A Technical Blueprint for a Web-Based Geometric Semantics Engine

Authored by: The Architectural Review Board Document Version: 1.0 Date: October 26, 2023

Executive Summary

This document presents a comprehensive technical blueprint for the development of a novel web application: an interactive Geometric Semantics Engine. The primary objective of this project is to create a functional, browser-based tool that operationalizes the theoretical framework of Geometric Semantics, as detailed in the foundational papers on Color Geometry and its axiomatic system. The application will serve as a dynamic environment for defining, composing, and reasoning about concepts, translating abstract semantic principles into tangible, interactive geometric operations within the perceptually uniform Oklab color space.

The proposed system is founded on a decoupled, "worker-first" architecture designed to ensure a fluid and responsive user experience. All computationally intensive tasks, such as the numerical optimization required for causal inference, will be offloaded from the main user interface thread to a dedicated background Web Worker. This core computational engine will leverage the power of WebAssembly to run a high-performance, industry-standard Quadratic Programming (QP) solver, bringing near-native speed for complex mathematical operations directly into the browser environment.

The technology stack has been meticulously selected to prioritize performance, developer experience, and alignment with the project's unique scientific visualization requirements. The frontend will be developed using the Svelte framework, chosen for its compile-time optimizations and minimal runtime overhead. The interactive 3D visualization of the Oklab manifold will be rendered using Three.js, a flexible and lightweight WebGL library. These will be supported by a suite of specialized JavaScript libraries for advanced color science, computational geometry, and state management.

This report is structured to guide a technical audience from high-level architectural decisions to granular implementation details. Part I details the system architecture and provides a rigorous justification for the selected technology stack. Part II focuses on the practical implementation of the Oklab perceptual manifold and its physical constraints. Part III forms the core of the document, providing a detailed breakdown of how each of the five axioms of Geometric Semantics will be translated into a specific, functional, and interactive UI module. Finally, Part IV outlines a phased implementation roadmap to guide the development process from foundational engine-building to full system integration and testing.

Part I: System Architecture and Technology Stack

This section defines the application's high-level structure and the specific technologies chosen for its implementation. The architectural design and technology stack are direct responses to the unique computational demands and interactive nature of a Geometric Semantics Engine,

ensuring both performance and fidelity to the underlying theoretical model.

1.1. Conceptual Architecture: A Decoupled, Worker-First Approach

The theoretical model's core computational process involves solving a geometric inverse problem for each causal inference, which translates to a constrained numerical optimization task. Executing such an intensive calculation directly on the browser's main thread would inevitably lead to a frozen, unresponsive user interface, rendering the application unusable for interactive exploration. Consequently, a "worker-first" architecture is not merely a performance enhancement but a fundamental design requirement.
This architecture decouples the user interface from the computational engine, ensuring that the UI remains fluid and responsive at all times. The entire computational workload is encapsulated within a Web Worker, which operates on a separate background thread. Communication between the frontend and this background engine occurs asynchronously via a message-passing interface, preventing any blocking of the main execution thread.
The system is composed of three primary components:

- **Interactive Frontend (Main Thread):** This component is responsible for all user-facing aspects of the application, including rendering the 3D visualization of the Oklab manifold and managing all user interactions. It will be built using the Svelte framework and the Three.js rendering library. Its primary role is to manage the UI state, capture user input, and dispatch computation requests to the Semantic Computation Engine. It does not perform any heavy calculations itself; it only visualizes the results returned by the worker.
- **State Management Layer (Main Thread):** A centralized store will be implemented to manage the global state of the application. This includes the collection of defined symbols (point vectors), abstract concepts (geometric regions), user-defined compositions, and the results of causal analyses. A dedicated state management solution ensures data consistency and predictability across all UI modules, simplifying development and preventing synchronization issues.
- **Semantic Computation Engine (Web Worker):** This is the analytical core of the application, running entirely within a dedicated background thread. It exposes a clean, asynchronous API to the main thread for handling all complex calculations. This engine will house the WebAssembly-based numerical solver, color science utilities, and computational geometry libraries. It will receive requests from the frontend, such as "solve for causality" or "calculate convex hull," perform the necessary computations, and post the results back to the main thread for visualization.

