Phonon: Focused Implementation Plan

Priority Order: 6 ▯ 1 ▯ 3 ▯ 4 ▯ 5 **Dropped**: Phase 2 (wet/dry - workaround with signal routing)

Phase 6: Interactive Tab Completion (FIRST - Foundation)

Why First?

Better developer experience makes everything else easier to build and use.

Deliverables

1. **Auto-generate completion metadata from source**
    - Parse doc comments from `src/nodes/*.rs`
    - Extract parameters from `new()` signatures
    - Generate metadata automatically (no manual maintenance)
2. **Parameter completion**

```
User types: lpf <TAB>
Shows: :cutoff Pattern :resonance Pattern=0.7

User types: limiter <TAB>
Shows: :input Signal :threshold Pattern :ceiling Pattern
```

3. **Template insertion**

```
User types: compressor <TAB> (selects template)
Inserts: compressor :input ___ :threshold ___ :ratio ___ :attack ___ :release ___
(Cursor at first ___ placeholder)
```

Implementation

Step 1: Standardize Doc Comments (All nodes)

````
/// CompressorNode - Reduces dynamic range with threshold and ratio
///
/// Classic downward compression with attack/release envelopes.
///
/// # Parameters
/// - `input`: Audio signal to compress
/// - `threshold`: Level above which compression starts (dB)
/// - `ratio`: Compression ratio (2.0 = 2:1, 10.0 = 10:1)
/// - `attack`: Attack time in seconds (default: 0.01)
/// - `release`: Release time in seconds (default: 0.1)
///
/// # Example
/// ```phonon
/// ~compressed: signal # compressor -20 4.0 0.01 0.1
/// ```
pub fn new(input: NodeId, threshold: NodeId, ratio: NodeId, attack: NodeId, release: NodeId
````

Step 2: Build-Time Parser (`build.rs`)

```
// Parse all src/nodes/*.rs files
// Extract doc comments + function signatures
// Generate src/modal_editor/completion/generated_metadata.rs
```

Step 3: Parameter Completion Logic - Detect context: `function_name <cursor>` - Show parameters with types and defaults - Support template insertion with placeholders

Files: - `build.rs` (new) - Doc comment parser - `src/modal_editor/completion/generated_metadata.rs` (generated) - `src/modal_editor/completion/parameter.rs` (new) - Parameter completion - Update all `src/nodes/*.rs` with standard doc format

Estimate: 15-20 hours - Parser: 6-8 hours - UI logic: 4-5 hours - Standardize docs: 5-6 hours (batch update all nodes) - Testing: 1 hour

---

Phase 1: Sidechain Compression (SECOND - High-Value Feature)

Deliverables

SidechainCompressorNode

```
pub struct SidechainCompressorNode {
    main_input: NodeId,         // Signal to compress
    sidechain_input: NodeId,    // Signal controlling compression
    threshold: NodeId,
    ratio: NodeId,
    attack: NodeId,
    release: NodeId,
    state: CompressorState,
}
```

Use Cases:

```
-- EDM ducking: kick ducks bass
~kick: s "bd*4"
~bass: saw 55
~ducked_bass: ~bass # sidechain_comp ~kick -10 4.0 0.01 0.1

-- Podcast: voice ducks music
~voice: s "voice_sample"
~music: saw "55 82.5"
~mixed: ~music # sidechain_comp ~voice -20 10.0 0.001 0.5
```

SidechainNoiseGateNode (Bonus - quick win)

```
pub struct SidechainNoiseGateNode {
    main_input: NodeId,
    sidechain_input: NodeId,
    threshold: NodeId,
    attack: NodeId,
    release: NodeId,
}
```

Files: - `src/nodes/sidechain_compressor.rs` - `src/nodes/sidechain_noise_gate.rs` - `tests/test_sidechain_compressor.rs` - `tests/test_sidechain_noise_gate.rs`

Estimate: 8-10 hours - SidechainCompressor: 5 hours - SidechainNoiseGate: 3 hours - Tests: 2 hours

---

Phase 3: FM Cross-Modulation (THIRD - Creative Synthesis)

Deliverable

FMCrossModNode - Use any audio signal as FM modulator

```
pub struct FMCrossModNode {
    carrier: NodeId,      // Audio to modulate
    modulator: NodeId,    // Audio doing the modulation
    mod_depth: NodeId,    // Modulation amount
}
```

Use Cases:

```
-- Drums modulating bass (rhythmic timbral changes)
~kick: s "bd*4"
~bass: saw 55
~modulated: fmcrossmod ~bass ~kick 2.0

-- LFO modulating pad (audio-rate vibrato)
~lfo: sine 8
~pad: saw "55 82.5"
~vibrato: fmcrossmod ~pad ~lfo 50.0

-- Voice modulating synth (vocoder-like effects)
~voice: s "voice"
~synth: pulse 110 0.5
~talking_synth: fmcrossmod ~synth ~voice 1.0
```

Implementation: - Phase modulation (varies instantaneous phase based on modulator) - Clean implementation: `output[i] = carrier[i] * cos(2π * mod * modulator[i])` - Different from classic FM (which uses internal oscillators)

Files: - `src/nodes/fm_crossmod.rs` - `tests/test_fm_crossmod.rs`

Estimate: 4-5 hours

Phase 4: Fundsp Individual Nodes (FOURTH - Unique Sounds)

Strategy

Port ONLY fundsp units that provide unique value (no duplication).

