# Metrized Symbol Docs

## General Information

### Overview

The Metrized Symbol Counter project officially began in September 2023, the project addresses a problem originally posed by Owen:

- Imagine a company constructing a building. They might solicit designs from multiple contractors, each proposing a different blueprint. To estimate the cost of the building, it's necessary to count the various components—such as plugs, lights, and other items—represented by symbols on the blueprints. Manually counting these symbols is time-consuming and error-prone, leading us to explore an automated solution using computer vision.

### Problem, Initial Approach & Current Approach

The challenge is to detect and count symbols in PDF floor plans to estimate building costs. After exploring several approaches, we settled on using one of the YOLO (You Only Look Once) models—starting with YOLOv5. We made continuous improvements, training with more data (manually labeling PDFs), utilizing newer models like v8 and v10.

Initially, our approach included a training server that we would make requests to whenever a detection was requested. This was limited as it was extremely slow and not flexible—meaning if the user wanted to add new symbols to detect or modify the existing symbol set, the entire training process would need to be repeated each time. Additionally, training is a computationally expensive task.  If multiple users wanted to detect symbols at the same time, they would essentially be placed in a growing queue, leading to extremely long wait times. These limitations eventually lead to our transition to a one-shot detection approach.

After extensive research, we were able to create a one-shot pipeline that leveraged **pre-trained MobileNetV4 vision models, Leiden community detection algorithms, and synthetic data augmentation** to detect symbols. This process was significantly faster and allowed us to remove the need for a training server altogether. This new Leiden pipeline, along with several other key algorithms (template matching, text-search, distinct classes generation) that were developed later on, are discussed in detail on the inference server page. We were able to create a fast, responsive, and accurate application.

### Web Application

We built a web application using React and Next.js as the frontend framework, with FastAPI powering the backend server. React was chosen for its modern development ecosystem and component-based architecture, while Next.js was selected for its server-side rendering capabilities and built-in middleware support. FastAPI serves as our backend API, handling authentication validation, subscription management, and business logic.

For authentication, we implemented Supabase, which provides comprehensive auth functionality including login, signup, password reset, and user profile management. Our backend validates JWT tokens issued by Supabase using shared encryption keys, ensuring secure communication between the frontend and backend services.

For billing and subscription management, we integrated Stripe due to its robust API and built-in features like subscription checkout pages, customer management portals, and webhook handling. Our implementation includes trial period management, subscription status tracking, and automated billing updates through Stripe webhooks.

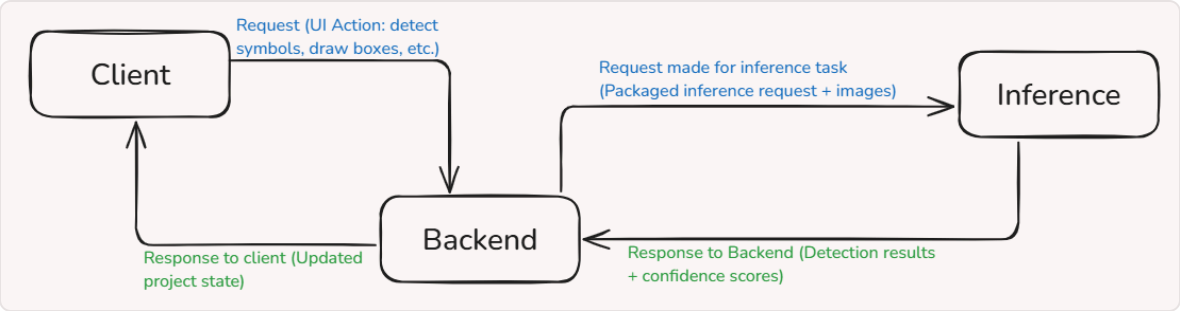
**Security Flow:** When a logged-in user makes a request, the frontend retrieves a JWT token from Supabase. This token is sent to our FastAPI backend, where it's validated using the shared SUPABASE_JWT_SECRET. The backend then processes subscription status checks and handles the business logic accordingly. This architecture ensures user data remains secure while maintaining seamless authentication across our application.

### Application Architecture

The application is split into 3 servers:

1. **Frontend - Client (Next.js/React)** Handles all UI/UX interactions and serves as the presentation layer. The client communicates with the backend through RESTful API calls, sending requests for project operations, symbol detection, and data retrieval. All user interactions—such as drawing boxes, selecting symbols, and viewing detection results—are handled through this interface.
2. **Server - Backend (FastAPI)** Serves as the central orchestrator and data persistence layer. This server:
    - Stores all project data as JSON files and images (due to Supabase storage limitations)
    - Maintains the authoritative "source of truth" for project state
    - Handles user authentication and subscription validation via Supabase JWT tokens
    - Manages all project operations (undo/redo, symbol management, measurement scales)
    - Orchestrates inference requests by packaging data and forwarding to the inference server
    - Processes and stores inference results returned from the inference server
    - Implements real-time state synchronization where the client reflects backend state changes
3. **Server - Inference (FastAPI + AI/ML Pipeline)** Dedicated to processing AI/ML workloads with specialized capabilities:
    - Handles multiple detection types: legend detection, template matching, page-wide detection, and text search
    - Implements a round-robin queue system for managing concurrent inference tasks across projects
    - Processes different request formats (single images, batch processing, region-specific detection)
    - Utilizes computer vision models including YOLO, Siamese networks, and custom algorithms
    - Sends results back to the backend server via HTTP callbacks
    - Maintains model caches and handles GPU/CPU resource management

**Communication Flow:** The architecture follows a request-response pattern where the frontend triggers actions, the backend validates and orchestrates the request, the inference server processes AI/ML tasks, and results flow back through the backend to update the client state. This separation ensures scalability, maintainability, and allows for independent scaling of compute-intensive inference workloads.




## Deployment

### Basic Setup

The servers are really just running on different terminals on the same computer at Owen's house. They operate on the same machine and use the same resources. There are options to separate the servers so that are on different computers, therefore giving us more computational access.  If anyone were to close those terminal windows, the web application would become unavailable for our users.

To access the server, we can go remote into it at IP: 10.0.0.5. We can remote into the computer using one of these passwords as long as we are connected to Owen's VPN. The VPN is necessary to allow us to access systems on Owen's network.

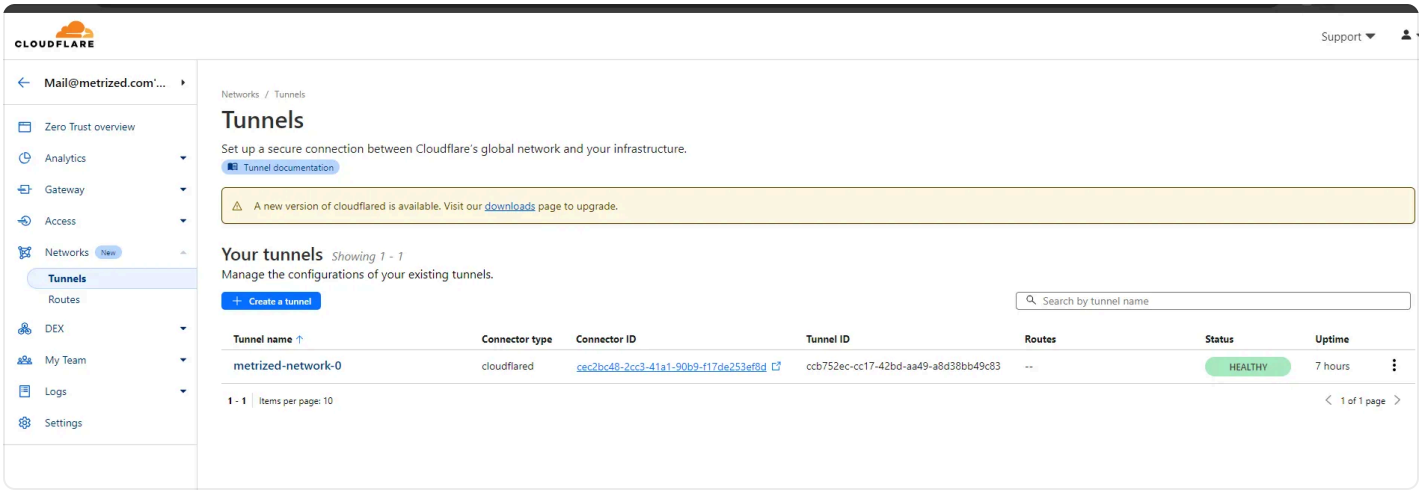
On Owen's computer (`metrized_server_0`) we use a cloned version of the metrized-symbol-counter to run the application. Typically, we just have a VScode instance open to make development and debugging easier. Keep in mind that each server has specific setup files and instructions (reference the ReadMe on GitHub).