This architectural separation of concerns provides a robust and scalable foundation. The UI can evolve independently of the computational logic, and the engine itself can be optimized or even replaced without affecting the frontend, so long as the API contract is maintained.

1.2. The Semantic Computation Engine: High-Performance In-Browser Numerics

The central computational task of the application is to solve the geometric inverse problem defined by Axiom 3 (Causality). This is formalized as a k-sparse Non-Negative Least Squares (NNLS) problem, a specific type of constrained Quadratic Program (QP). The need to solve this problem efficiently, repeatedly, and entirely within the client's browser dictates the technological composition of the Semantic Computation Engine.

A survey of the JavaScript ecosystem reveals a significant gap in mature, high-performance libraries for numerical optimization. While some native JavaScript QP solvers exist, such as quadprog (a direct port of older Fortran code), they often lack the performance, robustness, and modern feature set required for this application. The state-of-the-art in numerical optimization resides in libraries written in low-level languages like C and C++ for maximum efficiency, with OSQP being a prime example of a modern, robust solver for convex QP problems.
To bridge this gap, the engine will leverage WebAssembly (Wasm). WebAssembly is a binary instruction format that serves as a compilation target for languages like C/C++, enabling their code to run in modern web browsers at near-native speeds. By compiling a battle-tested C library like OSQP into a Wasm module using a toolchain such as Emscripten , we can embed a powerful and reliable numerical solver directly into our browser-based engine. This approach is the only viable path to achieving the required performance and accuracy for real-time semantic analysis.
The Semantic Computation Engine will consist of the following components, all operating within the Web Worker:

- **Wasm QP Solver:** The compiled OSQP module. This will be the heart of the engine, responsible for solving the constrained optimization problem that underpins the Geometric Reachability Analyzer.
- **Color Science Module:** A suite of functions built upon a dedicated color science library for high-precision conversions between sRGB and Oklab, and for implementing the "gamut oracle" function (detailed in Part II) required for physical feasibility checks.
- **Computational Geometry Module:** A collection of functions for geometric primitives, including 3D convex hull generation and point-in-polyhedron tests, which are essential for implementing the causal reachability axiom.
- **API Layer:** A JavaScript wrapper that abstracts the complexities of interacting with the Wasm module and the other engine components. This layer will expose a clean, promise-based API to the main thread (e.g., engine.solveCausality(antecedents, consequent)), simplifying communication and making the engine easy to use from the frontend.

1.3. Technology Stack Rationale and Selection

The selection of each technology is based on a rigorous evaluation of alternatives against the project's core requirements: high performance for interactive visualization, fidelity to the underlying color science, and an efficient development workflow.

- **Frontend Framework: Svelte** Svelte is chosen over more established frameworks like React for its fundamentally different approach. Instead of using a virtual DOM and performing diffing at runtime, Svelte is a compiler that shifts much of the work to the build step. This results in highly optimized, vanilla JavaScript code, leading to significantly smaller bundle sizes and superior runtime performance. Benchmarks indicate that Svelte applications can load up to 30% faster and handle larger component counts with less memory usage than their React equivalents. For a performance-critical, data-heavy visualization application, these advantages are paramount. Furthermore, Svelte's "less code" philosophy, simplified syntax, and built-in reactivity model reduce boilerplate and cognitive overhead, accelerating development cycles and improving maintainability.
- **3D Rendering Library: Three.js** Three.js is selected for its balance of power, flexibility, and performance. It is a lightweight, low-level library that provides fine-grained control over the WebGL rendering pipeline, making it ideal for creating a highly customized and

optimized visualization of the Oklab manifold. While an alternative like Babylon.js offers a more feature-rich, all-in-one engine, its larger bundle size (approximately 1.4 MB vs. Three.js's 168 kB) presents a significant performance penalty for a web application. The vast ecosystem and massive community behind Three.js ensure extensive documentation, examples, and a rich set of third-party helper libraries (such as drei) that can simplify common development tasks without sacrificing the core library's lean footprint.

