects according to their kind's declared rules (Section 6.12). Re-anchoring rules (Section 6.7) define how cross-cutting structures and attachments respond to tombstoned references at reduction time.

6.5 Conflict Records

When an operation cannot apply cleanly, the reduction produces a `Conflicted` effect referencing a `ConflictRecord` in the score's conflict registry. Conflict records are first-class graph objects: stable, addressable, visible to users, and addressable by subsequent operations.

6.5.1 Conflict Record Type

```
pub struct ConflictRecord {
    /// Content-derived identifier (Section below).
    pub id: ConflictId,

    /// The operations that participated in producing this conflict.
    /// At least two for a true conflict; one for an operation that
    /// failed precondition checking under reduction.
    pub caused_by: Vec<OperationId>,

    /// The kind of conflict, governing what resolution options
    /// exist.
    pub kind: ConflictKind,

    /// Objects affected by this conflict, for diagnostic and UI
    /// navigation purposes.
    pub affected_objects: Vec<TypedObjectId>,

    /// Current resolution state. Conflicts begin Unresolved; a
    /// later ResolveConflict operation transitions them to Resolved
    /// or Dismissed.
    pub resolution_state: ConflictResolutionState,
}

pub enum ConflictKind {
    /// Two concurrent operations attempted to write the same
    /// non-LWW field. The winning operation's effect is in the
    /// materialized state; the losing operation's effect is in the
    /// conflict record for user inspection.
    StructuralFieldCollision {
        winner: OperationId,
        loser: OperationId,
        field: FieldPath,
    },
}
```

```

    },

    /// A transaction's primitive members were partially applicable
    /// under reduction; the whole transaction was rejected.
    TransactionConflict {
        transaction: TransactionId,
        failed_members: Vec<OperationId>,
    },

    /// An operation targeted a tombstoned object and could not be
    /// repaired by the kind's declared rules.
    TombstonedTarget {
        target: TypedObjectId,
        operation: OperationId,
    },

    /// Re-anchoring failed: no deterministic target could be found.
    ReanchorFailure {
        original_referent: TypedObjectId,
        referencing_object: TypedObjectId,
    },

    /// A change-region-time-model operation produced contained
    /// events whose coordinate kinds are incompatible with the new
    /// time model; the migration could not be performed
    /// deterministically.
    TimeModelMigrationFailure {
        region: RegionId,
        incompatible_events: Vec<EventId>,
    },

    /// A registered extension operation's reduction failed in a
    /// way described by the extension. Opaque to the core.
    ExtensionConflict {
        kind_id: ConflictKindRegistryId,
        details: SerializedExtensionConflict,
    },
}

pub enum ConflictResolutionState {
    Unresolved,
    Resolved { by: OperationId, action: ResolutionAction },
    Dismissed { by: OperationId },
}

pub enum ResolutionAction {
    /// Accept the losing operation's effect (replacing the winner).
    AcceptLoser,
    /// Reapply the winner explicitly (no semantic change; clears the
    /// conflict).
    KeepWinner,
    /// User-authored replacement that supersedes both.
    Override { override_operation: OperationId },
}

```

```

    /// Re-anchor to a user-chosen target.
    Reanchor { new_target: TypedObjectId },
    /// Custom resolution for registered conflict kinds.
    Registered(ResolutionRegistryId),
}

```

6.5.2 The Conflict Registry

The score graph carries a top-level conflict registry alongside the cross-cutting registry:

```

pub struct ConflictRegistry {
    pub records: Vec<ConflictRecord>,
}

```

REQUIREMENT

The conflict registry is part of canonical materialized state. Conflict records persist until they are resolved or dismissed by explicit `ResolveConflict` operations. Conflict records **MUST NOT** be silently removed; their lifetime is governed by the operation set, not by UI state.

The layout pipeline (Chapter 7) **SHOULD** provide visual indication of unresolved conflicts at their affected objects; this is the user-facing manifestation of the conflict registry's contents.

6.5.3 ConflictId Derivation

Conflict records are produced during deterministic reduction, not authored by replicas. Their identifiers **MUST** therefore be content-derived, not minted from a local replica-plus-counter sequence. Otherwise, two replicas reducing the same operation set would produce conflict registries with disagreeing identifiers, breaking canonical state.

REQUIREMENT

Each `ConflictId` **MUST** be derived deterministically from the conflict's content via BLAKE3 truncation:

```

fn derive_conflict_id(
    kind: &ConflictKind,
    causing_operations: &[OperationId],
    affected_objects: &[TypedObjectId],
) -> ConflictId {
    let mut preimage = Vec::new();
    preimage.extend_from_slice(b"MUSCCONF");
    preimage.extend_from_slice(&kind.canonical_bytes());

    // Sort causing operations lexicographically by canonical bytes
    let mut sorted_ops = causing_operations.to_vec();
    sorted_ops.sort_by(|a, b| a.canonical_bytes().cmp(&b.canonical_bytes()));
}

```

```

    for op in &sorted_ops {
        preimage.extend_from_slice(&op.canonical_bytes());
    }

    // Sort affected objects lexicographically by canonical bytes
    let mut sorted_objs = affected_objects.to_vec();
    sorted_objs.sort_by(|a, b| a.canonical_bytes().cmp(&b.canonical_bytes())
    );
    for obj in &sorted_objs {
        preimage.extend_from_slice(&obj.canonical_bytes());
    }

    let hash = blake3(&preimage);
    ConflictId(u128::from_be_bytes(
        hash[0..16].try_into().unwrap()
    ))
}

```

The canonical `ConflictKind` byte representation **MUST** include the kind discriminant and all kind payload (winning operation, losing operation, field path, transaction id, tombstoned target, original referent, region, extension kind identifier, etc.), so that distinct conflicts have distinct preimages by construction. Implementations **MUST NOT** introduce an ordinal or local counter into the preimage; conflicts that share kind, causing operations, and affected objects are by definition the same conflict.

RATIONALE

Content-derived identifiers are the only mechanism that makes the conflict registry a deterministic materialized fact. Every replica reducing the same operation set produces the same conflict records with the same identifiers, addressable by subsequent `ResolveConflict` operations whose payloads carry those identifiers by reference. Truncation from 256 to 128 bits is safe: the birthday-collision probability for any practical conflict count is vanishingly small, and the same truncation discipline already governs other 128-bit identifiers in the format.

6.5.4 Conflict Resolution Operations

```

pub struct ResolveConflictPayload {
    pub target: ConflictId,
    pub action: ResolutionAction,
}

```

REQUIREMENT

A `ResolveConflict` operation transitions a conflict's state from `Unresolved` to `Resolved` (with the declared action applied) or `Dismissed` (with no semantic change to the materialized graph beyond the conflict record's state).

Concurrent `ResolveConflict` operations on the same conflict are themselves subject to canonical reduction: the earlier operation under the canonical order is applied; the later operation reduces to `NoOp` with reason `AlreadyApplied` if its action is equivalent, or to `Conflicted` with a meta-conflict record if its action differs.

6.6 Transactions

Transactions are first-class replicated operation groups. They reduce atomically: either every primitive member applies, or the whole transaction conflicts.

```
pub struct TransactionDescriptor {
    pub id: TransactionId,

    /// Human-readable label, for the undo history and diagnostics.
    pub label: String,

    /// Optional categorization (used by UIs and analytics).
    pub category: Option<TransactionCategory>,
}

pub struct TransactionId(pub u128);
```

Each primitive operation belonging to a transaction carries the transaction’s identifier in its envelope’s `transaction` field. The transaction descriptor itself is a separate operation envelope (`DeclareTransaction`) whose payload contains the `TransactionDescriptor`.

6.6.1 Transaction Descriptor Causal Ordering

The canonical reduction order is causal-first, then HLC, then operation id (Section 6.3.3). For the descriptor to be available when its members are reduced, the ordering between descriptor and members must be guaranteed by causal dependency, not by HLC stamps.

REQUIREMENT

Every primitive operation member of a transaction **MUST** causally depend on the transaction’s `DeclareTransaction` envelope: the descriptor’s `OperationId` **MUST** appear in the member’s `causal_context` (either in the per-replica DVV range or among its dots). Authoring implementations **MUST** observe and include the descriptor in the causal context of each member they author.

Delivery order need not respect this dependency: an out-of-order delivery in which a member arrives before its descriptor is normal and is handled by the canonical reduction algorithm, which sorts by causal order before reducing.

At reduction time:

- If the descriptor is present in the operation set and causally precedes every member: the transaction is well-formed; it is reduced atomically per the rule below.
- If a primitive operation declares membership in a transaction whose

`DeclareTransaction` envelope is absent from the operation set, or whose envelope is present but does not causally precede the member: the member is malformed against the transaction model. It reduces to `NoOp{reason: TransactionConflict}` and contributes to a synthesized conflict record of kind `ConflictKind::TransactionConflict` naming the malformed member and the missing-or-misordered descriptor. The member is preserved in the operation set (the CRDT property prohibits removal) but produces no effect on canonical state beyond the recorded conflict.

RATIONALE

Requiring causal dependency rather than HLC ordering matches how transactions actually arise at the authoring replica: a user begins an edit (descriptor), then commits its primitives. The primitives can causally observe the descriptor because they are authored after it on the same replica. The canonical reduction order then handles them correctly without any special transaction-extraction phase.

This rule also makes equivocating descriptors impossible: a member cannot claim membership in a transaction whose descriptor it has not seen.

REQUIREMENT

Transactions reduce atomically. During reduction:

- All primitive members of a transaction are gathered into a transaction block at the point of the earliest member's canonical position.
- The block is reduced as a single unit. Each member is evaluated against the working state produced by prior members within the same block.
- If every member's invariant preconditions are satisfied, the transaction reduces to `Applied`; each member's individual effect is recorded as part of the transaction's composite effect.
- If any member's invariant preconditions fail, the entire transaction reduces to `Conflicted`: a conflict record is created with kind `ConflictKind::TransactionConflict`, listing the failed members; no member's effect is applied; each member's individual effect is recorded as `NoOp{reason: TransactionConflict}`.

Primitive members of a transaction **MUST NOT** be observable as independently committed score states. Observers see either the pre-transaction state or the fully applied post-transaction state; they never see a partial intermediate.

RATIONALE

Atomic transaction reduction is essential for the tuplelet compensation model (Section 6.12.2): the delete operation and its compensating insert (or rewrite, or cascade) are committed together as a single transaction, so the score is never observably broken between them.

Local materialization may stream a transaction’s primitives to the UI for responsive feedback, but the materialized canonical state visible to remote observers transitions only at transaction boundaries.

6.7 Re-Anchoring

When an operation tombstones an object, every other object referencing it must respond deterministically. The response is specified as a total deterministic function.

6.7.1 The Re-Anchor Function

```
pub enum ReanchorResult {
    /// The reference is replaced with a reference to a new target.
    Reanchored {
        new_target: TypedObjectId,
        reason: ReanchorReason,
    },

    /// The reference is preserved but its target is tombstoned.
    /// The referencing object remains in the graph in a
    /// tombstoned-target state.
    TombstonedTarget,

    /// The referencing object is marked orphaned but retained.
    Orphaned,

    /// Re-anchoring cannot be performed deterministically; a
    /// conflict is recorded.
    Conflicted { conflict: ConflictId },

    /// The referencing object is cascade-deleted (tombstoned).
    CascadeDeleted,
}

pub enum ReanchorReason {
    SameVoiceNearer,
    SameStaffInstanceNearer,
    SameStaffNearer,
    SameRegionNearer,
    ExplicitFallback,
    DeclaredByExtension(ReanchorReasonRegistryId),
}
```

6.7.2 Total Ordering for “Nearest”

The notion of “nearest surviving anchor” is defined as a total ordering. There is no discretionary search and no implementation freedom to choose among “equally near” candidates.

REQUIREMENT

For an object kind K requiring re-anchoring to the nearest surviving K , “nearest” is computed as the strict lexicographic minimum over the surviving candidates of:

1. Containment proximity (smaller is closer): same voice (0), same staff instance (1), same staff (2), same region (3), same canvas (4). Candidates further away than the kind’s declared maximum proximity are excluded.
2. Absolute time distance from the tombstoned referent’s resolved position, measured under a canonical conversion (musical time within metric regions, wall-clock time within proportional regions, declared per-region for aleatoric regions).
3. Direction preference: forward (0) before backward (1), unless the kind declares the opposite preference.
4. Object identifier (lexicographic on the typed identifier’s canonical byte form, ascending) as the final tie-break.

If no candidate satisfies the kind’s declared proximity bound, the result is `Orphaned` (for user-content kinds such as comments and analytical annotations) or `CascadeDeleted` (for kinds whose existence requires their target, such as ties and beams).

Numeric proximity bounds, direction preferences, and the orphaned-versus-cascade choice are declared per kind in the re-anchoring rule table below; they are part of the kind’s conformance contract and **MUST NOT** be reinterpreted by implementations.

6.7.3 The Re-Anchoring Rule Table

The following table specifies, for each (referencing kind, referent kind) pair, the re-anchoring action under canonical reduction. The table is normative.

Referencing	Referent	Action	Parameters
Slur	Endpoint event	Re-anchor to nearest forward event in same voice; cascade-delete if fewer than two members survive	proximity max: same voice; direction: forward preferred
Slur	Interior event	No action	—
Tie	Either endpoint	Cascade-delete	Tie’s existence requires both endpoints.
Beam	Member event	Truncate. Cascade-delete if fewer than two members remain	proximity max: same voice

Referencing	Referent	Action	Parameters
Tuplet	Member event	Operation MUST provide tuplet compensation (Section 6.12.2); otherwise the operation conflicts at precondition check	—
Spanner	Anchor	Re-anchor to nearest surviving anchor of same kind; cascade-delete if no surviving anchor within proximity bound	proximity max: same staff instance; bound: per-spanner-kind
Marker	Anchor	Re-anchor to nearest event in same staff instance	proximity max: same staff instance; orphan on failure
Comment	Anchor	Orphan	User content never silently deleted.
Cue event	Source event	Cascade-delete	A cue with no source is meaningless.
Graphic gesture	Anchor event	Re-anchor to nearest surviving event of same staff instance; for Free anchoring, no action; for Range anchoring, truncate	proximity max: same staff instance

Referencing	Referent	Action	Parameters
Trajectory event	Endpoint pitch	Re-anchor: replace <code>EventPitch(PitchId)</code> with <code>ExplicitPitch</code> capturing the tombstoned pitch's last known value	—
Analytical annotation	Anchor	Re-anchor to time range preserving original extent; orphan if range cannot be reconstructed	—
Spelling attachment	Target pitch	Transition to tombstoned- target state	Attachment is preserved for diagnostic and undo purposes.
Decomposition attachment	Target event	Transition to tombstoned- target state	Attachment preserved for diagnostic and undo purposes.
Voice	Containing staff instance	Cascade- delete	—
Event	Containing voice	Cascade- delete	—
Region	Canvas	Refuse (canvas is not deletable)	—
Part definition	Referenced staff	Remove staff from part's staff list; orphan if part becomes empty	—

REQUIREMENT

Re-anchoring **MUST** be performed as part of the same reduction step that tombstones the referent. The resulting graph state **MUST** satisfy every invariant in Section 5.11. Re-anchoring actions **MUST** be recorded as `RepairRecord` entries in the triggering operation's effect.

6.8 Undo

Undo is not literal time travel: it is the authoring of a new operation that compensates for a previously-committed transaction against the current materialized state. This preserves the append-only operation set and is correct under concurrent edits.

6.8.1 UndoTransaction Payload

```
pub struct UndoTransactionPayload {
    /// The transaction being undone.
    pub target: TransactionId,

    /// Policy governing how to undo when the target's effects
    /// have been partially superseded by later operations.
    pub policy: UndoPolicy,
}

pub enum UndoPolicy {
    /// Compute the inverse semantic operations against the current
    /// materialized state. If any cannot apply cleanly (because the
    /// target objects have been tombstoned, fields have been
    /// further modified, etc.), the entire undo conflicts.
    StrictInverse,

    /// Undo what remains valid; record conflicts or no-ops for the
    /// rest. The undo operation succeeds, but partial.
    BestEffort,

    /// Also undo operations that are causally or semantically
    /// dependent on the target transaction. Dependency is computed
    /// from causal contexts and explicit dependency links;
    /// implementations MUST NOT infer broad causal dependence
    /// heuristically.
    Cascade,
}
```

6.8.2 Undo Semantics

REQUIREMENT

An `UndoTransaction` operation is committed to the operation set like any other operation; it does not modify history. Its reduction computes a compensating edit against the materialized state at the point of its canonical position:

- For `StrictInverse`: the inverse of each primitive in the target transaction is computed against the current working state. If every inverse applies cleanly, the undo transaction reduces to `Applied` with all compensating effects recorded. If any inverse cannot apply (target tombstoned, field modified, etc.), the undo transaction reduces to `Conflicted` with a conflict record.

- For `BestEffort`: inverses that apply cleanly are applied; inverses that cannot are recorded as `NoOp` effects with their reasons. The undo transaction reduces to `AppliedWithRepair`.
- For `Cascade`: the dependency set is computed first (causal closure plus explicit dependency links). The transaction's inverses, plus inverses of dependents in reverse order, are applied per the chosen sub-policy (typically `StrictInverse`).

Equality of the resulting graph state to the pre-target state is defined as *content equivalence*: every live object's fields, every attachment, every cross-cutting structure matches. Operation identifiers, stamps, undo stacks, caches, and storage layout are *not* required to match.

RATIONALE

Defining undo as a forward operation eliminates the contradiction in earlier passes where undo had to “restore previous state” while remote edits had been integrated. The compensating-edit approach respects the operation set's append-only nature and generalizes cleanly to concurrent collaboration: undoing a transaction whose effects have been entwined with remote edits is expressible as a deterministic compensating operation, possibly partial.

6.9 LWW Discipline

Last-writer-wins reduction is permitted only for explicitly-marked advisory scalar fields. Structural musical objects are governed by their kind's operation-specific reduction rules.

6.9.1 The `LwwAdvisory` Marker

Fields eligible for LWW reduction are explicitly marked as `LwwAdvisory`. No unmarked field may use LWW semantics.

```
/// Marker attribute, applied to fields whose concurrent modifications
/// are resolved by last-writer-wins under the canonical reduction
/// order.
#[attribute_only]
pub struct LwwAdvisory;

// Example field declarations:
//
//   #[lww_advisory]
//   pub title: Option<String>
//
//   #[lww_advisory]
//   pub composer: Option<String>
```

6.9.2 Eligible Fields

REQUIREMENT

The following classes of field **MAY** be marked `LwwAdvisory` and reduced by last-writer-wins:

- Score metadata: title, subtitle, composer, lyricist, copyright, work number, opus, dedication, free-form annotations.
- View preferences: which layers are active, which parts are shown, viewport, zoom level (per-replica state in addition; only the explicitly shared portions are LWW).
- Layout hint preferences: default page size, default margins, default staff spacing recommendations.
- Page-break and system-break preferences on layout-semantic structures (the user's stated preference, distinct from the engraver's decision).
- Optional advisory display hints on cross-cutting structures: line color preference, line-style preference, label-position preference. Endpoints, target identities, semantic class, and attachment ownership **MUST NOT** be LWW.

All other fields, including every field carrying musical structure, identity, ownership, or semantics, **MUST** use operation-specific reduction rules defined per operation kind. Implementations **MUST NOT** default to LWW for unmarked fields.

RATIONALE

LWW is acceptable for fields where the worst case of arbitrary deterministic resolution is benign: two authors editing the composer's name concurrently is a metadata collision, not a musical loss. LWW is catastrophic for fields where wholesale replacement destroys musical meaning: two authors editing a chord's pitch list concurrently must not produce "one author's chord wins." The marker discipline makes the boundary explicit and auditable.

6.10 Validation Modes

Operations are subject to precondition checking. Two modes distinguish authoring contexts:

Authoring mode. Interactive edits. All preconditions are enforced: invariant preconditions (which preserve graph invariants) and advisory preconditions (which encode user-intent constraints, range checks, and style policy).

Replay mode. Replaying historical operations or applying remote operations under reduction. Only invariant preconditions are enforced; advisory preconditions are skipped, since they represent the authoring replica's local policy at the moment of authoring, not invariants of the canonical state.

REQUIREMENT

Every operation specification **MUST** classify each precondition as invariant or advisory. Invariant preconditions **MUST** hold in all modes; advisory preconditions **MUST** hold in authoring mode and **MAY** fail silently in replay mode. Implementations **MUST NOT** reclassify a precondition without a corresponding revision to this specification.

6.11 Per-Kind Reduction Summary

The following table summarizes the reduction discipline for each representative operation kind. Detailed specifications appear in the next section.

Operation	Reduction Rule	Concurrency Outcomes
InsertEvent	Position-keyed insertion. Concurrent same-position inserts whose durations overlap promote the lexicographically-greater operation to a system-promoted voice (Section 5.6.3)	Voice promotion; non-overlap invariant preserved by allocation.
DeleteEvent	Tombstone the event and contained pitch identifiers. Apply tuplet compensation from the operation. Re-anchor referencing structures	Delete-wins against concurrent field modifications on the same event; concurrent same-target deletes are idempotent.
RespellPitch	Field overwrite by canonical reduction order; the later operation wins. Earlier respellings of the same pitch are preserved in the conflict record if they differ	Conflict record records the losing spelling for user inspection.
Transpose	Apply in canonical reduction order. Pitch identifiers preserved. Operations do not commute	Reduction order decides; conflict record entries for non-trivial intervals only when explicit dependency rules require.

Operation	Reduction Rule	Concurrency Outcomes
CreateCrossCutting	Set union. Concurrent additions of distinct cross-cutting structures all succeed	Tombstoned endpoints cause re-anchoring or cascade-delete per kind.
ChangeRegionTimeModel	Structural migration. Contained events whose coordinate kinds become incompatible cause the operation to conflict	Concurrent contained edits authored before the migration may conflict; explicit re-base rules per migration kind.
SetUserSystemBreak	LwwAdvisory on per-location break preference	Last-writer-wins on the break state at the same anchor.

6.12 Representative Operations

The remainder of the chapter specifies a representative selection of operations, one from each major category. The full catalog is delivered as a separate conformance specification.

These examples illustrate the contract that every operation in the catalog **MUST** satisfy. Each operation specification includes its reduction rule under the canonical reduction model (Section 6.3).

6.12.1 InsertEvent

Category: Event operations.

```
pub struct InsertEventOp {
    /// The voice the event is inserted into.
    pub voice: VoiceId,

    /// Position within the voice.
    pub position: EventPosition,

    /// The event to insert. Its EventId and any contained PitchIds
    /// are minted at operation-authoring time as a single causal
    /// step. The event's voice field must match the target voice.
    pub event: Event,
}
```

Invariant preconditions:

- The target voice exists.
- The event's `voice` field equals the target voice's id.
- The event's `id` is not present elsewhere in the arena.
- Every `PitchId` contained in the event (in an `IdentifiedPitch`, a trajectory endpoint, or a stepwise trajectory shape) is unique within the score's pitch-identity index.

- The position is within the enclosing region’s time extent.
- The event’s position and duration variants agree with the enclosing region’s time model.
- Inserting at the position does not produce an event overlap within the voice. Concurrent inserts that would otherwise produce overlap are resolved per the reduction rule below.

Advisory preconditions (authoring mode only):

- For pitched events, every pitch is within the instrument’s declared range, if any.
- The event’s duration does not extend past a region boundary in a way that would require splitting.

Effect: The event is added to the arena; the voice’s event list is updated to include the event in sorted position. Every contained `PitchId` is registered as live in the pitch-identity index. Any attachments declared on the event (spellings, decompositions) are added to their respective indexes.

Postconditions:

- The voice’s events list remains sorted and non-overlapping.
- The event arena’s lookup index resolves the new event.
- The event time index includes the new event for its region and voice.
- Every contained pitch is resolvable through the pitch-identity index.

Inverse: `DeleteEvent` of the inserted event, with captured state equal to the event itself plus any attachments added.

Reduction rule: Position-keyed insertion with voice promotion on collision. Concurrent inserts at the same position whose durations would overlap resolve as follows: the operation with the lexicographically-greater `OperationId` is placed in a newly allocated voice on the same staff instance, with `VoiceOrigin::SystemPromoted` (Section 5.6.3). The new voice’s `VoiceId` is derived deterministically from the staff instance, the original target voice, and the two operations’ identifiers, so all replicas independently allocate the same promoted voice. The non-overlap invariant is preserved by allocation rather than by rejection. Subsequent user normalization operations can merge the promoted voice back into the target, rename it, or accept it as an intentional second voice.

6.12.2 DeleteEvent

Category: Event operations.

```
pub struct DeleteEventOp {
    pub event: EventId,

    /// Compensation for tuplet membership, if any. Required when the
    /// target event is a member of one or more tuplets.
    pub tuplet_compensation: TupletCompensation,
}

pub enum TupletCompensation {
    /// Target is not in any tuplet; no compensation required.
```

```

    NotInTuplet,

    /// Replace the deleted event with a rest of the same duration.
    /// The rest receives a freshly-minted EventId; the deleted
    /// event's EventId and PitchIds are tombstoned.
    ReplaceWithRest { new_rest: Rest },

    /// Rewrite the enclosing tuplet(s) to remain structurally
    /// consistent. The list specifies the rewritten tuplets in full.
    RewriteTuplets { rewrites: Vec<TupletRewrite> },

    /// Cascade-delete the entire tuplet group(s) containing the
    /// target event. The listed tuplet ids are cascaded.
    CascadeDeleteTuplets { tuplets: Vec<TupletId> },
}

pub struct TupletRewrite {
    pub tuplet: TupletId,
    pub new_ratio: TupletRatio,
    pub new_members: Vec<EventId>,
}

```

Invariant preconditions:

- The target event exists in the arena (i.e., is live, not tombstoned).
- If the target event belongs to one or more tuplets, the `tuplet_compensation` field **MUST** be a variant other than `NotInTuplet`, and the chosen compensation **MUST** preserve the tuplet structural consistency invariant (Section 3.9) after the delete is applied.
- If the target event does not belong to any tuplet, the `tuplet_compensation` field **MUST** be `NotInTuplet`.
- For `ReplaceWithRest`: the replacement rest's duration **MUST** equal the deleted event's duration; the rest's voice and position **MUST** match the deleted event.
- For `RewriteTuplets`: every listed tuplet's resulting member-duration sum **MUST** match the tuplet's structurally-required total.
- For `CascadeDeleteTuplets`: every tuplet listed **MUST** contain the target event as a member.

Effect: The event is removed from the arena and from its voice's event list. The event's `EventId` and every contained `PitchId` are tombstoned in their respective identity indexes: they remain resolvable to "deleted" markers and **MUST NOT** be reused for new objects. The chosen tuplet compensation is applied in the same delta: a replacement rest is inserted, the tuplet is rewritten, or the tuplets are cascade-deleted as specified. Cross-cutting structures referencing the event or its pitches are re-anchored per the rule table (Section 6.7). Live attachments targeting the event or its pitches transition to tombstoned-target state and are preserved for replay, undo, and historical reference; they are not silently deleted.

Postconditions:

- The event is no longer live in the arena.

- Every cross-cutting structure that referenced the event has been re-anchored or cascade-deleted.
- The event’s identifier and every contained pitch identifier are tombstoned and resolvable only to a deletion record.
- The tuple structural consistency invariant holds for every tuple that was previously consistent.

Inverse: `InsertEvent` reconstructing the event from captured state, re-promoting the tombstoned identifiers to live, reversing the tuple compensation (delete the replacement rest; restore the original tuple structure; un-cascade tuple deletions), and re-creating any cross-cutting structures that were cascade-deleted as a consequence. Re-promotion preserves all stable identifiers; new identifiers **MUST NOT** be minted by the inverse.

Reduction rule: Set intersection with delete-wins. Concurrent deletion and modification of the same event resolves to deletion; the modification is dropped. Concurrent re-anchoring on the same referent resolves by the deterministic total ordering specified in Section 6.7.

RATIONALE

Requiring explicit tuple compensation prevents primitive deletion from leaving the score in a state that violates the tuple structural consistency invariant. The compensation is part of the same delta as the deletion, so concurrent observers see either the pre-delete state or a fully consistent post-delete state, never an intermediate broken state.