**Tips for troubleshooting:**

- [ ] make sure to launch servers with the host option like `uvicorn wsgi:app --reload --port 8000 --host 0.0.0.0`
- [ ] make sure you're connected to the VPN depending on how you're accessing the server
- [ ] have all .env variables setup correctly

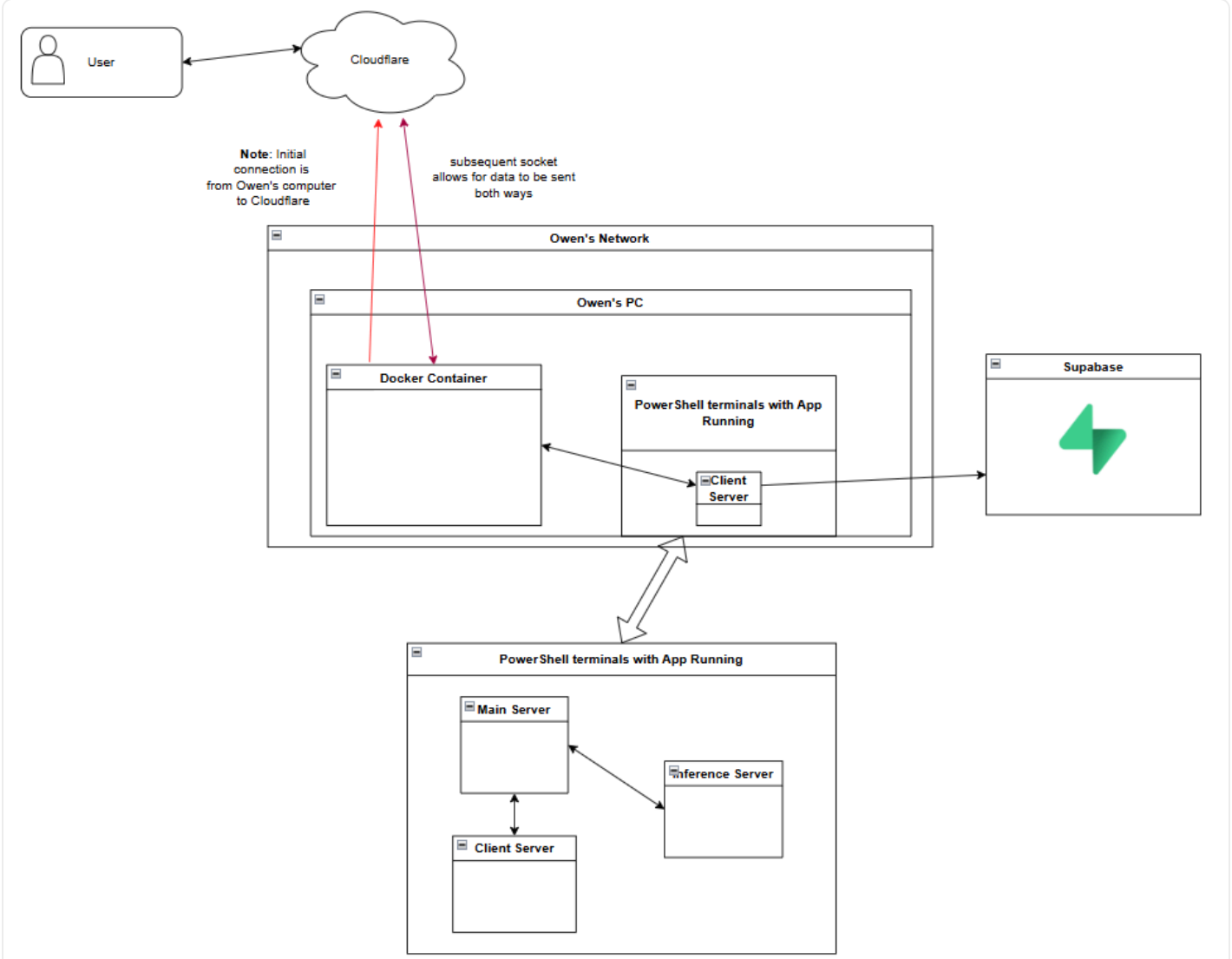
### CloudFlare & Docker

Now, let's look at how we get our website accessible through the internet:




First, note that we use a shared Cloudflare account to handle all request routing and redirections. We own the domain name "metrized.com." This allows us to use the main domain (as with our company website) or to add prefixes via CNAME records on Cloudflare. For instance, we have specified "symbol" as a prefix for our Symbol application. Any requests to symbol.metrized.com are rerouted to a designated domain (you can verify this by checking the DNS Records on Cloudflare). We also utilize a Cloudflare Tunnel setup. This setup involves running a Docker container on Owen's computer, which hosts a daemon. This daemon's purpose is to establish and maintain a secure connection between Owen's computer (which is otherwise inaccessible from external networks) and Cloudflare's servers. When a user makes a request to symbol.metrized.com , the request is securely tunneled to the daemon on Owen's Docker container. The container processes the request locally, and the response is sent back through the tunnel to the user. This configuration ensures secure and reliable access to the app with minimal setup.

**This shows the overall setup of our deployed application:**




Learn more about Cloudflare Tunnels here. For more information on how exactly our daemon was initialized refer to Noah.

# Frontend - Client

## Overview

The front-end of our application is designed with two main purposes in mind

1. Providing electrical contractors with an intuitive and efficient interface for performing ML-assisted labelling of electrical drawings
2. Allowing the developers of this application to easily generate and modify floorplan image labels for training the ML models that are used for symbol detection.

> ⚠️ **Setup: Follow and complete the setup instructions from the Metrized-Symbol-Counter GitHub Repo and ensure all servers are running locally without issue before proceeding.**

## Tech Stack

---

### TypeScript

TypeScript Handbook | Google TypeScript Style Guide

We use React with TypeScript for our frontend codebase to leverage the benefits of static typing. This choice enhances code quality and developer productivity, ensuring type safety and facilitating easier maintenance and debugging.

### Next.js

Next Docs

Next.js serves as our React framework, chosen for its enhanced performance, server-side rendering capabilities, and easy routing. Additionally, Next.js makes it easy to do SEO, but this capability isn't utilized.

> ℹ️ At the time of writing, we are using Next 14 with the App Router.

### Tailwind CSS

Tailwind Docs

For styling, we've (mostly) adopted Tailwind CSS, a utility-first CSS framework that enables us to build custom designs inline with JSX code. This approach speeds up the development process and avoids complicating styling with single use classes.

### Supabase

Supabase Docs

Supabase is a backend as a service platform that provisions and manages PostgreSQL databases that can be interacted with through their APIs (or directly with SQL queries). We lean on Supabase's auth services to manage user sign-ups and logins (including with Google OAuth) to restrict access to our API services to the appropriate authenticated users. We also use their database services to manage user profiles (names, avatar_urls) and store uploaded avatars into Supabase storage (a managed S3 bucket). Supabase has a free-tier email limit, so we've setup our own SMTP email server. Owen has control over this server. See Project Settings > Authentication > SMTP Settings

> At the time of writing, Supabase allows up to 50,000 users, 1GB storage, 5GB bandwidth and unlimited API calls under their free tier making it suitable for our application with the ability to be scaled up or migrated elsewhere.

> For more information on our setup, including middleware and App routing, check out this documentation.

## Supabase - Authentication and Profile Management

There are **three** key parts to our authentication and authorization system which ensures that only the owner of a project can access and modify its data:

- **Supabase Auth:** provides a hosted PostgreSQL database that includes user management and authentication capabilities. It allows for easy setup of sign-in/sign-up workflows, secure management of user sessions, and integration with third-party providers. Through its authentication API, developers can handle user data securely without having to implement complex authentication logic. Supabase Auth uses JWTs (JSON Web Tokens) for maintaining user sessions, ensuring that once a user is logged in, they receive a token which is used to authenticate subsequent requests to the server.
- **Supabase Database:** stores and manages subscription data in the `public.subscriptions` table, which tracks user subscription status, plan details, trial periods, and Stripe integration data. This serves as the source of truth for determining user access levels and feature permissions. The backend validates subscription status by querying this table before processing protected requests, ensuring only subscribed users can access premium features.
- **JWT (JSON Web Token):** is a compact means of securely sending JSON data between two parties. In our authentication system, JWTs are used to ensure that a user has been authenticated and to securely transmit information about their session. When a user logs in using Supabase Auth, they are issued a JWT which contains a set of claims including the user's identity and any additional metadata required (email, id etc.). This token is then sent with each request to the main server where it is validated against the JWT Secret to confirm the validity of that JWT. This validation process is essential for authorizing access to sensitive user data and for actions such as modifying project details specific to the authenticated user. The use of JWTs enables stateless authentication, meaning that the server does not need to keep a record of user sessions, thereby reducing server overhead and improving scalability.