- **Core Libraries:**
    - **Color Science: Color.js (colorjs.io)** This library is the definitive choice for handling the application's color science requirements. Co-authored by an editor of the CSS Color specification, it provides an authoritative and comprehensive implementation of modern color theory. Its key features include robust support for Oklab and Oklch, a wide array of other color spaces, and, most critically, sophisticated gamut mapping algorithms that go beyond naive clipping. This advanced gamut handling is essential for accurately implementing the "gamut droplet" as a physical feasibility constraint, as it allows for perceptually-aware mapping of out-of-gamut colors.
    - **Convex Hull: quickhull3d** This library is a direct JavaScript implementation of the robust and efficient Quickhull algorithm, which has an average time complexity of O(n \log n). It is lightweight, well-tested (having been incorporated into Three.js itself), and provides the exact functionality required by the theoretical model. Crucially, it includes a helper function, isPointInsideHull, which directly implements the geometric containment check needed for the causal reachability test of Axiom 3.
    - **QP Solver (for Wasm Compilation): OSQP** The Operator Splitting Quadratic Program (OSQP) solver is a modern, high-performance C library designed for the exact type of convex QP problems this application must solve. It is known for its efficiency and robustness and is distributed under the permissive Apache 2.0 license, making it suitable for any application. Its library-free, self-contained nature simplifies the compilation process to WebAssembly.

The following table provides a summary of the evaluation process for the primary technology choices.

Category	**Technology**	**Performance**	**Ecosystem & Maturity**	**Developer Experience**	**Project-Specific Fit**	**Rationale for Selection**
Frontend Framework	**Svelte**	Excellent	Good	Excellent	Excellent	Superior runtime performance, smaller bundles, and reduced boilerplate make it ideal for a fast, interactive visualization tool.
	React	Good	Excellent	Good	Good	Larger ecosystem

Category	Technology	Performance	Ecosystem & Maturity	Developer Experience	Project-Specific Fit	Rationale for Selection
						but runtime overhead of virtual DOM is a disadvantage for this specific use case. Steeper learning curve with JSX and hooks.
	Vue	Very Good	Very Good	Very Good	Good	A strong "middle ground" option, but Svelte's compile-time approach offers a clearer performance advantage for this project.
3D Rendering Library	**Three.js**	Excellent	Excellent	Good	Excellent	Lightweight, flexible, and high-performance. Its modularity allows for a custom, optimized renderer. Vast community support.
	Babylon.js	Very Good	Good	Excellent	Good	All-in-one engine with great tooling, but its significantly larger bundle

Category	Technology	Performance	Ecosystem & Maturity	Developer Experience	Project-Specific Fit	Rationale for Selection
						size is a major drawback for web deployment.

Part II: The Perceptual Manifold: Oklab Implementation

This section details the implementation of the foundational geometric space of the application. The Oklab manifold is not merely a coordinate system; it is the substrate for all semantic operations. Its correct and efficient implementation, including its physical boundaries, is paramount.

2.1. Oklab Space and Color Management Core

The application's core logic relies on the perceptual uniformity of the Oklab space, where geometric distance corresponds directly to perceived color difference. Therefore, all internal representations and computations involving color must be performed exclusively in Oklab coordinates. The Color.js library will be used to build a dedicated color management module that encapsulates all necessary transformations and calculations.
This module will expose a set of key functions to the rest of the application:

- sRGBToOklab(rgb: number): number: This function will take a standard sRGB color, typically from a UI element like a color picker (represented as an array [r, g, b] with values from 0 to 1), and convert it into its corresponding Oklab coordinate vector [L, a, b].
- oklabToSRGB(lab: number): number: The inverse function, which converts an Oklab coordinate vector back to the sRGB color space. This function will be used exclusively at the final rendering stage to display colors on the user's screen.
- getPerceptualDistance(lab1: number, lab2: number): number: This function calculates the perceptual difference, \Delta E, between two Oklab color vectors. It implements the standard Euclidean distance formula, \Delta E = \sqrt{(L_1-L_2)^2 + (a_1-a_2)^2 + (b_1-b_2)^2}, which is the geometric distance in the 3D Oklab space.

By centralizing these operations, the application ensures that all semantic logic operates within the perceptually uniform manifold, maintaining fidelity to the theoretical framework.

