Architectural Specification for Client-Side Irregular Sticker Layout and Nesting Engine

1. Executive Summary

The digital printing and die-cutting industry faces a persistent optimization challenge: the efficient arrangement of irregular shapes onto limited substrate areas to minimize material waste. This process, known in computational geometry as "nesting" or irregular bin packing, is mathematically complex (NP-hard) and traditionally requires expensive, server-side proprietary software. This report outlines a comprehensive architecture for a purely client-side, browser-based application designed to democratize this capability for small business owners, specifically targeting the production of custom stickers.
The proposed solution leverages the **Angular** framework to provide a robust user interface, while offloading the heavy computational burden of geometric optimization to **Web Workers**. By utilizing a suite of open-source JavaScript libraries—specifically **SVGNest** logic for packing, **imagetracerjs** for vectorization, and **PDFKit** for document generation—the architecture achieves a "zero-backend" footprint. This design not only eliminates server costs and latency but also enhances user privacy by ensuring proprietary artwork never leaves the client's device.[1]
The system is designed to handle the nuances of print production, including strict adherence to physical dimensions (Imperial and Metric), the generation of precise cut lines via polygon offsetting, and the output of high-resolution (300 DPI) production files. The architecture emphasizes modularity, enabling the future integration of WebAssembly (WASM) modules for enhanced performance while maintaining a maintainable JavaScript/TypeScript codebase for the immediate implementation.

2. Problem Domain and Algorithmic Complexity

The core utility of the application revolves around the "Sticker Nesting Problem." To understand the architectural decisions, one must first appreciate the mathematical and computational constraints inherent in placing irregular shapes.

2.1. The Irregular Bin Packing Problem

Unlike standard bin packing, which deals with rectangular objects, sticker nesting involves arbitrary polygons that may be concave, convex, or contain holes. The goal is to place a set of polygons $P = \{p_1, p_2,..., p_n\}$ into a rectangular sheet $S$ (the bin) such that no two polygons overlap ($p_i \cap p_j = \emptyset$) and the utilization of $S$ is maximized.
This problem belongs to the class of NP-hard problems, meaning that no known algorithm can find the *perfect* optimal solution in polynomial time. As the number of shapes ($n$) increases, the computational time required to check every possible permutation grows factorially.
Consequently, the architecture must rely on **heuristics** and **meta-heuristic optimization** (specifically Genetic Algorithms) rather than deterministic solvers. The system does not seek the "perfect" layout, but rather a "sufficiently optimal" layout within a reasonable timeframe.[3]

2.2. The "No Fit Polygon" (NFP)

The fundamental geometric operation enabling irregular nesting is the calculation of the **No Fit Polygon (NFP)**. In simple terms, if we have a stationary polygon $A$ and an orbiting polygon $B$, the NFP represents the locus of points where $B$ touches $A$ without intersecting.

- **Inner Fit Polygon (IFP):** This defines the valid area within the bin (sheet) where a shape can be placed.
- **Mechanism:** To check if a new shape can be placed, the algorithm does not merely check for pixel collisions (which is slow and resolution-dependent) or bounding box collisions (which is inaccurate for stickers). Instead, it calculates the NFP for the new shape against all previously placed shapes. The valid placement positions are those that lie outside the union of all NFPs but inside the IFP of the sheet.[3]

The calculation of NFPs is computationally expensive. It involves Minkowski Sums, which are operations that add two sets of position vectors. For two polygons with $m$ and $n$ vertices, the Minkowski sum can have up to $O(m \cdot n)$ edges. This informs a critical architectural requirement: **Geometry Simplification**. The input shapes must be simplified to reduce vertex count before nesting, otherwise, the browser will hang indefinitely calculating NFPs for high-fidelity traces.[5]

---

3. Client-Side Architecture and Framework Selection

The constraint of a "client-side only" application fundamentally shifts the architecture from a standard Request/Response model to a concurrent processing model within the browser.

