Batcherbird - Product Requirements Document

Executive Summary

Batcherbird is an open-source auto-sampling tool for hardware synthesizers that automates the tedious process of creating high-quality sample libraries. Named after the butcherbird's ability to mimic other birds' songs, Batcherbird captures and processes synthesizer voices through intelligent batch sampling.

Product Vision

Create the definitive open-source solution for hardware synth sampling that rivals commercial offerings like SampleRobot while maintaining simplicity and extensibility.

Core Requirements

Functional Requirements

1. MIDI Control

- Send precise MIDI note sequences with configurable timing
- Support velocity layers (minimum 1-127 range)
- Program change and CC automation for patch selection
- Sub-millisecond timing accuracy
- Support for multi-channel MIDI devices

2. Audio Capture

- High-quality audio recording (up to 192kHz/32-bit)
- Real-time monitoring with level meters
- Automatic gain staging suggestions
- Multi-channel recording support
- Buffer overflow protection

3. Intelligent Sample Detection

- Energy-based onset detection with configurable threshold
- Spectral flux detection for complex sounds
- Automatic silence trimming
- Preserve natural attack and release characteristics
- Manual override capabilities

4. Batch Processing

- Queue-based sampling workflow
- Pause/resume functionality
- Progress persistence (resume interrupted sessions)
- Parallel processing where applicable
- Error recovery without data loss

5. Export Capabilities

- Industry-standard formats: WAV, AIFF, FLAC
- Sampler-specific formats: Kontakt, EXS24, SFZ
- Automatic file naming conventions
- Metadata embedding (root note, velocity, etc.)
- Batch export with progress tracking

Non-Functional Requirements

Performance

- Process samples in real-time during recording
- Handle sessions with 1000+ samples efficiently
- Memory usage under 500MB for typical sessions
- Start-up time under 2 seconds

Reliability

- Graceful handling of MIDI/audio device disconnection
- Automatic session backup every 10 samples
- Crash recovery with minimal data loss
- Comprehensive error logging

Usability

- Single-window interface with clear workflow
- Preset system for common synthesizers
- Keyboard shortcuts for power users
- Clear visual feedback for all operations

