Skald: A CodeGen Audio Tool for Odin

Technical Report and Implementation Plan v2.0

Document Version: 2.0
Date: June 12, 2024
Status: DRAFT
Author: Senior Technical Architect

1. Introduction

Skald is a visual, node-based development tool designed to accelerate the creation of complex audio processing graphs for the Odin programming language. The project's core vision is to provide audio engineers, game developers, and creative coders with an intuitive graphical interface to design, prototype, and generate high-performance, boilerplate-free Odin audio code. Users can visually construct signal chains, manipulate parameters, and instantly generate the underlying source code, drastically reducing manual effort and the potential for errors.

This document outlines the revised technical architecture for Skald. The project is now architected as a decoupled system, comprising a modern desktop frontend and a focused, headless backend. The frontend is a desktop application built with **Electron, TypeScript, and React**, providing a rich, interactive user experience. The backend is a lightweight, command-line **Odin application** (skald_codegen) that functions as a pure code generation engine. Audio previewing is handled directly within the frontend using the browser-native **Web Audio API**, enabling rapid, iterative sound design. This decoupled approach maximizes development velocity, ensures a superior user experience, and provides a clear path for future expansion into a web-based platform.

2. Rationale for Technology Selection

The choice of technologies for Skald is driven by a strategy that plays to the strengths of each component, optimizing for user experience, development speed, and code generation robustness.

2.1. Odin for Headless Code Generation Engine

Odin remains the ideal choice for the core code generation engine, but its role has been focused into a small, headless command-line interface (CLI) tool. This approach leverages Odin's key strengths:

- **Performance:** The Odin compiler produces highly optimized, native binaries, ensuring that the code generation process is nearly instantaneous, even for large and complex audio graphs.
- **Low-Level Control:** Direct memory management and a clear, explicit syntax give us precise control over the structure and quality of the generated code.

- **AST Manipulation:** Odin's robust metaprogramming and introspection capabilities are critical. The ability to natively parse and manipulate Abstract Syntax Trees (AST) allows the codegen engine to perform complex, context-aware code construction, ensuring the output is not just text but is syntactically correct and idiomatic Odin code.

2.2. Electron, TypeScript, and React for User Interface

For the graphical user interface (GUI), we have selected a modern web technology stack, which offers significant advantages over native UI toolkits for this application.

- **Rapid Development:** The vast ecosystem of libraries, tools, and community support for React and TypeScript dramatically accelerates the development lifecycle.
- **Superior UI/UX:** HTML and CSS provide unparalleled control over layout, styling, and responsiveness, enabling the creation of a polished, modern, and aesthetically pleasing user interface that is difficult to achieve with traditional immediate-mode GUI libraries.
- **Purpose-Built Libraries:** The availability of mature, specialized libraries is a key factor. **React Flow** is a feature-rich, production-ready library specifically designed for building node-based editors. Leveraging it saves thousands of hours of development time and provides a robust foundation for our core UI.

2.3. Web Audio API for Real-time Audio Preview

The Web Audio API, a standard feature in all modern browsers (and thus in Electron), provides the audio processing backend for the frontend application.

- **Fast, Interactive Prototyping:** It enables real-time audio synthesis and processing directly within the UI. This allows users to hear an *approximation* of their graph's output instantly as they connect nodes and tweak parameters, providing a tight feedback loop for creative exploration.
- **Cross-Platform Consistency:** The API works identically across Windows, macOS, and Linux. This simplifies development and ensures a consistent user experience.
- **Future-Proofing:** By using a browser-native API, the entire audio previewing system is immediately portable to the future web application version of Skald with no modification required.

3. Core Concepts and Architecture

3.1. Project Philosophy

- **Visual First:** The primary mode of interaction is the visual graph.
- **Immediate Feedback:** Users should see and hear changes instantly.
- **Clean Code Output:** The generated code must be clean, readable, and performant.

