# Technical Architecture and Development Timeline: IDTAP

## Data Model and Core Entities

IDTAP is built around a sophisticated data model designed to represent complex musical relationships in up to four simultaneous instrumental tracks while maintaining precise temporal and harmonic relationships.

### Core Musical Entities

**Trajectory** represents the fundamental unit of musical expression. Each trajectory contains arrays of Pitch objects, duration arrays for temporal subdivision, and articulation dictionaries for performance techniques. The system includes 14 distinct types (IDs 0-13) including pitch-stable trajectories, various slide types with adjustable slopes, vibrato patterns, and ornaments. Each trajectory stores pitch data as both frequency values and logarithmic representations.

**Pitch** objects encode microtonal information using swara (scale degree), octave, and raised/lowered boolean flags, with computed properties for frequency, logarithmic frequency, and various representational formats. The system supports arbitrary fundamental frequencies and custom tuning ratios.

**Phrase** serves as the organizational unit containing trajectory arrays across multiple tracks, chikari objects for sympathetic strings, and analytical group arrays. The system supports both structured labeling (8 phrase types like Mohra, Asthai, Antara; 13 elaboration types like Vistar, Tan, Tihai) and ad-hoc labeling for flexible annotation.

**Piece** coordinates multiple phrase arrays across instruments while maintaining temporal synchronization. Each track contains phrase arrays, duration arrays, section markers, and category arrays. Pieces maintain relationships to Raga objects for harmonic context and Meter objects for rhythmic analysis.

**Raga** encapsulates harmonic frameworks through rule sets specifying active scale degrees, tuning ratios for microtonal precision, and fundamental frequencies. Supports both equal temperament and just intonation systems.

**Meter** provides sophisticated rhythmic analysis through hierarchical pulse structures supporting up to four temporal layers. Contains hierarchy arrays for patterns, tempo specifications, and PulseStructure objects organized by layer.

### Database Architecture

MongoDB implementation with 13 primary collections handles complex relational needs while maintaining query performance:

**Core Content Collections:** - **transcriptions**: Complete musical transcription documents with nested phrase arrays, trajectory data, duration arrays, section categorization, timing data, metadata, instrumentation arrays, raga objects, and comprehensive permission structures - **audioRecordings**: Individual audio file metadata including duration, SA frequency estimates, octave offsets, musician mappings, hierarchical date/location objects, raga timing information, and parent references to audioEvents - **audioEvents**: Organizational containers for grouped recordings with nested structures using dynamic keys, event type classification, and cascading update/delete operations

**User and Organizational Collections:** - **users**: Google OAuth integration with profile data, waiver agreements, owned transcription arrays, saved multiQueries, display preferences, and access history tracking - **collections**: User-curated content organization with purpose enumeration (Pedagogical, Research, Creative, Personal), permission objects, content arrays, and invite code systems

**Reference and Lookup Collections:** - **musicians**: Biographical data with hierarchical name structures, birth/death years, gharana affiliations, gender classification, primary instrument, and comprehensive instrument arrays - **ragas**: Musical scale/mode definitions with name identifiers, rules objects for scale degree configurations, and update tracking - **instruments**: Reference data with name and kind classifications supporting instrument type filtering and validation - **gharanas**: Musical school/lineage information with name identifiers and members arrays for genealogical context - **phonemes**: Comprehensive phonetic data with type classification, IPA symbols, English transliterations, and Hindi/Urdu representations - **location**: Hierarchical geographic structures with dynamic continent/country/city organization

**Specialized Collections:** - **audioEventTypes**: Standardized terminology for event classification - **performanceSections**: Performance structure annotation supporting consistent metadata entry

The architecture implements aggregation-based derived collections, bidirectional reference management, complex permission systems, and hybrid embedding/referencing strategies.

### Type System

Comprehensive TypeScript interfaces in shared/types.ts ensure cross-language compatibility with 200+ interfaces covering musical elements, display types, analysis structures, and database schemas. The Python API porting project systematically translates TypeScript classes into Python equivalents for computational research.

## Technical Stack and Infrastructure

### Deployment Architecture and System Interconnectivity

IDTAP operates as a distributed web application deployed across multiple cloud infrastructure providers to ensure scalability, reliability, and optimal performance. The system architecture follows a three-tier deployment model with clear separation of concerns between presentation, application logic, and data persistence layers.

The **frontend application** is developed using Vue.js 3 with TypeScript and deployed via Vite’s optimized build system. Vite provides hot module replacement during development and sophisticated tree-shaking and code-splitting for production builds, resulting in minimal bundle sizes and optimal loading performance. The application is served through a content delivery network (CDN) to ensure global accessibility and reduced latency for users worldwide.

The **backend server** runs on a DigitalOcean virtual Linux machine (Ubuntu 20.04 LTS) hosting a Node.js Express application. This server acts as the central coordination layer, managing API endpoints, user authentication, file processing, and orchestrating communication between the frontend application and various backend services. The Express server implements comprehensive middleware for request logging, error handling, CORS management, and security headers, ensuring robust production deployment.