Technical Architecture

Technology Stack

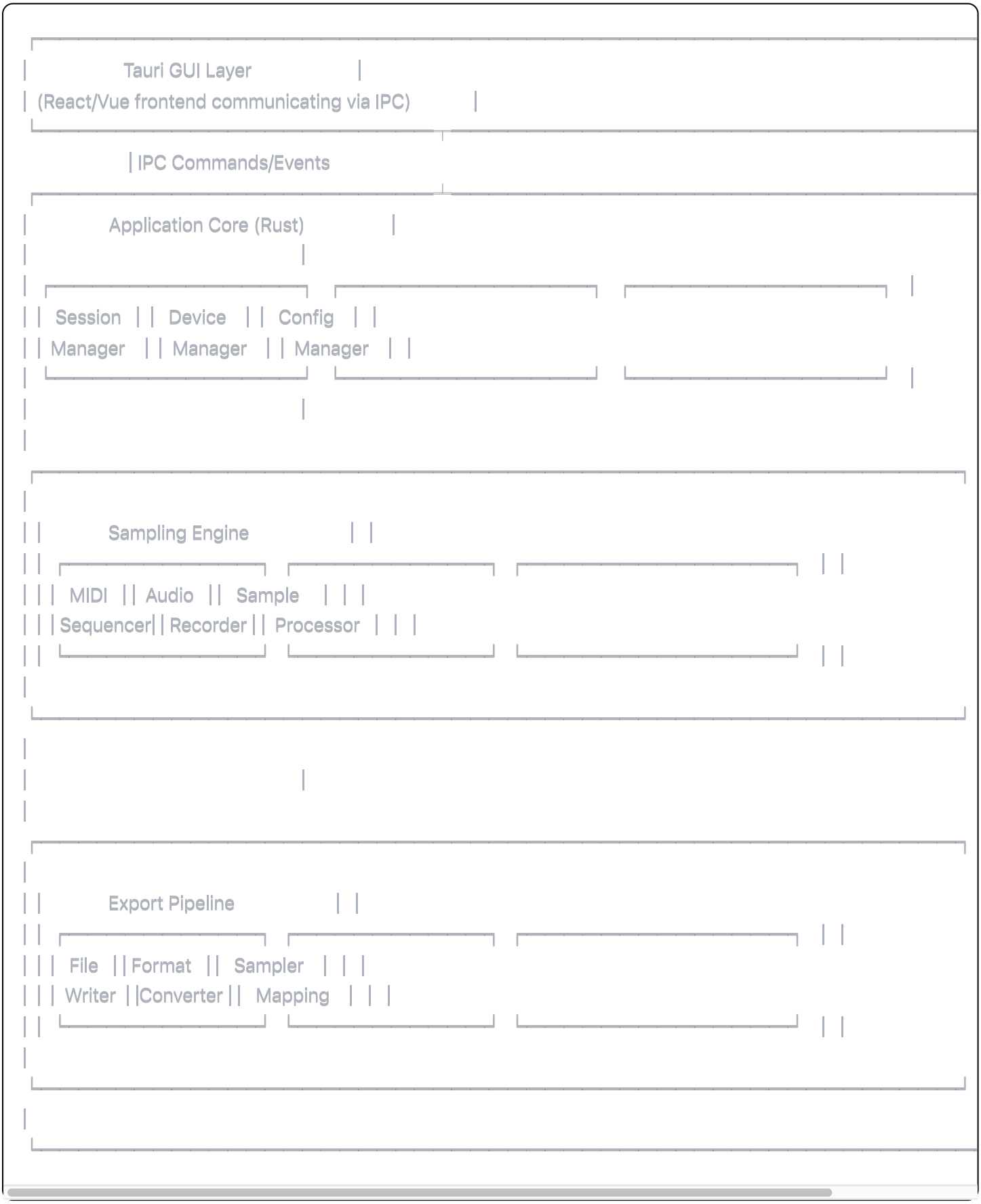
Core: Rust

- Performance-critical operations
- Cross-platform compatibility
- Single binary distribution

Key Dependencies:

```toml
midir = "0.10"        # MIDI I/O
cpal = "0.15"         # Audio I/O
hound = "3.5"         # WAV file handling
tokio = "1.35"        # Async runtime
serde = "1.0"         # Serialization
clap = "4.4"          # CLI interface
tauri = "2.0"         # GUI framework
```

System Architecture



Core Components

Session Manager

- Maintains sampling session state
- Handles persistence and recovery
- Coordinates between subsystems

- Event bus for UI updates

Device Manager

- MIDI and audio device enumeration
- Hot-plug detection
- Device capability querying
- Connection state management

Sampling Engine

- Orchestrates MIDI sequencing and audio recording
- Maintains precise timing synchronization
- Manages recording buffers
- Implements backpressure for processing pipeline

Sample Processor

- Plugin architecture for processing algorithms
- Built-in processors: onset detection, normalization, fade
- Parallel processing with thread pool
- Quality metrics calculation

Export Pipeline

- Format-specific exporters implementing common trait
- Streaming export for large sessions
- Progress reporting
- Validation before write