2.2. The Gamut Droplet as a Physical Oracle

The theoretical framework elevates the sRGB gamut from a simple display limitation to a fundamental law of semantic feasibility, termed the "gamut droplet" \mathcal{D}. A symbol's meaning is only considered physically realizable if its Oklab coordinate vector \mathbf{x} lies within this volume: \mathbf{x} \in \mathcal{D}. This check is a critical component of Axiom 3 (Causality) and must be performed frequently and efficiently.
A naive implementation—converting an Oklab color to sRGB and then checking if the R, G, and B components are all within the `` range—is computationally expensive. This process involves

complex matrix multiplications and non-linear gamma correction functions for every single check, making it a bottleneck for real-time interaction.
To overcome this, a high-performance "gamut oracle" function will be implemented. This function, isOklabInSRGBGamut(lab: number), will determine if a given Oklab point is within the sRGB gamut without performing the full, slow conversion. The implementation will be based on the mathematical principles described by Björn Ottosson, whose work on gamut clipping details how the sRGB gamut boundary can be described in Oklab space using efficient cubic polynomials. This allows for a much faster geometric check, transforming the gamut boundary from an implicit result of conversion into an explicit, queryable mathematical object.
In addition to the computational oracle, a visual representation of the gamut droplet is essential for user comprehension. A 3D mesh representing the surface of the sRGB gamut will be generated and rendered in the main Three.js scene. This can be achieved by:

1. Generating a dense set of sample points on the surface of the unit sRGB cube (e.g., on the faces where R, G, or B is 0 or 1).
2. Converting each of these sRGB points to its Oklab coordinate.
3. Using a convex hull algorithm on this cloud of Oklab points to generate a 3D mesh that approximates the gamut droplet's surface.

This semi-transparent mesh will serve as the visual boundary of physical possibility within the 3D scene, making the concepts of "in-gamut" and "out-of-gamut" immediately intuitive to the user.

Part III: Translating Axioms into Interactive UI Modules

This section provides the detailed specification for the application's user interface, with each module designed as a direct, functional implementation of one of the five core axioms of Geometric Semantics. This ensures that the final product is not merely a visualization tool but a true interactive engine for exploring the principles of the theory.
The following table provides a high-level map from the theoretical axioms to their corresponding UI modules and computational underpinnings.

Axiom	**UI Module**	**Core User Interaction**	**Key Computations**	**Primary Libraries**
Axiom 1: Grounding	Symbol Palette & Inspector	Create, select, and inspect color symbols as 3D points in the Oklab manifold.	Oklab \leftrightarrow sRGB conversion, Gamut Oracle check.	Three.js, Color.js
Axiom 2: Compositionality	Convex Mixer	Interactively mix multiple symbols using weighted averages to create composite symbols.	Weighted vector averaging (convex combination).	Three.js
Axiom 3: Causality	Geometric Reachability Analyzer	Test if a "consequent" symbol is causally reachable from a set of "antecedent"	3D convex hull generation, point-in-polyhedron test, Gamut Oracle check.	quickhull3d, OSQP (Wasm)

Axiom	UI Module	Core User Interaction	Key Computations	Primary Libraries
		symbols.		
Axiom 4: Abstraction	Conceptual Region Builder	Define and visualize abstract concepts as geometric regions within the manifold.	Filtering points based on channel values, set operations (intersection, union).	Three.js
Axiom 5: Closed Loop	Integrated Workflow	Utilize the outputs of one module as the inputs for another in a complete reasoning cycle.	Orchestration of all above computations.	Svelte (State Mgmt)

3.1. Axiom 1 (Grounding): The Symbol Palette & Inspector

Theoretical Basis: *Axiom 1: Each elementary symbol is defined as a point vector \mathbf{x}=(L,a,b) in the OKLab space, subject to the constraint that this point must lie within the gamut droplet: \mathbf{x} \in \mathcal{D}*.
This module provides the foundational capability to define and inspect the elementary symbols that form the vocabulary of the semantic engine.
UI Specification:

- **3D Interactive View:** The central component of the UI will be a Three.js canvas rendering the 3D Oklab space. The axes (L, a, b) will be clearly labeled. The pre-computed sRGB gamut droplet mesh will be rendered as a semi-transparent, solid volume, visually defining the space of physically possible meanings.
- **Symbol Palette:** A sidebar panel will list all user-defined symbols. Each entry will display the symbol's name (e.g., "Primary Red," "Sky Blue") and a small swatch of its color.
- **Symbol Creation:** A "+" button will open a standard sRGB color picker. When the user selects a color, it will be instantly converted to its Oklab coordinate vector. This vector is then added to the palette as a new symbol. The UI will enforce the grounding constraint by preventing the creation of symbols from colors that are outside the sRGB gamut.
- **Interaction and Inspection:**
    - Clicking a symbol in the palette will highlight its corresponding point in the 3D view, visually connecting the abstract name to its geometric locus.
    - A dedicated "Inspector" panel will display detailed information for the selected symbol, including its name, its Oklab coordinates (L, a, b), and its sRGB coordinates (R, G, B). This allows for direct inspection of the symbol's mathematical identity.

3.2. Axiom 2 (Compositionality): The Convex Mixer

Theoretical Basis: *Axiom 2: The semantic composition of a set of symbols is defined as their convex combination. The emergent meaning of the composition, \mathbf{x}_v, is the point in OKLab space given by the weighted average of the constituent symbols: \mathbf{x}_v = \sum_{u} \alpha_{vu} \mathbf{x}_u*.

This module allows users to explore the principle of compositionality by interactively mixing symbols to create new, composite meanings.

UI Specification:

- **Mixing Workspace:** A dedicated panel where users can drag and drop symbols from the Symbol Palette. These symbols act as the "antecedents" or ingredients of the mixture.
- **Weight Controls:** For each antecedent symbol in the workspace, the UI will provide a slider to control its weight, \alpha_{vu}, in the composition. The sliders will be interlinked to enforce the simplex constraint (\sum \alpha_{vu} = 1), so that increasing one weight automatically decreases the others proportionally.
- **Real-Time Visualization:** As the user adjusts the weights, the application will continuously compute the weighted average of the antecedent Oklab vectors. The resulting composite symbol, \mathbf{x}_v, will be visualized in real-time as a distinct point in the 3D view. The path traced by this point as weights are adjusted will lie within the convex hull of the antecedent points.
- **Physical Feasibility Feedback:** The inspector panel for the composite symbol will show its Oklab and sRGB values. If the calculated composite point \mathbf{x}_v falls outside the gamut droplet \mathcal{D}, the UI will display a prominent warning (e.g., "Chromatic Impossibility: Resulting color is not physically realizable"). This provides a direct, interactive demonstration of how composition can lead to semantically valid but physically impossible states.

3.3. Axiom 3 (Causality): The Geometric Reachability Analyzer

Theoretical Basis: *Axiom 3: A set of antecedent symbols is a valid cause for a consequent symbol if and only if the consequent is geometrically reachable. A state is reachable if it lies within the intersection of the convex hull of its causes and the gamut droplet: \mathbf{x}_v \in \text{conv}(\{\mathbf{x}_u\}) \cap \mathcal{D}*.

This module is the application's primary reasoning tool, providing a concrete, testable definition of a causal link through a geometric query.

UI Specification:

- **Analysis Setup:** The user interface will allow the user to select a set of two or more "antecedent" symbols and a single "consequent" symbol to test the causal relationship.
- **Analysis Execution and Visualization:** Upon clicking an "Analyze" button, the application will dispatch a request to the Semantic Computation Engine, which will perform the following steps:
    1. Compute the 3D convex hull of the Oklab coordinate vectors of the antecedent symbols using the quickhull3d library.
    2. Return the vertices and faces of this hull to the main thread.
    3. The frontend will then render this convex hull in the Three.js scene as a transparent, wireframe polyhedron that encloses the antecedent points.
    4. The engine will simultaneously test for the two conditions of reachability:
        - **Directional Check:** Use the isPointInsideHull function from quickhull3d to determine if the consequent symbol's vector is geometrically contained within the computed convex hull.
        - **Physical Check:** Use the high-performance isOklabInSRGBGamut oracle to determine if the consequent symbol's vector lies within the gamut droplet \mathcal{D}.
- **Interpretable Feedback:** The primary goal of this module is to provide clear and

interpretable results. The UI will display one of three distinct outcomes, each with a corresponding visual cue:

- **Success:** A message stating "Causality Valid: Consequent is reachable." The 3D view will show the consequent point located inside both the rendered convex hull and the gamut droplet.
- **Failure Mode 1 (Directional Insufficiency):** A message stating "Causality Invalid: Directional Insufficiency." The UI will clearly show that the consequent point lies outside the rendered wireframe convex hull, visually demonstrating that no possible mixture of the causes could produce the effect.
- **Failure Mode 2 (Chromatic Impossibility):** A message stating "Causality Invalid: Chromatic Impossibility." The UI will show the consequent point *inside* the convex hull but *outside* the semi-transparent gamut droplet mesh, illustrating that while the result is directionally achievable, it is too saturated to be physically real.