**Database services** utilize MongoDB Atlas, MongoDB’s fully managed cloud database service, providing automatic scaling, backup management, and global distribution capabilities. The Atlas deployment includes replica sets for high availability and automated failover, ensuring continuous service availability. Database connections are managed through MongoDB’s native Node.js driver with connection pooling and automatic reconnection handling.

### Programming Languages and Core Technologies

The IDTAP platform employs a polyglot architecture utilizing different programming languages optimized for specific computational requirements:

**JavaScript/TypeScript** serves as the primary development language for both frontend and backend components. The frontend utilizes TypeScript exclusively for type safety and enhanced developer experience, while the backend combines JavaScript and TypeScript depending on component requirements. TypeScript’s static typing system ensures consistency across the large codebase and enables confident refactoring and maintenance.

**Python** handles computationally intensive audio processing tasks that require specialized signal processing libraries. Python scripts are spawned as child processes from the Node.js server for tasks including pitch extraction, spectrogram generation, and melograph creation. The Python environment includes scientific computing libraries such as NumPy for numerical operations, Essentia for audio analysis, and Librosa for music information retrieval tasks.

**Web Assembly (WASM)** integration enables high-performance audio processing directly in the browser. The RubberBand time-stretching library is compiled to WASM for real-time audio manipulation without server-side processing requirements.

### Frontend Framework and Development Environment

The frontend application is built using **Vue.js 3** with the Composition API, providing a reactive and component-based architecture. Vue’s reactivity system enables efficient rendering of complex musical data structures with minimal performance overhead. The application leverages **Vuex** for centralized state management, handling complex application state including user authentication, transcription data, audio playback state, and analysis results.

**Vite** serves as the build tool and development server, providing near-instantaneous hot module replacement during development and optimized production builds. Vite’s ES module-based architecture eliminates the need for complex bundling during development, significantly improving developer productivity and build performance.

The component architecture utilizes **Vue Router** for single-page application navigation, implementing lazy loading for route-based code splitting. This approach ensures that users only download JavaScript code relevant to their current workflow, reducing initial page load times and improving perceived performance.

### Backend Infrastructure and API Design

The backend utilizes **Node.js** with the **Express** framework, providing a lightweight and scalable foundation for RESTful API development. Express middleware handles authentication via **Google OAuth 2.0**, implementing secure token-based authentication with refresh token rotation and session management.

**Mongoose** serves as the Object Document Mapper (ODM) for MongoDB integration, providing schema validation, query building, and relationship management. The ODM layer implements custom validation rules for musical data structures and automatic population of related documents through sophisticated aggregation pipelines.

File upload handling utilizes **Multer** middleware with custom storage engines for audio file processing. The system supports multiple audio formats (WAV, MP3, FLAC) with automatic conversion via **FFmpeg** integration. Audio processing includes peak normalization, format standardization, and metadata extraction for database storage.

### Major Libraries and Framework Integration

**D3.js** powers the sophisticated data visualization system throughout the platform. The transcription editor utilizes D3’s path generation, zoom behavior, and event handling for creating responsive musical notation displays. D3’s scale systems enable logarithmic frequency representation essential for microtonal music visualization. Analysis components leverage D3’s statistical functions, color interpolation, and SVG manipulation for generating publication-quality charts and graphs.

**Web Audio API** integration enables real-time audio synthesis and processing directly in the browser. Custom **AudioWorklet** implementations provide high-performance audio processing with sub-sample precision timing. The system implements three specialized synthesis engines: Karplus-Strong physical modeling for plucked strings, dual delay line synthesis for bowed instruments, and Klatt formant synthesis for vocal modeling.

**Web Workers** handle computationally intensive tasks without blocking the user interface. The spectrogram processing worker implements sophisticated algorithms for real-time scaling, intensity adjustment, and colorization using D3 color maps. Priority queuing ensures visible spectrogram regions are rendered first while background processing continues for off-screen areas.

**Intersection Observer API** enables lazy loading optimization for large musical content, ensuring only visible portions of spectrograms and transcriptions are actively rendered. This optimization is crucial for handling hour-long audio recordings without performance degradation.

### Database Technology and Data Management

**MongoDB Atlas** provides the primary data persistence layer with sophisticated aggregation pipelines for derived data generation. The system implements compound indexing strategies for complex musical queries, enabling sub-second search times across large transcription databases. The database architecture balances document embedding for frequently accessed related data with references for stable entities.

Collection design implements **aggregation-based derived collections** where audioRecordings are generated from audioEvents through complex pipelines that flatten hierarchical data while preserving relationships. This approach provides benefits of both hierarchical organization and flat query performance for analytical workflows.

The permission system implements **role-based access control** with invite-based sharing and audit logging for collaborative research workflows. Database operations include automated maintenance through scheduled aggregation tasks and cascading update/delete operations for maintaining data consistency.

### Audio Processing and Analysis Pipeline

**Python-based audio analysis** utilizes **Essentia** for music information retrieval, implementing constant-Q transforms for logarithmic frequency analysis with 72 bins per octave for microtonal precision. The system employs multiple concurrent pitch detection algorithms including PitchYin for fundamental frequency estimation and PredominantPitchMelodia for polyphonic melody extraction.