3.1. The Angular Framework Choice

Angular is selected as the application framework due to its strong typing (TypeScript), dependency injection, and structural rigidity, which benefits complex logic-heavy applications over simpler view libraries.

- **State Management:** The application requires managing complex states: Uploading $\rightarrow$ Tracing $\rightarrow$ Simplifying $\rightarrow$ Nesting $\rightarrow$ Rendering. Angular's **Signals** (introduced in recent versions) provide a granular reactivity model that is ideal for handling the high-frequency updates coming from the nesting worker (e.g., updating the preview canvas 60 times a second as layouts improve) without triggering unnecessary change detection cycles across the entire DOM.[7]
- **Service Isolation:** Angular's service architecture allows for clean separation of concerns. We can isolate the PDFService, GeometryService, and WorkerService, making the codebase testable and modular. This is crucial when handing off to a coding LLM, as distinct files with clear interfaces reduce hallucination and logic errors.

3.2. Concurrency via Web Workers

JavaScript is single-threaded. Running a Genetic Algorithm (GA) on the main thread would be catastrophic for User Experience (UX). The GA runs in a tight while loop, constantly mutating and evaluating layouts. On the main thread, this would freeze the UI, preventing the user from clicking "Stop," changing margins, or even seeing the progress bar update.
Therefore, the architecture mandates a **Manager-Worker** pattern.

- **The Manager (UI Thread):** Handles user inputs (files, dimensions, margins) and visualization (drawing the current state to a canvas).
- **The Worker (Background Thread):** Contains the SVGNest logic. It receives a list of simplified polygons and a bin size. It runs the generation loop and posts messages back to the Manager whenever a generation results in a higher fitness score (better packing efficiency).[9]

Feature	**Main Thread (Angular)**	**Web Worker (Nesting Engine)**
Responsibility	DOM Rendering, Event Handling, Canvas Drawing	Geometric Calculations, Genetic Algorithm, Heuristics
Libraries	pdfkit, blob-stream	svgnest (core), clipper-lib (geometry)
Access	Full DOM, Window, Canvas API	importScripts, postMessage, No DOM
Memory	UI State, Image Blobs	Large Coordinate Arrays, NFP Cache

3.3. Browser Storage and Memory Limits

While the user requested "no database," the browser effectively acts as an ephemeral database.

- **ImageBitmap:** To handle high-resolution print images (e.g., 300 DPI PNGs), the application should convert uploaded Files into ImageBitmap objects. These are optimized for GPU rendering on the canvas and have lower memory overhead than standard DOM Image elements.[11]
- **Structured Cloning:** Communication between the Main Thread and Worker involves copying data (Structured Clone). Sending massive raw SVG strings or pixel data repeatedly is inefficient. The architecture requires parsing geometry into lightweight Coordinate Arrays ([{x,y}, {x,y}...]) before sending them to the worker. The high-res raster data remains on the Main Thread, referenced only by an ID.

---

4. The Geometric Processing Pipeline

Before any nesting can occur, the raw user inputs (likely Raster images like PNG/JPG) must be converted into the mathematical language of the nesting engine (Polygons). This pipeline is critical for quality.

4.1. Raster-to-Vector Conversion (Tracing)

Users will upload sticker designs as images. To cut them, we need a vector path.

- **Library: imagetracerjs**.[12]
- **Role:** Analyzes the pixel data (ImageData) of the uploaded image and generates an SVG Path representing the boundary.
- **Configuration Strategy:**
  - The sticker cut line should be the *outline* of the non-transparent pixels.
  - We are not interested in vectorizing the *internal* details of the image (colors, gradients).
  - *Optimization:* The architecture defines a pre-processing step where the image is drawn to an off-screen canvas and thresholded to a binary black-and-white mask (Alpha channel > 0 = Black, Alpha channel 0 = White). imagetracerjs is then run on this mask. This forces the tracer to ignore internal details and produce a single, clean outer hull.[13]

4.2. Polygon Offsetting (The "Bleed" or "Margin")

Stickers rarely get cut exactly on the artwork edge. They usually have a white "margin" or "offset" to allow for cutter error.