6.12.3 CreateCrossCutting (Slur)

Category: Cross-cutting structure operations.

```
pub struct CreateCrossCuttingOp {
    pub structure: CrossCuttingStructure,
}

pub enum CrossCuttingStructure {
    Slur(Slur),
    Tie(Tie),
    Beam(Beam),
    Spanner(Spanner),
    Marker(Marker),
    Repeat(RepeatStructure),
    Analytical(AnalyticalAnnotation),
    Comment(Comment),
    GraphicGesture(GraphicGesture),
}
```

Invariant preconditions (Slur case):

- The slur’s id is unique within the cross-cutting registry.
- The slur’s start and end events both exist.
- The start and end events are in the same voice.

- The start event precedes (or equals) the end event in voice order.

Advisory preconditions (Slur case):

- The slur does not span a region boundary, unless explicitly permitted by region configuration.

Effect: The slur is added to the cross-cutting registry’s slur list. The cross-cutting reference index is updated.

Postconditions:

- The slur is queryable by id and by referenced events.

Inverse: DeleteCrossCutting of the created slur.

Reduction rule: Set union. Concurrent creation of multiple slurs over overlapping ranges is valid: both slurs exist.

6.12.4 RespellPitch

Category: Pitch and tuning operations.

```
pub struct RespellPitchOp {
  pub pitch: PitchId,
  pub new_spelling: PitchSpelling,
}
```

Invariant preconditions:

- The target pitch exists in some event in the arena (i.e., is live, not tombstoned).
- The new spelling is valid in the active pitch space: every accidental in the spelling’s `accidentals` stack belongs to the active accidental registry, the nominal exists in the space’s nominal registry, and the octave is in range.
- Duplicate accidentals in the spelling’s stack are absent unless the accidental’s registered combination semantics explicitly permit repetition.
- The new spelling represents the same scale position as the pitch’s current spelling, or is enharmonically equivalent under the active tuning system. (Respelling does not change sounding pitch.)

Effect: A spelling attachment with source `UserChosen` is created (or replaces an existing `UserChosen` attachment) targeting the pitch. The spelling attachment index is updated. The pitch’s `PitchId` is unaffected.

Postconditions:

- Spelling resolution for the pitch yields the new spelling, subject to the precedence configuration.
- The pitch’s scale position is unchanged.
- The pitch’s acoustic realization is unchanged.
- The pitch’s `PitchId` is unchanged.

Inverse: A `RespellPitch` restoring the prior spelling (captured), or a `RemoveAttachment` if no prior `UserChosen` attachment existed.

Reduction rule: Field overwrite by canonical reduction order. Concurrent respellings of the same live pitch resolve to the operation later in canonical order; the materialized spelling is that operation’s choice. The losing respelling is recorded in a `ConflictRecord` of kind `StructuralFieldCollision` if the two respellings represent different spellings (i.e., differ in nominal, accidental stack, or octave). Identical concurrent respellings reduce idempotently: the later operation produces `NoOp{reason: AlreadyApplied}` with no conflict recorded.

6.12.5 ChangeRegionTimeModel

Category: Region operations.

```
pub struct ChangeRegionTimeModelOp {
    pub region: RegionId,
    pub new_time_model: RegionTimeModel,
    pub position_remapping: PositionRemapping,
}

pub enum PositionRemapping {
    /// Preserve absolute time positions where possible. Events that
    /// cannot be remapped (e.g., metric events under a proportional
    /// model with no implied tempo) are converted by the supplied
    /// conversion function.
    PreserveTime { fallback: PositionConverter },

    /// Discard event positions and reinsert by user-provided
    /// position mapping.
    Reassign(HashMap<EventId, EventPosition>),
}
```

Invariant preconditions:

- The target region exists.
- The new time model is internally consistent.
- The position remapping covers every event in the region, or the remapping is `PreserveTime` with a defined fallback.

Effect: The region’s `time_model` is replaced. Every event in the region has its `position` field updated according to the remapping. The tempo map’s applicability is recomputed.

Postconditions:

- Every event in the region has a valid position in the new time model.
- The region’s invariants (events sorted, non-overlapping, within time extent) hold.

Inverse: `ChangeRegionTimeModel` restoring the prior time model, with a remapping captured from the pre-state.

Reduction rule: Structural migration. The operation is applied at its canonical position. Contained events whose coordinate kinds become incompatible with the new time model after mi-

gration cause the operation to reduce to `Conflicted` with a `ConflictKind::TimeModelMigrationFailure` record listing the incompatible events; the migration does not apply. Concurrent operations on the same region that author events in the prior time model and reduce after the migration may themselves reduce to `Conflicted` if their coordinate kinds disagree with the new model. Concurrent same-target migrations resolve by canonical order; the earlier operation applies and the later operation reduces to `Conflicted` with a structural-field-collision conflict (the two migrations are not generally commutable).

RATIONALE

Time-model migration is a structural rebase, not a field set. Treating it as LWW would silently destroy contained musical content whose coordinate kinds depend on the prior time model. Conflicted reductions force the user to address the incompatibility explicitly.

6.12.6 Transpose

Category: Pitch and tuning operations. This is a compound operation, defined as a sequence of primitives, listed here as an illustration of compounds.

User-facing parameters:

- Target scope (events, voice, staff, region, or selection).
- Interval (diatonic, chromatic, or compound).
- Spelling policy (preserve user spellings, respell to fit a target key, or apply a custom rule).
- Tuning policy (preserve, recompute under new tuning, or apply offsets).

Primitive decomposition: A transpose operation expands into:

1. For each target pitch, a `ReplaceEvent` updating the inner `Pitch` values while preserving every `IdentifiedPitch`'s `PitchId`. Transposition does not mint new pitch identifiers; it rewrites the contents of existing identified pitches.
2. For each pitch whose spelling is recomputed, a `RespellPitch` targeting the unchanged `PitchId` with source `Propagated`.
3. Where the transposition crosses a clef boundary, optional `ChangeStaffConfiguration` operations to insert clef changes.

The compound operation is recorded as a single transaction. Each primitive within the transaction has its own re-anchoring and reduction semantics. Identifier preservation ensures that attachments, analytical layers, and cross-cutting references targeting the transposed pitches continue to resolve correctly after the operation.

6.12.7 SetUserSystemBreak

Category: Layout-semantic operations.

```
pub struct SetUserSystemBreakOp {
    pub anchor: TimeAnchor,
```

```
pub region: RegionId,
pub present: bool,
}
```

Invariant preconditions:

- The region exists and is staff-based.
- The anchor resolves within the region.

Effect: The region's `user_system_breaks` list is updated to include or exclude the anchor. Layout caches that depend on system breaks are invalidated.

Postconditions:

- The region's user system breaks list reflects the change.
- Layout output, if produced subsequently, accounts for the change subject to the engraver's override rules (Chapter 7).

Inverse: `SetUserSystemBreak` with the opposite `present` value.

Reduction rule: `LwwAdvisory` on per-anchor break preference. Two concurrent operations setting the same break to `true` reduce idempotently. Concurrent set-true and set-false on the same anchor resolve by canonical reduction order; the later operation's preference is materialized. No conflict record is produced; the break preference is an advisory field eligible for LWW reduction (Section 6.9).

6.13 Operation Catalog Conformance

This chapter specifies the framework. The full catalog of operations is delivered as a separate conformance specification that extends this chapter.

REQUIREMENT

A conforming implementation **MUST** provide every operation enumerated in the conformance catalog. Each operation **MUST** satisfy the framework requirements stated in this chapter:

- Declared preconditions classified as invariant or advisory.
- Specified effect on the score graph.
- Specified postconditions, including which invariants are preserved.
- A defined inverse operation.
- A declared reduction rule under the canonical reduction model (Section 6.3).
- Specified re-anchoring behavior for any deletions or moves it triggers.

OPEN QUESTION

The conformance catalog is presently under development as a separate document. It is expected to comprise approximately 60–80 primitive operations and an open-ended set of compound operations. The catalog is normative once published; this specification is non-final until the catalog is delivered.

6.14 Forward References

- Operation envelope storage on disk (envelope block layout, causal-summary metadata, the operation index, snapshot canonical-base semantics) is specified in Chapter 8.
- Conflict-resolution algorithms (how a `ResolveConflict` operation transitions a conflict's state, and the full conflict-record contract) are specified in this chapter (Section 6.5) and in the Operation Catalog companion (Appendix A).
- The interaction between layout-semantic operations (system breaks, page breaks, engraving overrides) and the layout pipeline is specified in Chapter 7.
- The conformance catalog of all operations is a separate deliverable that extends this chapter.

Layout Intermediate Representation

This chapter specifies the layout intermediate representation (Layout IR): the data structures and pipeline that transform the score graph into a typeset visual layout. The IR sits between the score graph and two downstream consumers: the constraint solver (Chapter 9), which resolves spacing and positioning, and the renderer (specified outside this document), which produces final visual output (pixels, PDF, SVG).

The IR is a pipeline of distinct stages, each with its own type and each with a well-defined contract for the next. The pipeline is the spine of the engraving system; the engraving algorithms themselves (beam grouping, slur curvature, accidental ordering, casting-off) are the consumers and producers of stages and are specified outside this document as engraving-algorithm specifications layered atop this IR.

7.1 Design Principles

Pipeline of stages. The transformation from score graph to visual output is a pipeline of four IR stages: `LogicalLayoutIR` (structural projection without spatial information), `ConstrainedLayoutIR` (with constraint inputs but unresolved positions), `ResolvedLayoutIR` (with all positions resolved), and `RenderIR` (renderer-bound primitives, defined only at the interface level here). Each stage has its own type. Compilation between stages is unidirectional.

Shared provenance. Every IR object at every stage carries a reference back to the score graph object that originated it. This enables selection, editing back-references, error reporting, and incremental layout invalidation.

Engraving decisions are explicit. When the engraver makes a decision (stem direction, accidental ordering, beam consolidation), the decision is recorded in the IR. The decision can be inspected, overridden, and traced.

Overrides exist in both worlds. User-set engraving overrides are authoritative in the score graph and projected into the IR during the logical stage. The IR may also generate transient overrides for downstream stages.

Spring-based spacing. Time-to-space projection produces elastic spring parameters per time slot. The constraint solver resolves springs system-wide. This applies to both horizontal and vertical layout.

Staff-space units, f32 precision. Spatial coordinates within the IR are expressed in staff spaces using single-precision floating point. Conversion to absolute units (points, millimeters) occurs only at the render boundary.

Glyph metrics live elsewhere. The IR references glyphs by identifier and queries metrics from a font catalog. Metrics are not embedded in IR objects.

Incremental layout, fine-grained. The IR supports incremental re-engraving via per-object dependency tracking on score graph objects. Edits invalidate only the layout objects whose dependencies changed.

Uniform region containers. Staff-based, free-graphic, and hybrid regions produce structurally identical `LayoutRegion` containers with differing time projections and content kinds.

7.2 Spatial Primitives

7.2.1 Units

The IR uses two unit types:

```
/// Staff space: the fundamental unit of music engraving.
/// One staff space equals the distance between adjacent staff lines.
#[derive(Copy, Clone, PartialEq, PartialOrd, Debug)]
pub struct StaffSpace(pub f32);

/// Point: 1/72 inch. Used at the render boundary for absolute
/// dimensions (page sizes, font sizes, output coordinates).
#[derive(Copy, Clone, PartialEq, PartialOrd, Debug)]
pub struct Point(pub f32);

/// Scaling context for converting between staff spaces and points.
pub struct ScaleContext {
    /// Points per staff space. Depends on the staff size chosen for
    /// the score; typical values are 4--10 points per staff space.
    pub points_per_staff_space: f32,
}
```

REQUIREMENT

Spatial coordinates within the IR (in stages `LogicalLayoutIR`, `ConstrainedLayoutIR`, and `ResolvedLayoutIR`) **MUST** be expressed in staff spaces. Conversion to points occurs at the `RenderIR` boundary.

Single-precision floating point (f32) **MUST** be used for IR coordinates. Engraving precision requirements are bounded by printer resolution (typically 600–2400 DPI); single precision provides over 7 significant figures, comfortably exceeding any engraving precision requirement.

7.2.2 Geometric Types

```
#[derive(Copy, Clone, Debug)]
```

```

pub struct Point2D {
    pub x: StaffSpace,
    pub y: StaffSpace,
}

#[derive(Copy, Clone, Debug)]
pub struct Size2D {
    pub width: StaffSpace,
    pub height: StaffSpace,
}

#[derive(Copy, Clone, Debug)]
pub struct Rect {
    pub origin: Point2D,
    pub size: Size2D,
}

#[derive(Copy, Clone, Debug)]
pub struct BoundingBox {
    pub left: StaffSpace,
    pub right: StaffSpace,
    pub top: StaffSpace,
    pub bottom: StaffSpace,
}

#[derive(Copy, Clone, Debug)]
pub struct Transform2D {
    pub matrix: [[f32; 3]; 3],
}

```

7.3 Provenance

Every IR object carries a provenance record tracing it back to the score graph object that originated it. Provenance is uniform across all four stages; provenance records are shared structurally between stages so that an object passing through the pipeline retains a consistent identity.

```

pub struct Provenance {
    /// The primary source: the score graph object this IR object
    /// represents or derives from.
    pub source: TypedObjectId,

    /// For IR objects that derive from synthesizing engraving
    /// decisions rather than directly from score graph objects, the
    /// synthesis kind. Examples: an automatically inserted natural
    /// sign, a generated rest, a system break.
    pub synthesis: Option<SynthesisKind>,

    /// Additional source dependencies. An IR object may depend on
    /// multiple score graph objects (e.g., a beam depends on its
    /// member events). All dependencies are tracked for

```

```

    /// incremental layout.
    pub dependencies: Vec<TypedObjectId>,

    /// IR object's stable identifier across re-layouts where the
    /// underlying source is unchanged.
    pub stable_id: LayoutObjectId,
}

pub enum SynthesisKind {
    /// An accidental inserted to cancel a prior alteration.
    CancellationAccidental,

    /// A natural sign generated by key signature rules.
    KeySignatureNatural,

    /// A rest inserted to fill a voice gap.
    GeneratedRest,

    /// A system or page break selected by the engraver.
    EngravedBreak,

    /// A multimeasure rest combining consecutive empty measures.
    MultimeasureRest,

    /// A cautionary key or time signature at a system break.
    Cautionary,

    /// A custom synthesis declared by a layout extension.
    Registered(SynthesisRegistryId),
}

pub struct LayoutObjectId(pub u128);

```

REQUIREMENT

Every layout object at every IR stage **MUST** carry a non-empty `Provenance` record. Objects whose direct source does not correspond to a score graph object (engraver-synthesized objects) **MUST** declare a synthesis kind. The `dependencies` list **MUST** include every score graph object whose change should invalidate this layout object.

7.4 The Stage Pipeline

7.4.1 Pipeline Overview

```

ScoreGraph
|
|   Engraving pass:
|   - Project structure
|   - Make engraving decisions (stems, beams, accidentals)
|   - Apply user overrides

```

```

    v
LogicalLayoutIR
|
|   Spacing pass:
|   - Compute spring parameters per time slot
|   - Compute glyph bounding boxes
|   - Build collision constraints
    v
ConstrainedLayoutIR
|
|   Constraint solving (Chapter 9):
|   - Resolve horizontal positions
|   - Resolve vertical positions
|   - Resolve page and system breaks
    v
ResolvedLayoutIR
|
|   Render projection (out of scope):
|   - Convert to renderer primitives
|   - Apply page templates
|   - Generate draw calls
    v
RenderIR

```

7.4.2 Stage Contracts

REQUIREMENT

Each pipeline stage **MUST** accept its input stage as a pure function: identical inputs (plus identical configuration) **MUST** produce identical outputs. The stages are deterministic by contract.

Stage transitions **MUST** preserve provenance: every output object **MUST** carry a `Provenance` record either inherited from its input or newly synthesized with appropriate `synthesis kind`.

7.5 LogicalLayoutIR

`LogicalLayoutIR` is the structural projection of the score graph into layout objects, with all engraving decisions made but spatial positions unresolved. It is the output of the engraving pass and the input to the spacing pass.

7.5.1 Top-Level Structure

```

pub struct LogicalLayoutIR {
    /// Source score graph reference (typically a snapshot pointer
    /// or version identifier).
    pub source: ScoreVersion,

    /// Layout regions, one per score graph region.

```

```

pub regions: Vec<LayoutRegion>,

/// Engraving decisions made during this pass.
pub engraving_decisions: Vec<EngravingDecision>,

/// User overrides projected from the score graph.
pub overrides: Vec<EngravingOverride>,

/// Cross-region layout objects (slurs spanning regions, etc.).
pub cross_region: Vec<CrossRegionObject>,
}

```

7.5.2 Layout Regions

```

pub struct LayoutRegion {
    pub provenance: Provenance,

    /// Coordinate system local to this region. The region's
    /// transform maps local coordinates to canvas coordinates.
    pub coordinate_system: LocalCoordinateSystem,

    /// Time axis: how time positions map to horizontal positions.
    /// A tagged union over the three built-in axis kinds plus a
    /// registered variant for extension-defined axes. The enum
    /// form is canonical for serialization, hashing, and
    /// conformance comparison; implementations MAY use a trait
    /// object internally for dynamic dispatch.
    pub time_axis: TimeAxisModel,

    /// Vertical extent: the set of staff bands this region occupies.
    pub vertical_extent: VerticalExtent,

    /// Layout objects within this region.
    pub objects: Vec<LayoutObject>,
}

/// Canonical representation of a region's time axis. Variants
/// correspond to the three built-in region time models; the
/// Registered variant carries extension-defined axes by typed
/// registry id and serialized payload.
pub enum TimeAxisModel {
    Metric(MetricTimeAxis),
    Proportional(ProportionalTimeAxis),
    Aleatoric(AleatoricTimeAxis),
    Registered(TimeAxisRegistryId, SerializedRegisteredAxis),
}

```

7.5.3 The Time Axis Trait

For dynamic dispatch and ergonomic implementation, the `TimeAxisModel` enum's payload types share a trait:

```

pub trait TimeAxis: Send + Sync {
    /// The time model kind this axis implements.
    fn kind(&self) -> TimeAxisKind;

    /// Project a time position to a horizontal spring slot.
    /// Returns a slot identifier; spring parameters are queried
    /// separately.
    fn project(&self, time: TimePoint) -> SpringSlotId;

    /// Enumerate the spring slots in time order.
    fn slots(&self) -> &[SpringSlotId];

    /// For incremental layout: which slots are affected by a change
    /// to the given time range?
    fn affected_slots(&self, range: TimeRange) -> Vec<SpringSlotId>;
}

pub enum TimeAxisKind {
    Metric,
    Proportional,
    Aleatoric,
    Registered(TimeAxisRegistryId),
}

pub enum TimePoint {
    Musical(MusicalPosition),
    WallClock(WallClockTime),
}

pub enum TimeRange {
    Musical { start: MusicalPosition, end: MusicalPosition },
    WallClock { start: WallClockTime, end: WallClockTime },
}

```

The three built-in implementations:

```

pub struct MetricTimeAxis {
    /// Measure boundaries and their projected spring slots.
    pub measures: Vec<MeasureProjection>,

    /// Per-time-slot spring parameters, indexed by slot id.
    pub slots: Vec<SpringSlot>,
}

pub struct ProportionalTimeAxis {
    /// Wall-clock duration of the region.
    pub duration: WallClockDuration,

    /// Linear horizontal projection: space per second.
    pub space_per_second: StaffSpace,

    pub slots: Vec<SpringSlot>,
}

```

```

pub struct AleatoricTimeAxis {
    /// DAG of event ordering.
    pub ordering: EventOrderingDAG,

    /// Spring slots are organized by topological-order layers.
    pub slots: Vec<SpringSlot>,
}

```

7.5.4 Layout Objects

A `LayoutObject` is the unit of engraving. At the `LogicalLayoutIR` stage, layout objects are composite: a single `Note` object contains its notehead, stem, flag, accidental, articulations, and so on. At the `ConstrainedLayoutIR` stage these are flattened into individual glyph objects.

```

pub enum LayoutObject {
    /// Composite note object: notehead(s), stem, flag, accidental(s),
    /// articulations, ornaments, dot(s).
    Note(NoteLayout),

    /// Composite chord object: multiple noteheads sharing stem and
    /// articulations.
    Chord(ChordLayout),

    /// Rest with its glyph kind (whole, half, quarter, ...).
    Rest(RestLayout),

    /// Beam group: multiple notes/chords beamed together.
    BeamGroup(BeamGroupLayout),

    /// Tuplet visual: bracket and number.
    TupletDisplay(TupletDisplayLayout),

    /// Slur or phrase mark: control points and style.
    Slur(SlurLayout),

    /// Tie: same as slur but connecting same-pitch events.
    Tie(TieLayout),

    /// Spanner: hairpins, octave lines, pedal lines, etc.
    Spanner(SpannerLayout),

    /// Marker: rehearsal marks, section labels, tempo marks.
    Marker(MarkerLayout),

    /// Bar line: simple, double, final, repeat, etc.
    BarLine(BarLineLayout),

    /// Clef, key signature, time signature glyphs.
    Clef(ClefLayout),
    KeySignature(KeySignatureLayout),
    TimeSignatureDisplay(TimeSignatureDisplayLayout),
}

```

```

    /// Staff: the staff lines themselves.
    Staff(StaffLayout),

    /// Text: lyrics, dynamics text, expressive text.
    Text(TextLayout),

    /// Free graphic: path, shape, image, stroke.
    Graphic(GraphicLayout),

    /// Multimeasure rest: consolidated empty measures.
    MultimeasureRest(MultimeasureRestLayout),

    /// Cue: small-print rendering of source material.
    Cue(CueLayout),

    /// Trajectory: glissando, portamento line.
    Trajectory(TrajectoryLayout),

    /// Generic group: implementation-defined.
    Group(GroupLayout),
}

```

7.5.5 Note Layout: A Worked Example

To illustrate the composite-object pattern at the logical stage:

```

pub struct NoteLayout {
    pub provenance: Provenance,

    /// Reference to the score graph event.
    pub event: EventId,

    /// The notehead glyph and its staff position.
    pub notehead: NoteheadLayout,

    /// The stem, if any.
    pub stem: Option<StemLayout>,

    /// Flag, if any (single-note events with eighth-or-shorter
    /// duration that are not part of a beam group).
    pub flag: Option<FlagLayout>,

    /// Accidental glyphs attached to the note, in stacking order.
    pub accidentals: Vec<AccidentalLayout>,

    /// Augmentation dots.
    pub dots: Vec<DotLayout>,

    /// Articulations attached to the note.
    pub articulations: Vec<ArticulationLayout>,
}

```

```

    /// Ornament glyphs.
    pub ornaments: Vec<OrnamentLayout>,

    /// Ledger lines required to display the note's staff position.
    pub ledger_lines: Vec<LedgerLineLayout>,

    /// Engraving decisions made for this note.
    pub decisions: NoteDecisions,
}

pub struct NoteDecisions {
    pub stem_direction: StemDirection,
    pub accidental_visible: bool,
    pub accidental_parenthesized: bool,
    pub notehead_shape: NoteheadShape,
    /// Voice membership influences default stem direction;
    /// overrides may change it.
    pub stem_direction_source: DecisionSource,
}

pub enum DecisionSource {
    /// Decision derived from automatic engraving rules.
    Automatic,

    /// Decision derived from a user override in the score graph.
    UserOverride(EngravingOverrideId),

    /// Decision derived from an IR-stage override.
    IrOverride,
}

```

7.5.6 Engraving Decisions

The pipeline records engraving decisions explicitly so they can be inspected, overridden, and replayed.

```

pub struct EngravingDecision {
    pub id: EngravingDecisionId,
    pub target: LayoutObjectId,
    pub kind: EngravingDecisionKind,
    pub source: DecisionSource,
}

pub enum EngravingDecisionKind {
    StemDirection(StemDirection),
    BeamGrouping(BeamGroupingChoice),
    AccidentalOrdering(Vec<AccidentalId>),
    SlurDirection(SlurDirection),
    LedgerLineCount(u8),
    EnharmonicSpelling(PitchSpelling),
    SystemBreak,
    PageBreak,
}

```

```
NoteheadShape(NoteheadShape),
RestPosition(StaffPosition),
// ... extensible
}
```

7.6 Engraving Overrides

Engraving overrides are user assertions that the engraver’s default decision should be replaced. Overrides are authoritative in the score graph and projected into the IR during the logical stage.

7.6.1 Override Definition

```
pub struct EngravingOverride {
  pub id: EngravingOverrideId,

  /// What this override targets. May reference a score graph object
  /// (typical) or an IR-synthesized object (transient overrides).
  pub target: OverrideTarget,

  /// The override itself.
  pub kind: OverrideKind,

  /// Binding strength.
  pub priority: OverridePriority,

  /// Provenance: who set the override and when.
  pub origin: OverrideOrigin,
}

pub enum OverrideTarget {
  /// Targets a score graph object. Authoritative; survives reload.
  ScoreGraph(TypedObjectId),

  /// Targets an IR-synthesized object. Transient; cleared on
  /// re-engraving unless explicitly re-applied.
  IrSynthesized(LayoutObjectId),
}

pub enum OverrideKind {
  StemDirection(StemDirection),
  SlurDirection(SlurDirection),
  SlurCurvature(CurvatureOverride),
  AccidentalParenthesized(bool),
  AccidentalVisible(bool),
  NoteheadShape(NoteheadShape),
  BeamGeometry(BeamGeometryOverride),
  SystemBreak,
  PageBreak,
  HiddenObject,
  CustomPosition(Point2D),
}
```

```

    EnharmonicSpelling(PitchSpelling),
    LedgerLineSuppression,
    DotPlacement(DotPlacement),
    // ... extensible
}

pub enum OverridePriority {
    /// The engraver MUST honor this override. Failure produces an
    /// error rather than a silent override-of-the-override.
    Hard,

    /// The engraver SHOULD honor this override. The engraver may
    /// override it only under documented exceptional conditions
    /// (system overflow, unresolvable collision); when it does, the
    /// override is recorded in the resulting decisions with source
    /// IrOverride.
    Soft,
}

pub enum OverrideOrigin {
    User { author: AuthorId, timestamp: Timestamp },
    Import { format: ForeignFormatId },
    Plugin { plugin: PluginId },
    Internal,
}

```

7.6.2 Override Resolution

REQUIREMENT

Overrides **MUST** be applied during the engraving pass producing `LogicalLayoutIR`. Hard overrides **MUST** be honored or the pass **MUST** report an error. Soft overrides **MUST** be honored except where doing so would produce an invalid layout; when a soft override is not honored, the resulting decision **MUST** record the override and the reason for its non-application.

REQUIREMENT

Overrides whose targets reference objects in the score graph (the `ScoreGraph` variant) **MUST** be preserved across re-engraving. Overrides targeting IR-synthesized objects (the `IrSynthesized` variant) are transient; they are cleared on re-engraving unless explicitly re-applied. Implementations **MUST** not silently promote IR-synthesized overrides to score-graph overrides without explicit user action.

7.7 ConstrainedLayoutIR

`ConstrainedLayoutIR` is the output of the spacing pass: the logical IR with composite objects flattened to individual glyphs, each glyph carrying its bounding box, anchor, and constraint

inputs to the solver.