▼ More on JWTs

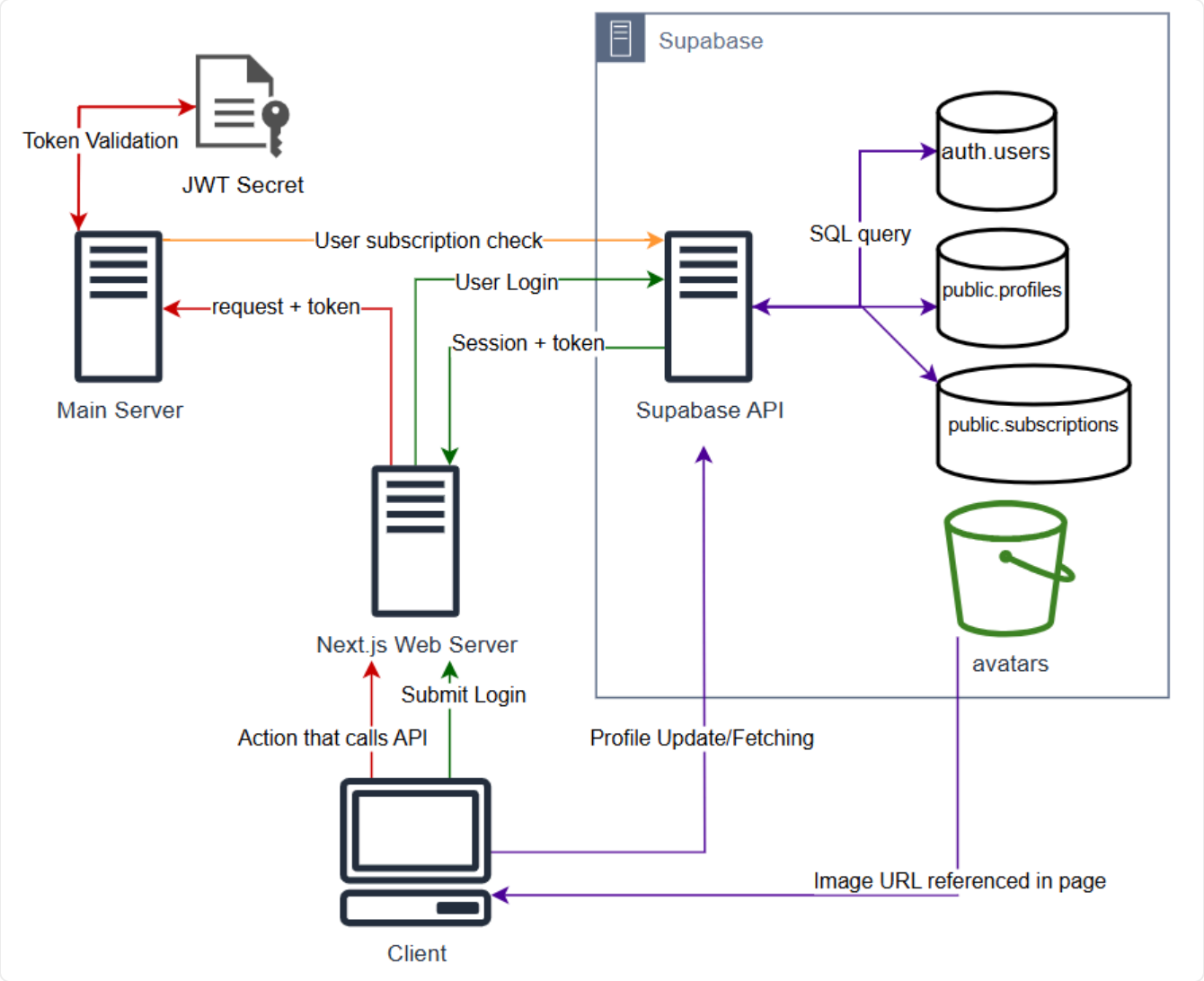
- The JSON Web Token is made of three parts that are combined into a single string
  i. **Header**: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 (HS256).
  ii. **Payload**: The payload contains the claims. These claims are statements about an entity (typically, the user) and additional data. The payload is a JSON object that has been Base64Url encoded, not encrypted or hashed, which means it can be easily decoded to reveal the claims in clear text.
  iii. **Signature**: To create the signature part, the encoded header, the encoded payload, a secret, and the algorithm specified in the header are used. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
- When the JWT is sent to the server, the server will decode the header and payload, then verify the signature by using the secret key to sign the header and payload again. If the signature created by the server matches the signature on the token, it confirms that the token is valid and the data has not been tampered with.
- It's important to note that while the payload is encoded, encoding is not the same as encryption. The contents of the payload can be read by anyone who intercepts the token. That's why sensitive information should not be stored in the JWT payload. The main purpose of the JWT is not to hide the data but to ensure the authenticity and integrity of the data.
- The JWT is the stored in the browser's cookies. Most importantly, it contains an access token which gives authenticated users access to the application until expiry (currently 1 hour). It also contains the refresh token which

automatically refreshes the JWT access token after expiry.

With the incorporation of Next.js 14, which includes app routing capabilities, the Next.js web server processes requests using route handlers. These handlers serve as the point of contact for requests coming from the client, making the Next.js server the primary entity that interfaces with the Supabase API and main server (our backend). This setup enhances security by not exposing the Supabase API, Supabase anon key and the main server directly to the client. Here is the authentication flow with Next.js route handlers:

1. **User Sign-Up/Login**: The user interacts with the client application, providing credentials through the interface built with Next.js
2. **Next.js Route Handler**: When the user attempts to sign up or log in, the client application sends the request to a specific Next.js route handler designed for authentication purposes - `/auth/login` and `/auth/signup`
3. **Supabase Authentication**: The Next.js route handler takes the credentials and communicates with Supabase Auth via Supabase's API to perform the authentication. Once authenticated, Supabase generates a JWT and sends it back to the Next.js route handler
4. **Token Handling**: The JWT is then relayed from the Next.js route handler back to the client, where it is stored in cookies, to maintain the session (middleware is used to refresh these tokens).
5. **Making Authenticated Requests**: Whenever the client needs to perform actions that require backend access, it sends a request to the `/api` route handler with the details of the request being performed.
6. **Next.js Server As Proxy**: The Next.js server, upon receiving the request details from the client, attaches the JWT and forwards the requests to the main server. By doing this, Next.js acts as a proxy, thus concealing the direct endpoints of the backend services and any other secrets.
7. **Server Token Validation**: The main server validates the JWT with the secret key. Valid tokens confirm the user's identity and permissions for the requested operation.
8. **Subscription Authorization:** For protected features, the main server queries the Supabase `subscriptions` table to verify the user has an active subscription or valid trial status before processing the request.
9. **Authorization and Processing**: If the user is validated and authorized, the main server performs the requested actions, such as retrieving or updating project data.
10. **Responding to the Client**: After the main server processes the request, it sends the results back to the Next.js server. The Next.js route handler then forwards this response to the client.
11. **User profiles:** separately, user profiles are managed in Supabase in the `public.profiles` table which contains user information such as names and avatar_urls which are retrieved on the client as required.
12. **Subscription Management:** User subscription data is stored in the `public.subscriptions` table, automatically synchronized with Stripe via webhooks. This includes subscription status, plan types, trial periods, and billing information.
13. **Profile Management:** users can modify these profile details through forms in the profile page which call on Next.js server actions, updating the data in this table via the Supabase API. Supabase also provides storage (a managed S3 bucket) where we upload avatar images to. The objects in this bucket are made public and linked to the `avatar_url` field in the profiles table so that we can refer to them in the images used in our UI.

This process is detailed below:




## Project/File Structure

Next.js Recommendations | React Recommendations | Shadcn Recommendations

Outside of the file structure requirements for using the Next.js App Router that have each folder under `app/` define a new route so long as a `page` file is found within (and an optional `layout` file for creating inherited layout components), there are no strict requirements on where JavaScript (or TypeScript in this case) files must be stored in a Next.js project. The file structure that we have adopted is outlined below.

```
.
├── public
└── src/
    ├── app/
    │   ├── new-project/
    │   │   └── page.tsx
    │   ├── project/
    │   │   └── [id]/
    │   │       ├── layout.tsx
    │   │       └── page.tsx
```

```
│   ├── layout.tsx
│   └── page.tsx
├── components/
│   ├── canvas/
│   │   └── ...
│   ├── projects-table/
│   │   └── ...
│   └── ui/
│       └── ...
├── contexts/
│   └── ...
├── hooks/
│   └── ...
├── lib/
│   └── ...
└── services/
    └── ...
```