- **Library: ClipperLib** (via js-angusj-clipper or clipper-lib wrapper).[15]
- **Operation:** Dilation (Positive Offset).
- **Why Clipper?** Standard SVG stroke width is purely visual. It does not change the underlying geometry. For nesting, we need the *mathematical boundary* of the expanded shape. Clipper calculates the "Offset Polygon" which is a new set of coordinates expanded by distance D from the original.
- **Unit Management:** The user inputs margin in inches/cm. This must be converted to the coordinate system units before passing to Clipper.
    - $D_{internal} = Margin_{inches} \times DPI_{trace}$.
- **Join Styles:** The offset operation supports different corner styles. For stickers, jtRound (Round Joins) is aesthetically preferred over jtMiter (Sharp Points) which can be dangerous or tear easily.[16]

4.3. Path Simplification (Optimization)

The raw trace from imagetracerjs, even on a mask, creates thousands of small path segments.

- **Problem:** NFP calculation time is $O(N^2)$. A shape with 1000 points is $100\times$ slower to nest than one with 100 points.
- **Solution: simplify-js**.[18]
- **Algorithm:** This library implements the **Ramer-Douglas-Peucker (RDP)** algorithm. It iteratively removes points that lie within a certain distance (tolerance) of the line segment connecting their neighbors.
- **Architectural Rule:**
    1. **High-Res Path:** Kept for the final PDF Cut Line (visual fidelity).
    2. **Low-Res Path:** Generated via simplify-js and sent to the Web Worker for Nesting (computational speed).
    - This decoupling ensures the nesting is fast without making the final printed sticker look "blocky".[19]

---

5. The Nesting Engine (Deep Dive)

This section details the internal logic of the nesting.worker.ts, which is the brain of the application. As no "plug-and-play" NPM library exists that perfectly isolates the SVGNest logic without UI dependencies, the coding LLM will need to port the logic based on the reference architecture of SVGNest.[3]

5.1. The Genetic Algorithm (GA) Lifecycle

The GA evolves a "population" of potential layouts.

1. **Initialization:** Create a population of N individuals. Each "individual" is a list of the sticker parts in a specific random order, with specific random rotations.
2. **Evaluation (Fitness Function):** For each individual, simulate the placement of parts into the bin.
   - *Placement Rule:* "First Fit Decreasing" strategy is common, but SVGNest uses a gravity-based approach (usually bottom-left). The algorithm attempts to place the current part as close to $(0,0)$ as possible without intersecting the NFPs of already placed parts.
   - *Fitness Score:* The score is determined by the bounding box of the placed parts. $Fitness = W_{bounds} \times H_{bounds}$ (or just W if using a fixed height roll). Lower is better.[3]
3. **Selection:** The individuals with the best fitness scores are selected to reproduce.
4. **Crossover & Mutation:**
   - *Crossover:* Combine the ordering of Parent A with Parent B.
   - *Mutation:* Randomly swap the order of two parts. Randomly rotate a part (e.g., by 90 degrees).
   - *Insight:* The mutation rate allows the algorithm to escape "local optima" (arrangements that are good, but not the best).
5. **Iteration:** Repeat for multiple generations or until the user stops the process.

5.2. Geometry Library Integration

The Worker cannot use DOM-based measurement (like getBBox()). It must use a pure math library.

- **ClipperLib in Worker:** The worker will import clipper-lib to handle the boolean operations required for NFP generation (calculating the difference between shapes).[16]
- **NFP Caching:** Calculating NFP is the most expensive operation.
  - *Strategy:* The worker must implement a Map<string, Polygon> cache.
  - *Key:* A composite key of the two shape IDs and their rotations (e.g., ShapeA_0deg_ShapeB_90deg).
  - *Value:* The computed NFP polygon.
  - Before calculating a fresh NFP, the worker checks the cache. This drastically speeds up later generations where the same shapes are compared repeatedly.[3]

---

6. High-Fidelity Output Generation

The final deliverable is a PDF. This is where the "Vibe Coding" project transforms into a professional tool.