Data Flow

1. User Configuration
   └→ Session Manager validates and initializes session

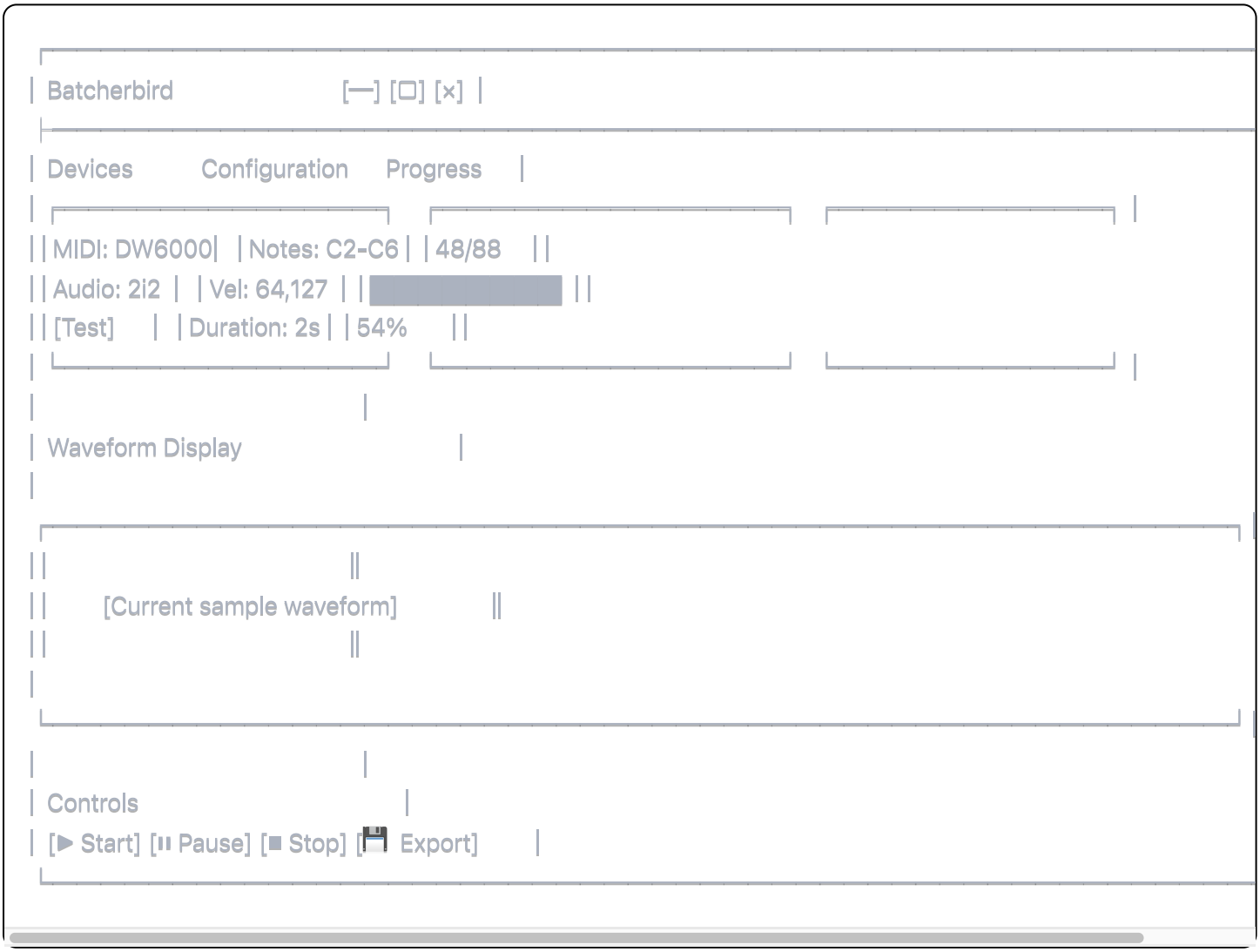
2. Sampling Loop (per note/velocity)
   ├→ MIDI Sequencer sends note-on
   ├→ Audio Recorder captures to ring buffer
   ├→ MIDI Sequencer sends note-off (after duration)
   ├→ Audio Recorder continues (release time)
   └→ Raw audio chunk queued for processing

3. Processing Pipeline (parallel)
   ├→ Onset/offset detection
   ├→ Trimming and fade application
   ├→ Normalization (if enabled)
   └→ Quality validation

4. Export Stage
   ├→ Format conversion
   ├→ Metadata injection
   ├→ File writing with verification
   └→ Sampler mapping generation

User Interface Design

Main Window Layout



Workflow States

1. **Configuration State**
   - Device selection
   - Parameter setup
   - Test note functionality
2. **Sampling State**
   - Real-time progress
   - Current note display
   - Pause/resume controls
3. **Review State**
   - Sample grid view
   - Individual sample editing
   - Batch operations
4. **Export State**
   - Format selection
   - Export progress

- Completion summary

Development Phases

Phase 1: Core Engine (MVP)

Timeline: 6 weeks

- Basic MIDI sequencing
- Audio recording to WAV
- Simple onset detection
- CLI interface
- Single velocity layer

Phase 2: Processing Pipeline

Timeline: 4 weeks

- Advanced onset detection
- Sample trimming and fades
- Normalization
- Quality validation
- Batch processing

Phase 3: GUI Implementation

Timeline: 6 weeks

- Tauri application shell
- Device management UI
- Real-time waveform display
- Progress tracking
- Session management

Phase 4: Export Formats

Timeline: 4 weeks

- Kontakt export
- SFZ export
- Sampler mapping files
- Metadata management

Phase 5: Advanced Features

Timeline: Ongoing

- Multi-velocity layers
- Round-robin sampling
- Modulation sampling
- Cloud preset sharing