▼ created with https://tree.nathanfriend.com/

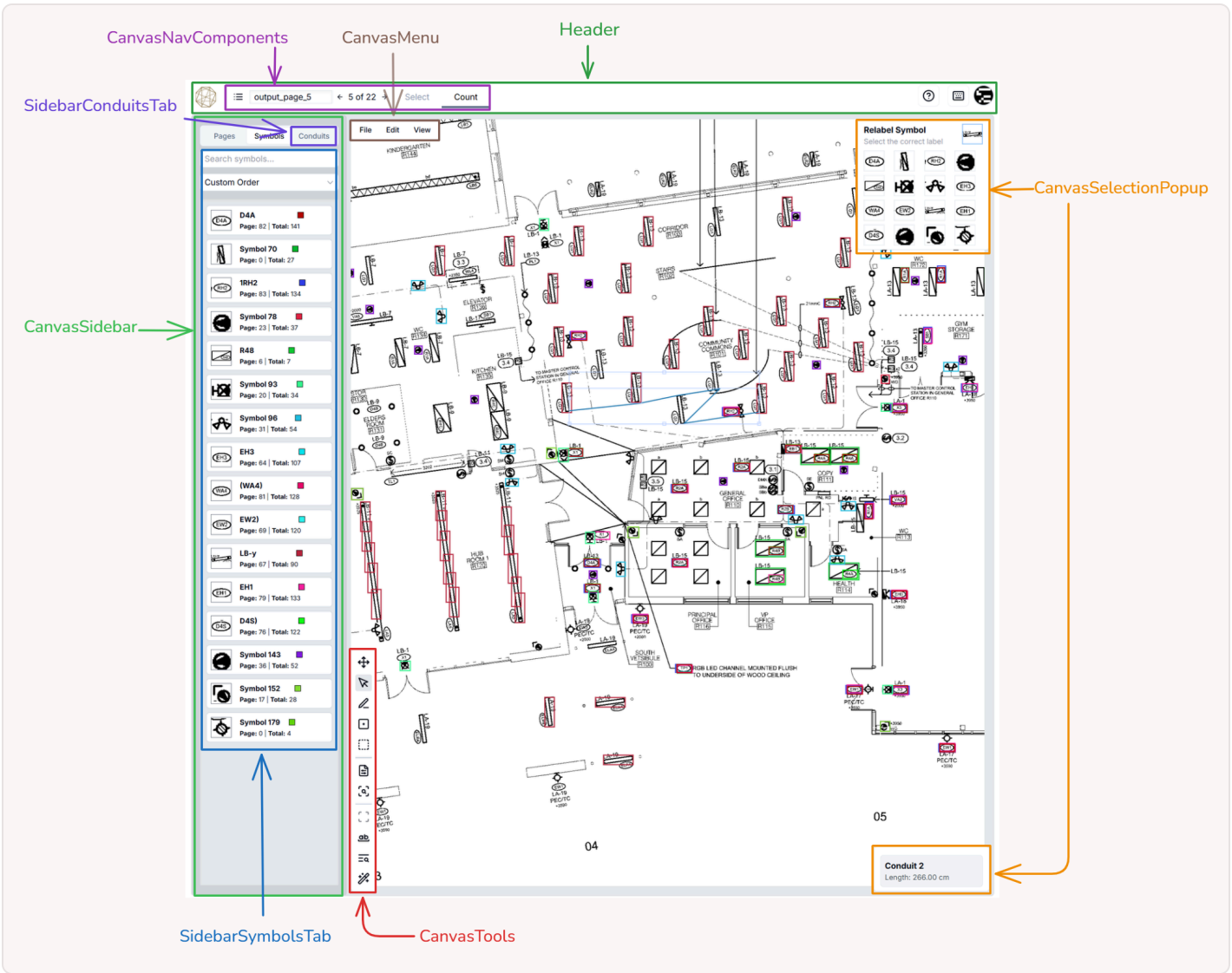
```
public
src
  app
    new-project
      page.tsx
    project
      [id]
        layout.tsx
        page.tsx
    layout.tsx
    page.tsx
  components
    canvas
      ...
    projects-table
      ...
    ui
      ...
  contexts
    ...
  hooks
    ...
  lib
    ...
  services
    ...
```

- `components/` contains the various React components that are used across the application. Subfolders are used for grouping components that belong to a specific 'feature' when doing so improves ease of use

- `contexts/` contains different React context definitions that are used for different aspects of state management within the project. (Contexts allow you to access project details from any component that's wrapped in the context provider. The suboptimal alternative is to manually pass data through child parent and child components)
- `hooks/` contains custom **React hooks** that are used for encapsulating different logic and data (`useTools` handles logic and design of the tool column and `useCanvasManager` handles changes on the canvas. `useLiveProjectStates` handles live updates of proejcts on the dasboard)
- `lib/` contains types.ts (all the custom type definitions) and any util files with miscellaneous TypeScript functions
- `services/` contains functions for making calls to the various backend endpoints associated with different user actions grouped based on which objects are being interacted with (for example `updateSymbolBox` in `symbolServices.ts` updates the box location on the backend)

### Components

Components are made pretty arbitrarily based on when it seems to "make sense" but below shows some of the key components that are used in the `/project/[id]` page




### Contexts (State Management)

Everything that makes the app responsive and modify what is rendered on the screen (where symbols are, what page is selected, what tabs are open etc.) is a part of React state. There are certain component-specific state for determining things

like managing a toggle value or opening a drawer, but the the most important parts of the applications state that are accessed and modified across many components are organized within six main Contexts:

- **BillingContext**: manages subscription-related UI and global billing events. Listens for "subscription-required" events from the backend (when APIs return 402 status) and automatically displays billing prompts to users
- **CanvasContext:** pieces of state relevant to the Fabric.js canvas (where is the user's cursor, what is the current scale, where is the background image placed relative to the canvas)
- **CertificationContext**: handles symbol certification workflows with optimistic updates. When users certify/uncertify symbols, the UI updates immediately while API requests happen in the background, with automatic rollback on errors
- **ModelContext:** configuration parameters for inference (confidence and IOU threshold)
- **ProjectContext:** mainly houses projectData state which is a mirror of all the data stored on a project in the server (project names, symbol details and locations, detection details and locations etc.)
  - note that projectData changes originate from the server - the client never directly modifies this data, only requests changes through API calls
- **UIContext:** state relevant to changing what the user sees on the screen without modifying projectData (what tab is selected, which symbols are selected, what tool is selected and much more)

When using the app, such as drawing boxes around symbols, you'll notice that there's no noticeable loading time, allowing the app to update its state smoothly. This is achieved through several mechanisms:

**Optimistic Updates**: While drawing a box around a new symbol, the box is visually created immediately on your device (client-side state). Once you finish drawing and release the cursor, a request is sent to the server to update the project's data with the new box. The server then sends back the updated project data, which replaces the current data on your screen. If the server request fails, the client-side changes are automatically reverted.

**Live Updates**: We use a polling mechanism to get live project updates from the server, particularly for inference requests. This ensures the frontend immediately reflects any new changes or data received from the inference server, keeping all users synchronized.

**State Separation**: UI state (selections, tool modes, sidebar visibility) is managed separately from project data, allowing immediate visual feedback while data changes are processed asynchronously.

These contexts work together through custom hooks that provide clean, type-safe access to state and ensure proper error handling when components try to access context outside of their providers.

### Hooks

Hooks are custom functions that encapsulate complex business logic and provide clean interfaces for components to interact with application state and functionality.

▼ **useCanvasManager**

The core hook that manages the Fabric.js canvas and all user interactions with it. This hook:

- Initializes and configures the canvas with background images and drawing objects
- Handles all mouse and keyboard events (drawing, selecting, zooming, shortcuts)
- Manages the complete tool system (drawing tools, detection tools, selection tools)
- Coordinates real-time polling for ML inference updates
- Handles optimistic updates and error recovery for canvas operations
- Syncs canvas object states with React context state

▼ **useTools**

Defines and manages the application's tool ecosystem. This hook:

- Returns an array of tool definitions with their properties (name, icon, shortcuts, hover content)
- Handles dynamic tool behavior that changes based on application state
- Manages tool availability and disabled states based on context
- Provides click handlers that integrate with canvas functionality and API calls
- Controls tool-specific UI elements like hover tooltips and configuration options

▼ **useLiveProjectStates**

Manages real-time data synchronization across the application. This hook:

- Establishes and maintains WebSocket connections with authentication
- Receives live project updates from the server
- Merges incoming data with existing project state
- Handles connection lifecycle (heartbeat, reconnection, cleanup)
- Used primarily in the dashboard to show live project status updates

These three hooks work together to create the interactive canvas experience: `useTools` defines what actions are available, `useCanvasManager` executes those actions on the canvas, and `useLiveProjectStates` keeps everything synchronized with the server in real-time.

### Services

The services layer handles all backend API communication, organized by domain functionality with consistent authentication and error handling patterns. Services are grouped into four main categories: **symbolServices** manages symbol detection, creation, and classification operations; **projectServices** handles project data retrieval, lifecycle management, exports, and history operations; **groupServices** provides organization functionality for both symbol and conduit groups; and **serviceUtils** contains core utilities including `authorizedFetch()` for automatic authentication, custom error handling with status codes, and file operation helpers. All services follow consistent patterns for request structure, error propagation, and authentication, providing a clean interface between React components and the backend API while automatically handling common concerns like token injection and billing error detection.

**Consistent Structure**:

```
export const serviceName = async (projectId: string, ...params) => {
  const path = `/api/endpoint/${projectId}`;
  const body = JSON.stringify(data);
  return await authorizedFetch(path, { method, headers, body });
};
```

**Authentication**:

- All services use `authorizedFetch()` for automatic token injection
- Bearer token authentication with Supabase integration

### Usage Pattern

```
try {
  const result = await serviceCall(params);
  setProjectData(result);
```

```
  } catch (error) {
    handleError(error); // Component handles display
  }
```

Services provide a clean, type-safe interface between React components and the backend API, with consistent patterns for authentication, error handling, and data transformation across all domains.

# Server - Backend

## Overview

The Metrized Symbol Counter backend is a comprehensive system designed to automatically detect, classify, and count symbols in engineering drawings and technical documents. The system processes PDF documents by converting them to images and using machine learning models to identify recurring symbols, enabling users to quickly quantify elements across large document sets.

> ⚠️ Setup: Follow and complete the setup instructions from the Metrized-Symbol-Counter GitHub Repo and ensure all servers are running locally without issue before proceeding.

