Calamus Architecture v1.3

The Three Pillars

Calamus is built on three interconnected systems:

Sound Engine — Generates audio through physics-based synthesis

- Container-based architecture (HarmonicGenerator, FormantBody, PhysicsSystem, etc.)
- Renders phrases to audio buffers
- Real-time playback of pre-rendered content
- Live synthesis only for actively-edited content

Data Layer — The contract between systems

- Defines all data structures (Note, Phrase, Sounit, Curve, Scale, etc.)
- Manages persistence (save/load)
- Provides the vocabulary both engines speak

Input Engine — Captures gesture, displays notation

- Wacom pen input (6 dimensions)
- Scale-centric staff display
- Curve-based note visualization
- Editing tools

Pre-Render Model

Calamus is a compositional tool, not a real-time performance instrument. This enables a pre-render approach:

Background rendering:

- When a phrase is modified, it's marked 'dirty'
- A background thread renders dirty phrases to audio buffers
- Rendered audio is stored in RAM
- Playback mixes pre-rendered buffers — trivial CPU load

Live synthesis exception:

- The currently-selected note/phrase plays through live sound engine
- Enables real-time feedback while editing
- Only one phrase live at a time — manageable load

Benefits:

- Unlimited complexity in sound design (1400 harmonics? No problem)
- Smooth playback regardless of project size
- No real-time pressure during composition

Threading Model

Three threads with distinct responsibilities:

Thread	Priority	Responsibility	Never Does
Audio	Highest (real-time)	Mix pre-rendered buffers, run live sound engine for active selection	Allocate memory, wait on locks, touch UI

Thread	Priority	Responsibility	Never Does
UI	Normal (Qt main)	User interaction, display updates, command processing	Heavy computation, long operations
Render	Low (background)	Pre-render dirty phrases, queue results	Touch UI directly

Thread Communication

UI ↔ Render: Qt signals/slots (safe, convenient)

UI → Audio: Lock-free SPSC (single-producer single-consumer) queue for commands

Audio → UI: Lock-free SPSC queue for status updates

Render → Audio: Lock-free queue for rendered buffers

Critical principle: Audio thread never waits. If data isn't ready, it plays silence or continues with what it has. No locks, no allocations, no exceptions in the audio callback.

Scope: Not a DAW

Calamus generates sound. Post-processing happens elsewhere.

Sound Engine DOES:

- Physics-based synthesis
- Pre-rendering to buffers
- Basic per-sounit level and pan
- Master level
- Export to WAV (stems or mix)

Sound Engine does NOT:

- Reverb, delay, chorus
- EQ, compression, limiting
- Buses, sends, inserts
- Mastering

Workflow: Compose in Calamus → Export stems → Post-process in Reaper/Audacity/DAW of choice

Preferences and Limits System

No hard-coded limits. Reasonable defaults with full configurability.

The 1,400 harmonics story: attempting an extreme value created standing waves in the room. Arbitrary limits prevent such discoveries. Instead:

Principle: All numeric limits are configurable. Reasonable defaults are provided, but the user can extend any range.

Implementation

- Limits class provides min/max values for all configurable parameters
- Containers query Limits rather than using magic numbers
- Preferences UI allows adjustment
- Can be set globally or per-project
- Future-proof: as CPUs improve, users adjust limits

Example Configurable Limits

Parameter	Default Min	Default Max	Notes
numHarmonics	1	64	Try 1400 for standing waves
F1 frequency	200 Hz	1000 Hz	Extend for special effects
F2 frequency	500 Hz	3000 Hz	Extend for special effects
Rolloff power	0.1	3.0	Wider range = more extreme brightness
Physics mass	0.0	1.0	Higher = more sluggish response
Drift rate	0.1 Hz	10.0 Hz	Faster = more obvious beating
Polyphony per sounit	1	16	Memory/CPU tradeoff
Attack time	0.001 s	5.0 s	Extend for slow swells
Release time	0.01 s	10.0 s	Extend for long decays

UI Layout

Main Window Tabs

- Composition — where music is written
- Sound Design — where sounits are created/edited
- Preferences — global and project settings, limits

Composition Window (Top to Bottom)

Timeline: Bar markers, time position, click to set 'now' marker, loop region shown as shaded area

Sounit Selector (left panel): Colored vertical bars representing each sounit. Height = register range, color = note color. Active sounit solid, others ghosted for context.