Success Metrics

Performance Targets

- Sample 88 notes in under 10 minutes
- Process samples with <100ms latency
- Export 1GB of samples in <30 seconds

Quality Targets

- 99.9% successful sample capture rate
- <1ms timing jitter on MIDI events
- Bit-perfect audio capture

Adoption Targets

- 1000 GitHub stars in first year
- 50+ contributed synthesizer presets
- Active community of 100+ users

Risk Mitigation

Technical Risks

- **MIDI timing on different OS**: Extensive testing matrix
- **Audio driver compatibility**: Fallback to OS defaults
- **Large session memory usage**: Streaming architecture

User Experience Risks

- **Complex workflow**: Guided mode for beginners
- **Device configuration**: Auto-detection with manual override
- **Loss of work**: Aggressive auto-save strategy

Open Source Strategy

Repository Structure

```
batcherbird/
├── CONTRIBUTING.md      # Contribution guidelines
├── ARCHITECTURE.md      # Technical documentation
├── crates/
│   ├── batcherbird-core/    # Core library
│   ├── batcherbird-gui/     # GUI application
│   └── batcherbird-cli/     # CLI application
├── presets/             # Community presets
└── docs/                # User documentation
```

Community Engagement

- Discord server for real-time support
- Monthly development updates
- Preset sharing platform
- Video tutorials and documentation

Licensing

- Dual license: MIT/Apache 2.0
- CLA for major contributions
- Patent grant for contributors

Appendix: Technical Specifications

Audio Specifications

- Sample rates: 44.1, 48, 88.2, 96, 176.4, 192 kHz
- Bit depths: 16, 24, 32 bit integer, 32 bit float
- Channel configurations: Mono, Stereo, Multi-channel

MIDI Specifications

- Full MIDI 1.0 compliance
- 1ms timing resolution
- Multi-port support
- SysEx capability for device control

File Format Specifications

- WAV: RIFF/WAVE with BWF metadata

- AIFF: Standard AIFF-C format
- Kontakt: NKI 5.0+ compatibility
- SFZ: Version 2.0 specification

Configuration Schema

toml

```toml
# batcherbird.toml
[project]
name = "DW6000 Factory Presets"
author = "Your Name"
version = "1.0"

[midi]
device = "Korg DW6000"
channel = 1
program_change_delay_ms = 50

[audio]
device = "Scarlett 2i2"
sample_rate = 48000
bit_depth = 24
channels = 2

[sampling]
note_range = { start = 36, end = 84 }  # C2 to C6
velocities = [64, 96, 127]
note_duration_ms = 2000
release_time_ms = 1000
pre_delay_ms = 100

[processing]
detect_onset = true
onset_threshold = 0.001
detect_offset = true
offset_threshold = 0.0001
normalize = true
fade_in_ms = 2
fade_out_ms = 10

[export]
format = "wav"
naming_pattern = "{synth}_{patch}_{note}_{velocity}"
output_directory = "./samples"
create_mapping = true
```

API Contracts

Core Trait Definitions

rust

```rust
// Sample processor trait for plugin architecture
pub trait SampleProcessor: Send + Sync {
    fn process(&self, sample: &mut Sample) -> Result<()>;
    fn name(&self) -> &str;
    fn description(&self) -> &str;
}

// Exporter trait for format plugins
pub trait Exporter: Send + Sync {
    fn export(&self, session: &Session, path: &Path) -> Result<()>;
    fn file_extension(&self) -> &str;
    fn format_name(&self) -> &str;
}

// Session state for persistence
#[derive(Serialize, Deserialize)]
pub struct SessionState {
    pub id: Uuid,
    pub config: SamplingConfig,
    pub completed_samples: Vec<SampleInfo>,
    pub current_note: Option<u8>,
    pub current_velocity: Option<u8>,
    pub started_at: DateTime<Utc>,
    pub last_updated: DateTime<Utc>,
}
```

Error Handling Strategy

rust

```rust
#[derive(Error, Debug)]
pub enum BatcherbirdError {
    #[error("MIDI device error: {0}")]
    MidiError(#[from] midir::InitError),

    #[error("Audio device error: {0}")]
    AudioError(#[from] cpal::BuildStreamError),

    #[error("Sample processing error: {0}")]
    ProcessingError(String),

    #[error("Export error: {0}")]
    ExportError(#[from] std::io::Error),

    #[error("Configuration error: {0}")]
    ConfigError(#[from] toml::de::Error),
}
```