### What It Does

The backend handles the complete lifecycle of symbol detection and counting:

**Document Processing**: Accepts PDF uploads, converts pages to images at various thresholds, and generates preview versions for efficient client-side rendering.

**Symbol Detection**: Uses computer vision models to automatically identify symbols in documents, with support for both graphical symbols and text elements through OCR.

**Manual Classification**: Provides tools for users to manually define legend symbols, correct detection results, and organize symbols into groups for better management.

**Counting & Analysis**: Maintains real-time counts of each symbol type across all document pages, with the ability to track changes and maintain detection history.

**Export & Integration**: Offers multiple export formats (CSV, YOLO labels, classification data) for integration with other engineering workflows.

## Core Workflows

### 1. Project Creation & Setup

- User uploads a PDF document
- System converts PDF to images at multiple threshold levels
- Auto-detection attempts to identify inference regions (areas likely to contain symbols)
- Project structure is created with pages, folders, and metadata

### 2. Legend Definition

- Users manually select symbol examples from the document
- Symbols are cropped, stored, and assigned names/categories
- Symbol groups can be created for organizational purposes
- Templates are prepared for automated detection

### 3. Automated Detection

- **Page Detection**: Scans entire pages or regions for known symbol types
- **Template Matching**: Uses cross-correlation for precise symbol matching

- **Distinct Class Detection**: Automatically discovers new symbol types using clustering
- **Text Search**: OCR-based detection for finding specific text symbols or labels

### 4. Result Refinement

- Manual symbol addition/removal on pages
- Re-classification of detected symbols
- Batch operations for efficiency
- Undo/redo system for safe experimentation

### 5. Analysis & Export

- Real-time symbol counts across all pages
- Statistical summaries and reports
- Export in various formats for downstream tools
- Project archival and sharing capabilities

## System Architecture

The backend consists of two cooperating servers:

**Main Server** (`localhost:8000`): Handles user requests, project management, file storage, authentication, and coordinates the overall workflow. Acts as the primary API gateway for frontend clients.

**Inference Server** (`localhost:8001`): Dedicated to machine learning operations including symbol detection, OCR processing, similarity computation, and other compute-intensive tasks. Uses queue-based processing to handle multiple concurrent requests efficiently.

## Asynchronous Programming Patterns

All of our servers utilize asynchronous code to handle computationally expensive operations (creating projects, running inference) without blocking requests for other users.

> ⚠️ **Important**: Our FastAPI server runs with only one worker to avoid conflicting states between workers and prevent race conditions.

### Why Async Matters

Consider sending a symbol detection request to the inference server. It makes little sense to have the main server and client wait minutes for a response while machine learning models process large documents. Instead, we quickly validate the request, then send a positive response. The main server records that detection is happening, the client shows an updated state, and the inference server starts an asynchronous function that processes the detection task.

For example, when a user runs "detect symbols on all pages," the system might need to:

- Process dozens of high-resolution images
- Run computer vision models on each image section
- Apply similarity matching against legend symbols
- Compute final symbol counts and classifications

This entire pipeline can take several minutes for large documents, but the user gets immediate feedback that their request was accepted and will receive real-time updates when it the inference completes.

### Pattern 1: Fire-and-Forget Tasks

For operations where completion time is not critical for the response:

```
# Note on async best practices:
# - Modify shared data all at once or use locks to manage access
# - Yielding control (await) can allow other tasks to run, potentially causing race conditions

async def some_task(some_args):
    """
    Performs an operation asynchronously, not blocking the thread.
    Completion time is not critical for the client response.
    """
    # Task logic here (e.g., symbol detection, file processing)

def function_to_run_after_task_finishes():
    """
    Callback function for post-task actions, e.g., flagging necessary client updates
    """
    mark_to_be_updated(project_id)  # Notify clients of changes

@app.post("/some/endpoint")
async def some_endpoint_func():
    """
    Endpoint that quickly returns while scheduling 'some_task' asynchronously.
    """
    validate_request()
    data = "Response data"
    task = asyncio.create_task(some_task(some_args))
    task.add_done_callback(lambda t: function_to_run_after_task_finishes())
    return data
```

### Pattern 2: Critical Completion Tasks

For operations where task completion is critical before responding:

```
async def critical_task(some_args):
    """
    Performs a critical operation where completion time matters (I/O operations).
    """
    async with httpx.AsyncClient() as client:
        response = await client.post(api_url, files=files, data=params, timeout=timeout_config)
        if response.status_code != 200:
            raise HTTPException(status_code=500, detail="Error from server")

    # Additional async operations
    data = await read_file_async()
    return "Task result data"
```

```python
@app.post("/another/endpoint")
async def another_endpoint_func():
    """
    Endpoint that awaits critical task completion before returning.
    """
    validate_request()
    data = await critical_task(some_args)
    return data
```

### Pattern 3: Process-Based Tasks

For CPU-intensive operations that need separate processes:

```python
def worker_wrapper():
    """
    Creates a process pool for CPU-intensive tasks like PDF processing.
    """
    pool = ProcessPoolExecutor(max_workers=4)

    async def worker(args):
        name, temp_file_path, user_data = args
        loop = asyncio.get_running_loop()
        return await loop.run_in_executor(pool, process_project, name, temp_file_path, user_data)

    return worker

worker_that_does_task = worker_wrapper()

async def setup_process(args):
    """
    Sets up a process asynchronously, handling data post-completion.
    """
    data = await worker_that_does_task(args)
    # Handle or save the data in main process context
    projects[data.id] = data
    save_project(data.id)

@app.post("/process/endpoint")
async def process_endpoint_func():
    """
    Quickly returns while setting up an asynchronous process.
    """
    validate_request()
    args = ("Project Name", "/path/to/temp/file", {"user": "data"})
    task = asyncio.create_task(setup_process(args))
    task.add_done_callback(lambda t: function_to_run_after_task_finishes())
    return "Process initiated"
```

### Real Examples in Our Codebase

**Symbol Detection**: Uses Pattern 1 - endpoint returns immediately, detection runs in background, polling notifies clients when complete.

**File Uploads**: Uses Pattern 2 - must complete file validation and initial processing before responding.

**PDF Processing**: Uses Pattern 3 - creates separate processes for CPU-intensive PDF conversion and image processing.

## Tech Stack

### FastAPI

FastAPI is a modern, fast web framework for building APIs with Python. It is designed for high performance and is built on standard Python type hints, making it easy to create robust applications. FastAPI automatically generates interactive API documentation and simplifies data validation and serialization.

**Key Features in Our Application:**

- **Route Decorators**: We use decorators like `@app.get("/some-url")` or `@app.post("/some-url")` to define endpoints
- **Automatic Documentation**: Visit `/docs` when the server is running to see interactive API documentation
- **Type Validation**: Request/response models are automatically validated using Pydantic
- **Dependency Injection**: We use `Depends()` for authentication, database connections, etc.

### PyTorch

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab. It provides tools and libraries for deep learning, and is known for its flexibility and ease of use. PyTorch supports dynamic computation graphs that allow modifications to be made on-the-fly during processing.

**Installation**: Our servers require CUDA-enabled (GPU) PyTorch:

```
pip3 install torch torchvision --index-url <https://download.pytorch.org/whl/cu118>
```

## Key Components

### Server Startup

Each FastAPI server contains a lifespan function that manages startup and shutdown:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup code - runs when server starts
    try:
        # Load project states from JSON files
        for file in os.listdir(SAVED_PROJECTS_DIR):
            project = read_project_from_json(json_path)
            projects[project.id] = project
        yield
    finally:
        # Shutdown code - runs when server stops
        # Clean up resources, save final state
        pass
```

**Why Lifespan Functions Are Critical:**

- **Multi-processing Safety**: Python creates new processes that re-run the entire file. Without lifespan functions, startup code would run multiple times
- **Resource Management**: Ensures proper loading/cleanup of models, database connections, and file handles
- **State Consistency**: Guarantees project data is loaded once and properly synchronized

# Main Server

## Overview

The main server is the central hub of the Metrized Symbol Counter system, serving as the sole point of contact for client applications and orchestrating all operations across the platform. It acts as the authoritative source for project data and coordinates requests between clients and our authentication + inference servers.

## Responsibilities

The main server handles all core application functionality:

**Project State Management**: Maintains complete project data in memory during runtime, including pages, legend symbols, detected symbols, groups, and user preferences. All state modifications flow through the main server to ensure consistency.

**File Storage & Management**: Stores and serves all project-related files including PDF conversions, page images at multiple thresholds, symbol crops, legend symbols, and JSON project snapshots.

**Real-Time Communication**: Manages WebSocket connections to provide live updates during long-running operations like symbol detection, ensuring clients stay synchronized with server state.

**API Gateway**: Provides the complete REST API surface for clients, handling authentication, validation, and business logic for all user operations.

**Inference Coordination**: Dispatches detection requests to the inference server and processes results, managing the complete detection pipeline from request to final symbol counts.

**Authentication & Authorization**: Integrates with external authentication services to verify users and enforce project-level access controls.

**Export Services**: Generates various export formats (CSV reports, YOLO labels, classification datasets) for integration with external tools and workflows.

## Data Storage Architecture

### In-Memory State

The server maintains several critical data structures in memory:

```
projects: Dict[str, ProjectData] = {}           # Active project states
histories: Dict[str, History] = {}              # Undo/redo history
user_to_project_ids: Dict[str, List[str]] = {}  # User access mapping
tracker = InferenceTracker()                    # Inference job tracking
```

### Persistent Storage

Projects are persisted to disk in two ways:

**JSON Snapshots**: Complete project state saved to `data/projects/saved/{project_id}.json` for reload on server restart.

**File Assets**: Organized directory structure per project:

```
data/projects/{project_id}/
├── conversion/
```

```
│   ├── originals/           # Original PDF pages as images
│   ├── transformed/         # Default threshold images
│   ├── transformed_threshold_{value}/  # Specific threshold versions
│   └── previews/           # Low-resolution sidebar images
├── legend_symbols/         # User-defined symbol templates
└── symbols/               # Temporary detection crops
```

### State Persistence Rules

- **Memory Only**: Newly created projects exist only in memory until explicitly saved
- **Auto-Save**: Most operations automatically trigger `save_project()` for persistence
- **History System**: Every significant change calls `add_current_state_to_history()` for undo/redo support

## Data Model

All data structures inherit from Pydantic `BaseModel` for type safety and validation. The core project structure:

```python
class ProjectData(BaseModel):
    id: str
    user_id: str = ""
    name: str
    measurement_scale: MeasurementScaleRequest = {}
    inference_template: Dict[str, int] = {"x1": 0, "y1": 0, "x2": 0, "y2": 0}
    pages: Dict[str, PageData] = {}
    symbol_groups: Dict[str, SymbolGroup] = {}
    conduit_groups: Dict[str, ConduitGroup] = {}
    legend_symbols: Dict[str, LegendSymbol] = {}
    curr_symbol_num: int = 1
    curr_group_num: int = 1
    curr_conduit_num: int = 1
    status: ProjectStatus = ProjectStatus.LEGEND
    pending_distinct_popup: bool = False
    pending_text_search_popup: bool = False
```

**Key Principles:**

- **Single Source of Truth**: Only the main server modifies project data
- **Immutable Operations**: Other servers process requests and return results; they never directly alter project state
- **Centralized State**: All project information resides in one location, eliminating redundancy and conflicts

## Authentication & Security

The server implements a dependency injection pattern for authentication:

python

```python
@app.post("/some-endpoint")
async def endpoint(user_data: UserData = Depends(get_current_user)):
    check_project_id(project_id)
```

```
    check_user_authentication(project_id, user_data)
    # Endpoint logic...
```

**Authentication Flow:**

1. Client sends Bearer token in Authorization header
2. `get_current_user()` validates token with external auth service
3. `check_user_authentication()` verifies project ownership
4. Request proceeds or returns 401/403 error

## Key Endpoint Categories

### Project Lifecycle

- `POST /create-project` - Upload PDF and initialize project structure
- `GET /project/{project_id}` - Retrieve complete project state
- `POST /save-project/{project_id}` - Persist project to disk
- `POST /delete-project/{project_id}` - Remove project and all files

### Symbol Detection

- `POST /detect-legend-symbols-box/{project_id}` - Detect legend symbols on page
- `POST /detect-page-symbols-box/{project_id}` - Symbol Detection
- `POST /detect-page-distinct-classes/{project_id}` - Detect and identify distinct symbols
- `POST /search-for-text-symbol/{project_id}` - OCR-based text detection
- `POST /detect-page-template/{project_id}` - Detects inference region (template)

### Symbol Management

- `PATCH /add-legend-symbol/{project_id}` - Create manual legend symbol
- `PATCH /add-page-symbol/{project_id}` - Create manual detection symbol
- `PATCH /delete-legend-symbol/{project_id}` - Deletes legend symbols
- `PATCH /update-page-symbol-classes-without-page/{project_id}` - Bulk relabel symbols
- `PATCH /promote-symbol-to-legend/{project_id}` - Convert detection to legend template

### Subscription / Billing Management

- `PATCH /checkout-session` - Create a Stripe hosted checkout page
- `PATCH /manage-subscription-billing-portal` - Allows user to access Stripe hosted billing portal

### Real-Time Updates

- `WebSocket /live-project-states` - Live project state synchronization
- `GET /poll-project-model-updates/{project_id}` - Polling-based state updates

## Concurrency & Threading

**Thread Safety**: The server uses a `data_lock` for protecting project data modifications in async functions that yield control.

**Process Isolation**: CPU-intensive operations (PDF processing) run in separate processes via `ProcessPoolExecutor` to avoid blocking the main thread.

**Async Task Management**: Long-running operations use `asyncio.create_task()` with callbacks for completion notification:

```
task = asyncio.create_task(handle_detection(project_id, data))
task.add_done_callback(lambda t: mark_to_be_updated(project_id))
```

## Performance Optimizations

**Image Caching**: Multiple threshold versions pre-generated and cached for fast access.

**WebSocket Efficiency**: Only sends project updates when actual changes occur, tracked via state comparison.

**File Serving**: Uses ETags and conditional requests to minimize unnecessary image transfers.

**Batch Processing**: Groups related operations (like multi-page detection) into single inference requests.

## Integration Points

**Inference Server**: Sends detection requests via HTTP POST with multipart form data containing images and parameters. Results return asynchronously to `/provide-inference/` endpoints.

**Authentication Service**: Validates user tokens and manages subscription status through external HTTP API calls.

**Client Applications**: Serves as the complete API backend, providing REST endpoints for all functionality plus WebSocket connections for real-time updates.

The main server's architecture ensures all data flows through a single, authoritative source while efficiently coordinating complex multi-step workflows across distributed services.

# Inference Server

The inference server is a specialized machine learning service dedicated to processing computer vision and OCR tasks for the Metrized Symbol Counter system. It operates independently from the main server, handling computationally intensive operations through a round robin queue-based architecture that ensures efficient resource utilization and concurrent processing.

## Overview

The inference server processes all machine learning workloads including symbol detection, template matching, OCR text recognition, and similarity analysis. It maintains no persistent state, functioning purely as a processing engine that receives requests, executes ML operations, and returns results to the main server.

## Responsibilities

**Symbol Detection**: Runs RF-DETR based object detection models to identify symbols in document images, supporting various detection modes including full-page, region-based, and template-guided detection.

**Template Matching**: Performs advanced cross-correlation template matching with multi-angle rotation support (0°, 90°, 180°, 270°) and two-stage refinement for precise symbol localization.

**OCR Processing**: Executes optical character recognition pipelines for text detection and extraction, including multi-stage OCR with error correction and mislabeled character handling.

**Similarity Analysis**: Computes symbol similarities using embedding-based approaches, Siamese networks, and cosine similarity metrics for symbol classification and merge suggestions.

**Distinct Class Discovery**: Implements clustering-based approaches (HDBSCAN + OCR) to automatically discover new symbol types without manual classification.

**Region Detection**: Automatically identifies inference regions in documents to focus detection efforts on relevant areas.

## Architecture

### Queue-Based Processing

The server uses a queue system to handle concurrent requests efficiently:

```
def round_robin(queue: Queue):
    """
    Continuously drains the global input queue, sorting tasks into per-project queues.
    Once no new tasks arrive for a short period, it processes tasks in round robin.
    """
    project_queues = defaultdict(deque)
    # Process one task per project queue to ensure fairness
    for project_id, q in project_queues.items():
        if len(q) == 0:
            continue
        request = q.pop()
        inference_task(request)
```

**Benefits:**

- **Fair Processing**: Round-robin ensures no single project monopolizes resources
- **Batch Efficiency**: Accumulates related tasks before processing
- **Resource Management**: Prevents memory overload from concurrent heavy operations
- **Project Isolation**: Each project's tasks are processed independently
- **Real Life Example**: Consider Person A who starts a 25-page inference task, then Person B starts a 1-page inference task. Without this queue system, Person B would have to wait for all 25 pages from Person A to finish. With the round-robin queue, Person B only has to wait for 1 page from Person A before their task gets processed, ensuring fair access to computing resources.

## Machine Learning Models

### RF-DETR: General Symbol Detection

```
ROBOFLOW = od.RoboflowModel(
    weights=MODEL_WEIGHTS,
    confidence_threshold=0.15,
    iou_threshold=0.20
)
```

**Purpose**: Main symbol detection using RF-DETR architecture
**Input**: Document page images (sliced into 728x728 crops)
**Output**: Symbol bounding boxes with confidence scores

### RF-DETR: Inference Region Detection

```
ROBOFLOW_INFERENCE_REGION = od.RoboflowModel(
    weights=INFERENCE_REGION_WEIGHTS,
    confidence_threshold=0.80,
    iou_threshold=0.20
)
```

**Purpose**: Automatically identifies areas likely to contain symbols
**Input**: Full document pages
**Output**: Region bounding boxes for focused detection

### Siamese Similarity Network

```
SIAMESE = SiameseLit(
    checkpoint_path=SIAMESE_PATH,
    small=False
)
```

**Purpose**: Computes precise similarity scores between symbol patches
**Input**: Pairs of symbol image crops
**Output**: Similarity confidence scores (0-1)

### PaddleOCR Engine