Staff Canvas (center): Scale degree lines (color-coded by harmonic function), Melodyne-style note blobs. Shape shows selected parameter evolution, color identifies sounit.

Transport (bottom): Play/stop/rewind/forward, position display, tempo, loop toggle

Mix Strip (below transport): Per-sounit: volume slider, pan knob, mute button (color matches sounit). Master level. Export button.

Now Marker and Loop Mode

Normal playback:

- Click timeline → set 'now' position
- Play → starts from 'now'
- Stop → returns to 'now'

Loop mode:

- Press 'L' → enter loop mode
- Click timeline → set loop end (auto-swap if before 'now')
- Shaded region shows loop
- Play → loops between 'now' and loop end
- Press 'L' → exit loop mode, clear loop region

Sound Design Window

- Container graph — visual node editor for connecting containers

- Parameter panel — adjust selected container's settings
- DNA selector — harmonic distribution presets and custom
- Test transport — plays selected phrase from Composition (no separate test area)

Key Data Relationships

Note vs Voice

Note (Data Layer):

- Compositional atom — what you see on the canvas
- Persistent — saved in project file
- Contains curves defining parameter evolution
- Belongs to a Phrase

Voice (Sound Engine):

- Runtime instance playing a Note
- Ephemeral — exists only during playback
- Has state: attack/sustain/release/off
- Has physics state (positions, velocities)
- Has 'age' for voice stealing when polyphony exceeded

Track and Sounit

1:1 relationship. Each track has its own sounit instance.

- Multiple tracks CAN start from the same sounit template
- But each gets an independent copy
- Can drift into variations over time
- No sharing — changes to one don't affect others

Phrase

- Container for Notes with shared timing context
- Unit of pre-rendering (phrase → buffer)
- Also used as test material in Sound Design
- Can be marked 'dirty' to trigger re-render

Container Architecture

See Container Port Specification v1.1 for complete details.

Container Categories

Essential (2): HarmonicGenerator, SpectrumToSignal

Shaping (4): RolloffProcessor, FormantBody, BreathTurbulence, NoiseColorFilter

Modifiers (5): PhysicsSystem, EasingApplicator, EnvelopeEngine, DriftEngine, GateProcessor

Key Principles

- Everything modulatable is an input port with a default value
- Unconnected input → uses default
- Connected input → modulated by source

- Multiple instances of any container allowed
- Signal path must end with Signal output
- PhysicsSystem provides expression AND stability (smooths discontinuities)

Connection Functions

When connecting output to input:

- passthrough — source replaces input entirely
- add — input + source × weight
- multiply — input × source × weight
- subtract — input - source × weight
- replace — input × (1-weight) + source × weight
- modulate — bipolar modulation around input

Inheritance Decision

Containers are implemented as subclasses, not generic Container with type parameter.

Reasoning:

- DSP code fundamentally different per container type
- Type safety — compiler catches connection errors
- Clear organization — FormantBody.cpp contains FormantBody code
- Easy debugging — open the file for the container you're fixing

Technical Platform

Language: C++

UI Framework: Qt

Audio: RtAudio (cross-platform, low-latency)

Build: CMake

Target platforms: Windows, macOS, Linux

Escape hatch: If specific components need different treatment, JUCE remains available for audio-specific modules.

File Structure (Planned)

- .calamus — Complete project (composition, sounits, settings)
- .sounit — Exported sounit definition (shareable)
- .phrase — Exported phrase (shareable)
- .wav — Rendered audio (stems or mix)

Format decision (JSON vs binary vs XML) deferred to implementation.

Open Questions

- File format details (JSON structure, binary format)
- Polyphony handling — per-voice container instances or shared with voice indexing?
- Control rate vs audio rate — which parameters need sample-accurate modulation?
- Default physics values (mass, spring, damping starting points)
- Undo/redo architecture

Version History

v1.0 — Initial architecture (three pillars, pre-render model)

v1.1 — Added foundational principles section

v1.2 — Clarified scope (not a DAW), eliminated Mixing tab, test phrase simplification

v1.3 — Added Preferences/Limits system, threading model, container inheritance decision

December 2025