Fundsp Units Analysis

Unit	Have It?	Unique Value?	Action
organ_hz	□ No	□ Additive organ synthesis (unique timbre)	**PORT**
reverb_stereo	Partial (mono only)	□ True stereo reverb	**CONSIDER**
softsaw_hz	□ No	□ Softer saw (minor variation)	Skip
dlowpass_hz	□ No	□ Nonlinear lowpass (Jatin Chowdhury)	**CONSIDER**
moog_hz	□ Yes (MoogLadderNode)	□ Duplicate	Skip

Unit	Have It?	Unique Value?	Action
saw/square/tri	□ Yes (VCONode)	□ Duplicate	Skip
noise/pink	□ Yes (NoiseNode)	□ Duplicate	Skip
chorus	□ Yes (ChorusNode)	□ Duplicate	Skip

Recommendation: Port These 2

1. OrganNode (Highest Priority) - Additive synthesis with harmonics - Classic organ sound (impossible with current oscillators) - **Why**: Unique timbre, classic sound

2. NonlinearLowpassNode (dlowpass_hz) - Nonlinear filtering (drive-dependent behavior) - Jatin Chowdhury's design (high-quality) - **Why**: Different character than linear filters

Skip: StereoReverbNode (current reverb works, stereo can wait)

Implementation Pattern

```rust
// src/nodes/organ.rs
use fundsp::prelude::*;

struct OrganState {
    unit: Box<dyn AudioUnit>,  // Type-erased
    last_freq: f32,
}

impl OrganState {
    fn new(freq: f32, sample_rate: f64) -> Self {
        let mut unit = organ_hz(freq);
        unit.reset();
        unit.set_sample_rate(sample_rate);
        Self {
            unit: Box::new(unit),
            last_freq: freq
        }
    }

    fn tick(&mut self) -> f32 {
        self.unit.tick(&[]).0
    }
}

pub struct OrganNode {
    frequency: NodeId,
    state: OrganState,
}

impl AudioNode for OrganNode {
    fn process_block(&mut self, inputs: &[&[f32]], output: &mut [f32], sample_rate: f32, _c
        let freq_buffer = inputs[0];

        for i in 0..output.len() {
            // Recreate if frequency changed significantly
            if (freq_buffer[i] - self.state.last_freq).abs() > 1.0 {
                self.state = OrganState::new(freq_buffer[i], sample_rate as f64);
            }

            output[i] = self.state.tick();
        }
    }
}
```

Files (per node): - `src/nodes/organ.rs` - `src/nodes/nonlinear_lowpass.rs` - `tests/test_organ.rs` - `tests/test_nonlinear_lowpass.rs`

Estimate: 8-10 hours total - OrganNode: 4-5 hours - NonlinearLowpassNode: 4-5 hours

---

Phase 5: Legacy Code Study (FIFTH - Cleanup)

Goal

Determine if `unified_graph.rs` (14,146 lines) can be removed.

Investigation Steps

1. Check if old architecture is called

```
# Search for SignalNode usage in compiler/main
grep -r "SignalNode::" src/main.rs src/compositional_compiler.rs src/*parser*.rs

# Search for old graph construction
grep -r "UnifiedGraph::new\|add_node" src/main.rs src/compositional_compiler.rs

# Search for eval_signal calls
grep -r "eval_signal" src/main.rs src/compositional_compiler.rs
```

2. Check test dependencies

```
# See if tests depend on old architecture
grep -r "SignalNode" tests/ | wc -l
grep -r "UnifiedGraph" tests/ | wc -l
```

3. Determine status - **If unused**: Delete immediately □ Save 14K lines! - **If used**: Document usage, create migration plan - **If partially used**: Identify what still needs it

Deliverable

Report documenting: - Current usage status - Dependencies (what still uses it) - Migration plan (if needed) - OR: Pull request removing it (if safe)

Estimate: 2-3 hours - Investigation: 1 hour - Documentation/removal: 1-2 hours

---

Timeline & Estimates

Phase	Feature	Hours	Week
6	Tab completion	15-20	1-2
1	Sidechain	8-10	3
3	FM cross-mod	4-5	3-4

Phase	Feature	Hours	Week
4	Fundsp (2 nodes)	8-10	4-5
5	Legacy study	2-3	5

Total: 37-48 hours (~5 weeks at 10 hours/week)

Success Criteria

Phase 6: Tab Completion ▯

☐ All nodes have standardized doc comments
☐ Completion metadata auto-generated from source
☐ `function <TAB>` shows parameters with types/defaults
☐ Template insertion works with placeholders
☐ Zero manual metadata maintenance required

Phase 1: Sidechain ▯

☐ SidechainCompressorNode passes 10+ tests
☐ SidechainNoiseGateNode passes 8+ tests
☐ Example: kick ducking bass (clear ducking effect audible)

Phase 3: FM Cross-Mod ▯

☐ FMCrossModNode passes 8+ tests
☐ Example: drums modulating bass (timbral changes audible)
☐ Works with any audio input (not just oscillators)

Phase 4: Fundsp ▯

☐ OrganNode produces classic organ sound
☐ NonlinearLowpassNode has distinct character vs linear filters
☐ Each has 8+ tests
☐ Performance acceptable (sample-by-sample overhead < 10%)

Phase 5: Legacy Study ▯

☐ Report documents current usage
☐ Decision made: remove or migrate
☐ If removed: all tests pass, -14K lines

Key Design Decisions

1. **Tab completion FIRST** - Foundation for better DX on everything else
2. **Only 2 fundsp nodes** - Organ (unique) + Nonlinear lowpass (high quality)
3. **Skip wet/dry audit** - Workaround with signal routing for now
4. **Study before remove** - Don't break things accidentally

Next Action

Start Phase 6: Standardize doc comments across all nodes, then build parser.