7.7.1 Glyph-Level Objects

```
pub struct GlyphObject {
    pub provenance: Provenance,

    /// The glyph to render. Reference into a font catalog; metrics
    /// are queried from the catalog rather than embedded here.
    pub glyph: GlyphReference,

    /// The spring slot this glyph belongs to (horizontal time slot).
    pub horizontal_slot: SpringSlotId,

    /// The vertical band this glyph belongs to.
    pub vertical_band: VerticalBandId,

    /// Bounding box of the glyph relative to its anchor, in staff
    /// spaces. Looked up from the font catalog at IR construction
    /// time and cached here.
    pub bounding_box: BoundingBox,

    /// Anchor point: where the glyph's reference position sits
    /// within its bounding box.
    pub anchor: Point2D,

    /// Layer ordering: higher draws on top.
    pub layer: i32,

    /// Visual style applied to the glyph.
    pub style: GlyphStyle,
}
```

7.7.2 Spring Slots

A spring slot is a time slot's spacing parameters: the constraint solver's input for resolving horizontal positions.

```
pub struct SpringSlot {
    pub id: SpringSlotId,
    pub time: TimePoint,

    /// Minimum width: the smallest the slot may compress to. The
    /// constraint solver MUST NOT compress below this.
    pub min_width: StaffSpace,

    /// Preferred width: the natural width under unconstrained
    /// spacing. Derived from Gourlay-style proportional rules.
    pub preferred_width: StaffSpace,

    /// Maximum width, if bounded. None means unbounded above.
    pub max_width: Option<StaffSpace>,
}
```

```

    /// Stretch factor: how readily this slot stretches when more
    /// width is available than preferred. Higher = more elastic.
    pub stretch_factor: f32,

    /// Compress factor: how readily this slot compresses when less
    /// width is available than preferred. Higher = more compressible.
    pub compress_factor: f32,

    /// Glyphs belonging to this slot, ordered by stacking
    /// considerations (accidentals to left of notehead, etc.).
    pub members: Vec<GlyphObjectId>,
}

```

7.7.3 Vertical Bands

Vertical layout uses the same spring model. A *vertical band* is a horizontal slice of the canvas (typically a staff or an inter-staff gap) with its own spring parameters.

```

pub struct VerticalBand {
    pub id: VerticalBandId,
    pub kind: VerticalBandKind,

    pub min_height: StaffSpace,
    pub preferred_height: StaffSpace,
    pub max_height: Option<StaffSpace>,
    pub stretch_factor: f32,
    pub compress_factor: f32,

    /// Glyphs belonging to this band.
    pub members: Vec<GlyphObjectId>,
}

pub enum VerticalBandKind {
    Staff(StaffId),
    InterStaffGap,
    InterSystemGap,
    MarginBand,
}

```

7.7.4 Constraints

The IR carries explicit constraint inputs for the solver:

```

pub struct ConstrainedLayoutIR {
    pub source: ScoreVersion,

    pub regions: Vec<ConstrainedLayoutRegion>,

    /// Horizontal spring slots across all regions.
    pub horizontal_slots: Vec<SpringSlot>,
}

```

```

/// Vertical bands.
pub vertical_bands: Vec<VerticalBand>,

/// Glyph-level layout objects.
pub glyphs: Vec<GlyphObject>,

/// Additional constraints not captured by spring parameters.
pub constraints: Vec<LayoutConstraint>,

/// Engraving decisions, carried forward from the logical stage.
pub engraving_decisions: Vec<EngravingDecision>,
}

pub enum LayoutConstraint {
/// Two glyphs must not overlap in 2D space.
NoCollision { a: GlyphObjectId, b: GlyphObjectId },

/// A glyph must align with another along an axis.
Align {
    a: GlyphObjectId,
    b: GlyphObjectId,
    axis: Axis,
},

/// A glyph must be positioned within a region of space.
PositionWithin {
    glyph: GlyphObjectId,
    region: Rect,
},

/// A range of slots must end at a system boundary.
SystemBreakAt { slot: SpringSlotId, kind: BreakKind },

/// A range of slots must end at a page boundary.
PageBreakAt { slot: SpringSlotId, kind: BreakKind },

/// Custom constraint declared by a layout extension.
Registered(ConstraintRegistryId, ConstraintParameters),
}

pub enum BreakKind {
    Hard,
    Soft,
}

```

7.8 ResolvedLayoutIR

ResolvedLayoutIR is the output of the constraint solver: every glyph has a definitive position. This is the IR consumed by the renderer.

```

pub struct ResolvedLayoutIR {
    pub source: ScoreVersion,

```

```

    /// Pages of the resolved score.
    pub pages: Vec<ResolvedPage>,

    /// Glyph-level objects with resolved positions.
    pub glyphs: Vec<ResolvedGlyph>,

    /// Engraving decisions, including any made by the constraint
    /// solver (e.g., soft system breaks placed by casting-off).
    pub engraving_decisions: Vec<EngravingDecision>,
}

pub struct ResolvedPage {
    pub provenance: Provenance,
    pub number: u32,
    pub size: Size2D,
    pub margins: Margins,
    pub systems: Vec<ResolvedSystem>,
    pub free_objects: Vec<GlyphObjectId>,
}

pub struct ResolvedSystem {
    pub provenance: Provenance,
    pub bounding_box: Rect,
    pub staves: Vec<ResolvedStaff>,
    pub measures: Vec<ResolvedMeasure>,
}

pub struct ResolvedGlyph {
    pub provenance: Provenance,
    pub glyph: GlyphReference,
    pub position: Point2D,
    pub transform: Option<Transform2D>,
    pub bounding_box: BoundingBox,
    pub style: GlyphStyle,
    pub layer: i32,
}

```

7.9 RenderIR (Interface Only)

The `RenderIR` stage is the renderer's input. Its full specification is delivered separately; this section defines only the interface contract.

```

pub trait RenderIRProducer {
    /// Convert resolved IR to renderer-bound primitives.
    fn produce(
        &self,
        resolved: &ResolvedLayoutIR,
        scale: ScaleContext,
        config: RenderConfiguration,
    ) -> RenderIR;
}

```

```
pub struct RenderConfiguration {
    /// Target output: PDF, SVG, on-screen, print.
    pub target: RenderTarget,

    /// Color space and color management options.
    pub color: ColorConfiguration,

    /// Anti-aliasing and rasterization options.
    pub rasterization: RasterizationConfiguration,
}
```

REQUIREMENT

Implementations producing RenderIR **MUST** preserve provenance from ResolvedLayoutIR: every renderer primitive **MUST** be traceable to its originating ResolvedGlyph (and therefore to its score graph source). This is the basis of hit-testing, selection, and back-reference navigation in the UI.

The full RenderIR type and its production rules are specified in the renderer specification, outside the scope of this document.

7.10 Incremental Layout and Caching

The pipeline supports incremental layout via per-object dependency tracking. An edit to a score graph object invalidates only the IR objects whose provenance lists it as a dependency.

7.10.1 The Layout Cache

```
pub struct LayoutCache {
    /// Reverse index: maps score graph object ids to the layout
    /// objects depending on them.
    pub dependencies: DependencyIndex,

    /// Cached IR stages, partitioned for granular invalidation.
    pub logical: HashMap<RegionId, LogicalRegionCache>,
    pub constrained: HashMap<RegionId, ConstrainedRegionCache>,
    pub resolved: HashMap<SystemId, ResolvedSystemCache>,

    /// Cached layout-stage outputs at finer granularity for
    /// performance-critical paths.
    pub fine_cache: FineLayoutCache,
}

pub struct DependencyIndex {
    /// For each score graph object, the layout objects depending
    /// on it. Reverse-indexed to make invalidation O(k) where k is
    /// the number of dependents.
    pub forward: HashMap<TypedObjectId, Vec<LayoutObjectId>>,

    /// For each layout object, its dependencies. Used to verify and
```

```

/// rebuild the reverse index on cache writes.
pub reverse: HashMap<LayoutObjectId, Vec<TypedObjectId>>,
}

```

7.10.2 Invalidation Rules

REQUIREMENT

When a semantic operation (Chapter 6) is applied to the score graph, the following invalidation **MUST** occur:

1. Every layout object whose `Provenance` lists a mutated score graph object in its `source` or `dependencies` is invalidated.
2. Invalidation propagates: layout objects that depend on invalidated layout objects (via cross-region references, spanning structures, etc.) are also invalidated.
3. Invalidated objects are cleared from the cache at their respective stages.
4. Invalidated systems trigger casting-off re-evaluation for themselves and any downstream systems whose horizontal extent may shift.

Re-engraving runs from the first invalidated stage forward, using cached values for non-invalidated objects.

7.10.3 Frame-Budget Requirements

REQUIREMENT

The incremental layout system **MUST** support the frame-budget requirements established in Chapter 10: an edit affecting a single system on a 100-page orchestral score **MUST** re-layout in under 16.7 ms on the reference hardware profile.

Achieving this requires (a) fine-grained dependency tracking as specified above; (b) cached glyph metrics in the font catalog; (c) parallel re-engraving where the affected systems are independent; (d) the constraint solver supporting partial re-resolution of affected slots only (see Chapter 9).

7.11 Region Uniformity

Staff-based, free-graphic, and hybrid regions produce structurally identical `LayoutRegion` containers. The differences are expressed in:

- The `time_axis`: metric, proportional, or aleatoric.
- The `objects mix`: staff-based regions contain primarily notation objects; free-graphic regions contain primarily `Graphic` objects; hybrid regions contain both.

REQUIREMENT

All three region kinds **MUST** use the same `LayoutRegion` container type. The constraint solver **MUST** treat regions uniformly: it consumes spring slots and constraints without branching on region kind.

RATIONALE

Uniform containers permit the constraint solver to be agnostic to region kind, simplifying its implementation. The differences between region kinds are encoded in the `TimeAxis` implementation and the object mix, not in container structure. This is the same insight that makes the file format and semantic operations layers tractable: structural uniformity at the containers, polymorphism in the contents.

7.12 Glyph Catalog Interface

The IR references glyphs by identifier; glyph metrics are queried from a font catalog at IR construction. This section defines the interface; the catalog implementation is outside the core specification.

```
pub trait GlyphCatalog: Send + Sync {
    /// Resolve a glyph reference to its metrics.
    fn metrics(&self, glyph: &GlyphReference) -> Option<GlyphMetrics>;

    /// Resolve a glyph reference to its rendering data.
    fn render_data(&self, glyph: &GlyphReference) -> Option<GlyphRenderData>;

    /// SMuFL version supported.
    fn smufl_version(&self) -> SmuflVersion;
}

pub struct GlyphMetrics {
    pub bounding_box: BoundingBox,
    pub advance_width: StaffSpace,
    pub anchors: HashMap<AnchorName, Point2D>,
}

pub struct GlyphRenderData {
    /// Outline data: path commands suitable for vector rendering.
    pub outline: Vec<PathCommand>,

    /// Optional pre-rendered bitmap for raster targets.
    pub bitmap: Option<GlyphBitmap>,
}
```

REQUIREMENT

Implementations **MUST** use a glyph catalog for metric and render-data lookups during IR construction. Metrics **MUST NOT** be duplicated in the IR; the IR carries glyph identifiers and queries the catalog. Implementations **MAY** cache metrics within IR objects as a performance optimization provided cached values are invalidated when the underlying catalog changes.

7.12.1 Glyph Catalog Identity for Layout Conformance

Layout output depends on glyph metrics. For reproducible layout, the exact glyph catalog used **MUST** be identifiable.

```

/// A reproducibility-quality identifier for the glyph catalog
/// used to produce a layout. Required for any layout conformance
/// claim that depends on byte-equal output across runs.
pub struct GlyphCatalogIdentity {
    /// SMuFL version targeted.
    pub smufl_version: SmuflVersion,

    /// Identifier of the specific font in use (e.g., Bravura,
    /// Petaluma, Leland).
    pub font_id: FontId,

    /// Optional semantic version of the font, if the font
    /// publisher uses versioned releases.
    pub font_version: Option<SemVer>,

    /// Content hash of the metric data made available to the
    /// layout solve. The hash is over a canonical serialization
    /// of every glyph's metrics (bounding box, advance width,
    /// named anchors) for every glyph referenced by the
    /// ConstrainedLayoutIR delivered to the solver, including
    /// glyphs that are candidates but do not appear in the final
    /// ResolvedLayoutIR. Computed by BLAKE3 with the domain
    /// tag "MUSCFNTM".
    pub metrics_hash: ContentHash,
}

```

REQUIREMENT

Any layout conformance claim subject to byte-equal output (within-implementation determinism per Chapter 9) **MUST** declare the `GlyphCatalogIdentity` under which it was produced. Two layouts produced with different `metrics_hash` values are layouts under different inputs; byte equality is not expected.

Implementations that cannot identify their glyph catalog at this granularity **MUST** mark their layout output as *implementation-specific* or *non-canonical*; conformance suites **MUST NOT** accept such output as byte-equivalent to a reference solver's output.

The `metrics_hash` is computed over the glyphs that are *inputs* to the layout solve (those

referenced by the `ConstrainedLayoutIR`), not only over glyphs in the final layout. Two solver invocations with different available candidate-glyph metrics are operating on different inputs even if they happen to converge on the same final glyph set; this rule makes that difference visible in the `metrics_hash`.

The hash is computed only over glyphs the layout solve actually consulted, not over the entire font catalog, so that font additions (new glyphs unused by the solve) do not invalidate prior layouts.

7.13 Forward References

- The constraint solver consuming `ConstrainedLayoutIR` and producing `ResolvedLayoutIR` is specified in Chapter 9.
- The serialization of layout cache data for cross-session persistence is specified in Chapter 8.
- The complete `RenderIR` specification, the engraving algorithm specifications (beam grouping, slur curvature, casting-off, accidental ordering), the font catalog implementation, and the renderer are delivered as separate specifications layered on top of this core document.

File Format

This chapter specifies the on-disk file format. Chapter 6 established that the canonical document state is the deterministic reduction of an operation-envelope set. This chapter specifies how that operation set, along with acceleration structures, profile declarations, and extension declarations, is laid out on disk in a way that is crash-recoverable, atomically updatable, and forward-compatible.

The byte-level encoding details (record layouts, varint conventions, exact field bit widths) are deferred to a companion document, the Binary Format specification. This chapter specifies *what* is stored, how the parts relate, and how the format behaves under concurrent edits, crashes, and forward-compatibility scenarios; the Binary Format specifies *how the bits are laid out*.

The file format's name is `musc` and its file extension is `.musc`.

8.1 Design Principles

Canonical state is the operation set. The canonical document is the operation-envelope set (Chapter 6). After pruning, the canonical document is a retained base snapshot plus the operation envelopes after that snapshot's causal frontier. Everything else stored in the bundle (snapshots, layout caches, indexes, rendered artifacts) is an acceleration structure and **MUST** be discardable.

Fixed prelude, atomic superblock commit. The file begins with a fixed-size header followed by two fixed-size superblock slots. The superblocks are the only mutable on-disk objects in the bundle. Commits append new chunks, write a new manifest chunk, then flip the active superblock by writing to the currently-inactive slot. This makes commits atomic with respect to crashes.

Content-addressed immutable chunks. Every other on-disk object is content-addressed by BLAKE3 of its uncompressed payload (with domain-separated preimage). Chunks are immutable once written. Duplicate content shares storage automatically.

Manifest as a table of roots. The manifest does not carry user-facing data. It carries the operation-set roots, the retained snapshot references, blob references, profile declarations, extension declarations, and the text-projection root. User-facing metadata (title, composer, etc.) lives in operation envelopes or in materialized state, not in the physical header.

Crash recovery is by superblock selection. Recovery is simply: read both superblocks, vali-

date, and select the highest valid generation. Anything not reachable from the selected superblock is unreachable garbage and can be reclaimed by a later conservative GC pass.

Forward compatibility through opaque preservation and edit barriers. Unknown extension chunks are preserved across reads and writes. Unknown required extensions either prevent open or force read-only mode. Unknown optional extensions are preserved unless an edit crosses a declared edit barrier.

Streaming reads. The bundle is designed so that opening a large score touches the prelude, the active superblock, the manifest, and a small set of bootstrap chunks (the most recent canonical-base snapshot plus the envelope index). Per-region content loads on demand.

Deterministic text projection of semantic content. The format admits a deterministic projection to a canonical s-expression form for diff tooling and version control. The text projection preserves canonical document semantics (operation envelopes, profile declarations, extension declarations, reduced state). It does *not* preserve chunk offsets, compression choices, cache chunks, garbage regions, or superblock generation numbers.

No encryption, no DRM. The format **MUST NOT** include encryption, access control, or rights-management features. Privacy at rest is the responsibility of the filesystem and operating system. The format is open and offers no mechanism by which user content can be locked away from its author or from other implementations.

Non-Goal

Digital rights management (DRM), content encryption, hardware-binding, license enforcement, and any mechanism that could lock user content to a particular vendor or platform are explicitly out of scope. The format is designed for openness, durability, and user ownership of data. Implementations **MUST NOT** add proprietary extensions intended to restrict access to or use of user content.

8.2 Durable Writes

This chapter refers throughout to “durable flushes” of file contents. The definition is platform-abstracted.

Durable flush. The platform operation that makes previously-written bytes durable according to the host filesystem contract: `fsync` on POSIX-like systems, `FlushFileBuffers` or `NtFlushBuffersFileEx` on Windows, equivalent calls on other platforms. A successful durable flush guarantees that bytes written before the call are on the persistent storage medium when the call returns.

Requirement

Implementations **MUST** use the platform’s durable flush primitive at the points required by the atomic write protocol (Section 8.10). Implementations **MAYNOT** rely on filesystem ordering guarantees or write-back caching alone; durability **MUST** be requested explicitly at the commit point.

8.3 The Bundle Layout

A `.musc` bundle is a single file whose first 576 bytes are a fixed prelude: a 64-byte header followed by two 256-byte superblock slots. The remainder of the file holds variable-length content: chunk payloads, the manifest chunk, blob payloads, the text projection (if present), and unreachable garbage from prior commits that has not yet been collected.

```
pub struct BundleLayout {
    /// Fixed 64-byte header at offset 0.
    pub header: FixedHeader,

    /// Two fixed-size superblock slots at offsets 64 and 320.
    pub superblock_a: Superblock, // at offset 64
    pub superblock_b: Superblock, // at offset 320

    /// Variable-length content begins at offset 576.
    /// All chunks, the manifest, and blobs live here.
    pub body: ContentRegion,
}
```

8.3.1 The Fixed Header

The header is exactly 64 bytes at file offset zero. It identifies the format and locates the superblock slots. The header never changes after the file is created.

```
pub struct FixedHeader {
    /// Magic bytes: ASCII "MUSCBND\0" (8 bytes).
    pub magic: [u8; 8],

    /// Major format version. Major version changes are
    /// non-backward-compatible.
    pub format_major: u16,

    /// Minor format version. Minor version changes are
    /// backward-compatible.
    pub format_minor: u16,

    /// Length of this header in bytes (currently always 64).
    /// Reserved for future expansion via header_length growth.
    pub header_length: u32,

    /// Offset of superblock slot A. Currently always 64.
    pub superblock_a_offset: u64,

    /// Offset of superblock slot B. Currently always 320.
    pub superblock_b_offset: u64,

    /// 128-bit file UUID, set at creation of the physical bundle.
    /// Changes on Save As (the physical bundle is a new file).
    /// Distinct from document_id, which identifies the logical work.
    pub file_uuid: FileUuid,
```

```

    /// Reserved bytes at offsets 48..60. MUST be zero on write;
    /// MUST be ignored on read. Future minor versions MAY use
    /// these bytes for additional fields whose absence is
    /// interpreted as zero.
    pub reserved: [u8; 12],

    /// CRC-32C of bytes 0..60 of this header (i.e., all preceding
    /// fields plus the reserved bytes). At fixed offset 60..64.
    pub header_crc: u32,
}

```

REQUIREMENT

The header **MUST** be exactly 64 bytes at file offset zero. Its contents **MUST NOT** change after file creation, except for the case where a major-version upgrade rewrites the entire file. Readers **MUST** verify the magic bytes and the header CRC before consulting any other part of the file.

Reserved bytes **MUST** be zero when written by current implementations. Readers **MUST** ignore the values of reserved bytes (future versions may use them).

The `file_uuid` is fixed for the lifetime of a physical bundle and persists across commits to that bundle. A filesystem-level byte copy preserves the `file_uuid` (the copy is the same physical bundle duplicated). Save As, export-as-new-bundle, and derivative-work creation **MUST** assign a fresh `file_uuid`; the resulting bundle is a new physical bundle, distinct from its source.

The `file_uuid` identifies the physical bundle, not the logical work. Two bundles with the same `file_uuid` are byte-level copies of the same physical bundle; two bundles with the same `document_id` (defined in the manifest; see Section 8.4.2) are versions of the same logical work, which may or may not share their `file_uuid`.

8.3.2 The Superblock Slots

Each superblock slot is exactly 256 bytes. The bundle has two slots (A at offset 64, B at offset 320). One is active; the other is the write target for the next commit. Atomic commit is the act of writing the inactive slot and durably flushing it.

```

pub struct Superblock {
    /// Magic bytes: ASCII "MUSCSUPR" (8 bytes).
    pub magic: [u8; 8],

    /// Generation counter. The active superblock is the one with
    /// the highest valid generation.
    pub generation: u64,

    /// Offset of the manifest chunk for this generation.
    pub manifest_offset: u64,

    /// Length of the manifest chunk in bytes.
    pub manifest_length: u64,
}

```

```

    /// BLAKE3 content hash of the manifest chunk's uncompressed
    /// payload, with domain-separated preimage
    /// (Section~\ref{sec:format:hashing}).
    pub manifest_hash: ContentHash,

    /// Schema version of the manifest at this generation.
    pub manifest_schema_version: SchemaVersion,

    /// Reduction algorithm version that produced any
    /// canonical-base snapshots in this manifest.
    pub reduction_algorithm_version: ReductionAlgorithmVersion,

    /// Profile under which this superblock is valid.
    pub profile_id: ProfileId,

    /// Commit state. Distinguishes a fully-committed superblock
    /// from one that was written but whose commit had not yet
    /// completed when the writer crashed (rare; the protocol does
    /// not normally leave such a state visible).
    pub commit_state: CommitState,

    /// Wall-clock timestamp at commit, in canonical units.
    /// Advisory only; superblock selection is by generation, not
    /// by timestamp.
    pub commit_timestamp: WallClockTime,

    /// CRC-32C of the bytes of this superblock up to (but not
    /// including) the CRC field. Lets readers reject torn writes.
    pub superblock_crc: u32,

    /// Reserved padding to 256 bytes. MUST be zero on write;
    /// MUST be ignored on read.
    pub reserved: [u8; /* computed */ 256 - HEADER_FIELDS_SIZE],
}

pub enum CommitState {
    /// Fully committed. The manifest at manifest_offset is the
    /// canonical state for this generation.
    Committed,

    /// Reserved for future use; writers MUST NOT produce
    /// non-Committed values in this version of the format.
    Reserved(u32),
}

```

8.3.3 Superblock Selection

REQUIREMENT

On open, readers **MUST** perform the following selection:

1. Read both superblock slots.
2. For each slot, validate: magic bytes, CRC, that the referenced manifest chunk's actual BLAKE3 hash matches the declared `manifest_hash`, and that `commit_state` is `Committed`. A slot failing any of these checks is *not valid* for ordinary selection.
3. A slot whose `commit_state` is not `Committed` is invalid for ordinary selection but **MAY** be inspected in diagnostic recovery mode. Writers in this format version **MUST NOT** produce such states under normal commit; observing one indicates either a crashed writer (recovery falls back to the other slot per the atomic-write protocol) or a non-conforming writer.
4. If both slots are valid and their generations differ by more than one, the reader **MUST** report an integrity anomaly (the file may have been tampered with, accidentally merged, or written by a non-conforming implementation). The reader **MAY** continue in read-only recovery mode using the highest-generation valid slot; it **MUST NOT** silently treat the file as normal.
5. If both slots are valid and their generations differ by at most one, the slot with the higher generation is active.
6. If exactly one slot is valid, that slot is active.
7. If neither slot is valid, the file is corrupt; readers **MUST** surface this as a hard error and **MUST NOT** synthesize state.

8.4 The Manifest

The manifest is a table of roots and declarations. It is itself a chunk (content-addressed, immutable). Each commit writes a new manifest chunk and points the inactive superblock slot at it.

```
pub struct Manifest {
    /// Identifier of the logical work. Stable across Save As
    /// operations and across copies that are intended to remain
    /// versions of the same work. May persist across forks if
    /// the user explicitly preserves it.
    pub document_id: DocumentId,

    /// Optional identifier of a shared ancestor document. Used by
    /// version-control workflows and fork-tracking to express
    /// genealogy. Absent for documents with no declared ancestry.
    pub lineage_id: Option<LineageId>,

    /// Identifier of this manifest. Each commit produces a new
    /// ManifestId; old manifests are retained in the chunk store
    /// until garbage collection removes them subject to the
    /// retention policy.
```

```

pub manifest_id: ManifestId,

/// Generation counter matching the superbloc that references
/// this manifest.
pub generation: u64,

/// Operation-envelope blocks defining the canonical document.
/// Block ordering is irrelevant for canonical state (envelopes
/// are a set), but writers SHOULD order blocks by min_stamp
/// for predictable scanning.
pub operation_roots: Vec<ChunkRef>,

/// Optional operation index chunk: maps OperationId to
/// (block, offset) for fast random access. If absent,
/// implementations rebuild on demand by scanning blocks.
pub operation_index_root: Option<ChunkRef>,

/// The active canonical-base snapshot, if pruning has occurred.
/// At most one canonical base is active at a time. When
/// present, the canonical document is the base snapshot's
/// materialized state plus the deterministic reduction of
/// envelopes whose OperationIds are not covered by the base's
/// causal frontier.
pub canonical_base: Option<SnapshotRef>,

/// Acceleration snapshots: caches that materialize state at
/// various causal frontiers for fast incremental loading.
/// These MAY be discarded freely without affecting canonical
/// state.
pub acceleration_snapshots: Vec<SnapshotRef>,

/// Blob references for large opaque content (audio, images,
/// custom fonts).
pub blob_roots: Vec<BlobRef>,

/// Profile declarations: which conformance profiles this
/// manifest claims to satisfy.
pub profile_declarations: Vec<ProfileDeclaration>,

/// Extension declarations: which extensions contributed to
/// this manifest, with their required/optional flag and
/// edit barriers.
pub extension_declarations: Vec<ExtensionDeclaration>,

/// Text projection root chunk, if a text projection is
/// maintained in this bundle. Non-canonical accelerator.
pub text_projection_root: Option<ChunkRef>,

/// Optional integrity index root chunk: cross-references chunk
/// hashes for faster integrity verification of large bundles.
/// Non-canonical accelerator; absence is permitted.
pub integrity_root: Option<ChunkRef>,
}

```

8.4.1 Canonical and Non-Canonical Manifest Roots

The manifest’s fields partition cleanly into two groups by their relation to canonical state.

Canonical roots (define the document):

- `operation_roots`: operation-envelope blocks containing the canonical operation set.
- `canonical_base`: the active canonical base snapshot, if pruning has occurred.
- `blob_roots`: blobs referenced by canonical operations or by canonical reduced state are themselves canonical; blobs referenced only by acceleration structures are non-canonical.
- `document_id`, `lineage_id`, `profile_declarations`, `extension_declarations`: structural metadata governing how the canonical document is interpreted.

Non-canonical reachable roots (accelerators only):

- `operation_index_root`: rebuildable from operation envelope blocks.
- `acceleration_snapshots`: caches at causal frontiers other than the canonical base.
- `text_projection_root`: derivable from canonical state.
- `integrity_root`: rebuildable from chunk hashes.

REQUIREMENT

The manifest **MUST** list every chunk that contributes to the current canonical state, transitively through its canonical roots (operation roots, canonical base, canonical blobs and extension chunks). It **MAY** additionally list non-canonical reachable roots (acceleration snapshots, operation index, text projection, integrity index, layout caches).

Chunks not reachable from any manifest root, canonical or non-canonical, are unreachable; readers **MUST NOT** rely on them and conservative garbage collection **MAY** reclaim them (Section 8.11).

Non-canonical reachable chunks **MAY** be rebuilt or replaced without altering canonical state. If a non-canonical chunk’s hash fails verification, it **MUST** be discarded and (if needed) rebuilt; failed verification of a non-canonical chunk is *not* a corruption of the bundle.

Failed verification of a canonical chunk (operation envelope block, canonical base snapshot, canonical blob, manifest) *is* corruption and **MUST** be surfaced as a hard error.

User-facing metadata (title, composer, lyricist, copyright, and similar fields) is *not* part of the manifest. Such metadata lives in the operation set and the materialized state. The manifest carries only the structural roots and declarations that the format needs to locate and validate content.

8.4.2 File, Document, and Lineage Identity

The format distinguishes three identifiers, each serving a different identity question:

file_uuid. Physical bundle identity. Set at file creation; persists for the lifetime of the physical file. **Changes on Save As:** a copy created by Save As is a new physical bundle and **MUST** be assigned a fresh `file_uuid`. Used by tools that need to identify “this exact file on disk” (caches, backup software, file fingerprinting).

document_id. Logical work identity. Stable across Save As copies that are intended to remain the same work (the canonical case: “I’m saving my piece to a backup folder, and it’s still the same piece”). **Persists across forks at the user’s discretion:** a fork that represents a derivative work **SHOULD** have a new `document_id`, while a fork that is a working copy of the original **SHOULD** preserve it. The format does not enforce a policy; the editor surfaces the choice.

lineage_id. Optional shared ancestor identity. Records that two documents share a common ancestor for version-control or genealogy purposes. Multiple documents **MAY** share a `lineage_id`; the field is set when a fork explicitly preserves ancestry information.

REQUIREMENT

Implementations **MUST** assign a fresh `file_uuid` on any Save As operation. They **MUST NOT** carry the source file’s UUID into the new file: doing so would conflate physical and logical identity and break tools that rely on `file_uuid` for content fingerprinting.