**Librosa** provides additional signal processing capabilities for pitch manipulation and time-domain processing. The pitch shifting implementation uses phase vocoder-based processing with 2048-point FFT and 512-sample hop length, supporting ±200 cents adjustment while maintaining temporal characteristics.

**NumPy** handles numerical operations and array processing for spectral data manipulation. Generated spectral data is compressed using **pako** (gzip) compression and served through CDN distribution for optimal web performance.

This comprehensive technical stack enables IDTAP to provide sophisticated musical analysis capabilities through a web-based interface while maintaining the performance and reliability required for academic research applications.

## Technical Architecture Overview

### Platform Philosophy

IDTAP’s core innovation is trajectory-based transcription replacing fixed-pitch notes with “trajectories”—archetypal paths between pitches that capture finely calibrated glissandi. This middle-level representation authentically captures microtonal inflections and expressive nuances while avoiding over-granular frequency data and over-simplified MIDI representation.

### Frontend Architecture

Vue.js 3 application with TypeScript, organized around sophisticated component-based architecture supporting multiple concurrent workflows. Uses Composition API, Vuex state management, and Vue Router.

**Editor Components**: EditorComponent coordinates specialized sub-components through event-driven architecture. Renderer manages visual display on logarithmic frequency scale with D3.js visualizations. TranscriptionLayer handles trajectory placement through sophisticated mouse/keyboard interactions. SpectrogramLayer and MelographLayer provide algorithmic pitch analysis with Web Workers and WebGL acceleration.

**Analysis Components**: AnalyzerComponent provides query systems for filtering transcriptions by musical criteria across 15 categories. Implements pitch prevalence visualization and pattern analysis with substring matching algorithms.

**Advanced Technologies**: Web Audio API integration powers real-time synthesis through custom AudioWorklets. D3.js enables interactive visualizations. Web Workers handle computationally intensive tasks. Intersection Observer API enables lazy loading optimization.

### Backend Infrastructure

Node.js Express server coordinates between web application, MongoDB, and Python audio processing. Implements microservices-inspired patterns with monolithic deployment.

**Database Management**: MongoDB with sophisticated aggregation pipelines for derived data. Compound indexing for complex queries with sub-second search times.

**Audio Processing Pipeline**: Spawns Python scripts for pitch extraction, spectrogram generation, and melograph creation. Implements job queuing with progress reporting.

**Analysis Services**: Powers query system with full-text indexing, structural indexing, and geospatial indexing.

### Audio Synthesis System

Custom AudioWorklet processors provide real-time, low-latency synthesis for transcription verification:

**Karplus-Strong Algorithm**: Circular delay buffer implementation with first-order lowpass filtering for realistic string damping. Supports real-time frequency modulation.

**Sarangi Synthesis**: Dual delay line architecture with cascaded filtering and resonant filtering at specific frequencies simulating sympathetic strings.

**Klatt Formant Synthesizer**: 47 controllable parameters including six formant frequencies. Supports vowel morphing through 15 configurations for phonetic transcription.

**Dynamic Deployment**: Automatically deploys appropriate engines based on instrumentation. Includes chikari modeling for sympathetic strings and capture system for looping playback.

### Python Audio Analysis

**Spectrogram Generation**: Essentia’s NSGConstantQ algorithm for 72 bins per octave with hierarchical compression.

**Multi-Algorithm Pitch Extraction**: PitchYin, PitchYinProbabilistic, and PredominantPitchMelodia with sub-10ms temporal resolution.

**Tonic Estimation**: Essentia’s TonicIndianArtMusic algorithm for automatic tonic detection with raga tuning integration.

## Development Timeline

### Phase 1: Foundation (2021-Early 2022)

Established theoretical framework and basic web implementation. Core Vue.js architecture, initial trajectory data structures, and basic audio integration using Web Audio API.

### Phase 2: Full-Time Development (Summer 2022)

Dramatic acceleration with editor redesign, phrase management, and synthesis system implementation. Achieved drag-and-drop trajectory manipulation, multi-instrument support, and custom synthesis engines.

### Phase 3: Feature Expansion (Late 2022-2023)

Developed collections system with role-based access control, comprehensive analysis suite with query system and pattern detection, multi-instrument support expansion, and Python integration for audio processing.

### Phase 4: Refinement and Testing (2024)

Established comprehensive test coverage with Vitest and Pytest, initiated Python API porting project, implemented performance optimizations including WebGL acceleration and lazy loading, and developed documentation infrastructure.

### Current State

IDTAP now provides real-time synthesis verification, flexible microtonal tuning systems, and comprehensive analytical capabilities. The trajectory-based system successfully captures musical nuances impossible to notate conventionally.

The platform demonstrates that building from first principles can produce more authentic solutions for representing non-Western cultural practices in digital environments, while showing the potential for browser-based applications to handle traditionally desktop-bound computational tasks.