3.2. User Experience Flow

1. **Construct:** The user adds and connects audio nodes (oscillators, filters, effects) on a canvas.
2. **Configure:** The user selects nodes to edit their parameters in a dedicated panel.
3. **Preview:** The user clicks a "Play" button to hear a real-time approximation of the audio

output via the Web Audio API.
4. **Generate:** The user clicks "Generate Code." The application converts the visual graph into a JSON structure.
5. **Compile:** The JSON is passed to the Odin CLI, which generates the corresponding Odin source code.
6. **Integrate:** The user copies the generated code into their Odin project.

3.3. Software Architecture

The Skald application is composed of two distinct processes that communicate via a well-defined contract.

- **Frontend (Electron/React Application):** This is the user-facing application. Its responsibilities include:
    - Rendering the entire graphical user interface, including the node editor, parameter panels, and menus.
    - Managing the state of the audio graph using React Flow.
    - Implementing the audio preview system using the Web Audio API.
    - Serializing the graph state into a JSON object.
    - Invoking the backend CLI process and capturing its output.
    - Displaying the generated code to the user.
    - Handling project file I/O (saving/loading the graph as a .json file).
- **Backend (Odin CLI - skald_codegen):** This is a headless, stateless command-line tool. Its sole responsibility is:
    - To read a JSON string representing an audio graph from its standard input (stdin).
    - To parse this JSON and build an internal representation of the graph.
    - To traverse the graph and generate valid, high-performance Odin source code.
    - To print the generated source code to its standard output (stdout).

3.3.1. Frontend-Backend Communication

The contract between the two components is simple and robust, relying on standard inter-process communication:

1. **Invocation:** The Electron application spawns the skald_codegen executable as a child process.
2. **Data Transfer (Write):** The frontend serializes the React Flow graph state into a JSON string. This string is then written to the stdin stream of the skald_codegen process.
3. **Data Transfer (Read):** The frontend listens to the stdout stream of the skald_codegen process. The Odin tool performs its code generation and prints the resulting Odin code to stdout. The frontend reads this stream until it closes.
4. **Error Handling:** Any errors encountered by the Odin CLI (e.g., malformed JSON, invalid graph logic) will be written to stderr. The frontend will listen to this stream and display any error messages to the user in a modal or notification panel.

3.4. Key Data Structures

The primary data structure is the audio graph, which will be defined as a set of TypeScript interfaces in the frontend. This same structure is serialized to JSON to be consumed by the Odin backend.

```
// Example TypeScript interfaces defining the JSON contract

interface NodeParameter {
  id: string;
  name: string;
  value: number | string;
  type: 'float' | 'int' | 'string';
}

interface AudioNode {
  id: string;         // Unique ID (e.g., from React Flow)
  type: string;       // Type of node (e.g., 'oscillator', 'lowpass_filter')
  position: { x: number; y: number };
  parameters: NodeParameter[];
}

interface Edge {
  id: string;
  sourceNodeId: string;
  sourceHandleId: string; // e.g., 'output_audio'
  targetNodeId: string;
  targetHandleId: string; // e.g., 'input_frequency'
}

interface AudioGraph {
  nodes: AudioNode[];
  edges: Edge[];
  // Global settings like sample rate can be included here
  metadata: {
    sampleRate: number;
  };
}
```

3.5. Visual Node Graph

The visual graph is the centerpiece of the UI. Implemented with **React Flow**, it will support standard interactions: node dragging, panning, zooming, and creating connections (edges) between node handles.

3.6. Code Generation Engine

The code generation engine *is* the Odin application. It's a focused tool that performs a single task perfectly. It will contain parsers for the JSON structure, logic for topologically sorting the graph to ensure correct processing order, and a set of templates or builders for generating the Odin code for each supported node type. It does not contain any UI, audio, or state management logic.

4. Key Features

The core feature set remains unchanged, but their implementation is now mapped to the new architecture.