```
ocr = PaddleOCR(use_angle_cls=True, lang='en', show_log=False)
```

**Purpose**: Text recognition and extraction from symbol regions
**Input**: PIL Images (individual symbol crops or full page images)
**Output**: Recognized text with confidence scores

## Detection / Inference Pipelines:

### Caching

```
cache = Cache(SYMBOL_DETECTOR_CACHE)
key = cache.compute(img, conf_threshold, iou_threshold)
if key in cache:
    return cache[key]
```

**Symbol Detection Cache**: Stores RF-DETR results based on image content and parameters
**OCR Cache**: Project-level caching of OCR results to avoid reprocessing
**Cache Keys**: Generated from image hashes and detection parameters

### Request Flow

1. Main server sends detection request to `/detect/{project_id}`
2. Inference server queues request and returns immediate response
3. Background worker processes request using appropriate ML models
4. Results sent back to main server via POST to `/provide-inference/...`

### Response Format

All detection results follow consistent structure:

```
fields = [
    "class_id",      # Symbol classification
    "x_center",      # Normalized x coordinate
    "y_center",      # Normalized y coordinate
    "width",         # Normalized width
    "height",        # Normalized height
    "confidence",    # Detection confidence (0-1)
    "similarity",    # Classification similarity (0-1)
    "page_id",       # Source page identifier
]
```

### LEGEND Detection

**Use Case**: Detect Legend Symbols on a specified region

**Process**:

1. Run pretrained RF-DETR general symbol detector on specified legend regions
2. Return raw detections to main server
3. Main server `/provide-inference` receives results and generates legend symbols according to the detection results.

### PAGE Detection

**Use Case**: Find specified legend symbols on document page(s)

**Process**:

1. Run RF-DETR inference on page regions
2. Fetch legend symbols from main server
3. Compute embeddings for detected symbols and legend symbols
4. Apply Leiden clustering for classification
5. Calculate final cosine similarity scores

**Template Input** - User provides single example of symbol to detect

**Crop Extraction** - Run pretrained YOLO to find all potential symbols in document

**Feature Embedding** - Generate embeddings for template and all crops using MobileNet

**Similarity Filtering** - Remove obviously different crops (cosine similarity threshold)

**Embedding Enhancement** - Apply custom enhancement using PCA and manifold learning

**Graph Construction** - Build similarity graph between remaining embeddings

**Community Detection** - Use Leiden algorithm to group similar embeddings

**Binary Classification** - Label crops as match/no-match based on community membership

### TEMPLATE Detection

**Use Case**: Precise matching using template matching (cross-correlation)

**Process**:

1. Extract template from legend or user selection
2. 1st stage: Perform multi-angle template matching (0°, 90°, 180°, 270°)
3. 2nd stage: Uses top 4 results from 1st step and runs multi-angle template matching again to extract even more potential positives (maximizes recall - may introduce + increase false positive rate)
4. Apply NMS and Siamese scoring for final results

### DISTINCT Detection

**Use Case**: Automatically discover unique symbol classes

**Process**:

1. Run RF-DETR across multiple pages
2. Crop all detected symbols
3. Apply HDBSCAN clustering on visual features
4. Use OCR to classify text vs. graphical symbols

5. Return representative symbols (each is a legend symbol) for each cluster

### TEXT_SEARCH Detection

**Use Case**: Find specific text patterns using OCR **Process**:

1. Run OCR across specified pages + build OCR dataset (cached per project)
2. Apply regex pattern matching
3. Filter results by inference region bounds
4. Return matching text locations with metadata

## Other Key Endpoints to Note

### Primary Detection Endpoint

```
@app.post("/detect/{project_id}")
async def detect(
    project_id: str,
    images: List[UploadFile] = File(...),
    data: str = Form(...),
    template: UploadFile | None = File(None),
):
```

**Purpose**: Unified endpoint for all detection types
**Input**: Images + JSON parameters specifying detection type and settings
**Output**: Queues request and returns immediate acknowledgment

### Symbol Comparison

```
@app.post("/compare-symbols/{project_id}")
async def compare_symbols(project_id: str, payload: dict):
```

**Purpose**: Find similar legend symbols for merge suggestions
**Process**: Uses embeddings + rotation-aware similarity computation
**Output**: List of similar symbol IDs ranked by similarity score

### Region Auto-Detection

```
@app.post("/detect-inference-region")
async def detect_inference_Region(image: UploadFile = File(...)):
```

**Purpose**: Automatically identify inference regions in documents
**Output**: Normalized YOLO bounding boxes for detected regions

## Inference Pipeline Flow Examples

### Pipeline Overview:

This is the general flow for all inference server related tasks once initiated by the client / frontend.

> 💡 Frontend → Main Server → Inference Server → ML Processing → Results → Main Server → State Integration → UI Update

### (1) Frontend Request Initiation

Users initiate detection through various functions in `symbolServices.ts`, each targeting different detection workflows:

**Box Selection Detection (Legend Symbols + Page Symbols)**

```
// Single page detection with bounding box
const detectSymbolsBox = async (
  type: CanvasType,           // "legend" | "page"
  coords: XyxyBox | null,     // Bounding box coordinates
  projectId: string,
  pageId: string,
  confThreshold: number,      // Detection confidence (0-1)
  iouThreshold: number,       // Overlap threshold for NMS
  selectedSymbols: string[],  // Filter to specific symbol classes
)

// Full project detection (multiple pages)
const detectSymbolsFullProject = async (
  pageIds: string[],          // Array of page IDs to process
  type: CanvasType,
  projectId: string,
  confThreshold: number,
  iouThreshold: number,
  selectedSymbols: string[],
)
```

**Draw Selection Detection (Legend Symbols + Page Symbols)**

```
// Freehand polygon selection detection
const detectSymbolsDraw = async (
  type: CanvasType,
  coords: [[number, number]] | null,  // Polygon points array
  projectId: string,
  pageId: string,
  confThreshold: number,
  iouThreshold: number,
  penToolWidth: number,               // Draw tool width
  scaleFactor: number,
  canvasZoom: number,
  selectedSymbols: string[],
)
```

**Template Matching Detection**

```
// Template matching on single page using inference template
const detectTemplatePage = async (
  type: CanvasType,
  projectId: string,
  pageId: string,
  confThreshold: number,
  iouThreshold: number,
  selectedSymbols: string[],
)

// Template matching across entire project
const detectTemplateFullProject = async (
  type: CanvasType,
  projectId: string,
  pageIds: string[],           // Pages to process
  confThreshold: number,
  iouThreshold: number,
  selectedSymbols: string[],
)

// Additional template detection with custom region
const detectAdditionalTemplate = async (
  type: CanvasType,
  coords: XyxyBox,            // Custom template region
  projectId: string,
  pageId: string,
  confThreshold: number,
  iouThreshold: number,
  selectedSymbols: string[],
)
```

**Distinct Classes Discovery**

```
// Automatically discover new symbol types across multiple pages
const detectDistinctClasses = async (
  pageIds: string[],          // Pages to analyze
  type: CanvasType,
  projectId: string,
  confThreshold: number,
  iouThreshold: number,
  selectedSymbols: string[],
)
```

**Text Search Detection**

```
// OCR-based text pattern search
const searchForTextSymbol = async (
  selectedPages: string[],    // Pages to search
  projectId: string,
```

```
  regexToMatch: string,       // Regular expression pattern
  confThreshold: number,
  iouThreshold: number,
)
```

### (2) Main Server Detection Request Handler

All detection types funnel through specialized handler functions that prepare requests for the inference server:

**Handler Function Routing**

```python
# Legend detection (RF-DETR to discover new symbols)
@app.post("/detect-legend-symbols-box/{project_id}")
async def detect_legend_symbols_box(project_id: str, data: RunBoxDetection):
    asyncio.create_task(handle_legend_detection(project_id, data, "box"))

@app.post("/detect-legend-symbols-draw/{project_id}")
async def detect_legend_symbols_draw(project_id: str, data: RunBoxDetection):
    asyncio.create_task(handle_legend_detection(project_id, data, "draw"))

# Page detection (find known symbols)
@app.post("/detect-page-symbols-box/{project_id}")
async def detect_page_symbols_box(project_id: str, data: RunBoxDetection):
    asyncio.create_task(handle_page_symbol_detection(project_id, data, "page"))

# Template detection (cross-correlation matching)
@app.post("/detect-page-template/{project_id}")
async def detect_template_page(project_id: str, data: RunBoxDetection):
    asyncio.create_task(process_detection_async(project_id, data, "template"))

# Distinct classes discovery (automatic clustering)
@app.post("/detect-page-distinct-classes/{project_id}")
async def detect_page_distinct_classes(project_id: str, data: AggregatedRunBoxDetection):
    asyncio.create_task(process_detection_async(project_id, data, "distinct"))

# Text search (OCR-based search)
@app.post("/search-for-text-symbol/{project_id}")
async def search_for_text_symbol(project_id: str, data: SearchTextDetection):
    asyncio.create_task(handle_search_for_text_symbol(project_id, data))
```

**Legend Detection Handler**

```python
async def handle_legend_detection(project_id, data, type_of_selection="box"):
    # Load project metadata and start tracking
    page = projects[project_id].pages[data.page_id]
    tracker.start_inference(project_id)