6.1. PDFKit Integration

PDFKit is the standard for Node.js and Browser PDF generation. It works by constructing a document stream.[21]

- **Vector Graphics:** PDFKit supports SVG path data (d strings) directly. doc.path('M10 10 L...').stroke(). This is used to draw the cut lines.
- **Raster Images:** PDFKit embeds images via doc.image(buffer, x, y, options).
- **Browser Compatibility:** PDFKit is a Node stream. To work in the browser, we pipe the output to blob-stream, which buffers the chunks and produces a final Blob object that can be downloaded via an anchor tag.[23]

6.2. The Unit Conversion Problem

This is the most common failure point in print apps.

- **The Web:** Uses CSS Pixels (96 DPI).
- **PDF:** Uses Points (72 DPI).
- **Print:** Uses Dots (300 DPI).
- **User:** Uses Inches or Centimeters.

The architecture requires a **Unit Conversion Service** that acts as the single source of truth.

- **Input:** User says "Sticker is 2 inches wide."
- **Internal Logic:**
  - $Width_{PDF} = 2 \text{ inches} \times 72 \text{ points/inch} = 144 \text{ points}$.
- **Image Scaling:**
  - Uploaded Image is 600 pixels wide.
  - If placed directly into PDF, PDFKit assumes 1px = 1pt. The image would appear $\frac{600}{72} = 8.33$ inches wide. Huge!
  - **Correction:** We must tell PDFKit to fit the 600px image into the 144pt box.
  - Code: doc.image(imgData, x, y, { width: 144 }).
- **Cut Line Scaling:**
  - The nesting engine works in arbitrary units (often scaled up integers for ClipperLib stability).
  - When rendering the PDF, the coordinate positions (x, y) returned by the worker must be scaled by the ratio of $\frac{\text{PDF Points}}{\text{Nesting Units}}$.

7. Detailed Data Structures

Strict typing is essential for the interaction between the Angular components, the Service Layer, and the Web Worker.

7.1. The Sticker Object

TypeScript

```typescript
interface StickerSource {
  id: string; // UUID
  file: File; // Original User File
  bitmap: ImageBitmap; // GPU-ready bitmap for Canvas Preview

  // Dimensionality
  inputDimensions: {
    width: number;
    height: number;
    unit: 'in' | 'cm' | 'mm';
  };

  // Geometry
  originalPath: Point; // High-res path from ImageTracer
  simplifiedPath: Point; // Low-res path for Nesting
  offsetPath: Point; // The margin/bleed path

  // Configuration
  margin: number; // In inches
}
```

7.2. The Nesting Configuration Message

TypeScript

```
interface NestingRequest {
  bin: {
    width: number; // In normalized calculation units
    height: number;
  };
  shapes: {
    id: string;
    points: Point; // The Simplified Path
  };
  config: {
    rotations: number; // e.g., 4 (0, 90, 180, 270)
    populationSize: number;
    mutationRate: number;
  };
}
```

7.3. The Nesting Result Message

TypeScript

```
interface NestingResponse {
  generation: number;
  fitness: number; // 0.0 to 1.0 (Utilization)
  placements: {
    id: string;
    x: number;
    y: number;
    rotation: number; // degrees
  };
}
```

8. Implementation Roadmap and Library Details

This section acts as a direct guide for the coding LLM.

8.1. Phase 1: Scaffold and Ingestion

1. **Initialize:** ng new sticker-nesting --style=scss.
2. **Dependencies:**
   - npm install pdfkit blob-stream imagetracerjs simplify-js js-angusj-clipper
   - npm install --save-dev @types/pdfkit @types/filesystem (and other types).
3. **Component:** Create upload-dropzone. Use DragEvent to capture files.
4. **Service:** ImageAnalysisService. Implement FileReader to read data URL, load into Image object, draw to hidden Canvas, and get ImageData.