- **Visual Node Graph Editor:** Implemented using the **React Flow** library within the React/Electron application. This provides a professional-grade canvas for all node-based interactions.
- **Extensive Node Library:** Node definitions and their corresponding code generation logic will be managed primarily in the Odin backend. The frontend will fetch a manifest of available nodes to populate the UI.
- **Real-time Parameter Editing:** A dedicated React component will display the parameters for the currently selected node. State changes will be managed by React's state hooks.
- **Instant Audio Preview:** A service within the React application will translate the React Flow graph into a corresponding Web Audio API graph, enabling real-time playback.
- **One-Click Code Generation:** A "Generate Code" button in the UI triggers the stdin/stdout communication process with the backend Odin CLI. The resulting code is displayed in a read-only code viewer component with a "Copy to Clipboard" feature.
- **Project Management:** The Electron frontend will use Node.js's fs module to handle saving and loading the graph state to and from the local filesystem as a .json file.

5. Implementation Plan

The project will be executed in a series of focused phases.

- **Phase 1: Odin CodeGen CLI Foundation**
  - Define the canonical graph.json format.
  - Create the skald_codegen Odin project.
  - Implement logic to read a hardcoded graph.json file from disk.
  - Implement the core codegen logic for a few basic nodes (e.g., Sine Oscillator, Gain, Output).
  - Refactor to read from stdin and write to stdout.
  - Write unit tests to validate output for sample JSON inputs.
- **Phase 2: Electron & React UI Foundation**
  - Set up a new project with Electron Forge, TypeScript, and React templates.
  - Install and configure react-flow-renderer.
  - Create the main application layout (e.g., sidebar for nodes, main canvas, parameter panel).
  - Render a basic, empty React Flow canvas.
- **Phase 3: Interactive Node Editor**
  - Implement UI logic for adding new nodes to the canvas from a library panel.

    - Enable connecting, moving, and deleting nodes and edges.
    - Create the parameter editing panel, which dynamically displays controls for the selected node.
    - Ensure all graph state changes are captured correctly in the React Flow state object.
- **Phase 4: Frontend-Backend Integration**
    - Implement the "Generate Code" button logic.
    - Add the function to serialize the React Flow state into the agreed-upon JSON format.
    - Write the Node.js logic to invoke skald_codegen, pipe the JSON to its stdin, and capture its stdout and stderr.
    - Create a code preview component (e.g., using react-syntax-highlighter) to display the captured output.
- **Phase 5: Audio Preview & Project I/O**
    - Develop a "Graph Interpreter" service that maps the React Flow state to a Web Audio API node graph.
    - Implement play/stop controls that connect/disconnect the generated Web Audio graph from the destination.
    - Implement "Save" and "Load" menu items that use Electron's dialogs to read/write the graph's JSON state to a file.
- **Phase 6: Web Application Deployment (Future Goal)**
    - **Task 1: Server Scaffolding:** Create a Node.js/Express.js server.
    - **Task 2: API Endpoint:** Create a /api/generate endpoint that accepts the graph JSON in its request body.
    - **Task 3: CLI Wrapper:** The endpoint handler will invoke the pre-compiled Linux binary of skald_codegen, pass the request body to it, and return the generated code in the API response.
    - **Task 4: Authentication:** Implement a user authentication layer (e.g., using Passport.js or a third-party service like Auth0) to protect the API and user projects.
    - **Task 5: Deployment Strategy:** Define a Dockerfile that bundles the Node.js server, the static React build artifacts, and the compiled Odin CLI binary. Outline the process for deploying this container to a cloud provider (e.g., AWS Fargate, Google Cloud Run).

6. Jira Storyboard

Epic	**User Story**	**Task**
SKALD-E1: Odin CodeGen CLI	As a developer, I want a CLI that transforms a JSON graph into Odin code so I can automate code generation.	SKALD-1: Define JSON schema. SKALD-2: Implement stdin reader. SKALD-3: Implement codegen