3.4. Axiom 4 (Abstraction): The Conceptual Region Builder

Theoretical Basis: *Axiom 4: Higher-level concepts and abstract properties are represented not as single points, but as regions (sets of points) within the OKLab space*. Logical operations on concepts are implemented as geometric set operations.
This module allows the system to move beyond concrete, point-like perceptions to abstract, region-based concepts.
UI Specification:

- **Region Definition:** A tool will allow users to define new abstract concepts. A concept will be defined by a set of constraints on the Oklab (or Oklch for more intuitive hue/chroma control) channels. For example, the concept "Warm Colors" could be defined by a hue h range of 0 \le h \le 90, and "Bright Colors" by a lightness L value greater than 0.8.
- **Region Visualization:** When a concept is selected, the application will visualize the corresponding region within the 3D view. This can be achieved by rendering a dense point cloud of all valid, in-gamut points that satisfy the concept's constraints, effectively "highlighting" a volume within the gamut droplet.
- **Set-Theoretic Operations:** The module will provide functionality to perform logical operations on these concepts:
  - **Conjunction (AND):** The user can select two concepts (e.g., "Bright Colors" and "Reds") and compute their intersection. The application will then visualize the resulting, smaller region representing "Bright Reds."
  - **Disjunction (OR):** Similarly, users can compute the union of two regions (e.g., "Reds" OR "Blues").
  - **Subsumption (IS-A):** The UI will allow for checking if one concept is a subset of another (e.g., checking if the region for "Scarlet" is fully contained within the region for "Red").

3.5. Axiom 5 (Closed Loop): System Integration and Workflow

Theoretical Basis: *Axiom 5: A complete grounded system is formed by a closed loop wherein (1) a subsymbolic perceptual system maps sensory inputs to coordinates... (2) a symbolic reasoning layer performs geometric operations... and (3) the resulting coordinates can be mapped back to the perceptual domain for verification....*
This axiom is not implemented as a single UI module but is realized through the seamless

integration and workflow between all other modules. The application design ensures that the output of any module can serve as the input for another, creating a self-consistent cycle of perception, composition, abstraction, and reasoning.

Example Integrated Workflow:

1. **Grounding (Axiom 1):** A user opens the **Symbol Palette** and defines three elementary symbols: Red, Yellow, and Blue, by picking them from a color picker. These appear as points in the 3D Oklab manifold.
2. **Composition (Axiom 2):** The user drags Red and Yellow into the **Convex Mixer**, adjusts their weights to 50% each, and observes the creation of a new composite color. They name this Orange and save it back to the Symbol Palette. The system now has a new, derived symbol.
3. **Abstraction (Axiom 4):** The user navigates to the **Conceptual Region Builder** and creates a new concept called "Primary Colors" by selecting the points corresponding to Red, Yellow, and Blue. They then create another concept, "Warm Colors," defined by a hue range that encompasses Red, Orange, and Yellow.
4. **Causality (Axiom 3):** Finally, the user opens the **Geometric Reachability Analyzer**. They set Red and Blue as antecedents and a new, user-defined Purple as the consequent. The system computes the convex hull of Red and Blue and confirms that Purple lies within it and within the gamut, validating the causal link.

This end-to-end workflow demonstrates the closed grounding loop in action. Symbols are grounded as geometric points, combined to form new symbols, grouped into abstract regions, and used as the basis for verifiable causal inference, all within the single, unified geometric framework.

Part IV: Implementation Roadmap

This section outlines a phased development plan designed to manage complexity, mitigate risk, and deliver functionality incrementally. The roadmap is structured into four distinct phases, each corresponding to a set of related development sprints.