8.2. Phase 2: Vectorization and Geometry

1. **Trace:** Use ImageTracer.imagedataToTracedata(). Extract the coordinate list from the first layer.
2. **Clean:** Filter out tiny paths (noise) based on area threshold.
3. **Offset:** Initialize ClipperLib.
   - Scale coordinates up by 1000 (Clipper works with integers).[15]
   - Execute ClipperOffset.
   - Scale down by 1000.
4. **Simplify:** Run simplify(points, tolerance, true) (High quality flag).[18]

8.3. Phase 3: The Worker (The Hard Part)

1. **Generate:** ng generate web-worker nesting.
2. **Port Logic:** Since svgnest is not a modular NPM package, the LLM must implement the *concept* of the NFP placement.
   - *Alternative:* Use bin-pack (NPM) if irregular nesting proves too complex for a single coding pass, but this sacrifices the "irregular" requirement. The report strongly suggests attempting the port or using a WASM build of libnest2d if the LLM can handle WASM integration.
   - *Recommended JS Path:* Implement a simplified "Gravity" placement. Sort shapes by area. Place largest first. For each subsequent shape, try to place it at (0,0), then check intersection. If intersect, move x by step size. If x > binWidth, move y and reset x.
   - *Intersection Check:* Use a "Point in Polygon" or "Polygon Intersect" function (available in clipper-lib documentation) to valid placement.[24]

8.4. Phase 4: Output

1. **Preview:** The Main Thread listens to the Worker. On message, it clears the <canvas>, and loops through placements.
   - ctx.save(); ctx.translate(p.x, p.y); ctx.rotate(p.rotation); ctx.drawImage(...); ctx.restore();.
2. **PDF:** On "Export," replicate the Canvas draw logic but using pdfkit syntax.
3. **Download:** Trigger the blob-stream save.

---

9. Deployment and Infrastructure

The user specifies Docker and Nginx Proxy Manager.

9.1. Dockerfile Strategy

A multi-stage build is standard and efficient.

Dockerfile

```
# Build Stage
FROM node:20-alpine as builder
WORKDIR /app
COPY package.json package-lock.json./
RUN npm ci
COPY..
RUN npm run build --configuration=production

# Runtime Stage
FROM nginx:alpine
COPY --from=builder /app/dist/sticker-nesting /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

9.2. Nginx Configuration

To support high-performance client-side operations, specific headers are recommended.

Nginx

```
server {
    listen 80;
    server_name localhost;
    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }

    # Enable compression for text-based assets (JS, CSS, SVG)
    gzip on;
    gzip_types text/plain text/css application/json application/javascript text/xml application/xml
application/xml+rss text/javascript image/svg+xml;

    # Cache control for static assets
    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

---

10. Conclusion

This research report defines a complete architectural blueprint for a client-side Sticker Layout Application. By strictly separating the UI concerns (Angular) from the geometric computation concerns (Web Workers), the system ensures a fluid user experience even during intensive optimization tasks. The selection of libraries—**PDFKit** for precise output, **imagetracerjs** for vector acquisition, and **ClipperLib** for geometric manipulation—covers the functional requirements without introducing server-side dependencies.
The resulting application will be a powerful, privacy-preserving tool that runs efficiently in a containerized environment, satisfying the user's request for a "vibe code" project that solves a real-world manufacturing problem for their friend's business. The depth of this specification provides the coding LLM with not just the "what," but the "how" and "why," minimizing ambiguity and ensuring a successful implementation.