The choice of whether to preserve `document_id` across a Save As is exposed to the user: the editor **SHOULD** default to preserving `document_id` (the common case is “save a copy of the same work to a new file”) and **SHOULD** offer a clear “Save As Derivative Work” or equivalent action that mints a new `document_id`.

The format **MUST NOT** treat documents with the same `document_id` as automatically synchronizable: that decision belongs to the collaboration transport (Appendix A).

8.4.3 Manifest Encoding

The manifest chunk is special: it is the bootstrap entry from the superblock into the chunk graph. A cold reader needs to decode the manifest before it has any information from the chunk store.

REQUIREMENT

In this format version, the manifest chunk **MUST** be stored *uncompressed*. Its `CompressionAlgorithm` on disk is implicitly `None`; the superblock’s `manifest_length` field is therefore both the on-disk length and the uncompressed payload length. Implementations **MUST** reject as malformed any bundle whose manifest payload is not directly parseable as a canonical manifest chunk’s uncompressed bytes.

This rule eliminates a bootstrap chicken-and-egg problem: a reader can decode the manifest using only information present in the fixed header and the active superblock, without first consulting the manifest to learn how the manifest is compressed.

All other chunks (operation envelope blocks, snapshots, extension data, layout caches, blobs, integrity indexes, operation indexes, text projection chunks) **MAY** be compressed. Future format major versions **MAY** permit manifest compression, but doing so requires explicit superblock fields (`manifest_compression`, `manifest_uncompressed_length`) and is a non-backward-compatible change.

RATIONALE

The manifest is small (a few kilobytes at most for typical scores), so compression yields negligible space savings. Eliminating the bootstrap dependency is far more valuable than the few hundred bytes of compression headroom.

8.5 Content Hashing

Epiphany uses BLAKE3 as its single content-hashing algorithm.

REQUIREMENT

All content hashes in the bundle **MUST** be BLAKE3-256 outputs (32 bytes). Implementations **MUST NOT** use any other algorithm for content addressing; SHA-256, SHA-3, and other algorithms **MUST NOT** appear in this version of the format. Future major versions **MAY** introduce additional algorithms; such changes require a new format major version.

8.5.1 Domain-Separated Preimages

A naive content-addressing scheme that hashes only payload bytes is vulnerable to collisions across chunks of different semantic kinds. Epiphany uses domain-separated preimages.

```

/// Computes the canonical hash preimage for a chunk's content.
/// The chunk's compression algorithm is metadata on the ChunkRef,
/// not part of the content identity: identical uncompressed
/// payloads produce identical hashes regardless of how they are
/// physically compressed in storage. This preserves deduplication
/// across compression choices.
fn hash_preimage(
    domain_tag: &[u8; 8],
    chunk_kind: ChunkKind,
    schema_version: SchemaVersion,
    uncompressed_length: u64,
    uncompressed_payload: &[u8],
) -> Vec<u8> {
    let mut preimage = Vec::new();
    preimage.extend_from_slice(domain_tag);
    preimage.extend_from_slice(&chunk_kind.canonical_bytes());
    preimage.extend_from_slice(&schema_version.canonical_bytes());
    preimage.extend_from_slice(&uncompressed_length.to_le_bytes());
    preimage.extend_from_slice(uncompressed_payload);
    preimage
}

```

REQUIREMENT

The content hash of a chunk **MUST** be BLAKE3 of the canonical preimage above. The domain tag, chunk kind, schema version, and uncompressed length are part of the preimage; the compression algorithm is *not*. Two chunks with identical uncompressed payloads, identical kinds, and identical schema versions **MUST** produce identical hashes regardless of their compression choices.

RATIONALE

Including compression algorithm in the hash would cause identical content stored with different compression to have different addresses, defeating deduplication and forcing churn during recompression or repack. Including domain tag, kind, and schema version in the preimage closes a real collision: two semantically different chunks with identical raw bytes are no longer interchangeable.

The domain tag is a fixed 8-byte string identifying the bundle format and protecting against cross-format collisions if the same BLAKE3 hash were reused in unrelated contexts. The canonical domain tag for `.musc` chunks is `"MUSCCHNK"`; manifest chunks use `"MUSCMANI"`; blob payloads use `"MUSCBLOB"`.

8.6 Chunks

A chunk is an immutable, content-addressed unit of storage. Chunks are referenced from the manifest by `ChunkRef`.

```
pub struct ChunkRef {
    /// Content identifier: BLAKE3 of the canonical preimage.
    pub id: ChunkId,

    /// Kind of chunk. The reader uses this to dispatch parsing.
    pub kind: ChunkKind,

    /// Schema version targeted by this chunk's payload encoding.
    pub schema_version: SchemaVersion,

    /// Offset of the chunk's compressed payload in the bundle file.
    pub offset: u64,

    /// Length of the on-disk compressed payload.
    pub compressed_length: u64,

    /// Length of the uncompressed payload (used to size buffers
    /// before decompression).
    pub uncompressed_length: u64,

    /// Compression algorithm. Part of the ChunkRef, not part of
    /// the chunk's identity.
    pub compression: CompressionAlgorithm,
```

```

    /// Restated content hash, redundant with the id field but
    /// included in the ChunkRef for fast verification without
    /// fetching the chunk.
    pub hash: ContentHash,
}

pub enum ChunkKind {
    /// A block of operation envelopes.
    OperationEnvelopeBlock,

    /// An operation-id-to-block index accelerator.
    OperationIndex,

    /// A materialized snapshot of canonical state (full or
    /// partial). Canonical if referenced as the manifest's
    /// canonical_base; cache (acceleration) otherwise.
    Snapshot,

    /// A large opaque blob (audio, image, font, ML model).
    Blob,

    /// Extension-defined data, preserved opaquely by core readers.
    ExtensionData,

    /// The text projection's root document.
    TextProjection,

    /// Layout cache: derivative from canonical state, always
    /// discardable.
    LayoutCache,

    /// Integrity index: cross-references chunk hashes.
    IntegrityIndex,

    /// Manifest chunk. Special: referenced from the superblock,
    /// not from another chunk.
    Manifest,
}

pub enum CompressionAlgorithm {
    /// No compression. Stored payload bytes equal uncompressed.
    None,

    /// Zstandard compression. Implementations MUST support reading.
    Zstd { level: u8 },

    /// Other algorithms reserved for future versions.
    Reserved(u8),
}

pub struct ContentHash(pub [u8; 32]);

/// Chunk identifier. A newtype around ContentHash: the chunk's

```

```

/// identifier is identical to its content hash, but the type
/// distinction is preserved to make role visible at use sites
/// (a ChunkId is what you store in a ChunkRef and what you look
/// up; a ContentHash is what you compute by hashing). Both occupy
/// the same 32 bytes.
pub struct ChunkId(pub ContentHash);

```

REQUIREMENT

Chunks **MUST** be immutable. Once a chunk is written and its bytes durably flushed, those bytes **MUST NOT** be modified. New state is encoded as new chunks with new identifiers; old chunks become unreachable garbage when their referencing manifests are no longer reachable.

Implementations **MUST** verify chunk hashes on first read of each chunk in a session: decompress the payload, compute the canonical preimage's BLAKE₃, and confirm the result matches the declared `hash`. Subsequent reads **MAY** skip verification if the chunk has been validated within the current session. A chunk whose hash does not match **MUST** be treated as corrupt: if it is a cache chunk, discard and rebuild; if it is a canonical chunk (operation block, retained snapshot, or blob referenced from the manifest), surface as hard corruption.

Readers **MUST** verify that decompression of a chunk's payload produces exactly `uncompressed_length` bytes. Length mismatch is treated as corruption.

8.7 Operation Envelope Blocks

The canonical document is stored as operation-envelope blocks. Each block is a chunk of kind `OperationEnvelopeBlock` containing a vector of envelopes, with summary metadata.

```

pub struct OperationEnvelopeBlock {
    /// Block identifier (same as its ChunkId in the chunk store).
    pub block_id: ChunkId,

    /// Operation envelopes in this block. Order within a block is
    /// not normative for canonical reduction (envelopes are a set)
    /// but writers SHOULD preserve authoring order within a block
    /// for diagnostic purposes.
    pub envelopes: Vec<OperationEnvelope>,

    /// DVV summary covering the envelopes in this block. Enables
    /// readers to determine causal coverage without parsing every
    /// envelope.
    pub dvv_summary: CausalContext,

    /// Earliest stamp in the block.
    pub min_stamp: OperationStamp,

    /// Latest stamp in the block.
    pub max_stamp: OperationStamp,
}

```

REQUIREMENT

Writers **SHOULD** begin a new operation-envelope block when adding another envelope would cause the uncompressed block payload to exceed 1 MiB, except when an individual envelope’s encoded size exceeds 1 MiB, in which case the envelope occupies its own block.

Readers **MUST** accept valid blocks of any size up to a profile- declared maximum (default: 64 MiB uncompressed). Profile declarations **MAY** raise or lower this bound. Blocks exceeding the active profile’s bound **MUST** be treated as malformed.

The `operation_roots` field of the manifest **MUST** list every operation-envelope block that contributes to the canonical document. The set of envelopes *is* the union of all envelopes across all referenced blocks; block boundaries are storage artifacts, not semantic structure.

8.7.1 The Operation Index

The optional `operation_index_root` chunk maps each `OperationId` to the `ChunkRef` of its enclosing block plus an offset within the block. It enables $O(\log n)$ lookup of an operation by id.

REQUIREMENT

The operation index is an acceleration structure, not canonical. If absent, readers rebuild it by scanning all blocks. If present but corrupt or stale, readers **MUST** reject the index and rebuild from blocks.

Writers **SHOULD** rebuild or incrementally update the operation index at commit time when the operation set has grown significantly since the last commit.

8.8 Snapshots and Canonical Bases

A snapshot is a materialized score state at a particular causal frontier. Snapshots have two distinct roles:

Acceleration snapshot. A cache of the materialized state at some point, used to skip reduction work. Listed in the manifest (or not), but *not* declared as a canonical base. Discardable.

Canonical base. A materialized state at a causal frontier that the manifest declares as the base after pruning. Together with the operation envelopes after the frontier, this constitutes the canonical document. Not discardable while it remains declared.

```
pub struct SnapshotRef {
    pub snapshot_id: SnapshotId,

    /// The causal frontier this snapshot materializes: the
    /// envelopes whose effects are baked into the snapshot.
    pub covers_causal_frontier: CausalContext,

    /// Reduction algorithm version under which the snapshot was
    /// produced. Snapshots produced under an earlier algorithm
```

```

    /// version cannot be used as canonical bases under a later
    /// algorithm without rebuilding.
    pub reduction_algorithm_version: ReductionAlgorithmVersion,

    /// Profile under which the snapshot was produced.
    pub profile_id: ProfileId,

    /// Root chunk of the snapshot's materialized state.
    pub root: ChunkRef,

    /// Hash of the snapshot's root chunk for fast verification.
    pub hash: ContentHash,
}

```

8.8.1 The Canonical Document Identity

REQUIREMENT

The canonical document is defined as follows:

- If `canonical_base` is `None`, the canonical document is the deterministic reduction (Section 6.3) of the union of envelopes in all `operation_roots` blocks.
- If `canonical_base` is `Some(base)`, the canonical document is the base snapshot's materialized state plus the deterministic reduction of all envelopes whose `OperationId` is *not covered* by the base's `covers_causal_frontier`. An `OperationId` is covered by a `CausalContext` if and only if it appears in the dotted version vector's per-replica contiguous range or among its dots (i.e., causal-precedes-or-equals under the DVV's induced partial order). HLC stamps **MUST NOT** be used to determine coverage; only the DVV frontier is canonical.
- A snapshot may serve as the canonical base only if its `reduction_algorithm_version` equals the active superblock's `reduction_algorithm_version` and its `profile_id` matches the manifest's profile declarations.

At most one `canonical_base` is active at any time. Pruning replaces the current canonical base with a new one whose frontier strictly dominates the previous frontier under the DVV partial order; the old base becomes unreachable and is subject to garbage collection.

Acceleration snapshots (`acceleration_snapshots`) **MUST** be ignored for the purposes of canonical state. They exist to accelerate cold open and incremental reduction; if their content disagrees with what the canonical reduction would produce, they **MUST** be rebuilt or discarded.

Caches, indexes, layout chunks, and other materialized artifacts in the bundle are acceleration structures only and **MUST** be ignored or rebuilt if they disagree with the canonical document.

8.8.2 Pruning

REQUIREMENT

Pruning is the act of removing operation envelopes that are causally covered by a new canonical base snapshot. Pruning:

1. Materializes a snapshot covering the chosen causal frontier (a DVV).
2. Replaces the current `canonical_base` field with a `SnapshotRef` naming the new snapshot.
3. Removes operation-envelope blocks whose envelopes are *entirely* covered by the new frontier. Blocks with mixed coverage **MUST** be split or retained whole; uncovered envelopes **MUST** remain reachable.
4. Commits the new manifest via the atomic write protocol (Section 8.10).

Pruning **MUST NOT** alter the canonical document state: the reduction of (base snapshot + post-frontier envelopes) under the active algorithm **MUST** equal the reduction of the pre-pruning envelope set.

Implementations **MUST NOT** prune envelopes outside the explicit pruning protocol. Operation envelopes are not silently garbage- collected based on age or other heuristics.

8.9 Blobs

The blob store holds large opaque content: audio reference tracks, embedded images, custom fonts, ML model checkpoints. Blobs are content-addressed identically to chunks (BLAKE3 of uncompressed payload, with the "MUSCBLOB" domain tag).

```
pub struct BlobRef {
    pub blob_id: BlobId,
    pub media_type: String, // RFC 6838 media type
    pub offset: u64,
    pub compressed_length: u64,
    pub uncompressed_length: u64,
    pub compression: CompressionAlgorithm,
    pub hash: ContentHash,

    /// Optional declared maximum size that readers MAY use to
    /// reject blobs that exceed reader resource limits.
    pub declared_max_uncompressed_length: Option<u64>,
}
```

REQUIREMENT

Readers **MUST** apply resource limits to blob loading: uncompressed length **MUST** be checked against the reader's policy before decompression begins; `media_type` **MUST** be validated against the reader's accepted list before any type-specific parsing.

Implementations **MUST NOT** eagerly load all blobs at open; blobs **MUST** be streamed on demand. Operations requiring blob content **MUST** be designed for lazy access.

Blob content is opaque to the core. The core stores, transmits, and integrity-checks blobs; it does not interpret their bytes.

8.10 The Atomic Write Protocol

A commit transforms the bundle from one canonical state to another. The protocol uses two superblock slots so that a crash at any point during commit leaves the bundle in either the pre-commit state (active superblock unchanged) or the post-commit state (active superblock now points at the new manifest); no intermediate state is visible.

REQUIREMENT

Commits **MUST** follow this sequence:

1. Write all new chunks (envelope blocks, possibly a new snapshot, possibly a new operation index, etc.) to file regions outside the prelude and outside currently-reachable chunks. Writers **MAY** append at end-of-file or fill known- unreachable regions, but **MUST NOT** overwrite any currently-reachable chunk.
2. Durably flush the file.
3. Compute the new manifest chunk's payload and write it.
4. Durably flush the file.
5. Compute the new Superblock with `generation = active_generation + 1` and `manifest_offset` pointing at the new manifest chunk.
6. Write the new superblock to the currently-inactive superblock slot (a 256-byte write at a fixed offset).
7. Durably flush the file. *This is the commit point.* Before this flush returns successfully, the active superblock is the old one; after it returns successfully, the active superblock is the new one.

After step 7, the previously-active superblock slot becomes the inactive slot for the next commit. Garbage collection is a separate, optional, deferred operation (Section 8.11).

8.10.1 Crash Recovery

REQUIREMENT

On open after any crash, recovery follows the superblock selection rule (Section 8.3). The bundle is always in a recoverable state:

- Crash between steps 1 and 7: the active superblock is unchanged; the new chunks and the new manifest are unreachable garbage; the document opens at the pre-commit state. A later GC reclaims the orphaned bytes.
- Crash during step 7: the superblock slot write may be torn. The CRC check rejects torn writes. Recovery falls back to the other slot. The document opens at the pre-commit state. The partially-written slot will be overwritten by the next successful commit.

- Crash after step 7: the new superblock is active. The document opens at the post-commit state. No special recovery is needed.

Recovery **MUST NOT** synthesize or guess: if neither superblock validates, the file is corrupt and **MUST** be reported as such.

8.11 Garbage Collection and Retention

Garbage collection reclaims bytes occupied by chunks that are no longer reachable from any retained manifest. Retention policy governs which old manifests are preserved for rollback.

8.11.1 Retention Policy

```
pub struct RetentionPolicy {
    /// Maximum number of previous manifests to retain beyond the
    /// active one. 0 means "no rollback retention"; the active
    /// manifest is the only one preserved. The default declared by
    /// the Full profile is profile-defined.
    pub retain_previous_manifests: u32,

    /// Optional wall-clock duration beyond which old retained
    /// manifests may be reclaimed regardless of count. None means
    /// no time-based eviction.
    pub retain_duration: Option<WallClockDuration>,

    /// Whether to preserve manifests explicitly marked as named
    /// checkpoints (user-invoked "save a version" action) beyond
    /// the count and duration limits.
    pub retain_named_checkpoints: bool,
}
```

REQUIREMENT

The active conformance profile **MUST** declare a `RetentionPolicy`. Implementations **MUST** honor the declared policy: chunks reachable from any retained manifest **MUST NOT** be garbage-collected, and retention selection (which manifests are retained) **MUST** be deterministic given the manifest history and the policy.

Rollback is the act of writing a new superblock that points at a previously-active manifest still reachable in the chunk store. The fixed layout has exactly two superblock slots; rollback does not require additional on-disk superblocks. It requires only that the target manifest's reachable chunks remain in the bundle, which the retention policy guarantees.

8.11.2 Garbage Collection Rules

REQUIREMENT

Garbage collection is a conservative, optional, deferred operation. It **MUST** preserve every byte reachable transitively from:

1. The active superblock’s manifest.
2. Every retained manifest selected by the active `RetentionPolicy`.
3. Any pending edit’s working set (chunks written but not yet committed).

Garbage collection **MUST NOT** run as part of a commit’s critical path. It **MAY** run as a background operation or as an explicit user-invoked compaction. Pre-GC and post-GC bundles **MUST** contain identical canonical state and identical retained-rollback reachability.

8.12 Forward Compatibility and Edit Barriers

The bundle must round-trip cleanly across implementations that may not all understand every extension referenced in it.

8.12.1 Extension Declarations

```
pub struct ExtensionDeclaration {
    pub extension_id: ExtensionId,
    pub version: SemVer,

    /// If true, an implementation that does not understand this
    /// extension MUST refuse to open the bundle for editing, but
    /// MAY open read-only.
    pub required: bool,

    /// Root chunks owned by this extension. Opaque to core
    /// readers; preserved across reads and writes.
    pub preserved_chunk_roots: Vec<ChunkRef>,

    /// Object kinds this extension contributes to or depends on,
    /// for edit-barrier evaluation.
    pub affected_object_kinds: Vec<ObjectKind>,

    /// Edit barriers declared by this extension.
    pub edit_barriers: Vec<EditBarrier>,
}

pub struct EditBarrier {
    /// Scope of the barrier: whole score, region, staff instance,
    /// analysis layer, object set, pitch space, tuning context, etc.
    pub scope: BarrierScope,

    /// Object kinds protected by this barrier.
    pub affected_object_kinds: Vec<ObjectKind>,
}
```

```

    /// Operation kind tags prohibited by this barrier. A barrier
    /// prohibits an operation class, not one exact payload, so it
    /// references OperationKindTag rather than OperationKind.
    pub prohibited_operation_kinds: Vec<OperationKindTag>,

    /// Additional condition narrowing the barrier (e.g., applies
    /// only when a specific attachment is present).
    pub condition: BarrierCondition,
}

/// A discriminator-only view of OperationKind (Chapter ch:semops),
/// without payload. Used by edit barriers, conformance reports,
/// and diagnostic surfaces where the operation's class matters
/// but its payload does not. There is a canonical mapping from
/// OperationKind to OperationKindTag.
pub enum OperationKindTag {
    InsertEvent,
    DeleteEvent,
    ModifyEvent,
    RespellPitch,
    Transpose,
    CreateCrossCutting,
    DeleteCrossCutting,
    ModifyCrossCutting,
    ChangeRegionTimeModel,
    InsertRegion,
    DeleteRegion,
    InsertStaffInstance,
    DeleteStaffInstance,
    SetUserSystemBreak,
    SetUserPageBreak,
    DeclareTransaction,
    Registered(OperationKindRegistryId),
}

pub enum BarrierScope {
    WholeScore,
    Region(RegionId),
    StaffInstance(StaffInstanceId),
    AnalysisLayer(AnalysisLayerId),
    ObjectSet(Vec<TypedObjectId>),
    PitchSpace(PitchSpaceId),
    TuningContext,
    /// Registered scope kind for grammar-specific scoping.
    Registered(BarrierScopeRegistryId),
}

/// Additional narrowing condition for an edit barrier. The barrier
/// applies only when its scope, affected object kinds, prohibited
/// operation kinds, and this condition all match the candidate edit.
pub enum BarrierCondition {
    /// The barrier applies unconditionally within its scope.

```

```

Always,

/// The barrier applies only while the named object exists
/// (not tombstoned). Used when an extension's invariant
/// depends on a specific object remaining live.
ObjectExists(TypedObjectId),

/// The barrier applies only when the named object carries
/// data declared by the named extension. Used when the
/// invariant protected by the barrier exists only for
/// objects participating in the extension.
ObjectHasExtensionData {
    object: TypedObjectId,
    extension: ExtensionId,
},

/// Conjunction: all conditions must match.
All(Vec<BarrierCondition>),

/// Disjunction: any condition matching is sufficient.
Any(Vec<BarrierCondition>),

/// Negation: the inner condition must not match.
Not(Box<BarrierCondition>),

/// Registered condition kind, evaluated by an extension-aware
/// implementation. Core implementations evaluate Registered
/// conditions as Always (conservative): they treat the barrier
/// as active whenever an unknown extension's condition would
/// otherwise narrow it.
Registered(BarrierConditionRegistryId),
}

```

8.12.2 Behavior Under Unknown Extensions

REQUIREMENT

When an implementation opens a bundle declaring an extension it does not understand:

- If the extension is `required = true`, the implementation **MUST** either refuse to open the bundle or open it strictly read-only. It **MUST NOT** edit.
- If the extension is `required = false`, the implementation **MAY** open the bundle for editing, but **MUST** preserve the extension's `preserved_chunk_roots` across reads and writes. The chunks are treated as opaque bytes.
- Edits **MUST** be checked against every active edit barrier. An edit matching a barrier's scope, affected object kinds, and operation kinds is prohibited unless the user explicitly performs an *unsafe edit*.
- An unsafe edit is an explicit user action that acknowledges loss of extension data. The unsafe-edit operation **MUST** tombstone the relevant extension chunks (so

they are no longer preserved) rather than silently breaking extension invariants. The unsafe-edit mechanism prevents unknown extensions from rendering a document permanently uneditable while still preserving extension data by default.

8.13 Text Projection

The format admits a deterministic projection to a canonical s-expression text form. The text projection is normative.

REQUIREMENT

The text projection **MUST** preserve, deterministically and bidirectionally with the binary form:

- All operation envelopes (with their identities, stamps, causal contexts, transaction groupings, and payloads).
- All profile declarations.
- All extension declarations, including their preserved chunk roots, with extension payloads encoded as base64 or another canonical text form.
- The active canonical base snapshot’s reference, frontier, and reduction algorithm version (the snapshot payload itself **MAY** be encoded compactly or referenced externally).
- All canonical reduced state.

The text projection **MUST NOT** be required to preserve:

- Chunk offsets within the file.
- Compression algorithm choices.
- Cache chunks (operation indexes, layout caches, integrity indexes).
- Garbage bytes from prior commits.
- Superblock generation numbers, slot assignments, or CRCs.

Two binary bundles whose canonical document semantics are identical **MUST** project to identical text. Two text projections that parse to identical canonical document semantics **MUST** re-serialize to binary bundles whose canonical document semantics are identical (though their physical layout, chunking, and compression **MAY** differ).

8.13.1 Use Cases

The text projection serves three primary use cases:

Version control. Text projections diff and merge in line-based tools. Merge conflicts surface at the operation- envelope level, which is the meaningful level for collaborative editing.

Format inspection and debugging. A canonical text form enables direct inspection of bundle contents without binary tooling.

Long-term archival. Text formats survive transitions between binary specifications. The text projection’s structure is stable across binary format major versions where the semantic

content is preserved.

8.14 Schema Versioning

Chunk payloads are encoded against a schema declared by each chunk's `schema_version` field. Schemas evolve under a semantic versioning discipline.

```
pub struct SchemaVersion {  
    pub major: u16,  
    pub minor: u16,  
}
```

REQUIREMENT

Schema versioning rules:

- Major version changes are non-backward-compatible. Readers that do not support the chunk's major schema version **MUST** handle it according to the chunk's role:
 - *Canonical chunks* (operation envelope blocks, the canonical base snapshot, the manifest, and blobs referenced by canonical operations or reduced state) **MUST** be either parseable or the bundle **MUST** be opened in read-only preservation mode (or refused). A reader that cannot parse a canonical chunk cannot know the canonical document.
 - *Non-canonical chunks* (acceleration snapshots, operation index, layout caches, text projection, integrity index, extension-defined chunks declared as preservable opaque content) **MAY** be treated as opaque preserved content; the reader does not need to understand them to open the bundle, and writes preserve them verbatim.
- Unknown canonical operation payloads (operation kinds not in the reader's understood catalog) require the relevant extension support or the reader **MUST** open the bundle read-only without materializing the unknown operations. Materializing an unknown operation **MUST NOT** proceed silently: doing so would fork canonical state against any reader that does understand the operation.
- Minor version changes are backward-compatible. A reader supporting major version N **MUST** accept any minor version $N.m$ for $m \leq$ the reader's supported minor. Minor version changes **MUST** only add optional fields or enumeration variants in a way preserving the wire format of prior values.
- The manifest's `manifest_schema_version` declares the schema of the manifest itself. Each chunk's `schema_version` field declares the schema of that chunk's payload.
- Schema evolution is governed by the Binary Format companion specification, which defines the wire encoding for each schema version.

8.15 Format Profiles

A format profile defines a conformance subset that implementations **MAY** promise to support. Profiles enable lightweight readers to declare their capabilities precisely.

```
pub struct ProfileDeclaration {
    pub profile_id: ProfileId,
    pub version: SemVer,

    /// Constraints imposed by this profile (maximum block size,
    /// permitted compression algorithms, required extensions, etc.)
    pub constraints: ProfileConstraints,
}

pub enum ProfileId {
    /// Full profile: all features supported, no restrictions.
    Full,

    /// Read-only profile: bundles produced under this profile MAY
    /// be opened by readers that cannot edit. Editing implementations
    /// SHOULD upgrade the profile on first edit.
    ReadOnly,

    /// Lite profile: a reduced feature set for embedded or mobile
    /// readers. Restricts maximum block sizes, snapshot complexity,
    /// and permitted extensions.
    Lite,

    /// Custom profile defined by a registered profile declaration.
    Custom(ProfileRegistryId),
}
```

REQUIREMENT

Every bundle **MUST** declare at least one profile in its manifest's `profile_declarations`. An implementation **MAY** open a bundle if it supports any of the declared profiles, subject to the required-extension rules (Section 8.12).

8.16 Compression

Each chunk **MAY** be stored compressed. Compression is per-chunk metadata, not part of content identity.

REQUIREMENT

Conforming implementations **MUST** support reading chunks compressed with Zstandard at any level `zstd` defines. Writers **MAY** choose to compress or not on a per-chunk basis; uncompressed chunks (`CompressionAlgorithm::None`) are always permitted.

Implementations **MUST NOT** require a specific compression algorithm for canonical conformance. Future major format versions **MAY** introduce additional algorithms via the `Reserved` variant; such changes require a new format major version. Decompression **MUST** verify the output length against the declared `uncompressed_length` and reject chunks whose decompressed size disagrees.

8.17 Streaming Reads

Cold open of a large score **MUST** be fast: under one second to first interactive frame on a 100-page orchestral score, per the performance requirements (Chapter 10). The format's prelude and chunk-graph design support this.