4.1. Phase 1: Core Engine and Manifold Foundation (Sprints 1-3)

Goal: To build the non-visual, computational backbone of the application. This phase focuses on establishing the high-performance engine and the architectural patterns that will support all subsequent development.

Key Tasks:

- **Task 1.1 (Wasm Integration):** Set up the Emscripten toolchain. Compile the OSQP C library into a WebAssembly module (.wasm) and its corresponding JavaScript glue code.
- **Task 1.2 (Engine API):** Create the Web Worker script. Develop a robust JavaScript wrapper around the compiled Wasm module to handle memory management and expose a clean, promise-based API for solving QP problems.
- **Task 1.3 (Color Science Module):** Integrate the Color.js library into the worker. Implement and unit-test the core conversion functions (sRGBToOklab, oklabToSRGB) and the perceptual distance metric.
- **Task 1.4 (Gamut Oracle):** Develop and benchmark the high-performance isOklabInSRGBGamut function based on the mathematical formulations for the sRGB gamut boundary in Oklab space.

- **Task 1.5 (Architecture Setup):** Initialize the Svelte project. Establish the main application structure, including the state management store and the communication channel between the main thread and the Web Worker.

4.2. Phase 2: Foundational UI Modules (Sprints 4-6)

Goal: To implement the core visual and interactive features of the application, allowing users to define and manipulate basic semantic elements.
Key Tasks:

- **Task 2.1 (3D Scene Setup):** Integrate Three.js into the Svelte application. Implement the main 3D canvas, including camera controls (orbit, pan, zoom) and basic scene lighting.
- **Task 2.2 (Gamut Visualization):** Generate the 3D mesh for the sRGB gamut droplet and render it as a semi-transparent object in the scene.
- **Task 2.3 (Symbol Palette & Inspector):** Build the UI for creating, listing, and deleting symbols (Axiom 1). Implement the interaction logic for selecting a symbol and displaying its properties in the inspector panel and its position in the 3D view.
- **Task 2.4 (Convex Mixer):** Build the UI for composing symbols via weighted averaging (Axiom 2). Implement the real-time calculation and visualization of the composite symbol as weights are adjusted.

4.3. Phase 3: Advanced Reasoning Modules (Sprints 7-9)

Goal: To implement the primary analytical tools of the semantic engine, focusing on causal inference and abstraction.
Key Tasks:

- **Task 3.1 (Geometry Library Integration):** Integrate the quickhull3d library into the Semantic Computation Engine within the Web Worker.
- **Task 3.2 (Reachability Analyzer - Backend):** Extend the engine's API with a function that takes antecedent and consequent symbols, computes the convex hull, and performs the two reachability checks (in-hull and in-gamut).
- **Task 3.3 (Reachability Analyzer - Frontend):** Develop the UI for the Geometric Reachability Analyzer (Axiom 3). This includes the logic for setting up an analysis, dispatching the request to the worker, and rendering the resulting convex hull and interpretable feedback messages.
- **Task 3.4 (Conceptual Region Builder):** Develop the UI for defining and visualizing abstract concepts as geometric regions (Axiom 4). Implement the point-cloud visualization for regions and the logic for performing set-theoretic operations.

4.4. Phase 4: Integration, Optimization, and Deployment (Sprints 10-12)

Goal: To ensure all modules function together as a cohesive system, refine the user experience, optimize performance, and prepare the application for deployment.
Key Tasks:

- **Task 4.1 (End-to-End Workflow Testing):** Implement and rigorously test the integrated workflows described in the Closed Loop section (Axiom 5). Ensure that symbols and concepts created in one module are immediately available and usable in others.

- **Task 4.2 (Performance Profiling):** Conduct comprehensive performance profiling of the application, focusing on rendering frame rates, worker response times, and memory usage. Identify and address any bottlenecks.
- **Task 4.3 (UI/UX Refinement):** Conduct user acceptance testing (UAT) to gather feedback on the application's usability and clarity. Refine UI elements, tooltips, and instructional text based on this feedback.
- **Task 4.4 (Deployment):** Configure the build process for production, including code minification and asset optimization. Deploy the application to a staging and then production web server.