		for 'Oscillator'. SKALD-4: Implement codegen for 'Filter'. SKALD-5: Print result to stdout.
SKALD-E2: React/Electron UI Foundation	As a user, I want a desktop app with a node canvas so I can start building my audio graph.	SKALD-6: Set up Electron/TS/React project. SKALD-7: Install & configure React Flow. SKALD-8: Create main window layout.
SKALD-E3: Interactive Node Editor	As a user, I want to add, connect, and configure nodes so I can design my signal chain.	SKALD-9: Implement "Add Node" from a list. SKALD-10: Implement node/edge deletion. SKALD-11: Create parameter editing panel.
SKALD-E4: Frontend-Backend Integration	As a user, I want to click a button and see the generated Odin code so I can use it in my project.	SKALD-12: Implement state-to-JSON serialization. SKALD-13: Create Node.js child process invoker. SKALD-14: Create code preview panel.
SKALD-E5: Audio Preview & Project I/O	As a user, I want to hear my graph and save my work so I can iterate on my designs.	SKALD-15: Build Web Audio graph interpreter. SKALD-16: Add play/stop UI controls. SKALD-17: Implement Save/Load file dialogs.
SKALD-E6: Web App Deployment	As an admin, I want to deploy Skald as a web service so users can access it from anywhere.	SKALD-18: Build Express.js server. SKALD-19: Create /api/generate endpoint. SKALD-20: Implement user authentication. SKALD-21: Create Dockerfile for deployment.

7. Build and Deployment Strategy

The decoupled architecture necessitates two distinct build and deployment pipelines.

Desktop Application

The desktop application will be bundled into standard OS-specific installers (e.g., .dmg for macOS, .exe for Windows, .deb/.AppImage for Linux).

1. **Backend Compilation:** The skald_codegen Odin tool will be compiled separately for each target platform (windows-amd64, darwin-amd64, linux-amd64).
2. **Frontend Build:** The React/TypeScript project will be built into static HTML/CSS/JS assets.
3. **Packaging:** The electron-builder tool will be configured to package the frontend assets and automatically include the correct compiled Odin binary from step 1 based on the build target. The final output is a single, self-contained distributable application.

Web Application (Future)

The web application will be deployed as a containerized service to a cloud provider.

1. **Backend Compilation:** A single skald_codegen binary will be compiled for the linux-amd64 target, as this is the standard for most Docker containers.
2. **Frontend Build:** The React application will be built into a directory of static assets (build or dist).
3. **Containerization:** A Dockerfile will be created with a Node.js base image. It will:
    - Copy the Node.js/Express server code.
    - Copy the compiled Linux Odin binary into the container's path (e.g., /usr/local/bin).
    - Copy the static frontend build assets.
    - Install Node.js dependencies (npm install).
    - Expose the necessary port and define the command to start the server.
4. **Deployment:** This container image will be pushed to a registry (e.g., Docker Hub, ECR) and deployed to a managed container service.

8. Future Considerations

This architecture is highly extensible. Future enhancements can be integrated smoothly.

- **Extensibility:** A plugin system can be developed where users or third parties can contribute new nodes. This could be achieved by publishing NPM packages that contain both the React component for the node and a JSON definition of its parameters, which the Odin backend can then use for code generation.
- **Cloud Synchronization:** User projects (the graph .json files) could be synchronized across devices by storing them in a cloud database (e.g., Firestore) linked to their user account.
- **CI/CD:** Automation pipelines can be established to trigger builds, run tests, and deploy both the desktop and web applications upon commits to the main branch.

9. Conclusion and Strategic Recommendations

The revised architecture represents a significant strategic pivot for the Skald project. By decoupling the UI from the code generation logic, we gain immense flexibility, accelerate development, and deliver a vastly superior user experience. The Odin language can be focused on what it does best—generating performant, low-level code—while the web technology stack provides a mature, powerful, and purpose-built platform for the graphical user interface.

Strategic Recommendations:

1. **Embrace the Decoupling:** Development teams for the frontend and backend can work in parallel with minimal friction, as the JSON contract is their only point of integration.
2. **Prioritize the Desktop Experience:** The initial goal is to deliver a polished, feature-complete desktop application. This provides the most value to users in the short term.
3. **Plan for the Web:** The choice of Electron, React, and the Web Audio API ensures that the majority of the frontend codebase is directly reusable for the future web application. This "start with desktop, plan for web" approach is both efficient and forward-thinking, positioning Skald for long-term success and accessibility.

This plan provides a robust and scalable foundation for building Skald, ensuring we can deliver a world-class tool to the Odin community.