8.17.1 Required Locatability

REQUIREMENT

The bundle **MUST** guarantee that:

- The fixed header is at offset zero of the file.
- Both superblock slots are at fixed offsets within the first 576 bytes.
- The active manifest is locatable from the active superblock without scanning.
- Operation-envelope blocks, the canonical base snapshot (if any), acceleration snapshots, blobs, and extension chunks are locatable from the manifest's roots without scanning.
- The operation index (if present) provides $O(\log n)$ operation-id lookup; if absent, scanning a small number of blocks suffices.

Locatable means: a reader can determine the offset and length of the chunk in the bundle using only previously read content.

8.17.2 Cold Open Procedure

The recommended cold open sequence:

1. Read the 64-byte fixed header. Verify magic and CRC.
2. Read both 256-byte superblock slots. Verify CRC on each. Select the active slot per the superblock selection rule (Section 8.3).
3. Read and verify the manifest chunk referenced by the active superblock. The manifest is stored uncompressed (Section 8.4.3), so this is a single read-and-parse.
4. If the manifest's `canonical_base` is `Some`, read and verify the canonical base snapshot. The snapshot provides the materialized state at its declared causal frontier.
5. Stream the operation-envelope blocks listed in `operation_roots`. Apply the canonical reduction (Chapter 6) over the envelopes whose `OperationId` is not covered by the canonical base's `covers_causal_frontier` (if a base is present), or over every envelope in the operation set (if no base is present). The result is the materialized score graph.

6. Invoke the layout pipeline (Chapter 7) over the materialized graph, producing the first system’s layout. Layout caches (if reachable from the manifest as accelerators) **MAY** be consulted to skip work; cache misses fall back to fresh layout computation.
7. Load remaining regions, envelope blocks, snapshot ranges, and layout caches lazily as the user navigates.

For unsynchronized first-frame display, implementations **MAY** present the canonical base’s materialized state immediately while the post-frontier reduction completes in the background; user edits **MUST** be deferred until reduction completes, since the visible state is provisional until then.

8.17.3 Memory Mapping

REQUIREMENT

Implementations **MAY** memory-map the bundle file for efficient random access to chunks and blobs. The format’s content-addressing and chunk immutability guarantees make memory mapping safe: chunks cannot change underneath a reader holding a pointer into them. Superblock slots are 256-byte aligned and never overlap reachable chunk content, so writers updating an inactive superblock cannot corrupt readers accessing the active state.

8.18 Binary Format Companion

The byte-level encoding (varint conventions, exact field bit widths, record alignment, endianness, string encoding details, struct layouts) is delivered as a companion specification, the Binary Format document. The Binary Format document is normative; an implementation cannot conform to the file format specification without conforming to the Binary Format specification.

OPEN QUESTION

The Binary Format companion is under development as a separate specification. Its development follows this chapter; the byte-level decisions depend on the structural decisions specified here.

8.19 Forward References

- The constraint-solver interface (Chapter 9) operates on the in-memory `ConstrainedLayout IR`; layout caches stored in the bundle are derivative and not part of the canonical state.
- The full performance requirements that this format must satisfy are in Chapter 10.
- The Binary Format companion specifies byte-level encoding.
- The Text Projection companion specifies the canonical s-expression form.
- The Profile Conformance companion specifies the feature lists per profile.

Constraint-Solver Interface

This chapter specifies the constraint-solver interface: the `ConstrainedLayoutIR` the solver consumes, the `ResolvedLayoutIR` it produces, the constraints it must respect, the determinism contract it must satisfy, the diagnostic information it must emit, and the conformance discipline by which implementations are judged.

The chapter deliberately specifies the *interface* and its contracts rather than a particular algorithm. Implementations are free to choose among approaches (Cassowary, linear programming relaxations, custom heuristics, differentiable layout, learned methods) and to evolve their choices over time. Conformance is established by hard-constraint validity, deterministic behavior, diagnostic completeness, and performance and quality on a reference suite of test scores.

9.1 Design Principles

The interface is normative; global optimality is not. The solver’s input/output contract, its constraint families, its diagnostic surface, and the discipline by which it is evaluated are normative. Global aesthetic optimality (proving no better layout exists) is neither decidable nor practical to verify for arbitrary scores; it is therefore replaced by reference-suite thresholds.

Hard constraints are validity; quality metrics are evaluation axes. Hard constraints are the conditions under which a layout is valid at all. Quality metrics are the axes along which valid layouts are evaluated and compared. The solver **MUST NOT** trade off a hard-constraint violation for a better quality score: a layout that violates a hard constraint is not a layout, regardless of its quality vector.

Determinism within an implementation; equivalence across implementations. A given implementation at a fixed version **MUST** produce identical output for identical input. Different conforming implementations **MAY** produce different layouts; they conform by satisfying hard constraints, meeting diagnostic contracts, behaving deterministically internally, and passing reference-suite thresholds. Cross-implementation byte-identity is explicitly not required.

Pareto frontiers as design target, not conformance obligation. Solvers **SHOULD** seek layouts on or near the Pareto frontier of the normative quality metrics. They are *not* required to prove they have done so universally; conformance is reference-suite-based.

Deterministic budgets, not wall-clock budgets. Wall-clock time **MAY** stop interactive work, but the returned layout’s canonical form depends on deterministic counters (iterations, constraint evaluations, search nodes), not on scheduler luck.

Incremental solving as observational equivalence. Incremental re-solving **MUST** be observationally equivalent to full solving of the declared invalidation scope under the same profile and budget. Implementations are free to widen scopes conservatively but **MAYNOT** return layouts that diverge from the canonical scoped result.

Structured diagnostics, never silent corruption. Solver failures are first-class outputs with diagnostic information sufficient for the editor to surface meaningful feedback. The score graph is never invalidated by a solver failure; a solver error means “we could not produce a layout,” not “the score is broken.”

9.2 The Solver Interface

```
pub trait ConstraintSolver: Send + Sync {
    /// The solver's identifying tier. Used by readers/editors to
    /// decide whether this solver is sufficient for the active
    /// score's declared solver tier requirement.
    fn tier(&self) -> SolverTier;

    /// The solver's implementation version. Identical input plus
    /// identical version produces identical output (within-
    /// implementation determinism).
    fn version(&self) -> SolverVersion;

    /// Solve from scratch.
    fn solve(
        &self,
        input: &ConstrainedLayoutIR,
        config: &SolverConfig,
    ) -> SolveReport;

    /// Solve incrementally over the declared invalidation scope.
    fn solve_incremental(
        &self,
        input: &ConstrainedLayoutIR,
        prior: &SolverState,
        invalidations: &InvalidationSet,
        config: &SolverConfig,
    ) -> SolveReport;
}

pub struct SolverConfig {
    /// Conformance profile under which to solve. Selects metric
    /// thresholds, profile-specific constraints, and the active
    /// extension set.
    pub profile: SolverProfile,

    /// Deterministic budget. Wall-clock time is advisory only.
    pub budget: SolverBudget,
```

```

    /// Tie-breaking weights for selecting among layouts of
    /// equivalent quality.
    pub tie_breaking: TieBreakingWeights,
}

pub struct SolverBudget {
    /// Maximum solver iterations (algorithm-defined unit; the
    /// reference algorithm counts Cassowary pivots).
    pub max_iterations: u64,

    /// Maximum search-tree nodes explored, for algorithms that
    /// perform discrete search.
    pub max_nodes: u64,

    /// Maximum constraint-evaluation count.
    pub max_constraint_evaluations: u64,

    /// Advisory wall-clock cap, in milliseconds. The solver MAY
    /// stop early when this is exceeded, but the layout it
    /// returns under stoppage MUST be the same layout that the
    /// deterministic budget would have produced at the
    /// corresponding deterministic counter state. Wall-clock time
    /// MUST NOT change the canonical layout.
    pub advisory_wall_time_ms: Option<u64>,
}

pub struct SolverBudgetUsed {
    pub iterations: u64,
    pub nodes: u64,
    pub constraint_evaluations: u64,
    pub wall_time_ms: u64,
}

```

REQUIREMENT

solve and solve_incremental **MUST** be pure functions of their inputs within the determinism contract (Section 9.7). They **MUST NOT** consult external state (system clock, environment, thread scheduling, random sources) in ways that influence the returned layout.

Implementations **MAY** use multi-threaded execution internally, but the result **MUST** be identical to a single-threaded execution under the same deterministic budget.

9.3 The Solver Report

Every solver invocation returns a `SolveReport`. There is no separate “error” return; failures and partial successes appear as report variants distinguished by `SolveStatus`.

```

pub struct SolveReport {
    pub status: SolveStatus,
}

```

```

    /// Whether every hard constraint is satisfied.
    pub satisfied_hard_constraints: bool,

    /// The layout produced. Always present; under failure
    /// statuses, the layout is diagnostic-only and MUST NOT be
    /// used as if it were valid.
    pub layout: ResolvedLayoutIR,

    /// Updated solver state for subsequent incremental calls.
    pub state: SolverState,

    /// Unsatisfied hard constraints, if any. Empty under
    /// SolveStatus::Solved.
    pub unsatisfied_constraints: Vec<ConstraintId>,

    /// Non-fatal warnings about the solution.
    pub warnings: Vec<SolverWarning>,

    /// Quality metric vector for the returned layout.
    pub metric_vector: QualityMetricVector,

    /// Budget consumed during this solve.
    pub budget_used: SolverBudgetUsed,
}

pub enum SolveStatus {
    /// All hard constraints satisfied, target quality reached.
    Solved,

    /// All hard constraints satisfied, but warnings were
    /// generated (e.g., a soft constraint had a large violation,
    /// or a layout decision was unusual and worth surfacing).
    SolvedWithWarnings,

    /// Deterministic budget exhausted before reaching target
    /// quality. The returned layout MUST still satisfy all hard
    /// constraints; it simply may not be optimal. If hard
    /// constraints could not be satisfied within budget, the
    /// status is Unsatisfiable, not PartialBudgetExhausted.
    PartialBudgetExhausted,

    /// Hard constraints cannot be simultaneously satisfied. The
    /// returned layout is diagnostic-only and MUST NOT be
    /// presented as a valid layout.
    Unsatisfiable,

    /// Solver bug or unexpected error. The returned layout is
    /// diagnostic-only. Editors SHOULD surface this as a defect
    /// to report.
    InternalError,
}

```

```
pub struct SolverWarning {
  pub kind: SolverWarningKind,
  pub affected_objects: Vec<TypedObjectId>,
  pub message: String,
}

pub enum SolverWarningKind {
  LargeSoftConstraintViolation { constraint: ConstraintId, magnitude: f64 },
  UnusualLayoutDecision(String),
  QualityFloorApproached { metric: QualityMetricKind },
  ExtensionWarning(ExtensionWarningId),
}
```

REQUIREMENT

The `SolveReport.layout` field is always populated, but its authority depends on status:

- **Solved** or **SolvedWithWarnings**: the layout is a valid layout that the editor may render.
- **PartialBudgetExhausted**: the layout is a valid layout (`satisfied_hard_constraints` is `true`) and the editor may render it, while noting that quality is below the profile's target.
- **Unsatisfiable** or **InternalError**: the layout is diagnostic-only. Editors **MAY** render it for the user to see what failed, but **MUST** clearly indicate the layout is not authoritative.

A solver **MUST NOT** return `PartialBudgetExhausted` with `satisfied_hard_constraints == false`. If hard constraints cannot be satisfied within the budget, the correct status is `Unsatisfiable`, with the partial layout marked as such.

9.4 Constraint Families

The solver consumes constraints in normalized form. The constraint families enumerated here are the normative core; implementations **MAY** support additional families via `LayoutConstraint::Registered`.

9.4.1 Strength Levels

```
pub enum ConstraintStrength {
  /// Hard constraint. The solver MUST satisfy this constraint,
  /// or return Unsatisfiable.
  Required,

  /// Soft constraint with an associated weight. The solver
  /// minimizes the weighted violation when optimizing.
  Preferred { weight: f64 },
}
```

REQUIREMENT

A solver **MUST NOT** treat a Required constraint as if it were Preferred for any reason, including for quality optimization. A layout that violates a Required constraint is not a valid layout, regardless of its quality vector.

9.4.2 Normative Constraint Families

The constraint catalog (spring, collision, alignment, containment, break, aesthetic, extension) is specified in Chapter 7. Implementations **MUST** support every family normatively required by the active solver tier (Section 9.6).

9.5 Quality Metrics

The solver's output is evaluated against a normative set of quality metrics. Each metric is normalized to the closed interval $[0.0, 1.0]$, with 0.0 representing the best achievable value and 1.0 representing the worst tolerable value before a profile threshold flags the metric as failing.

```

/// A quality metric value normalized to [0.0, 1.0].
/// Lower is always better.
pub struct NormalizedMetric(pub f64);

impl NormalizedMetric {
    /// Construct, panicking if the value is not finite or is out
/// of range. Conforming implementations MUST construct only
/// valid NormalizedMetric values.
    pub fn new(value: f64) -> Self {
        assert!(value.is_finite());
        assert!(0.0 <= value && value <= 1.0);
        NormalizedMetric(value)
    }
}

pub struct QualityMetricVector {
    pub collision_penalty: NormalizedMetric,
    pub spacing_distortion: NormalizedMetric,
    pub slur_shape_penalty: NormalizedMetric,
    pub beam_slope_penalty: NormalizedMetric,
    pub vertical_density_penalty: NormalizedMetric,
    pub system_break_penalty: NormalizedMetric,
    pub page_fill_efficiency: NormalizedMetric,
    pub casting_off_quality: NormalizedMetric,
    pub symbol_density_uniformity: NormalizedMetric,

    /// Additional metrics contributed by extensions. Each MUST be
/// a valid NormalizedMetric (finite, in [0.0, 1.0], lower is
/// better).
    pub extension_metrics: Vec<ExtensionMetric>,
}

pub struct ExtensionMetric {

```

```
pub metric_id: ExtensionMetricId,
pub value: NormalizedMetric,
}
```

REQUIREMENT

Every `NormalizedMetric` value **MUST** satisfy:

- It is finite (not NaN, not infinity).
- It lies in the closed interval $[0.0, 1.0]$.
- Lower values denote better layouts; higher values denote worse layouts.

Per-metric normalization functions (mapping raw measurements to $[0.0, 1.0]$) are specified in the Quality Metric Catalog companion document. Implementations **MUST** use the catalog's normalization; arbitrary normalization is non-conforming.

Extension metrics **MAY** define their own normalization function but **MUST NOT** change orientation or range: 0.0 remains best, 1.0 remains worst, and values must remain finite within range.

9.5.1 Pareto Frontiers as Design Target

Solvers should seek layouts on or near the Pareto frontier of the normative metrics. This is the design intent. It is not the conformance requirement.

REQUIREMENT

The normative metric set defines the axes along which layout quality is evaluated. Solvers **SHOULD** produce Pareto-efficient layouts with respect to these metrics: layouts for which no metric can be improved without worsening another. Because global Pareto optimality is not practically verifiable for arbitrary scores, conformance is determined by reference-suite thresholds (Section 9.9) rather than by proof of Pareto-optimality. When multiple Pareto-equivalent layouts exist, the configured `TieBreakingWeights` select among them. Tie-breaking is deterministic.

```
pub struct TieBreakingWeights {
  pub collision: f64,
  pub spacing: f64,
  pub slur_shape: f64,
  pub beam_slope: f64,
  pub vertical_density: f64,
  pub system_break: f64,
  pub page_fill: f64,
  pub casting_off: f64,
  pub symbol_density: f64,
}
```

REQUIREMENT

Tie-breaking weights **MUST** have normative defaults specified in the Quality Metric Catalog. Implementations **MAY** permit users to customize these weights; the defaults represent the reference engraving aesthetic and are the basis for reference-suite conformance.

9.6 Conformance Tiers

Solver conformance is graded into tiers, each declaring a subset of features the implementation supports. Tiers are *orthogonal* to file-format profiles (Chapter 8); a document may declare both an authoring file profile and a minimum solver tier, and the two axes are independent.

```
pub enum SolverTier {
    /// Minimal Layout Solver.
    Minimal,

    /// Standard Engraving Solver.
    Standard,

    /// Advanced / Extension-Aware Solver.
    Advanced,
}
```

9.6.1 Minimal Layout Solver

REQUIREMENT

A solver claiming the Minimal tier **MUST**:

- Satisfy every hard constraint or return `Unsatisfiable` (validity conformance).
- Be deterministic within its declared implementation version (Section 9.7).
- Produce well-formed `SolveReport` values with accurate `satisfied_hard_constraints`, `unsatisfied_constraints`, and metric vectors (diagnostic conformance).
- Support common-music-notation regions, metric time, and the standard constraint families.
- Pass the Minimal subset of the reference suite.

A Minimal solver **MAY** produce mediocre aesthetic quality; metric thresholds for the Minimal tier are relaxed relative to the Standard tier.

9.6.2 Standard Engraving Solver

REQUIREMENT

A solver claiming the Standard tier **MUST**:

- Satisfy every Minimal-tier obligation.
- Meet the Standard-tier metric thresholds on the reference suite for every quality metric.
- Support the incremental solving contract with at least Measure-local invalidation scope (Section 9.8).
- Support every Standard-tier constraint family declared in the Quality Metric Catalog.

The Standard tier corresponds to “professional engraving quality” for common-practice notation.

9.6.3 Advanced / Extension-Aware Solver

REQUIREMENT

A solver claiming the Advanced tier **MUST**:

- Satisfy every Standard-tier obligation.
- Support proportional and aleatoric region layout, not only metric regions.
- Support microtonal accidentals and their interactions with adjacent glyphs.
- Support extension-contributed constraint families and extension metric thresholds for each registered extension whose layout requirements are part of the Advanced reference suite.
- Support at least System-local invalidation scope under incremental solving.

9.6.4 Tier Declaration and Document Compatibility

REQUIREMENT

Documents **MAY** declare a minimum solver tier required for faithful engraving in their profile declarations (Chapter 8). A solver with tier lower than the document’s declared minimum **MAY** open the document but **SHOULD** surface a clear warning that engraving quality may not match the document’s intent.

Solver tier is independent of file-format profile. A document authored under the `Lite` file profile **MAY** declare a minimum solver tier of `Standard`; a document authored under the `Full` file profile **MAY** declare no minimum tier and be displayable by a `Minimal` solver.

9.7 Determinism

The solver’s determinism contract has two parts. *Within-implementation determinism* is a strict byte-equality obligation; *cross-implementation conformance* is reference-suite-based.

9.7.1 Within-Implementation Determinism

REQUIREMENT

A solver implementation at a fixed version **MUST** produce byte-for-byte identical `SolveReport` output for identical inputs, where “identical inputs” means:

- Identical `ConstrainedLayoutIR`.
- Identical `SolverConfig` (profile, budget, tie-breaking weights).
- Identical font and glyph metrics (referenced by version and content hash).
- Identical reduction algorithm version (Chapter 6).
- For incremental calls: identical prior `SolverState` and identical invalidation set.
- Identical solver implementation version (i.e., the same binary).

Determinism **MUST** hold across runs on the same machine, across cold and warm starts, and across hardware threads. Implementations **MUST NOT** use fast-math flags, processor- dependent SIMD without canonical-output guarantees, or any compiler options that vary floating-point results across builds on the same platform.

Where parallelism is used internally, the solver **MUST** serialize the merge step so that thread scheduling does not affect output. Deterministic budgets **MUST** be enforced from shared counters that capture all parallel contributions.

9.7.2 Cross-Implementation Conformance

REQUIREMENT

Cross-implementation determinism (byte-equality of outputs across different conforming solvers) is *not* required. Different conforming solvers **MAY** produce different layouts for the same input. Conformance across implementations is established by:

1. All hard constraints satisfied on every reference suite score.
2. The implementation is internally deterministic per the within-implementation rule above.
3. `SolveReport` values are well-formed and diagnostically accurate.
4. For each reference-suite score, every quality metric is within the tier’s declared threshold.

RATIONALE

Cross-implementation byte equality would freeze the ecosystem to whatever single algorithm first claimed conformance, and would preclude algorithmic innovation. Reference-suite thresholds give users a meaningful quality floor while permitting genuine imple-

mentation diversity. Within- implementation determinism remains essential for reproducible builds, regression testing, and predictable behavior under collaborative editing.

9.8 Incremental Solving

9.8.1 Invalidation Scopes

```
pub enum InvalidationScope {
    /// A single object's layout depends only on local context.
    ObjectLocal,
    /// Effects bounded within a single measure.
    MeasureLocal,
    /// Effects bounded within a single system.
    SystemLocal,
    /// Effects bounded within a single page.
    PageLocal,
    /// Effects bounded within a single region.
    RegionLocal,
    /// Effects may propagate across the entire score.
    WholeScore,
}

pub struct InvalidationSet {
    /// Declared scope. The solver MAY widen this scope
    /// conservatively; it MUST NOT narrow it.
    pub scope: InvalidationScope,

    /// Specific invalidated entries within the declared scope.
    pub slots: Vec<SpringSlotId>,
    pub bands: Vec<VerticalBandId>,
    pub constraints: Vec<ConstraintId>,
    pub glyphs: Vec<GlyphObjectId>,
}
```

9.8.2 Observational Equivalence

REQUIREMENT

For any invalidation set with declared scope S , `solve_incremental` **MUST** produce a layout observationally equivalent to `solve` called on the same input restricted to S , under the same profile, the same configuration, and the same deterministic budget. “Observational equivalence” means byte-identical `ResolvedLayoutIR` output for objects within S , and unchanged output for objects outside S .

The solver **MAY** widen the scope conservatively: an `ObjectLocal` invalidation may be solved as `MeasureLocal` if that simplifies the implementation, provided the widened result is also observationally equivalent to full solving at the widened scope.

The solver **MAYNOT** narrow the scope. If `SystemLocal` effects might propagate (e.g.,

an edit at the system's end shifts subsequent systems' contents), the invalidation **MUST** be declared at least `SystemLocal` or higher.

9.8.3 Propagation Documentation

REQUIREMENT

Implementations **MUST** document, per their published conformance contract, the maximum invalidation scope they can service for each constraint family. The documented bound is the implementation's contract for how far edit effects propagate before requiring whole-score re-solving.

9.9 Conformance: The Reference Suite

Solver conformance is established by performance on a normative reference suite of test scores. Each tier has its own subset of the suite and its own metric thresholds.

REQUIREMENT

The Reference Suite is delivered as a companion specification. Each suite entry consists of:

- A test score in canonical `.musc` form.
- Per-tier inclusion: whether the entry is required for Minimal, Standard, or Advanced conformance.
- Per-tier metric thresholds: the maximum permitted `NormalizedMetric` value for each quality metric.
- Optional fixed-expectation tests: specific layout properties (a particular bar's width, a specific system break location) that all conforming solvers **MUST** reproduce. Used sparingly, only where the suite intends to pin an unambiguous expectation.

A solver claiming a given tier **MUST** pass every entry required at that tier. Failure on any single entry is conformance failure at the claimed tier.

Suite versions are tied to specification versions; conforming implementations **MUST** declare which suite version they pass.

RATIONALE

The reference suite makes conformance checkable. The earlier universal-Pareto formulation quantified over all possible layouts of every possible score, which is undecidable. Reference-suite thresholds replace "no better layout exists" with "this layout is within tolerance on these specific scores," which is testable, falsifiable, and operationally useful. The cost is that conformance is only as good as the suite's coverage; that is acceptable, because suite coverage is visible, evolvable, and can be expanded as the spec matures.

9.10 Reference Algorithm

The reference algorithm is explanatory and diagnostic. Its purpose is to disambiguate constraint interactions where prose alone might leave a choice open, and to generate the expected outputs published with the reference suite. It is *not* the normative algorithm and *not* the aesthetic floor.

REQUIREMENT

Conforming implementations are *not* required to use the reference algorithm or to reproduce its layouts, except where a reference-suite entry explicitly fixes an expected behavior via a fixed-expectation test. Implementations **MAY** choose any algorithm satisfying the contract: hard constraints, determinism, diagnostics, reference-suite thresholds.

The reference algorithm is described in a non-normative companion document. A natural starting point uses the Cassowary linear- arithmetic constraint solver for the linear layer (positions, alignments, containment) together with a separate numerical optimization layer for aesthetic constraints (beam slopes, slur curvatures) that are nonlinear in nature. Hybrid approaches and alternative architectures (differentiable layout, constraint programming, learned methods) are all conforming choices.

9.11 Forward References

- End-to-end performance requirements, including frame budgets that the solver participates in, are in Chapter 10. Solver performance is measured in the context of the full layout pipeline.
- The Quality Metric Catalog companion document specifies the exact normalization functions for each metric, the default tie-breaking weights, and the per-tier thresholds.
- The Reference Suite companion document specifies the test scores, per-tier inclusion, and per-tier metric thresholds.
- The reference-algorithm companion document is non-normative and describes one starting-point implementation.

Performance Requirements

This chapter specifies performance contracts for the *core layer*. Targets are normative; they bound what conforming implementations promise and what tests measure for conformance. The chapter restates targets introduced informally in earlier chapters and adds the measurement methodology.

10.1 Core/Product Boundary

Chapter 1 excluded the editor UI, the audio engine, the renderer beyond the layout IR, and platform integrations from the core's scope. Many performance properties of a working notation system depend on those out-of-scope components: first interactive frame after launch involves UI bootstrapping, edit response involves UI event handling, scrolling and selection involve a real renderer, audio acquisition involves a working audio engine. The core cannot promise end-to-end timings for behavior it does not own.

This chapter therefore distinguishes three obligation classes:

Core obligations. The core data structures, layout IR, solver, and file format **MUST** admit the stated bounds. The core's own work (reduction, layout, file writes, snapshot acquisition) **MUST** fit within the budgets allocated to it in the end-to-end frame budget. The core **MUST** expose APIs suitable for each downstream component to do its own work inside its remaining budget (e.g., a lock-free immutable snapshot API for audio engines, an incremental invalidation interface for renderers, a cancellable solve API for UIs).

Product obligations. End-to-end UI, audio, and rendering targets (first interactive frame after launch, end-to-end edit-to-pixel latency, end-to-end scroll smoothness, audio-thread snapshot acquisition latency under load) are the obligation of the product layers built atop the core. Those targets are specified in product-specific companion documents, not here.

Performance Reference Suite obligations. The Performance Reference Suite (Appendix A) measures core behavior only: layout time per system, file write/read time, reduction throughput, solver budget consumption, snapshot acquisition latency. End-to-end UI traces are out of scope for the core's reference suite.

REQUIREMENT

Where this chapter states a performance target, the target is a *core obligation* unless explicitly labeled product obligation. Core obligations **MUST** be satisfiable by a conforming core implementation independent of the product layers built atop it; product-layer code is permitted to consume budget on top of the core's portion, but the core's own portion **MUST** fit within the stated bound.

10.2 Design Principles

Targets are part of the contract. Core performance requirements are not aspirations. A conforming core implementation **MUST** meet the stated targets on the reference hardware profile under the stated measurement conditions.

Frame budgets dominate. User experience is ultimately governed by frame timing. The core's portion of each frame budget is stated explicitly; the product layer's portion is specified separately.

Reference hardware profile. All targets are stated against a specified hardware profile. Implementations **MAY** document additional profiles (lower-end mobile, higher-end workstation); the reference profile is the basis of conformance.

Measurement methodology is normative. Without specified measurement methodology, performance targets are unenforceable. This chapter specifies how each target is measured.

Tail latency matters. Average performance is insufficient; targets are stated as percentiles (typically p99 or worst-case) to reflect what users actually experience.

10.3 Reference Hardware Profile

The reference hardware profile defines the platform against which performance conformance is measured. Implementations **MAY** target additional profiles; this profile is the basis of conformance.

Reference Hardware Profile (2026 baseline):

Desktop reference:

- Apple silicon (M-series) or x86-64 equivalent
- 8 performance cores (or equivalent throughput)
- 16 GB RAM
- NVMe SSD storage
- Discrete or integrated GPU with Metal/Vulkan support
- Display: 1440p at 60 Hz minimum

Tablet reference:

- iPad Pro M-series or equivalent
- 8 GB RAM minimum
- SSD storage
- Display: 120 Hz Pro Motion or equivalent

REQUIREMENT

Conforming implementations **MUST** meet the targets in this chapter on the reference hardware profile. Implementations **MAY** miss targets on hardware below the reference profile (e.g., older devices, low-end embedded targets); such gaps **MUST** be documented in the implementation’s published performance contract.

The reference profile is updated periodically as commodity hardware evolves. The version of the reference profile against which an implementation conforms **MUST** be declared.