Conclusions

This document provides a technically rigorous and exhaustive plan for the creation of a web application that serves as an interactive semantic engine based on the principles of Geometric Semantics. The proposed architecture, centered on a high-performance WebAssembly engine operating within a Web Worker, directly addresses the significant computational demands of the underlying theory while ensuring a fluid and responsive user experience. The selected technology stack—Svelte for the frontend, Three.js for 3D rendering, and specialized libraries for color science and numerical optimization—represents a carefully considered balance of performance, capability, and development efficiency.

The detailed breakdown of features demonstrates a clear and direct pathway from the abstract axioms of the source material to concrete, functional, and interactive UI modules. By translating concepts like "grounding," "compositionality," and "causality" into specific user interactions and visual feedback mechanisms, the proposed application will not only serve as a powerful tool for research and education but also as a compelling proof-of-concept for this novel approach to artificial intelligence. The emphasis on interpretability, particularly in the Geometric Reachability Analyzer, addresses a critical weakness in many contemporary AI systems by providing transparent, verifiable, and geometrically intuitive explanations for its reasoning processes.

The implementation roadmap provides a structured, phased approach to development that prioritizes foundational components and manages complexity, ensuring a clear path to a successful project delivery. By adhering to this blueprint, a development team can construct a faithful and powerful computational realization of the Geometric Semantics framework, creating a unique tool that allows users to directly interact with the "physics of meaning."

Works cited

1. WebAssembly - MDN Web Docs, https://developer.mozilla.org/en-US/docs/WebAssembly 2. WebAssembly - Wikipedia, https://en.wikipedia.org/wiki/WebAssembly 3. An Active-Set Algorithm for Convex Quadratic Programming Subject to Box Constraints with Applications in Non-Linear Optimization and Machine Learning - MDPI, https://www.mdpi.com/2227-7390/13/9/1467 4. albertosantini/quadprog: Module for solving quadratic programming problems with constraints - GitHub, https://github.com/albertosantini/quadprog 5. OSQP, https://osqp.org/ 6. osqp/osqp: The Operator Splitting QP Solver - GitHub, https://github.com/osqp/osqp 7. How to Use WebAssembly for Machine Learning in Browsers? - Clover Dynamics, https://www.cloverdynamics.com/blogs/how-to-use-web-assembly-for-machine-learning-in-browsers 8. WebAssembly, https://webassembly.org/ 9. OSQP solver documentation,

https://osqp.org/docs/ 10. React vs Vue vs Svelte: Choosing the Right Framework for 2025 - Medium, https://medium.com/@ignatovich.dm/react-vs-vue-vs-svelte-choosing-the-right-framework-for-2025-4f4bb9da35b4 11. Svelte vs React: A Comprehensive Comparison for Developers - Strapi, https://strapi.io/blog/svelte-vs-react-comparison 12. Comparing front-end frameworks for startups in 2025: Svelte vs React vs Vue - Merge Rocks, https://merge.rocks/blog/comparing-front-end-frameworks-for-startups-in-2025-svelte-vs-react-vs-vue 13. Three.js vs Babylon.js vs Verge3D: Which 3D Framework to Choose - Edana, https://edana.ch/en/2025/10/17/three-js-vs-babylon-js-vs-verge3d-which-to-choose-for-a-successful-3d-project/ 14. Babylon.js Vs. Three.js-comparison - Ansi ByteCode LLP, https://ansibytecode.com/babylon-js-vs-three-js-comparison/ 15. Three.js vs. Babylon.js: Which is better for 3D web development? - LogRocket Blog, https://blog.logrocket.com/three-js-vs-babylon-js/ 16. Color.js: Let's get serious about color • Color.js, https://colorjs.io/ 17. Gamut mapping - Color.js, https://colorjs.io/docs/gamut-mapping 18. CSS Color Module Level 4 - W3C, https://www.w3.org/TR/css-color-4/ 19. mauriciopoppe/quickhull3d: A JS library to find the convex hull of a finite set of 3d points - GitHub, https://github.com/mauriciopoppe/quickhull3d 20. Quickhull - Rosetta Code, https://rosettacode.org/wiki/Quickhull 21. The solver — OSQP documentation, https://osqp.org/docs/solver/index.html 22. sRGB gamut clipping - Björn Ottosson, https://bottosson.github.io/posts/gamutclipping/