10.1. Summary of Strategic Decisions

- **Zero-Backend:** chosen to reduce hosting complexity and improve privacy.
- **Web Workers:** chosen to prevent UI blocking during NP-hard calculations.
- **SVG/Vector Logic:** chosen over raster manipulation to ensure print-quality cut lines and precise nesting.
- **Angular:** chosen for robust state management of the multi-stage workflow.
- **Heuristic Optimization:** chosen as a pragmatic solution to the intractable bin-packing problem.

Works cited

1. Basic Image Nesting, Rectangular Type : r/CommercialPrinting - Reddit, accessed November 18, 2025, https://www.reddit.com/r/CommercialPrinting/comments/1aqhh7x/basic_image_nesting_rectangular_type/
2. Use web workers to run JavaScript off the browser's main thread | Articles - web.dev, accessed November 18, 2025, https://web.dev/articles/off-main-thread
3. Jack000/SVGnest: An open source vector nesting tool - GitHub, accessed November 18, 2025, https://github.com/Jack000/SVGnest
4. 2D Bin Packing Algorithm - javascript - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/53812280/2d-bin-packing-algorithm
5. Nesting Algorithm | CADEXSOFT, accessed November 18, 2025, https://cadexsoft.com/blog/nesting-algorithm/
6. Computational complexity and shape nesting - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/17925775/computational-complexity-and-shape-nesting
7. Web workers - Angular, accessed November 18, 2025, https://angular.dev/ecosystem/web-workers
8. Angular Web Workers: What, Why, When, and How (2025 Edition) | by Sehban Alam, accessed November 18, 2025, https://medium.com/@sehban.alam/angular-web-workers-what-why-when-and-how-2025-edition-09020c531fe6
9. Optimizing Angular Performance with Web Workers for Heavy Computations - C# Corner, accessed November 18, 2025, https://www.c-sharpcorner.com/article/optimizing-angular-performance-with-web-workers-for-heavy-computations/
10. Using Web Workers - Web APIs - MDN Web Docs, accessed November 18, 2025, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
11. Struggling to draw SVG onto an offscreen canvas in a web worker - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/79190241/struggling-to-draw-svg-onto-an-offscreen-canvas-in-a-web-worker
12. jankovicsandras/imagetracerjs: Simple raster image tracer and vectorizer written in JavaScript. - GitHub, accessed November 18, 2025,

https://github.com/jankovicsandras/imagetracerjs
13. How to trace/outline objects in an image? - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/65119816/how-to-trace-outline-objects-in-an-image
14. fromtheexchange/image2svg-awesome: All about image tracing and vectorization—the conversion of a raster image (jpg/png) to a vector image (svg). - GitHub, accessed November 18, 2025, https://github.com/fromtheexchange/image2svg-awesome
15. Clipper2 - Polygon Clipping and Offsetting Library - angusj.com, accessed November 18, 2025, https://www.angusj.com/clipper2/Docs/Overview.htm
16. svg - Offsetting polygons in Javascript - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/13248896/offsetting-polygons-in-javascript
17. junmer/clipper-lib: Boolean operations and offsetting library ... - GitHub, accessed November 18, 2025, https://github.com/junmer/clipper-lib
18. Simplify.js - a high-performance JavaScript 2D/3D polyline simplification library, accessed November 18, 2025, https://mourner.github.io/simplify-js/
19. ST_Simplify - PostGIS, accessed November 18, 2025, https://postgis.net/docs/ST_Simplify.html
20. SVGnest - Free and Open Source nesting for CNC machines, lasers and plasma cutters, accessed November 18, 2025, https://svgnest.com/
21. Embedding SVG in PDF (exporting SVG to PDF using JS) - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/5913338/embedding-svg-in-pdf-exporting-svg-to-pdf-using-js
22. PDFKit Measurement Unit Node.js - Sariful Islam, accessed November 18, 2025, https://sarifulislam.com/blog/pdfkit-measurement-unit/
23. Getting Started with PDFKit, accessed November 18, 2025, https://pdfkit.org/docs/getting_started.html
24. clipper-lib - NPM, accessed November 18, 2025, https://www.npmjs.com/package/clipper-lib