10.4 Frame Budgets

10.4.1 Interactive Edit Latency

REQUIREMENT

The *core’s portion* of a single-system edit (operation envelope construction, reduction, incremental layout through `ResolvedLayoutIR`) **MUST** complete within one frame (16.7 ms at 60 Hz) at the p99 percentile, measured on the reference hardware profile. This includes the case where the affected system is on a 100-page orchestral score: the core’s incremental layout machinery (Chapter 7) and the constraint solver’s incremental contract (Chapter 9) jointly satisfy this requirement.

On 120 Hz displays (tablet reference), the core’s portion **SHOULD** complete within 8.3 ms at p99. This is not required for conformance but is the target for first-class tablet performance.

End-to-end edit-to-pixel latency (input handling, hit testing, render submission, display flip) is a product-layer obligation. The core’s contribution is bounded by this target so that the product layer has the remaining frame budget to do its work.

10.4.2 Multi-System Edits

REQUIREMENT

Edits affecting up to ten systems (e.g., a transposition over a substantial passage) **MUST** complete within four frames (66.7 ms at 60 Hz) at p99. The four-frame budget gives the editor headroom to display a transient “thinking” indication without violating perception of responsiveness.

10.4.3 Full Re-Layout

REQUIREMENT

Full re-layout of a 100-page orchestral score (triggered, for example, by a tuning system change applied score-wide) **SHOULD** complete within two seconds at p99. This is a soft target; the hard target on full re-layout is that the operation be cancellable: a user beginning a different edit **MUST** cause the full re-layout to abort and the new edit to proceed.

10.5 Cold Open

REQUIREMENT

The *core's portion* of cold open of a 100-page orchestral score from a `.musc` bundle **MUST** reach *first-system-ready* state within one second on the reference hardware profile. First-system-ready is defined as: the prelude has been read, the active superblock and manifest validated, the canonical base snapshot (if any) acquired, post-frontier envelopes reduced for at least the first system's coverage, and the first system's `ResolvedLayoutIR` produced. Rendering, hit-testing, and UI bootstrapping that build atop this state are product-layer obligations. Background loading of remaining systems **MAY** continue after first-system-ready and **MUST NOT** block user interaction served from already-loaded state. not-yet-visible systems is permitted; full-score loading need not complete in one second.

10.5.1 Cold Open Measurement

Cold open is measured from process start (or, for warm processes opening a new score, from the open API invocation) to the first frame containing fully rendered content for the user's initial viewport.

10.5.2 Streaming Read Discipline

REQUIREMENT

The cold open target **MUST** be achieved through streaming reads per Chapter 8: the implementation **MUST NOT** load the full score into memory before displaying the first system. Implementations failing to meet the target by eager loading are non-conforming.

10.6 Memory Budgets

REQUIREMENT

Steady-state resident memory for a 100-page orchestral score **SHOULD** remain below 500 MB on the reference hardware profile, exclusive of evictable caches. Evictable caches **MAY** push total resident memory higher; they **MUST** be evicted under memory pressure without compromising correctness.

10.6.1 Memory Categories

For measurement, memory is partitioned into:

Core data. The score graph, attachments, indexes, and operation envelope set. Cannot be evicted without losing state.

Layout cache. The cached IR stages. Evictable; rebuilds on demand but at performance cost.

Glyph and font caches. Glyph render data and metric caches. Evictable; small-to-moderate size.

Audio buffers and reference tracks. Audio data loaded for playback or reference. Evictable.

Implementation overhead. Runtime, allocator metadata, OS-level mappings.

REQUIREMENT

Implementations **MUST** expose the core-data memory consumption separately from evictable caches. Reporting tools **MUST** be able to distinguish them. The 500 MB target applies to core data only.

10.6.2 Per-Object Memory Discipline

REQUIREMENT

Implementations **MUST NOT** embed glyph metrics in IR objects (Chapter 7). Implementations **MUST NOT** embed spelling-attachment payloads on pitch objects (Chapter 2); attachments are externally indexed. These discipline requirements bound per-object memory at known upper limits.

10.7 Core Data Structures for Audio Thread Use

The audio engine is outside the core specification. The core's obligation is to expose data structures that permit real-time audio safety; audio-engine performance is the obligation of the product audio layer (Section 10.1).

REQUIREMENT

The core's read-only data structures consumed by an audio thread (pitch resolution, tempo map lookup, event arena queries) **MUST** be designed for allocation-free, lock-free access. Specifically:

- Pitch frequency resolution given a tuning system **MUST** be performable without allocation.
- Tempo-map conversion (musical to wall-clock and vice versa) **MUST** be performable without allocation.
- Time-indexed event lookup for playback **MUST** be performable without allocation.

These are properties of the core's data layouts, not of any particular audio engine. A product-layer audio engine that consumes them is responsible for its own real-time discipline (no syscalls on the audio thread, no priority inversion, no unbounded loops); the core only promises that its data structures do not preclude that discipline.

10.7.1 Snapshot API**REQUIREMENT**

The core **MUST** provide an immutable-snapshot API: the ability for any thread to acquire a read-only view of the score graph at a consistent point in time, suitable for use by an audio engine. Snapshot acquisition by the snapshot reader **MUST** be lock-free in the sense that it never blocks on edit-thread progress: an edit in flight **MUST NOT** prevent the snapshot reader from obtaining a valid (possibly older) snapshot.

Snapshot *release* **MAY** be deferred to a non-audio thread. The snapshot mechanism is implementation-defined (double-buffered immutable copies, persistent data structures, reference-counted immutable trees, hazard pointers); only the externally observable lock-free acquisition property is normative here.

End-to-end audio-thread acquisition latency under load is a product-layer obligation; the core's contribution is bounded by the lock-free guarantee.

10.8 Constraint-Solver Performance Targets**REQUIREMENT**

The constraint solver (Chapter 9) **MUST** satisfy:

- Incremental solving of a single-system edit completes within 10 ms at p99 on the reference hardware profile. (This is a tighter target than the overall frame budget; the remaining 6.7 ms accommodates the rest of the pipeline.)
- Cold solving of a 100-page orchestral score completes within two seconds at p99.
- Incremental solving propagation is bounded as documented in the implementation's published contract.

10.9 File Format Performance

REQUIREMENT

File format operations **MUST** satisfy:

- Bundle write of a typical edit (one or more operation envelopes appended to the operation-envelope block stream, manifest rewrite, superblock flip) completes within 50 ms at p99 on the reference hardware profile.
- Bundle read of the manifest and bootstrap chunks (sufficient for first interactive frame) completes within 200 ms at p99 for a 100-page orchestral score.
- Operation-envelope reduction rate **MUST** exceed 10,000 envelopes per second on the reference hardware profile, measured during cold reduction from a fresh canonical base.

10.10 Measurement Methodology

REQUIREMENT

Performance conformance **MUST** be measured using:

- A reference test corpus including the conformance suite from Chapter 9, augmented with edit traces representative of typical authoring sessions.
- Measurements at the p99 percentile across at least 1000 iterations per scenario.
- Warm runs after a cold warm-up phase, unless the target is explicitly cold (cold open, cold load).
- Release builds with full optimization enabled, but without fast-math or precision-compromising flags.
- The reference hardware profile, with no other significant load on the system during measurement.

Implementations **MUST** publish their measurement results against this methodology for conformance claims.

10.11 Regression Testing

REQUIREMENT

Conforming implementations **SHOULD** maintain continuous performance regression testing against the reference corpus, with alerts for regressions exceeding 10% on any target. The regression suite **SHOULD** be a precondition for releases.

10.12 Forward References

- The full performance reference corpus is delivered as a companion specification, the Performance Reference Suite.
- Audio engine performance targets (latency, jitter, stream discipline) are specified in the audio engine specification, outside this document.
- Network and synchronization performance for the collaboration layer is specified in the collaboration specification, outside this document.

Extension Points

This chapter consolidates the extension points specified across prior chapters into a single reference. Each extension point admits implementation-defined or plugin-contributed content that the core treats opaquely, subject to the contracts stated here.

The chapter is a navigation aid rather than a source of new requirements: every extension point referenced here is defined in its originating chapter, and the originating definition is authoritative. This chapter pulls the points together for plugin authors and future specification editors.

11.1 Design Principles

Extensibility without forking. Every extension point permits content to be added without modifying the core specification or its implementations. Plugins extend; they do not fork.

Identifiers, not strings. Extension points are identified by typed registry identifiers, not free-form strings. Registries are explicit, versioned, and discoverable.

Contracts, not contents. Extension points specify what an extension *must satisfy* (interfaces, invariants, reduction rules, performance bounds) rather than what an extension *must contain*.

Capability-bound at the runtime. Plugin runtime permissions (read score, write score, network, filesystem) are declared per extension and granted per user. This is enforced by the plugin runtime, which is outside this specification; the extension points are designed to admit such enforcement.

Forward compatibility. Extensions referenced in a stored score **MUST** be preserved across reads and writes even when the reading implementation does not understand them (per Chapter 8). Extensions thus survive distribution even where they cannot be exercised.

11.2 Notation Grammar Extensions

The notation grammar layer supports extension at several points:

PositionStructure::Registered (Chapter 4). Custom position structures for pitch spaces whose organization does not fit the chromatic, diatonic-over-chromatic, or JI lattice families. Used for maqam, gamelan, raga, and other non-Western grammars.

IntervalAlgebra::Registered (Chapter 4). Custom interval algebras for grammar-specific arithmetic.

TranspositionBehavior::Registered (Chapter 4). Custom transposition rules.

SpellingAlgorithmId (Chapter 2). Registered spelling-inference algorithms for the spelling pre-pass.

NominalRegistry (Chapter 4). Score-level or grammar-level extensions to the catalog of named position-letters.

AccidentalRegistry (Chapter 4). Score-level extensions to the accidental catalog, subject to the registry's `extensible` flag.

11.3 Tuning System Extensions

TuningFunctionId (Chapter 4). Registered procedural tuning functions (historical temperaments, custom meantone variants, generative tunings).

AdaptiveTuningFunctionId (Chapter 4). Registered adaptive tuning functions taking harmonic context as input.

TuningFileFormat (Chapter 4). Recognized external tuning file formats (Scala, MIDI Tuning Standard, additional formats via registry).

CompatibilityMapping (Chapter 4). Declared compatibility between pitch spaces and tuning systems whose underlying spaces differ.

11.4 Glyph and Font Extensions

CustomGlyphId (Chapter 4). Custom glyphs not in SMuFL, defined per score or per plugin.

Composite glyph references (Chapter 4). Composite glyphs constructed from multiple primary glyph references.

ModificationRegistryId (Chapter 4). Registered pitch-space modifications for grammar-defined accidentals.

11.5 Score Graph Extensions

ExtensionOp (Chapter 6). Implementation-defined operation kinds extending the core's semantic operations. Extension operations **MUST** satisfy the operation framework's requirements (preconditions, effects, inverse, reduction rule, re-anchoring).

CustomMarkerId (Chapter 5). Custom marker kinds beyond the core's standard catalog.

CustomAnalyticalKindId (Chapter 5). Custom analytical annotation kinds.

Custom event variants (Chapter 5). The event taxonomy is closed in the core, but the `Group` layout object and the `Registered` fields throughout permit composition of core types into extended structures.

11.6 Layout IR Extensions

SynthesisRegistryId (Chapter 7). Custom synthesis kinds for engraver-generated objects.

ConstraintRegistryId (Chapter 7). Registered layout constraint kinds extending the standard constraint families.

MappingRegistryId (Chapter 5). Custom parameter mappings for graphic-object time bindings.

Custom view kinds (Chapter 5). The `ViewKind::Custom` variant admits user-defined view kinds.

11.7 Constraint-Solver Extensions

LayoutConstraint::Registered (Chapter 7). Custom constraint families consumed by the solver. Extension constraints **MUST** declare their strength. When soft, extension constraints contribute to the quality vector through an extension metric subject to the `NormalizedMetric` discipline (Chapter 9) and to the tier-specific reference- suite thresholds.

11.8 File Format Extensions

FormatProfile::Custom (Chapter 8). Profile definitions beyond Full, ReadOnly, and Lite.

Self-describing extension envelope (Chapter 8). Plugin-contributed and forward-compatible content is carried in self-describing extension records that the core preserves opaquely.

11.9 Extension Registry Contract

Every extension registry shares a common structural contract.

```
pub trait ExtensionRegistry {
    type Id;
    type Definition;

    /// Resolve an identifier to its definition.
    fn resolve(&self, id: Self::Id) -> Option<&Self::Definition>;

    /// Enumerate all registered definitions.
    fn enumerate(&self) -> Vec<&Self::Definition>;

    /// Version of the registry. Score files declare which version
    /// their references target.
    fn version(&self) -> RegistryVersion;
}
```

REQUIREMENT

Every extension registry **MUST** be versioned. Stored references to registered definitions **MUST** include the registry version they target. Loading a score with references to registry versions newer than the reader understands **MUST** preserve those references per the forward-compatibility rules of Chapter 8.

11.9.1 Identifier Stability**REQUIREMENT**

Registry identifiers **MUST** be stable across registry versions within a major version. Renumbering, reassignment, or repurposing of identifiers is forbidden within a major version. Removal of a definition **MUST** reserve its identifier; the identifier **MUST NOT** be reassigned to a different definition.

11.9.2 Discoverability

Registries **MUST** support enumeration: an implementation can list all known definitions and their identifiers. This enables tooling (plugin browsers, score migration assistants, conformance testers) to operate against unknown registries by inspection.

11.10 Plugin Runtime Considerations

The plugin runtime is outside this specification, but extension points are designed to admit a capability-bound runtime as described in the feature compendium:

- Extensions are isolatable: an extension's failure does not crash the host.
- Extensions are introspectable: their declared capabilities and inputs are inspectable before execution.
- Extensions are revocable: an extension can be disabled without invalidating the scores that reference it.

REQUIREMENT

Extension points **MUST** be designed such that disabling an extension does not invalidate stored content referencing it. The content remains in the score, opaque to the host, until the extension is re-enabled or the references are explicitly removed by the user.

11.11 Conformance and Extensions

Extensions do not affect core conformance directly, but they interact with it:

REQUIREMENT

A conforming implementation:

- **MUST** support every extension point as a structural slot (i.e., scores containing extension references are readable and writable).
- **MUST NOT** require any specific extension to be installed for core conformance.
- **MAY** require specific extensions for specific features (e.g., a particular plugin for a particular grammar).
- **MUST** clearly distinguish between core conformance and extension-dependent features in its documentation.

11.12 Forward References

- The plugin runtime (WebAssembly-based, capability-bound) is specified in the Plugin Runtime specification, outside this document.
- The complete extension registry catalog is delivered as a companion document.
- Implementation-defined extension contributions are governed by the implementation's published extension specification.

Intentionally Deferred Types and Specifications

This appendix catalogs types, concepts, and specifications that the core specification refers to but does not fully define. Each entry is *deliberately external*: it is the responsibility of a named companion specification or a future revision of this document. The purpose of this appendix is to distinguish deliberately external content from accidentally missing content, so that reviewers and implementers can quickly tell which unresolved references represent open work and which represent boundaries to other documents.

A.1 Companion Specifications

The following are companion specifications referenced throughout the core. None has been delivered as of this revision; each is named here with the chapter(s) that depend on it.

Binary Format companion. Byte-level encoding of all canonical chunks: varint conventions, exact field widths, record layouts, endianness rules, string encoding details. Referenced from Chapter 8. The on-disk representations specified in that chapter are structural; the Binary Format specifies how the bits are laid out.

Text Projection companion. The canonical s-expression text projection of the bundle: grammar, deterministic serialization rules, base64 encoding for opaque payloads, handling of extension data, round-trip semantics with binary form. Referenced from Chapter 8 Section 8.13.

Profile Conformance specification. The complete catalog of file-format profiles (Full, Read-Only, Lite, plus any normative custom profiles), with their constraints, permitted compression algorithms, maximum block sizes, required extensions, and conformance test discipline. Referenced from Chapter 8 Section 8.15.

Quality Metric Catalog. Per-metric normalization functions mapping raw measurements to `NormalizedMetric` values, default tie-breaking weights, per-tier metric thresholds, and the formal definition of each quality metric in the normative metric set. Referenced from Chapter 9.

Reference Suite. The collection of test scores against which solver conformance is established, with per-tier inclusion, per-tier metric thresholds, and any fixed-expectation tests. Versioned with this specification. Referenced from Chapter 9 Section 9.9.

Performance Reference Suite. The collection of edit traces and reference scores against which performance conformance is established. Referenced from Chapter 10.

Operation Catalog companion. The complete list of primitive operations and compound operations supported by the core, each with full preconditions, effects, inverse, and reduction rule. The representative operations in Chapter 6 illustrate the contract; the full catalog is delivered separately.

Reference Algorithm companion. A non-normative description of one starting-point implementation of the constraint solver. Explanatory and diagnostic only; conforming implementations are not required to use it. Referenced from Chapter 9 Section 9.10.

A.2 Externally Provided Components

The following are real components on which the core depends but which are owned by other specifications outside the Epiphany core scope.

SMuFL glyph catalog. The Standard Music Font Layout provides the normative glyph repertoire. Referenced from Chapter 4 and the layout chapters. SMuFL is maintained by the W3C Music Notation Community Group; the core spec declares its dependency on the SMuFL version targeted but does not redefine SMuFL.

Font metric provider. Implementations resolve glyph geometric data (bounding boxes, anchor points, advance widths) through a font metric provider. The provider's interface is part of the implementation contract; specific font metrics are external content.

Audio engine specification. Out of scope here per Chapter 1. The core exposes data structures designed for allocation-free, lock-free audio-thread access (Chapter 10); the audio engine's internals, latency obligations, and stream discipline are specified separately.

Plugin runtime specification. A capability-bound, sandboxed runtime hosts extension code. The runtime is out of scope here; the extension points (Chapter 11) are designed to admit such a runtime.

Collaboration transport. The wire protocol by which operation envelopes are exchanged between replicas is out of scope here. Chapter 6 specifies what must be exchanged and how it is reduced; the transport layer is separate.

User interface and editor. Editor commands, selection semantics, keyboard shortcuts, gesture grammar, and UI-level interaction patterns are out of scope. The core exposes a programmatic operation API; UI mapping is layered above.

A.3 Open Canonical Algorithms

The following algorithms affect canonical state but are not yet specified normatively, profile-declared, or marked non-canonical. Per Appendix D Section D.11, each **MUST** receive one of those dispositions before the document can claim full canonical score determinism across implementations.

Spelling pre-pass algorithm. The algorithm that computes default spellings from key signature, harmonic context, and notation grammar. Referenced from Chapter 2. Several established algorithms exist (Longuet-Higgins, Temperley pitch-spelling preference rules,

and others); the choice of normative algorithm or profile-declared algorithm set is open.

Notational decomposition algorithm. The algorithm that breaks a sounding duration into notehead values, augmentation dots, and ties under metric context. Referenced from Chapter 3. Conservatoire engraving manuals describe the conventions verbosely; the formalization is open.

Tempo curve integration. The numerical integration algorithm for converting musical position to wall-clock position under `TempoShape::Curve`. Referenced from Chapter 3. Possible candidates include Romberg integration and adaptive Gauss-Kronrod; the normative choice is open.

Wallclock-to-musical root-finding. The root-finding algorithm for the inverse of tempo curve integration. Referenced from Chapter 3. Possible candidates include bisection with monotonicity guarantees and Brent’s method; the normative choice is open.

A.4 Extension Registry Catalogs

The extension registries enumerated in Chapter 11 (notation grammar extensions, tuning system extensions, glyph and font extensions, score graph extensions, layout IR extensions, solver extensions, file format extensions) are referenced by typed registry identifiers in the core. The concrete catalogs of registered extensions (the contents of each registry) are delivered as separate, versioned registry documents. Each registry document specifies its identifier numbering, the version under which each registered identifier was added, and the registered content’s contract.

A.5 Support-Type Identity Semantics

Several identifier-like support types appear throughout the specification by name and shape, but their detailed encoding is delivered by companion specifications. Their semantic identity rules are normative here.

Type	Source	Semantic rule
<code>FileUuid</code>	128-bit random at bundle creation	Physical bundle identity. Fixed for the lifetime of the physical bundle; preserved by filesystem byte-copy; freshly minted on Save As, export-as-new-bundle, and derivative-work creation.
<code>DocumentId</code>	128-bit random at logical-work creation	Logical work identity. Stable across Save As copies intended to remain versions of the same work. May persist across forks at user discretion.
<code>LineageId</code>	128-bit random at fork-with-ancestry	Shared-ancestor identity. Optional. Multiple documents may share a <code>LineageId</code> declaring common ancestry for version-control or genealogy purposes.

Type	Source	Semantic rule
ManifestId	Content-derived from manifest payload	Unique per manifest version. Implementations MAY use <code>trunc128(BLAKE3 of canonical manifest preimage)</code> with domain tag <code>"MUSCMNIF"</code> ; the Binary Format companion defines the exact derivation.
SnapshotId	Content-derived from snapshot payload and frontier	Stable for a snapshot's root and causal frontier. Two snapshots with identical materialized state at identical frontiers have identical ids. The Binary Format companion defines the exact derivation.
BlobId	BLAKE3 of blob content with domain tag <code>"MUSCBLOB"</code>	Content hash of the blob's uncompressed payload, identical to the blob's <code>ContentHash</code> .
AuthorId	Implementation-defined; persists across replica reseatings	Identifies the human author of operations, distinct from <code>ReplicaId</code> (which identifies the device or session). One author may operate multiple replicas; one replica may host one author at a time. The collaboration transport companion defines authentication.
ReductionAlgorithmVersion	Semantic version	Selects canonical reduction semantics. Two replicas with different <code>ReductionAlgorithmVersion</code> may produce different canonical states from the same operation set; the active superblock declares the version under which the bundle's canonical base was materialized (Section 8.3).
SolverProfile	Registered identifier	profile Selects the solver's hard-constraint set, normalized-metric thresholds, tie-breaking weights, and active extension catalog (Chapter 9). The Quality Metric Catalog companion defines the registered profile catalog.
ProfileId (file format)	Registered identifier	profile Selects the file-format conformance subset (Full, ReadOnly, Lite, or registered Custom). The Profile Conformance companion defines the registered profile catalog.
ContentHash	BLAKE3-256 output (32 bytes)	Cryptographic content hash. The canonical hash for all content-addressed objects in the bundle.

Type	Source	Semantic rule
ChunkId	Newtype around ContentHash	Same 32 bytes as the underlying ContentHash; the type distinction makes role visible at use sites.
OperationId	64-bit ReplicaId + 64-bit counter, both randomly initialized; counter monotonic	Stable identity of an operation, set by the authoring replica at the moment of authoring. Never reassigned.
TransactionId	128-bit, author-minted	Identifies a transaction declared by a DeclareTransaction envelope. Member envelopes reference this id in their envelope transaction field.
ConflictId	Content-derived via BLAKE3 truncation with domain tag "MUSCCONF"	Identifier of a conflict record. Derived from canonical kind, sorted causing operations, sorted affected objects. Section 6.5.3.
Typed identifiers (EventId, PitchId, VoiceId, StaffId, etc.)	64-bit ReplicaId + 64-bit counter; or ReplicaId::SYSTEM_DERIVED + BLAKE3-derived counter for system-derived identifiers	Stable identity within their typed namespace. Section 5.3.2.

REQUIREMENT

Implementations **MUST** respect the semantic identity rules declared above. They **MAY** freely choose the bit-level encoding of each type (the Binary Format companion specifies the canonical wire encoding); they **MAYNOT** reinterpret the identity semantics. In particular, two distinct works **MUST NOT** share a `DocumentId`; two byte-distinct chunks **MUST NOT** share a `ContentHash`; two distinct operations **MUST NOT** share an `OperationId`.

A.6 Distinction from Open Questions

This appendix lists content that is *deliberately external* to the core specification: it is owned by a companion document, a future revision, or a third-party specification.

It does *not* list “open questions” in the core’s own design space. Those are marked with the *Open Question* callout where they appear in the body of the text, and they represent unresolved decisions *within* the core specification’s scope. The two categories overlap only at the four open canonical algorithms in Section A.3, which are unresolved in the core text *and* whose resolution will likely take the form of a companion specification or a profile declaration.

Glossary

This glossary defines key terms used throughout the specification. Definitions are drawn from their introducing chapters; the chapter reference points to the authoritative definition.

- Acoustic pitch** (Chapter 2). The frequency realization of a pitch under an active tuning system. One of the two intrinsic layers of a `Pitch`.
- Adaptive tuning** (Chapter 4). A tuning system whose frequency resolution depends on harmonic context, not on pitch-space position alone.
- Aleatoric time** (Chapter 3). A region time model in which events have partial or undefined ordering, expressed as a directed acyclic graph with optional interval bounds.
- Anchor** (Chapter 3). A reference to a point in time, expressed relative to an identified object (event, measure, region, wall-clock time) plus an offset. Anchors survive edits that do not delete their target.
- Arena (event arena)** (Chapter 5). The flat storage owned by a score holding all events of all kinds, with $O(1)$ lookup by `EventId`.
- Bundle** (Chapter 8). The on-disk container holding a score: header, manifest, chunk store, and blob store in a single `.musc` file.
- Canonical reduction** (Chapter 6). The deterministic procedure that materializes a score graph from the operation set: sort by canonical order, group transactions, apply each operation against working state, record effects.
- Canvas** (Chapter 5). The spatial root of a score: a 2D space partitioned into regions of staff-based, free-graphic, or hybrid content.
- Causal predecessors** (Chapter 6). The set of operations on which a given operation causally depends. Used by the CRDT layer to determine concurrency. Expressed in canonical form as a dotted version vector (`CausalContext`); see “Dotted version vector.”
- Chunk** (Chapter 8). An immutable content-addressed unit of storage within a bundle. Identified by the cryptographic hash of its uncompressed payload, with a domain-separated preimage.
- Compound operation** (Chapter 6). A user-facing operation that expands into a sequence of primitive operations applied as a transaction.
- Conflict record** (Chapter 6). A first-class graph object recording an operation that could not apply cleanly under canonical reduction. Stable, addressable, user-visible, and resolv-

able by subsequent `ResolveConflict` operations.

CRDT (Chapters 6, 8). Conflict-free replicated data type. In Epiphany, the *operation set* is a true CRDT (a grow-only set of operation envelopes); the materialized score graph is *not* itself a CRDT but is the deterministic reduction of that set.

Cross-cutting structure (Chapter 5). A first-class object spanning multiple tree-nodes via references: slurs, ties, beams, spanners, markers, repeats, analytical annotations, comments, graphic gestures, lyrics, chord symbols.

Decomposition (Chapter 3). The notational breakdown of an event’s sounding duration into notehead values, augmentation dots, and ties. Attached externally by the decomposition pre-pass.

Delta (Chapter 6). The structural record of an operation’s effect, sufficient to reconstruct the pre-state of every object the operation modifies.

Document identifier (Chapter 8). The logical work identity (`DocumentId`), stable across Save As copies that are intended to remain versions of the same work. May persist across forks at the user’s discretion. Distinct from `FileUuid` (physical bundle identity, changes on Save As) and `LineageId` (optional shared-ancestor identity).

Dotted version vector (Chapters 6, 8). The compact causal-context representation of operation history. Operational semantics (causal precedence, frontier coverage, monotonic growth) are specified in Chapter 6; the on-disk serialization is specified in Chapter 8.

Edit barrier (Chapter 8). A structured declaration by an extension that certain edits would invalidate the extension’s data. Implementations not understanding the extension refuse matching edits unless the user performs an explicit unsafe edit.

Engraving decision (Chapter 7). A choice made by the engraver (stem direction, beam consolidation, etc.) recorded explicitly in the layout IR for inspection and override.

Engraving override (Chapter 7). A user assertion that the engraver’s default decision should be replaced. Authoritative in the score graph; projected into the layout IR.

Event (Chapter 5). A rhythmic atom of the score: pitched, unpitched, rest, indeterminate, trajectory, graphic, or cue. Owned by exactly one voice.

Format profile (Chapter 8). A named subset of the file format that conforming implementations promise to support. Profiles enumerated: Full, ReadOnly, Lite, Custom.

Frame budget (Chapter 10). The time allowed for a layout-and-render operation to maintain 60 Hz (16.7 ms) or 120 Hz (8.3 ms) interactivity.

Glyph catalog identity (Chapter 7). The identifier (SMuFL version, font id, optional font version, metrics hash) under which layout output was produced. Required for any byte-equal-output layout conformance claim.

Graphic content (Chapter 5). Free-form vector graphics, drawn strokes, text, and images within free-graphic or hybrid regions.