MIDI/Audio Synchronization Strategy

1. **Pre-roll Recording**: Start audio recording 100ms before MIDI note
2. **Ring Buffer**: Use 10-second ring buffer for continuous recording
3. **Timestamp Alignment**: Mark MIDI events with system timestamps
4. **Latency Compensation**: User-configurable offset in milliseconds

Sample File Naming Convention

Default pattern: `{synth}_{patch}_{note}_{velocity}.wav`

Variables:

- `{synth}`: Synthesizer name (from config)
- `{patch}`: Current patch name or number
- `{note}`: MIDI note number
- `{note_name}`: Note name (e.g., "C3")
- `{velocity}`: MIDI velocity value
- `{date}`: Current date (YYYYMMDD)
- `{time}`: Current time (HHMMSS)

Testing Strategy

Unit Tests

- Sample detection algorithms
- MIDI message generation
- Audio buffer management
- Configuration parsing

Integration Tests

- MIDI loopback tests
- Audio recording with virtual devices
- Full sampling session simulation
- Export format validation

Hardware Testing Matrix

- OS: macOS 12+, Windows 10+, Ubuntu 22.04+
- Audio Interfaces: Test with 5+ popular models
- MIDI Interfaces: USB and DIN MIDI testing
- Synthesizers: Community-provided test results

Build and Release Process

Build Targets

- macOS: Universal binary (Intel + Apple Silicon)
- Windows: x64 binary with installer
- Linux: AppImage and .deb packages

CI/CD Pipeline

```yaml
# GitHub Actions workflow
- Rust formatting and linting
- Unit and integration tests
- Binary building for all platforms
- Automatic draft release creation
- Community preset validation
```

Minimum System Requirements

- **OS**: macOS 12+, Windows 10+, Linux (kernel 5.10+)
- **RAM**: 4GB minimum, 8GB recommended
- **Storage**: 100MB for application, 1GB+ for samples

- **CPU**: 2-core x64 processor
- **Audio**: ASIO/Core Audio/ALSA compatible interface
- **MIDI**: Class-compliant MIDI interface

Onset Detection Algorithm (Initial Implementation)

Energy-Based Detection with Spectral Flux

1. Calculate RMS energy in 10ms windows
2. Compute spectral flux between consecutive FFT frames
3. Combine energy and flux with weighted sum
4. Apply median filter to smooth detection function
5. Find peaks above adaptive threshold
6. Backtrack 5ms from peak for precise onset

GUI-Core Communication Protocol

Tauri IPC Commands

typescript

```
// Frontend → Backend Commands
interface Commands {
  // Device management
  list_midi_devices(): MidiDevice[]
  list_audio_devices(): AudioDevice[]
  test_connection(midi: string, audio: string): TestResult

  // Session control
  start_session(config: SessionConfig): SessionId
  pause_session(id: SessionId): void
  resume_session(id: SessionId): void
  stop_session(id: SessionId): void

  // Sample management
  get_sample_preview(id: SampleId): WaveformData
  update_sample_trim(id: SampleId, start: number, end: number): void

  // Export
  export_session(id: SessionId, format: ExportFormat): Progress
}

// Backend → Frontend Events
interface Events {
  session_progress: { current: number, total: number, note: string }
  sample_captured: { id: SampleId, note: number, velocity: number }
  device_connected: { type: 'midi' | 'audio', name: string }
  device_disconnected: { type: 'midi' | 'audio', name: string }
  error: { code: string, message: string, recoverable: boolean }
}
```

Session Persistence Format

json

```
{
  "version": "1.0",
  "session": {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "created": "2024-01-20T10:30:00Z",
    "updated": "2024-01-20T11:45:00Z",
    "config": {
      // Full sampling configuration
    },
    "state": "in_progress",
    "progress": {
      "completed_samples": 45,
      "total_samples": 88,
      "current_note": 72,
      "current_velocity": 127
    }
  },
  "samples": [
    {
      "id": "sample_001",
      "note": 60,
      "velocity": 96,
      "filename": "samples/DW6000_init_60_96.wav",
      "metrics": {
        "peak_level": -3.2,
        "rms_level": -18.5,
        "duration_ms": 2847,
        "onset_ms": 12,
        "offset_ms": 2835
      },
      "processing": {
        "trimmed": true,
        "normalized": true,
        "faded": true
      }
    }
    // ... more samples
  ]
}
```