Hard constraint (Chapter 9). A constraint that must be satisfied; violation produces a solver error.

Hybrid logical clock (Chapters 6, 8). A clock combining physical wall-clock time with a logical counter, used as a tie-breaker in canonical reduction order. Operational semantics

(per-replica monotonicity, role in canonical reduction ordering, malformed-envelope behavior) are specified in Chapter 6; the on-disk serialization is specified in Chapter 8.

Identifier (typed) (Chapter 5). A 128-bit stable identifier for a named object: `EventId`, `VoiceId`, `StaffId`, etc. Generated by replica-plus-counter scheme.

Incremental layout (Chapter 7). Re-layout of only the IR objects whose dependencies have changed, with cached values reused elsewhere.

Instrument (Chapter 5). An abstract definition of a sounding entity: name, transposition, range, default sound, default clef, default staff count.

Layout IR (Chapter 7). The pipeline of intermediate representations between score graph and renderer: `LogicalLayoutIR`, `ConstrainedLayoutIR`, `ResolvedLayoutIR`, `RenderIR`.

Manifest (Chapter 8). The mutable index of a bundle: a table of operation roots, optional canonical base snapshot, acceleration snapshots, blob roots, profile declarations, and extension declarations. Itself a content-addressed chunk; the active manifest is selected via the active superbloc slot (Chapter 8).

Metric time (Chapter 3). A region time model with measures, beats, and exact rational positions. The dominant model for Western notated music.

Notation grammar (Chapter 4). The declarative definition of a notation system's pitch space, intervals, accidentals, and spelling rules. CMN is the default; others extend.

Operation envelope (Chapter 6). A transmitted, stored, and reducible record carrying an operation's payload together with its identity, ordering stamp, causal context, and optional transaction grouping. The unit of replication.

Operation kind (Chapter 6). The tagged enumeration of primitive operation variants: `InsertEvent`, `DeleteEvent`, `RespellPitch`, `Transpose`, `ChangeRegionTimeModel`, and so on, plus `Registered` for extension-defined primitives. The full catalog is delivered separately.

Operation log (Chapter 8). The physical storage of the canonical operation-envelope set on disk, as operation-envelope blocks. The canonical document is the deterministic reduction of this envelope set, or, after pruning, of the manifest's `canonical_base` snapshot plus envelopes whose `OperationIds` are not covered by the base's causal frontier.

Operation set (Chapter 6). The replicated grow-only CRDT of operation envelopes. The canonical document is its deterministic reduction.

Pareto frontier (design target) (Chapter 9). The set of layouts for which no quality metric can be improved without worsening another. Solvers **SHOULD** seek layouts near the frontier; conformance is reference-suite-based, not proof of Pareto optimality.

Part (Chapter 5). A per-instrument or per-section view onto the score for printed extraction. Parts are projections, not storage.

Pitch space (Chapter 4). The analytical universe of a pitch system: what positions exist, how they relate, what accidentals are valid. Distinct from tuning system.

Pitch spelling (Chapter 2). The notational appearance of a pitch: nominal, accidental, octave. Attached externally by the spelling pre-pass.

Primitive operation (Chapter 6). The atomic unit of editing. Primitive operations compose

into compound operations and transactions.

Provenance (Chapter 7). The record on each layout IR object tracing back to its score graph source, enabling selection, editing back-references, and incremental layout invalidation.

Proportional time (Chapter 3). A region time model in which horizontal position is wall-clock time, with no measures or beats.

Re-anchoring (Chapter 6). The normative response of a cross-cutting structure when its referent is deleted: cascade delete, truncate, re-anchor, mark orphaned, or refuse.

Reference pitch (Chapter 4). The anchor converting tuning system structure to absolute frequencies: a position and a frequency in Hertz. A property of the score, not the tuning system.

Region (Chapter 5). A spatial-temporal container in the canvas declaring a time model (metric, proportional, aleatoric) and a content model (staff-based, free-graphic, hybrid).

Repair record (Chapter 6). A deterministic compensating change made during reduction to preserve graph invariants (re-anchoring, spanner truncation, attachment tombstoning, voice promotion, tuplet compensation). Recorded as part of canonical state via `OperationEffect::AppliedWithRepair`.

Replica (Chapter 5, 8). An instance of a score on a device. Each replica has a unique identifier and authors operations with monotonic local sequence numbers. The replica identifier `ReplicaId::SYSTEM_DERIVED` is reserved for deterministically-derived system identifiers.

Retention policy (Chapter 8). The declared rule governing how many old manifests (and their reachable chunks) are preserved in the bundle for rollback purposes. Rollback is achieved through retained manifests in the chunk store, not by retaining multiple on-disk superblock generations.

Scale position (Chapter 2). The analytical identity of a pitch within a pitch space. One of the two intrinsic layers of a `Pitch`.

Score graph (Chapter 5). The in-memory representation of all musical content in a score: canonical truth from which layout, serialization, and editing operations derive.

Semantic operation (Chapter 6). A mutation of the score graph: insert event, delete measure, transpose, respell pitch, etc.

SMuFL (Chapter 4). Standard Music Font Layout. The normative source for glyph references in the core.

Snapshot (Chapter 8). A materialized score state. As an acceleration snapshot (listed in `acceleration_snapshots`), derivative and discardable. As a canonical-base snapshot (named in the manifest's `canonical_base` field with declared causal frontier and reduction algorithm version), the canonical base for pruned documents, in which case the canonical state is the base snapshot plus envelopes whose `OperationIds` are not covered by the base's causal frontier.

Soft constraint (Chapter 9). A constraint with a weight; violation is permitted but penalized in quality metrics.

Sounding duration (Chapter 3). The exact rational duration an event occupies in time. In-

trinsic; notational decomposition is attached separately.

Spring slot (Chapter 7). A time slot's horizontal spacing parameters (min, preferred, max widths; stretch, compress factors) consumed by the constraint solver.

Staff (Chapter 5). The global, abstract staff identity persisting across the entire score. A staff is declared once at the score level and referenced by staff instances in any region where it appears.

Staff instance (Chapter 5). The region-local manifestation of a staff: the voices, measures, clef changes, key changes, and metric grid that belong to that staff for one region. A single Staff may have multiple StaffInstances across the score.

Tempo map (Chapter 3). The function mapping musical positions to wall-clock positions, piecewise over musical time with constant, linear, exponential, or curve segments.

Time anchor (Chapter 3). See "Anchor."

Time axis (Chapter 7). The polymorphic layout-IR component projecting time positions to horizontal positions. Three implementations: metric, proportional, aleatoric.

Transaction (Chapter 6). A replicated operation group that reduces atomically: either every member applies and the transaction reduces to `Applied`, or the entire transaction reduces to `Conflicted` with a recorded conflict.

Trajectory (Chapter 5). A continuous pitch-motion event: glissando, portamento, pitch bend. A sibling to pitched events under the event taxonomy.

Tuning system (Chapter 4). The mechanism mapping pitch-space positions to frequencies, given a reference pitch. Distinct from pitch space.

Tuplet (Chapter 3). A grouping object annotating irregular rhythmic groupings. Does not modify sounding durations; purely notational and analytical.

Typed object identifier (Chapter 5). A tagged union (`TypedObjectId`) over every identifier kind in the score graph. Used wherever an object is referenced generically (cross-cutting endpoints, conflict `affected_objects`, repair records, edit barriers).

Validation mode (Chapter 6). The strictness setting for operation precondition checking: strict authoring or lenient replay.

Vertical band (Chapter 7). A horizontal slice of the canvas with spring parameters for vertical spacing: staff bands, inter-staff gaps, inter-system gaps, margin bands.

View (Chapter 5). A recipe for displaying a configuration of the score: which layers are active, which parts are rendered, what overrides apply.

Voice (Chapter 5). A polyphonic line within a staff. Owns a sequence of events.

Wall-clock time (Chapter 3). Time measured by the playback engine or external sync source, in fixed-point nanoseconds. Distinct from musical time.

Bibliography and References

Engraving Treatises

- Behind Bars.** Elaine Gould. *Behind Bars: The Definitive Guide to Music Notation*. Faber Music, 2011. The principal modern reference for music engraving conventions.
- Music Notation in the Twentieth Century.** Kurt Stone. *Music Notation in the Twentieth Century: A Practical Guidebook*. W. W. Norton, 1980. Authoritative reference for contemporary notation.
- Essentials of Music Notation.** Tom Ross. *The Art of Music Engraving and Processing*. Hansen Books, 1970. Classical treatise on engraving conventions, still influential.
- Music Notation.** Gardner Read. *Music Notation: A Manual of Modern Practice*. Crescendo Publishers, 1969. Reference for engraving practices and history.

Music Theory and Analysis

- The Cognition of Basic Musical Structures.** David Temperley. MIT Press, 2001. Source of preference-rule approaches to spelling and meter analysis.
- Pitch Spelling and the Line of Fifths.** H. Christopher Longuet-Higgins. Foundational work on pitch-spelling algorithms.
- Tonal Pitch Space.** Fred Lerdahl. Oxford University Press, 2001. Pitch-space theory as analytical framework.
- A Generative Theory of Tonal Music.** Fred Lerdahl and Ray Jackendoff. MIT Press, 1983. Foundational work in formal music theory and the source of preference-rule methodology.

Tuning, Microtonality, and Just Intonation

- Genesis of a Music.** Harry Partch. University of Wisconsin Press, 2nd ed. 1974. Foundational text in extended just intonation and microtonal practice.
- Tuning, Timbre, Spectrum, Scale.** William Sethares. Springer, 2nd ed. 2004. Modern treatment of tuning systems and their psychoacoustic foundations.

Helmholtz-Ellis JI Pitch Notation. Marc Sabat and Wolfgang von Schweinitz. The HEJI accidental system referenced by the core’s built-in JI pitch spaces.

Sagittal: A Microtonal Notation System. George D. Secor and David C. Keenan. The Sagittal microtonal accidental system.

Constraint-Based Layout

Cassowary Linear-Arithmetic Constraint Solver. Greg J. Badros, Alan Borning, and Peter J. Stuckey. *The Cassowary Linear Arithmetic Constraint Solving Algorithm*. ACM Transactions on Computer-Human Interaction, 8(4):267-306, 2001. Source of the constraint-solver model referenced in Chapter 9.

Music Engraving with Constraint Programming. Multiple authors. Survey of constraint-based approaches to music engraving.

The Gourlay Algorithm. J. S. Gourlay. *A Language for Music Printing*. Communications of the ACM, 1986. Source of the proportional spacing model referenced in Chapter 7.

CRDT and Distributed Systems

A Comprehensive Study of Convergent and Commutative Replicated Data Types. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. INRIA Technical Report RR-7506, 2011. Foundational survey of CRDTs.

Dotted Version Vectors. Nuno Preguiça et al. Refinement of vector clocks used in the file format (Chapter 8).

Hybrid Logical Clocks. Sandeep S. Kulkarni et al. Clocks combining physical and logical time, used for total ordering of operation envelopes (Chapter 6).

Prior-Art Notation Formats

MusicXML. Recordare LLC, later W3C Music Notation Community Group. The dominant interchange format; surveyed for the interchange profile work referenced in Chapter 5.

MEI (Music Encoding Initiative). An XML-based encoding designed for scholarly editions and musicology. Influenced the spec’s handling of editorial apparatus and analytical layers.

MNX. W3C Music Notation Community Group draft. The successor effort to MusicXML; informed several decisions on layered data models.

LilyPond. David Bauer, Han-Wen Nienhuys, Jan Nieuwenhuizen. Open-source music engraving software using a text-based input format and a sophisticated rule-based engraver. Significant influence on the spec’s engraving philosophy.

SMuFL. W3C Music Notation Community Group. The Standard Music Font Layout, referenced as the normative glyph catalog in Chapter 4.

Additional References

The Anatomy of a Note. Multiple papers on the data representation of musical notation in computer systems.

Notation in Computer Music. Various proceedings of the International Computer Music Conference (ICMC) and the New Interfaces for Musical Expression (NIME) workshops.

Determinism Contract

This appendix consolidates the reproducibility obligations that appear throughout the specification. It defines exactly which values are admissible in canonical state, how they are serialized, how comparisons work, and where implementation freedom begins.

The appendix is the document’s legal code for reproducibility: where the body of the specification says “deterministic,” this appendix says what that means.

D.1 Thesis

REQUIREMENT

Canonical document state **MUST** be independent of platform, CPU, locale, thread scheduling, hash-map iteration order, floating-point environment, compression settings, and wall-clock timing.

Where exact equality is required, this specification defines canonical representations. Where numerical approximation is unavoidable, this specification defines deterministic tolerances, quantization, and reporting rules.

Implementations **MAY** freely choose internal data structures, algorithms, and parallelism strategies. The canonical externally observable result **MUST NOT** depend on those choices.

D.2 Layers of Determinism

The specification distinguishes five layers. Each layer has its own determinism contract; conflating them is the most common source of impossible requirements.

Canonical score determinism (Chapter 6). The reduction of an operation-envelope set to a materialized score graph is fully deterministic *when every algorithm contributing to that reduction has received one of the dispositions in Section D.11* (normatively specified, profile-declared by versioned identifier, or explicitly non-canonical). Under that condition, identical operation sets produce identical materialized score states across all conforming implementations, with identical object identifiers, tombstones, conflicts, and effects. For algorithms still in open-question state (Section D.11), this byte-equal cross-implemen-

tation guarantee **MUST NOT** be claimed; the output of such algorithms is either non-canonical or governed by the profile’s declared algorithm version.

Canonical serialization determinism (Chapter 8). The bundle format’s chunk hashes, manifest encoding, text projection, and the ordering of canonical maps and sets are deterministic. Two implementations producing the same canonical state **MUST** produce identical chunk content hashes and identical text-projection bytes.

Layout determinism (Chapter 9). Within a single solver implementation at a fixed version, with fixed profile, fixed budget, and fixed font/glyph metrics, layout output is byte-identical across runs. Across different solver implementations, layout output **MAY** differ aesthetically within reference-suite thresholds.

Conformance determinism . Reference-suite-based: conforming implementations need not produce byte-identical output to each other; they need to fall within declared metric thresholds on every reference-suite entry. See Chapter 9 Section 9.9.

Non-canonical cache determinism . Implementations maintain caches (operation index, layout cache, glyph metric cache, render artifact cache). These **MAY** vary across runs, platforms, and implementations. They **MUST NOT** affect canonical state and **MUST** be discardable (Chapter 6 Section 6.3.5).

REQUIREMENT

Byte-equality obligations **MUST** be applied only to the layer for which they are stated. Implementations **MUST NOT** interpret “deterministic” more strongly than its layer specifies, and **MUST NOT** interpret it more weakly either.

RATIONALE

The conditional in canonical score determinism is essential. Several algorithms affecting canonical state remain open questions in this draft: the spelling pre-pass (Chapter 2), the notational decomposition algorithm (Chapter 3), and the `wallclock_to_musical` root-finding algorithm (Chapter 3). Until those algorithms receive Section D.11 dispositions, cross-implementation byte equality cannot be promised for outputs derived from them. This is not a weakening of the determinism contract; it is a precise statement of what the contract currently covers. Once the open algorithms are specified or profile-declared, the conditional resolves and full cross-implementation byte equality applies to all of canonical state.

D.3 Floating-Point Values in Canonical State

D.3.1 Permitted Forms

REQUIREMENT

Canonical stored floating-point values **MUST** be finite IEEE 754 binary64 values. NaN and infinity **MUST NOT** appear in canonical chunks: implementations **MUST** reject them at serialization time and **MUST** treat their presence on read as data corruption. The value -0.0 (negative zero) **MUST** be canonicalized to $+0.0$ before storage. Two floating-point values that differ only in zero sign are equal under canonical comparison. Floating-point values appear in advisory, acoustic, tuning, tempo, layout, and quality-metric contexts. Where exact identity is needed (object identifiers, operation ordering, graph membership, hash identity), floating point **MUST NOT** be used.

D.3.2 Serialization

REQUIREMENT

Canonical `f64` values are serialized as little-endian IEEE 754 binary64 octets after applying the $-0.0 \rightarrow +0.0$ canonicalization. The eight serialized bytes are the canonical representation; two values whose canonical bytes are equal are canonically equal. Hash preimages incorporating floating-point values (Section 8.5) **MUST** use the canonical serialized bytes, not platform-native byte orders or representations.

D.3.3 Equality

REQUIREMENT

Canonical equality of two floating-point values is byte equality of their canonical serialized representations. The IEEE 754 notion that `NaN` $\neq$ `NaN` is irrelevant in canonical state, because canonical state cannot contain NaN. For `NormalizedMetric` values (Chapter 9), additional invariants apply: the value is finite, in $[0.0, 1.0]$, and lower is better. Comparisons of `NormalizedMetric` use IEEE 754 ordered comparison; the $-0.0 \rightarrow +0.0$ canonicalization applies before comparison.

D.4 Rounding and CPU Behavior

REQUIREMENT

Numerical operations whose results enter canonical state **MUST** use IEEE 754 round-to-nearest, ties-to-even. The ambient floating-point rounding mode (which on most platforms defaults to round-to-nearest-ties-to-even but can be changed) **MUST NOT** affect canonical results: implementations **MUST** either save/restore the rounding mode or use operations insensitive to it. Implementations **MUST NOT** compile canonical numerical algorithms with

`-ffast-math`, `-funsafe-math` optimizations, or equivalent flags that permit non-IEEE-conformant transformations. Implementations **MUST NOT** reorder, reassociate, or distribute floating-point expressions in ways that can change the canonical result. Implementations **MUST NOT** silently fuse multiplications and additions into fused-multiply-add operations unless the algorithm explicitly requests FMA. Implementations **MUST NOT** depend on, or be sensitive to, ambient floating-point exception flags or host rounding mode.

Where implementations use SIMD, GPU acceleration, or platform math libraries (`libm`, vendor math libraries) in canonical algorithms, they **MUST** produce results identical to a strict-IEEE single-threaded baseline. Differences in transcendental function implementations (`sin`, `cos`, `exp`, `log`) across `libm` versions are a real source of nondeterminism; implementations using transcendentals in canonical algorithms **MUST** use a documented portable implementation (e.g., a vendored math library or a software fallback path).

D.5 Exact and Quantized Representations

The specification prefers exact representations over floating-point wherever possible. The following are exact by design:

Time and duration. `MusicalPosition` and `MusicalDuration` are rational (Chapter 3). Inline-or-promoted representation; arithmetic is exact.

Operation ordering. Causal order from DVV; total order by HLC and identifier (Chapter 6). All integer-valued.

Identifiers. All graph identifiers are 128-bit values composed of replica plus counter (Chapter 5).

Hashes. All canonical hashes are BLAKE3-256 (Chapter 8).

D.5.1 Quantized Layout Coordinates

Spatial layout coordinates require numeric computation, but canonical output must be stable. The specification quantizes canonical coordinates to a fixed grid.

```
/// A canonical spatial coordinate, quantized to 1/1024 of a
/// staff space. Used for canonical serialized output of
/// ResolvedLayoutIR positions. Internal solvers MAY use floating
/// point during computation; canonical serialization rounds to
/// QuantizedCoord.
pub struct QuantizedCoord {
    /// Coordinate value in units of 1/1024 staff space.
    pub units: i64,
}
```

REQUIREMENT

The canonical spatial coordinate grid is 1/1024 staff space per unit. Layout coordinates emitted into canonical `ResolvedLayoutIR` **MUST** be quantized to this grid before serialization. The quantization rule is round-to-nearest, ties-to-even on the integer unit value. Implementations **MAY** use floating-point coordinates internally during layout computation. The act of quantization at serialization time absorbs all floating-point variation from the canonical output. Two implementations whose internal computations agree to better than 1/2048 staff space at every coordinate produce identical canonical output after quantization.

The grid spacing is fixed at 1/1024 staff space for this format version. Future major versions **MAY** adjust the spacing; doing so is a non-backward-compatible change.

RATIONALE

1/1024 staff space is finer than visual resolution at any practical printing density (a staff space is typically 1.5–2 mm, so 1/1024 is roughly 1.5–2 μm , well below print resolution). The grid is fine enough that quantization is inaudible musically and invisible visually, but coarse enough that floating-point noise from different solver implementations routinely rounds away.

D.6 Tolerance Classes

The specification refers to numerical tolerances in several places (acoustic comparison, layout quality thresholds, tempo integration, solver residual). All such tolerances belong to one of a small set of named classes, each with declared semantics.

```
pub enum ToleranceClass {
    /// Acoustic pitch comparison, in cents.
    AcousticCents,

    /// Layout coordinate comparison, in staff spaces.
    LayoutCoordinate,

    /// Quality metric comparison; absolute on the [0.0, 1.0]
    /// NormalizedMetric scale.
    QualityMetric,

    /// Tempo integration residual: maximum permitted error in
    /// musical_to_wallclock or wallclock_to_musical conversion.
    TempoIntegration,

    /// Solver residual: maximum permitted constraint violation
    /// for a soft constraint to be considered satisfied.
    SolverResidual,
}

pub struct Tolerance {
    pub class: ToleranceClass,
```

```

    /// Absolute tolerance. The unit is implied by the class.
    pub absolute: f64,

    /// Optional relative tolerance, applied to nonzero values.
    pub relative: Option<f64>,

    /// What the tolerance governs.
    pub governance: ToleranceGovernance,
}

pub enum ToleranceGovernance {
    /// Tolerance affects canonical equality (rare; usually only
    /// for diagnostic comparison).
    Equality,

    /// Tolerance is a validation threshold (constraint considered
    /// satisfied if violation is below tolerance).
    Validation,

    /// Tolerance affects only diagnostic output; canonical state
    /// is unaffected.
    Diagnostic,
}

```

REQUIREMENT

Specifications and conformance documents **MUST NOT** introduce ad-hoc epsilon constants. Every numerical tolerance that affects normative behavior **MUST** be declared as a named `Tolerance` value belonging to a defined `ToleranceClass`, with an explicit unit, absolute and optional relative bounds, and governance category.

Tolerances **MUST NOT** apply to identity: `PitchId`, `EventId`, `OperationId`, graph membership, hash identity, and operation ordering are exact, never within tolerance.

The Quality Metric Catalog, the Reference Suite, and the Performance Reference Suite enumerate the specific tolerance values for each profile and tier. Their values are normative per the companion specifications.

D.7 Ordered Iteration over Sets and Maps

Canonical output frequently depends on iterating a collection. Hash-map and B-tree implementation order leaks platform-specific behavior into canonical output if not controlled.

REQUIREMENT

Whenever canonical output, canonical serialization, or canonical hashing depends on iterating a set, map, or other collection, iteration **MUST** occur in a specified total order. The following total orders are normative:

- Typed identifiers (`EventId`, `PitchId`, `VoiceId`, etc.): lexicographic ascending on

the identifier's canonical byte form (16 bytes: 8-byte replica, 8-byte counter, big-endian).

- Operation envelopes: causal order first (from DVV closure), then HLC physical time, then HLC logical counter, then `OperationId` as final tie-break.
- Chunk references: ascending by `ChunkKind` discriminant, then by content hash lexicographic, then by file offset.
- Conflict records: ascending by `ConflictId`.
- Extension declarations: ascending by `ExtensionId`, then by semantic version lexicographic.
- Strings (text fields): byte-lexicographic on the UTF-8 NFC representation (Section D.8).
- Rationals (`RationalTime` sequences): by numeric value ascending.
- Composite keys: lexicographic on the canonical representation of the key tuple's component fields, in declaration order.

Hash-map iteration order, B-tree implementation differences, and similar implementation artifacts **MUST NOT** leak into canonical output.

D.8 Text and Unicode

Locale-dependent text behavior is a routine source of canonical divergence. The specification fixes this at the encoding layer.

REQUIREMENT

Canonical text fields **MUST** be encoded as UTF-8 with Unicode NFC (Normalization Form Canonical Composition) applied. Two texts whose NFC byte representations are equal are canonically equal regardless of decomposition state.

Comparisons of canonical text fields for identity **MUST** be byte comparisons of NFC-encoded UTF-8. User-facing display collation (sorting by locale rules, case-insensitive matching, diacritic folding) is non-canonical and **MUST NOT** affect canonical state.

Locale settings **MUST NOT** affect:

- Parsing of canonical text projections.
- Serialization of canonical text fields.
- Sorting of canonical map keys.
- Decimal formatting in canonical text projections (the text projection uses a fixed decimal-point convention; locale comma-vs-period preferences are irrelevant).
- Date and time formatting in canonical data (canonical timestamps use a fixed format; locale formats are for UI only).

Two implementations operating under different locale settings on the same canonical document **MUST** produce identical canonical output, including identical chunk hashes and identical text projections.

D.9 Randomness and Parallelism

D.9.1 Randomness

REQUIREMENT

Randomness **MUST NOT** affect canonical output unless the random seed is an explicit canonical input.

Algorithms whose results enter canonical state and that use pseudorandom generation (e.g., probabilistic constraint solvers, randomized rounding) **MUST** consume their seed as a declared algorithm parameter that becomes part of the algorithm's conformance configuration. Identical seed plus identical input yields identical output.

Identifier minting (`ReplicaId` initialization) is the only conforming use of platform-level randomness in this specification, and even there the random output enters canonical state only via the resulting identifiers.

D.9.2 Parallelism

REQUIREMENT

Parallel implementations of canonical algorithms **MUST** produce results identical to a strict single-threaded baseline. Race timing, work-stealing order, and thread count **MUST NOT** affect canonical state.

Where parallel reductions are used (summation over many contributions, parallel constraint evaluation), the merge step **MUST** be deterministic: ordered reduction over a canonical ordering of contributions, not arbitrary tree-merge order.

Implementations **MAY** use parallel execution as a performance optimization. They **MAYNOT** use it as a license for nondeterminism.

D.10 Compression and File Bytes

REQUIREMENT

Canonical content identity is the uncompressed payload bytes with the domain-separated preimage (Chapter 8). Compression algorithm choice and compression level **MUST NOT** affect canonical identity. Identical uncompressed content has identical `ContentHash` regardless of how it was physically compressed.

Canonical chunk payload encoding **MUST** be deterministic: the same in-memory canonical structure **MUST** produce identical uncompressed bytes across runs and across implementations, within the same schema version. The Binary Format companion specification defines the canonical encoding.

Compression bytes (the physical bytes stored on disk after `zstd` or another algorithm is applied) are *not* canonical unless a profile explicitly requires bit-identical bundle reproduction. Implementations **MAY** recompress chunks (e.g., during compaction or repack) without altering canonical state.

Readers **MUST** verify decompressed payload against the declared content hash before treating the payload as authoritative.

D.11 Open Algorithm Hooks

Several algorithms remain open questions in the body of the specification: the spelling pre-pass (Chapter 2), the notational decomposition algorithm (Chapter 3), tempo integration for arbitrary curves and root-finding for `wallclock_to_musical` (Chapter 3). These algorithms affect canonical state.

REQUIREMENT

Any algorithm that affects canonical state **MUST** satisfy one of:

- It is specified normatively in this document or a named companion specification. Conforming implementations **MUST** implement the specified algorithm.
- It is profile-declared: the active conformance profile names a specific algorithm by versioned identifier. Two conforming implementations on the same profile use the same algorithm. Profiles may differ in their algorithm choices.
- It is explicitly marked non-canonical or advisory: its output appears in the score only in non-canonical layers (caches, diagnostics, user-facing rendering hints) and **MUST NOT** appear in canonical chunks.

Implementations **MUST NOT** introduce algorithms affecting canonical state outside of these three categories. “Vendor’s spelling preference” or “proprietary decomposition heuristic” **MUST NOT** silently determine canonical content.

D.12 Conformance Statement

REQUIREMENT

Conforming implementations **MUST** declare, as part of their published conformance statement:

- Their platform and floating-point library, with the libm version or vendored math library reference.
- Their parallel execution strategy and the demonstrated equivalence to single-threaded baseline.
- Their handling of the -0.0 canonicalization and NaN/infinity rejection.
- Their adherence to round-to-nearest-ties-to-even.
- Their compliance with the canonical iteration orders in Section D.7.
- Their NFC normalization implementation reference.
- Their declared algorithms for any open-question area (spelling, decomposition, tempo curves), with versioned identifiers.

Revision History

Date	Section	Change
June 2026	17, All	Initial scaffold; chapter structure and front matter established.
June 2026	17, Ch. 2	Pitch chapter written: layered pitch type, scale position, acoustic realization, spelling subsystem with attachments, precedence, pre-pass, and analytical layers. Chapter structure expanded: pitch separated from time/tuning; semantic operations chapter added.
June 2026	17, Ch. 3	Time and duration chapter written: exact rational time with inline-or-promoted representation, distinct position and duration newtypes, wall-clock time as fixed-point nanoseconds, time anchors, time signatures with explicit beat groups including irrational meters, tempo maps with deterministic conversion, notational decomposition pre-pass, tuplets as grouping objects, and three region time models (metric, proportional, aleatoric).
June 2026	17, Ch. 4	Tuning systems and pitch spaces chapter written: pitch spaces as analytical universes distinct from tuning systems, position structures (chromatic, diatonic-over-chromatic, JI lattice, registered), interval algebras, nominal and accidental registries, SMuFL glyph references with versioning, tuning resolution variants including adaptive tuning, score-level reference pitch, hierarchical resolution rules, and the normative built-in catalog of pitch spaces and tuning systems.

Date	Section	Change
June 2026	17, Ch. 5	Score graph chapter written: hybrid tree-plus-cross-cutting topology, canvas-region-staff-voice-event hierarchy with regions carrying time and content models, typed 128-bit identifiers with replica-plus-counter generation, flat event arena, complete event taxonomy (pitched, unpitched, rest, indeterminate, trajectory, graphic, cue), cross-cutting registry partitioned by type, graphic content with region-local coordinates and time bindings, parts as projections separate from the canvas, analysis layers and views, graph invariants, and required indexes.
June 2026	17, Ch. 6	Semantic operations chapter written: operation framework with dual command/delta layering, primitive and compound operation distinction, transactions as atomic user units, precondition classification (invariant vs. advisory) with strict authoring and lenient replay modes, the normative re-anchoring rule table, inverse-based undo with single-replica scope, semantic merge semantics with five merge families, and representative operations illustrating each major category. The full operation catalog is deferred to a separate conformance specification.
June 2026	17, Ch. 7	Layout IR chapter written: pipeline of four stages (LogicalLayoutIR, ConstrainedLayoutIR, ResolvedLayoutIR, RenderIR-at-interface), shared provenance discipline, spatial primitives in staff spaces with f32 precision, polymorphic time axis trait with metric/proportional/aleatoric implementations, composite logical objects flattened to glyph-level constrained objects, spring-based horizontal and vertical layout, explicit engraving decision recording, hard and soft override mechanism with preservation rules, fine-grained dependency tracking with invalidation rules, glyph catalog interface, and uniform region containers across kinds.

Date	Section	Change
June 2026	17, Ch. 8	File format chapter written: bundle container structure with header, manifest, chunk store, and blob store; custom schema-defined binary format with self-describing extension envelope; content-addressed immutable chunks with manifest as the only mutable index; multi-level chunking (score-level, staff-region, locality-grouped cross-cutting, attachment chunks); operation log as canonical source of truth with dotted version vectors and hybrid logical clocks; snapshots as derivative caches; schema versioning with lazy migration and forward compatibility; format profiles (Full, ReadOnly, Lite); per-chunk zstd compression; atomic write protocol; deterministic s-expression text projection; streaming read guarantees. DRM and encryption explicitly excluded.
June 2026	17, Ch. 9	Constraint-solver interface chapter written: interface-and-contract specification rather than algorithm; the solver trait with from-scratch and incremental modes; Cassowary- style Required and Preferred constraint strengths; the constraint family catalog (spring, collision, alignment, containment, break, aesthetic, extension); the normative quality metric set (optical spacing, collisions, page fill, casting-off, beam angle, slur curvature, vertical balance, symbol density uniformity); Pareto- bounded quality target with configurable slack and tie-breaking weights for ambiguous cases; within-implementation byte-determinism with cross-implementation Pareto equivalence; incremental solving contract with propagation bounds; structured error reporting with partial solutions on budget exhaustion; reference suite for conformance; non-normative recommendation of Cassowary as a starting algorithm.

Date	Section	Change
June 2026	17, Chs. 10, 11	Performance Requirements chapter expanded from skeleton: reference hardware profile (2026 desktop and tablet baselines), frame budgets for interactive edits at p99, cold-open targets, memory budgets separating core data from evictable caches, audio thread discipline requirements crossing the core/audio boundary, constraint-solver performance targets, file format performance targets, measurement methodology, regression testing recommendations. Extension Points chapter written from skeleton: consolidated catalog of every extension point in the spec organized by chapter of origin (notation grammar, tuning system, glyph, score graph, layout IR, constraint solver, file format extensions), the extension registry contract requiring versioning and identifier stability, plugin runtime considerations, conformance interactions.
June 2026	17, Appendices	Glossary expanded with 50+ defined terms cross-referenced to their originating chapters. Bibliography populated with engraving treatises, music theory references, tuning and microtonality sources, constraint-based layout literature, CRDT references, and prior-art notation format citations.
June 2026	17, All	Platform named <i>Epiphany</i> . Typography refreshed to TeX Gyre Pagella (body), Heros (sans), and Cursor (mono). Color palette refined to deep teal, antique gold, warm parchment neutrals, and deep crimson accents. Title page redesigned with ornamental rules and an oldstyle classical feel. Status section, introduction, and PDF metadata updated to reflect the platform's name.

Date	Section	Change
June 2026	17, Pass 1 revision (Chs. 2, 3, 4, 5, 6)	Type-seam corrections in response to external review. Introduced <code>IdentifiedPitch</code> wrapping every embedded pitch with a stable <code>PitchId</code> ; updated <code>PitchedEvent</code> , <code>TrajectoryEndpoint</code> , and <code>TrajectoryShape::Stepwise</code> accordingly. Introduced <code>EventDuration</code> union covering musical, wall-clock, and indeterminate cases, with concrete-only <code>DurationBounds</code> avoiding recursive indeterminacy. Aleatoric regions now declare an explicit <code>AleatoricAnchoringDiscipline</code> . <code>PitchSpelling</code> carries a stack of accidentals (<code>Vec<AccidentalId></code>) with normative additive semantics and duplicate-detection invariants, replacing the prior <code>Option<AccidentalId></code> . <code>PitchSpacePosition</code> gains a <code>JiVector</code> variant for first-class just-intonation lattice positions. The default CMN pitch space is now correctly described as <code>cmn-12</code> (diatonic-over-chromatic), not pure chromatic. Chapter 2’s reference-pitch text rewritten to defer to Chapter 4, resolving the prior ownership contradiction. <code>MusicalPosition</code> and <code>MusicalDuration</code> now derive <code>Clone</code> , not <code>Copy</code> , with a permitted inline-or-pointer-tagged implementation noted as conforming. <code>Tie::pitch_pairing</code> now uses <code>Vec<(PitchId, PitchId)></code> instead of positional indices. The duplicate <code>AnalysisLayer</code> definition in Chapter 2 was removed in favor of the authoritative definition in Chapter 5. Graph invariants expanded to cover identified pitches, tombstoning, time-model agreement, and <code>PitchId</code> -pairing tie targets. Representative operations (<code>InsertEvent</code> , <code>DeleteEvent</code> , <code>RespellPitch</code> , <code>Transpose</code>) updated for atomic ID minting, pitch tombstoning, voice-promotion on concurrent insertion, and <code>PitchId</code> preservation through transposition. Status downgraded to “structural working draft.”

Date	Section	Change
June 2026	17, Pass 2 revision (Chs. 3, 5, 6)	Graph and time invariants strengthened in response to external review. <code>TimeAnchor</code> offsets generalized via the <code>AnchorOffset</code> union (<code>Musical</code>, <code>WallClock</code>, <code>Zero</code>), with normative agreement between offset variant and target region's time model. <code>TempoSegment</code> restructured to carry explicit <code>end</code>: <code>Option<TimeAnchor></code>, <code>start_tempo</code>, <code>end_tempo</code>: <code>Option<Tempo></code>, and <code>shape</code>; tempo segments are required to be monotonic in musical time so that wall-clock-to-musical conversion is single-valued. Polymeter admitted natively: <code>StaffBasedContent</code> now carries <code>Vec<StaffInstance></code> (region-local), plus <code>default_metric_grid</code>: <code>Option<MetricGrid></code> and <code>barline_alignment_groups</code>: <code>Vec<BarlineAlignmentGroup></code>. Measures and metric grids are per-staff-instance, not per-region. Staff identity split into the global <code>Staff</code> (declared once at score level) and the region-local <code>StaffInstance</code>; Score top-level structure expanded with <code>staves</code>: <code>Vec<Staff></code> and <code>staff_groups</code>: <code>Vec<StaffGroup></code>. <code>DeleteEvent</code> preconditions strengthened: target events belonging to tuplets require explicit <code>TupletCompensation</code> (<code>NotInTuplet</code>, <code>ReplaceWithRest</code>, <code>RewriteTuplets</code>, <code>CascadeDeleteTuplets</code>). Tie validation reframed from universal immediate-adjacency to class-specific profiles: <code>TieClass</code> (<code>Standard</code>, <code>Editorial</code>, <code>CrossVoice</code>, <code>LaissezVibrer</code>, <code>Registered</code>) admits editorial, cross-voice, and laissez-vibrer ties without relaxing standard-tie invariants. System-promoted voices formalized: <code>VoiceOrigin</code> enum (<code>UserDeclared</code>, <code>Imported</code>, <code>SystemPromoted</code>); promoted voices have deterministically-derived <code>VoiceIds</code>, are first-class graph objects, are visible to users, and are normalized only by explicit user operation. Graph invariants expanded from 14 to 18 normative invariants reflecting staff-instance ontology, per-staff measures, anchor-offset agreement, system-promoted voice identity, and barline-alignment-group containment. New identifier types: <code>StaffInstanceId</code>, <code>StaffGroupId</code>, <code>BarlineAlignmentGroupId</code>, <code>TimeSignatureId</code>.

Date	Section	Change
June 2026	17, Pass 3 revision (Ch. 6, 8)	Concurrent semantics rewritten from pairwise operation merge to operation-set deterministic reduction. The replicated operation set is now framed as a true CRDT (a grow-only set of envelopes); the materialized score graph is its deterministic reduction, not itself a CRDT. <code>OperationId</code> (replica + counter) separated from <code>OperationStamp</code> (HLC + id); canonical reduction order is causal-then-HLC-then-lexicographic. Causal contexts expressed as dotted version vectors (<code>CausalContext</code>), not exhaustive predecessor lists; exhaustive lists are explicitly forbidden as canonical representation. <code>OperationEnvelope</code> introduced with id, author, stamp, causal context, optional transaction, payload. Four-phase operation lifecycle specified: Prepare, Commit, Reduce, Report. Operation effects formalized: <code>Applied</code>, <code>AppliedWithRepair</code>, <code>Conflicted</code>, <code>TombstonedTarget</code>, <code>NoOp{reason}</code>; every replica reducing the same set records the same effects with the same reasons. Conflict records made first-class graph objects in a score-level <code>ConflictRegistry</code>: stable, addressable, user-visible, resolvable by subsequent <code>ResolveConflict</code> operations. Tombstones formalized with explicit retention rules (pending replication, undo, conflict resolution, attachment resolution). Re-anchoring rewritten as a total deterministic function: “nearest” is now a strict lexicographic minimum over (containment proximity, time distance, direction preference, object id), with no implementation discretion. Re-anchoring rule table updated for the new staff-instance ontology and the deterministic “nearest” function. Transactions formalized as atomic replicated groups: primitive members MUST NOT materialize as independently visible intermediate states; failures conflict the whole transaction. Undo redefined as a forward compensating operation (<code>UndoTransaction</code>) with three policies (<code>StrictInverse</code>, <code>BestEffort</code>, <code>Cascade</code>); bit-identical undo demoted to content equivalence. LWW reduction restricted to explicitly-marked <code>LwwAdvisory</code> scalar fields (metadata, view preferences, layout hints, system/page break preferences, optional cross-cutting display hints); structural musical fields MUST NOT default to LWW. Cache invalidation rule: caches are acceleration only, never canonical; invalidated on operation-set, causal-order, reduction-algorithm, or profile change. Per-operation merge families renamed to reduction rules and rewritten in the new <code>221</code> model; <code>ChangeRegionTimeModel</code> now correctly conflicts rather than silently coercing incompatible contained events; <code>SetUserSystemBreak</code> explicitly marked <code>LwwAdvisory</code>; <code>RespellPitch</code> concurrent col-

Date	Section	Change
June 2026	17, Pass 4 revision (Ch. 8)	File-format restructured around Pass 3’s operation-set canonical-document model. The earlier “single mutable manifest” story replaced with a fixed 64-byte header plus two fixed-size 256-byte superblock slots; the superblocks are the only mutable on-disk objects, and atomic commit is achieved by writing the inactive superblock slot with generation +1. Active superblock selected by highest valid generation. Generation gap greater than one detected as integrity anomaly with read-only recovery mode. Manifest reframed as a table of roots and declarations (<code>operation_roots</code>, <code>retained_snapshots</code>, <code>blob_roots</code>, <code>profile_declarations</code>, <code>extension_declarations</code>, <code>text_projection_root</code>, <code>integrity_root</code>); user-facing metadata explicitly removed from the manifest in favor of operation envelopes and reduced state. BLAKE3 fixed as the single content-hashing algorithm, replacing the prior “BLAKE3 or SHA-256” uncertainty. Hash preimage domain-separated and explicitly excludes <code>compression_algorithm</code> to preserve deduplication across compression choices. Operation-envelope blocks specified as the canonical storage unit with a soft 1 MiB uncompressed target size and a default 64 MiB upper bound per profile. Snapshots separated into acceleration snapshots (caches, discardable) and canonical-base snapshots (declared in <code>retained_snapshots</code>, required for pruning). Pruning protocol specified: canonical state preserved as (base snapshot + post-frontier envelopes). Atomic write protocol specified step-by-step with explicit durable-flush points; the durable flush is platform-abstracted to cover <code>fsync</code>, <code>FlushFileBuffers</code>, and equivalents. Crash recovery rules enumerated by failure point. Garbage collection specified as conservative, optional, deferred; never on the commit critical path. Edit barriers formalized as a structured first-class concept (<code>EditBarrier</code> with scope, affected object kinds, prohibited operation kinds, and condition), with an unsafe-edit escape hatch that tombstones extension data rather than silently corrupting it. Text projection reframed: semantic round-trip required (envelopes, declarations, reduced state); physical-layout fidelity (chunk offsets, compression choices, cache chunks, garbage regions, superblock generation) explicitly not required. Schema versioning, format profiles (Full, ReadOnly, Lite, Custom), and compression sections updated for the new architecture. Streaming-read cold-open procedure rewritten to use snapshot + envelope-stream model.

Date	Section	Change
June 2026	17, Pass 5 revision (Ch. 9)	Solver conformance reframed from universal Pareto optimality to reference-suite-based conformance with profile-specific thresholds. The earlier “no better Pareto layout exists” obligation, which quantified over an undecidable space of possible layouts, has been replaced with checkable, falsifiable thresholds on a reference suite. Solver obligations split into four classes: validity conformance (hard constraints, provenance, unit system, invariants), determinism conformance, diagnostic conformance (<code>SolveStatus</code>, <code>SolveReport</code>, warnings, unsatisfied-constraint reporting, metric vectors), and quality conformance (reference-suite thresholds). Three solver tiers introduced (<code>Minimal</code>, <code>Standard</code>, <code>Advanced</code>) with declared obligations per tier; tiers are orthogonal to file-format profiles. Deterministic budgets formalized (<code>max_iterations</code>, <code>max_nodes</code>, <code>max_constraint_evaluations</code>); wall-clock time demoted to advisory only. Wall-clock MUST NOT change the canonical layout. Quality metrics normalized: <code>NormalizedMetric</code> is a finite <code>f64</code> in <code>[0.0, 1.0]</code> with 0.0 best and 1.0 worst; orientation and range fixed across all metrics including extension metrics. Hard constraints explicitly separated from quality metrics: a solver MUST NOT trade off a hard-constraint violation for a better quality score. Within-implementation determinism (byte-equal output for identical input + version) separated from cross-implementation conformance (reference-suite thresholds, not byte equality). <code>SolveReport</code> unified the prior <code>Result<SolveOutput, SolverError></code> return: every invocation yields a report with a <code>SolveStatus</code> (<code>Solved</code>, <code>SolvedWithWarnings</code>, <code>PartialBudgetExhausted</code>, <code>Unsatisfiable</code>, <code>InternalError</code>). <code>PartialBudgetExhausted</code> preserves hard-constraint validity; budget exhaustion that cannot satisfy hard constraints returns <code>Unsatisfiable</code>. Six invalidation scopes specified (<code>ObjectLocal</code>, <code>MeasureLocal</code>, <code>SystemLocal</code>, <code>PageLocal</code>, <code>RegionLocal</code>, <code>WholeScore</code>); incremental solving contract reframed as observational equivalence to scoped full solving (solvers MAY widen scopes conservatively but MAYNOT narrow them). Reference algorithm demoted from “recommended starting point” to explanatory and diagnostic only; conforming implementations MAY choose any algorithm. Pass 4 rollback-language corrected: rollback is achieved by retained manifests in the chunk store, not by retaining multiple on-disk superblock generations (the fixed layout has only two slots).

Date	Section	Change
June 2026	17, Pass 6 revision (App. D)	New Determinism Contract appendix consolidates reproducibility obligations scattered across Chapters 2–9 into a single normative reference. Five layers of determinism distinguished: canonical score determinism (byte-equal cross-implementation), canonical serialization determinism (byte-equal chunk hashes and text projection), layout determinism (byte-equal within implementation version), conformance determinism (reference-suite-based, not byte-equal cross-implementation), and non-canonical cache determinism (caches may vary, MUST NOT affect canonical state). Canonical floating-point values restricted to finite IEEE 754 binary64; NaN and infinity prohibited from canonical chunks; -0.0 canonicalized to $+0.0$ before storage and hashing. Little-endian IEEE 754 binary64 serialization fixed. IEEE 754 round-to-nearest, ties-to-even mandated for canonical numerical algorithms; <code>-ffast-math</code> and equivalents prohibited; reassociation, distribution, and implicit fused-multiply-add forbidden where they could change canonical output. Quantized layout coordinates introduced (<code>QuantizedCoord</code> at 1/1024 staff space) so canonical <code>ResolvedLayoutIR</code> positions are stable across implementations whose internal computations agree to within half-quantum. Tolerance classes formalized (<code>AcousticCents</code>, <code>LayoutCoordinate</code>, <code>QualityMetric</code>, <code>TempoIntegration</code>, <code>SolverResidual</code>) with governance category (<code>Equality</code>, <code>Validation</code>, <code>Diagnostic</code>); ad-hoc epsilons prohibited; tolerances MUST NOT apply to identity. Canonical iteration order specified for every enumerable collection (identifiers lexicographic on canonical byte form; operations causal-then-HLC-then-id; chunk references by kind, hash, offset; conflicts by id; extensions by id then version; strings byte-lexicographic on NFC UTF-8). Text normalization fixed at UTF-8 NFC; locale prohibited from affecting canonical parsing, serialization, sorting, decimal formatting, or projection. Randomness MUST NOT enter canonical state unless the seed is an explicit canonical input. Parallelism permitted but MUST produce results identical to a strict single-threaded baseline through deterministic ordered reductions. Compression confirmed as non-canonical (identical uncompressed content has identical hash regardless of compression choice); recompression permitted without altering canonical state. Open algorithm hooks (spelling pre-pass, decomposition, tempo integration, root-finding) required to be either normatively specified, profile-declared by versioned

Date	Section	Change
June 2026	17, Pass 7 revision (editorial, all chapters)	Editorial consistency pass: no architectural changes, only cleanup. Resolved the contradiction between Appendix D’s “byte-equal canonical state cross-implementation” and the existence of open canonical algorithms by adding an explicit conditional on the disposition of Section D.11. Stale section-number cross- references replaced with symbolic labels (Section 2.4 → <code>sec:pitch:spelling</code> ; Section 2.4.4 → <code>sec:pitch:precedence</code> ; Section 5.9.3 → <code>sec:graph:timebinding</code> ; Section 6.6 → <code>sec:semops:conflicts</code> ; Section 6.8 → <code>sec:semops:undo</code>). Terminology normalized: “merge family” → “reduction rule” where the operation framework was meant; “Pareto-bounded quality model” → reference-suite/normalized-metric language in extension points; glossary entries for Operation log, Operation set, Operation envelope, Canonical reduction, Pareto frontier (design target), Staff, Staff instance, Conflict record, CRDT, Snapshot, and Transaction rewritten or added. New Appendix A (Intentionally Deferred Types and Specifications) catalogs companion specifications (Binary Format, Text Projection, Profile Conformance, Quality Metric Catalog, Reference Suite, Performance Reference Suite, Operation Catalog, Reference Algorithm), externally provided components (SMuFL, font metric provider, audio engine, plugin runtime, collaboration transport, user interface), open canonical algorithms, and extension registry catalogs; distinguishes deliberately external content from accidentally missing content. Status section rewritten to reflect structural stabilization: substantive review passes complete, architecture not under active redesign, remaining blockers are companion specifications and open canonical algorithms; the section enumerates what conformance can be established now and what awaits companion delivery. Revision history entries for Passes 1–7 read as the document’s architectural-stabilization log, consolidating the design decisions taken in response to external review.

Date	Section	Change
June 2026	17, Pass 8 revision (post-review correctness pass)	Determinism correctness pass addressing review of the Pass 7 output. <code>ConflictId</code> reframed as content-derived via BLAKE3 truncation with domain tag "MUSCCONF", fixing a determinism leak where each replica would have minted its own conflict identifiers from local replica-counter sequences; conflict records produced during reduction now have identifiers that agree across replicas by construction. <code>RepairRecord</code> defined (was referenced by <code>OperationEffect::AppliedWithRepair</code> without definition), with a <code>RepairKind</code> enumeration covering <code>Reanchored</code>, <code>SpannerTruncated</code>, <code>Orphaned</code>, <code>CascadeDeleted</code>, <code>AttachmentTombstoned</code>, <code>VoicePromoted</code>, <code>TupletCompensated</code>, and <code>Registered</code>. <code>TypedObjectId</code> defined as a tagged union over every identifier kind in the score graph, with canonical-bytes serialization for hashing/ordering/equality. <code>OperationKind</code> defined as a tagged enumeration of primitive operation variants, plus a <code>Registered</code> variant for extension-defined primitives. System-derived identifier namespace formalized: <code>ReplicaId::SYSTEM_DERIVED</code> reserved at <code>0xffff_ffff_ffff_ffff</code>, with the 64-bit counter derived deterministically via BLAKE3 truncation; domain tags for system identifiers ("MUSCSVCE" for system-promoted voices, "MUSCSPCH" for system-derived pitches) enumerated; user-authored replicas MUST NOT use the reserved value. Envelope-acceptance rules added: a well-formed envelope requires non-system replica id, finite physical time, well-formed DVV, deserializable payload; per-replica monotonicity enforced; clock-skew warnings allowed but re-ordering for plausibility prohibited; authentication deferred to collaboration transport. Canonical-base snapshot semantics repaired: manifest split into single <code>canonical_base: Option<SnapshotRef></code> plus <code>acceleration_snapshots: Vec<SnapshotRef></code>; coverage test specified as DVV-frontier membership, explicitly forbidding HLC-stamp comparison for canonicity determination; pruning replaces the canonical base atomically. Header CRC moved to the last 4 bytes of the 64-byte fixed header, resolving a circular-definition bug where the CRC field appeared inside its own coverage range. Manifest chunks MUST be stored uncompressed in this format version, resolving the bootstrap chicken-and-egg problem (a reader needs to know how to decode the manifest before it has any manifest information). <code>ChunkId</code> defined as a newtype around <code>ContentHash</code>, clarifying the prior comment that the hash and id were redundant. Three-layer identity

Date	Section	Change
June 2026	17, Pass 9 revision (correctness and consistency pass)	Operation-envelope duplicate handling formalized into three outcomes: <code>IdempotentDuplicate</code> (same id, byte-identical envelope: silently dropped), <code>FirstAcceptance</code> (new id: accepted after well-formedness check), and <code>EnvelopeEquivocation</code> (same id, different canonical bytes: divergent envelope rejected, integrity diagnostic surfaced, bundle non-conforming if equivocation appears on-disk). Well-formedness invariant <code>envelope.stamp.id == envelope.id</code> made normative. HLC monotonicity reframed as a property of the set of accepted envelopes, not arrival order: out-of-order delivery is fine; on detection of inconsistency, both envelopes are retained but the offending replica's stream from the violating counter onward is quarantined and the document is flagged with a replica-equivocation anomaly. The free-form <code>NoOpReason::PreconditionFailedUnderReduction</code> <code>reason: String</code> replaced with a typed <code>PreconditionFailureReason</code> enum, closing a determinism leak (free-form strings in canonical state could otherwise differ across implementations). Transaction descriptor ordering integrated into canonical reduction via causal dependency: every primitive member of a transaction MUST causally depend on its <code>DeclareTransaction</code> envelope; members lacking such a dependency reduce to <code>NoOp{TransactionConflict}</code> with a synthesized conflict record. The fixed-header <code>file_uuid</code> requirement reworded to match the Pass 8 identity model: fixed for the lifetime of a physical bundle, preserved by filesystem byte-copy, freshly minted on Save As. Manifest <code>integrity_root</code> made <code>Option<ChunkRef></code> (it is an accelerator); new “Canonical and Non-Canonical Manifest Roots” subsection partitions the manifest's fields by their role in canonical state, with the rule that failed verification of non-canonical chunks is rebuildable rather than corruption. Cold-open procedure rewritten to reflect the canonical-base / operation-envelope-block model (the prior text retained pre-Pass-3 region-chunks language). Streaming- read locatability list updated. System-derived counter collisions: mandatory collision check during reduction; collision becomes hard corruption via new <code>ConflictKind::SystemIdentifierCollision</code> record naming both colliding canonical-input sets. <code>GlyphCatalogIdentity</code> <code>metrics_hash</code> scope broadened²²⁷ hash covers glyphs referenced by the <code>ConstrainedLayoutIR</code> (solver inputs), not just glyphs in the final <code>ResolvedLayoutIR</code>. <code>CommitState</code> validation added to superblock selection: non-

Date	Section	Change
June 2026	17, Pass 10 revision (order-independence and integrity-anomaly cleanup)	<code>OperationId</code> equivocation reworked from a first-arrival rule (which made canonical state arrival-order-dependent across replicas observing the same envelopes in different orders) to an order-independent <code>OperationSlot</code> model. The slot is either <code>Single(OperationEnvelope)</code> or <code>Equivocated { operation_id, candidates: BTreeSet<EnvelopeHash> }</code>; observing a second distinct canonical envelope under an <code>OperationId</code> transitions the slot to <code>Equivocated</code> regardless of arrival order. <code>Equivocated</code> slots produce no canonical reduction effect; both candidates are retained for diagnostic recovery. Operations causally depending on an equivocated id are held pending resolution. Resolution is by explicit policy: transport-level credential revocation, local user reconciliation, or a profile-declared deterministic selection function. No arrival-order default applies. HLC monotonicity-anomaly handling sharpened: a new <code>AnomalousReplicaSegment</code> type captures the offending replica, the first violating counter, the reason, and the excluded envelope ids. Operations from the violating counter onward are retained but <i>excluded from canonical reduction</i>; canonical state is derived only from envelopes that satisfy the monotonicity invariant. The earlier “quarantined” wording is now backed by an explicit structural exclusion rule. System-derived ID collisions moved out of <code>ConflictKind</code> and into a new <code>IntegrityAnomaly</code> type with kinds <code>SystemIdentifierCollision</code>, <code>OperationSlotEquivocated</code>, and <code>ReplicaStreamQuarantined</code>. The distinction is that conflict records are ordinary canonical-state facts (a representable musical conflict between two valid edits) while integrity anomalies indicate the structural assumptions underlying canonical state have failed and require external recovery; an integrity anomaly takes the document out of ordinary canonical operation. Stale “refined in a later revision” parenthetical removed from the <code>DeleteEvent</code> reduction rule; the re-anchoring total order is part of the current architecture and needs no further refinement. Typo fixed in the superblock-selection requirement (<code>MAY-be-valid</code> replaced with the prose “invalid for ordinary selection but may be inspected in diagnostic recovery mode”). Glossary chapter references for “Dotted version vector” and “Hybrid logical clock” updated 128 “Chapters 6, 8” with explicit “defined operationally in Chapter 6; serialized in Chapter 8” wording, reflecting that the Pass 3 architectural shift made causal context part of operation semantics

Date	Section	Change
------	---------	--------