    # Build request with region information
    page_info = {
        "page_id": data.page_id,
```

```
            "conf_threshold": data.conf_threshold,
            "iou_threshold": data.iou_threshold,
            "selected_legend_symbols": data.selected_legend_symbols,
        }

        # Add region data (box vs polygon selection)
        if type_of_selection == "box":
            page_info["region"] = {"region_type": "box", "data": data.box}
        else:
            page_info["region"] = {"region_type": "draw", "data": data.polygon}
```

**Page Symbol Detection Handler**

```
async def handle_page_symbol_detection(project_id: str, data, detection_type: str):
    # Handle single-page or batch processing
    requests_list = data.requests if hasattr(data, "requests") else [data]

    # Build payload for each page
    requests_payload = []
    for req in requests_list:
        page_info = {
            "page_id": req.page_id,
            "conf_threshold": req.conf_threshold,
            "iou_threshold": req.iou_threshold,
            "selected_legend_symbols": req.selected_legend_symbols,
        }

        # Add region (full page if no box specified)
        if hasattr(req, "box") and req.box:
            page_info["region"] = {"region_type": "box", "data": req.box}
        else:
            page_info["region"] = {"region_type": "box", "data": {"x1": 0, "y1": 0, "x2": 0, "y2": 0}}

        requests_payload.append(page_info)
```

**Unified Request Transmission**

All handlers use the same HTTP interface to communicate with the inference server:

```
async def handle_search_for_text_symbol(project_id: str, data: SearchTextDetection):
    tracker.start_inference(project_id)

    # Process all pages (inference server filters by selected_pages)
    requests_payload = []
    for page_id in projects[project_id].pages.keys():
        page_info = {
            "page_id": page_id,
            "conf_threshold": data.conf_threshold,
            "iou_threshold": data.iou_threshold,
```

```
                "region": {"region_type": "box", "data": {"x1": 0, "y1": 0, "x2": 0, "y2": 0}}
            }
            requests_payload.append(page_info)

        # Include search parameters
        detection_data = {
            "detection_type": "text_search",
            "symbol_to_search_for": data.symbol_to_search_for,
            "selected_pages": data.selected_pages,
            "requests": requests_payload
        }
```

### (3) Inference Server Request Processing

Once the main server sends detection requests, the inference server receives them through the unified `/detect` endpoint and routes them to specialized processing functions:

#### Unified Detection Endpoint

```
@app.post("/detect/{project_id}")
async def detect(
    project_id: str,
    images: List[UploadFile] = File(...),
    data: str = Form(...),
    template: UploadFile | None = File(None),
):
    parsed_data = json.loads(data)
    detection_type = parsed_data.get("detection_type")
    requests_list = parsed_data.get("requests", [])

    # Route to appropriate handlers based on detection type
    if detection_type in ["text_search", "distinct"]:
        # Queue-based batch processing
        if detection_type == "text_search":
            return await handle_text_search_request(project_id, images, parsed_data)
        else:
            return await handle_batch_request(project_id, images, requests_list, detection_type)
    else:
        # Single detection processing (page, legend, template)
        return await handle_single_request(project_id, images[0], parsed_data, template)
```

#### Request Queuing System

All requests enter a round robin queue management system for optimal GPU utilization:

```
input_queue = Queue()

def round_robin(queue: Queue):
    """
    Manages per-project queues to ensure fair processing across multiple projects.
    Accumulates tasks briefly, then processes one task per project in round-robin fashion.
```

```
    """
    project_queues = defaultdict(deque)

    while run or any(len(q) != 0 for q in project_queues.values()):
        # Drain global queue into project-specific queues
        while True:
            try:
                task = queue.get(timeout=0.1)
                project_id, req = task
                project_queues[project_id].appendleft(req)
            except Empty:
                break

        # Round-robin: process one task per project
        for project_id, q in project_queues.items():
            if len(q) == 0:
                continue
            request = q.pop()
            inference_task(request)  # Execute the actual ML processing
```

**Detection Type Routing**

The `inference_task` function dispatches to specialized handlers:

```
def inference_task(request):
    """Routes requests to appropriate detection algorithms"""
    try:
        if isinstance(request, list):
            # Batch processing for multiple pages
            if request and hasattr(request[0], 'type') and request[0].type is DetectionType.TEXT_SEARCH:
                result = text_search_detection(request)
            else:
                result = distinct_detection(request)
        else:
            # Single request processing
            if request.type is DetectionType.PAGE:
                result = page_detection(request)
            elif request.type is DetectionType.LEGEND:
                result = legend_detection(request)
            elif request.type is DetectionType.TEMPLATE:
                result = template_detection(request)

        # Send results back to main server
        send_results_to_main_server(result, request)

    except Exception:
        logger.exception("Error during inference")
        send_error_response(request)
```

### (4) Inference Server Detection Algorithms

**Legend Symbol Detection**

```
def legend_detection(request: DetectRequest):
    """
    Discovers new symbols in user-selected regions for manual classification.
    Simple RF-DETR detection.
    """
    img = preprocess(request.image, request.region)
    predictions = run_model_inference(img, request.conf_threshold, request.iou_threshold)
    return raw_detections_with_placeholders(predictions, request.page_id)
```

**Leiden Symbol Detection**

```
def page_detection(request: DetectRequest):
    """
    Finds instances of known legend symbols using ML similarity matching.
    Pipeline: RF-DETR → Crop symbols → Embeddings → Leiden clustering → Cosine similarity
    """
    img = preprocess_with_template(request.image, request.region, request.project_id)
    yolo_predictions = run_model_inference(img, thresholds...)
    classified_symbols = predict_scores(yolo_predictions, legend_symbols, embeddings...)
    return classified_symbols_with_confidence(classified_symbols, request.page_id)
```

**Leiden ITC Algorithm: File:** `leiden_itc.py`

**What is "Leiden" Detection?**

- **Leiden** refers to the Leiden algorithm - a sophisticated community detection method for finding clusters in graphs. In our context, it's used to group similar symbol embeddings together, then classify unknown symbols based on which "community" they belong to.

**The Complete Pipeline Explained:**

**Step 1: Initial Detection**

```
img = preprocess_with_template(request.image, request.region, request.project_id)
yolo_predictions = run_model_inference(img, thresholds...)
```

- **What happens:** RF-DETR finds all potential symbols in the page image
- **Output:** Raw bounding boxes with confidence scores
- **Purpose:** Locates and finds all general symbol candidates

**Step 2: The Leiden ITC Core Process**

```
classified_symbols = predict_scores(yolo_predictions, legend_symbols, embeddings...)
```

This calls the `one_shot_leiden_itc()` function, which does the heavy lifting:

**A) Embedding Generation**

```
# For each legend symbol, create 20+ augmented training examples
train_embeddings = create_augmentations(legend_symbol)  # Rotations, flips, synthetic variations
test_embeddings = extract_embeddings(yolo_detected_crops)  # From RF-DETR results
```

- **Training set:** 60+ augmented versions of the single legend symbol
- **Test set:** All the symbols RF-DETR found on the page
- **Purpose:** Create a robust representation of what the legend symbol "looks like"

**B) Similarity Filtering (First Gate)**

```
# Exclude symbols that are clearly not similar to the legend
cos_sims = test_embeddings.dot(legend_embedding)
include_mask = cos_sims >= 0.445  # Only keep reasonably similar symbols
```

- **What this does:** Immediately excludes symbols that are obviously different
- **Threshold 0.445:** Empirically determined cutoff for "possibly the same symbol"
- **Purpose:** Reduce noise before expensive graph operations

**C) Graph Construction**

```
def build_semantic_graph(embeddings):
    # 1. k-reciprocal re-ranking for distance refinement
    refined_dist = k_reciprocal_re_ranking(embeddings, k1=20, k2=6)
    # 2. Build similarity graph with top-k connections
    # 3. Prune weak edges (bottom 40% of similarities)
    # 4. Weight edges by node degree (popular nodes get lower weights)
```

**What k-reciprocal re-ranking does:**

- **Problem:** Raw cosine distance can be noisy
- **Solution:** If symbol A is similar to B, and B is similar to A, strengthen that connection
- **k1=20:** Look at top 20 neighbors for each symbol
- **k2=6:** Refine using top 6 reciprocal neighbors
- **Result:** More reliable similarity measurements

**Graph structure:**

- **Nodes:** All embeddings (training + test)
- **Edges:** Weighted by refined similarity scores
- **Pruning:** Remove weakest 40% of connections to focus on strong relationships

**D) Leiden Community Detection**

```
partition = leidenalg.find_partition(graph, leidenalg.ModularityVertexPartition)
```

- **What happens:** Algorithm finds "communities" - groups of highly connected nodes
- **Typical result:** Training examples form one community, similar test symbols join them
- **Key insight:** Symbols that cluster together are likely the same type

**E) Majority Voting Classification**

```
for community in communities:
    training_labels_in_community = get_training_labels(community)
    majority_label = most_common(training_labels_in_community)
    assign_label_to_all_test_symbols_in_community(majority_label)
```

- **Logic:** If most training examples in a community are "positive" (legend symbol), then test symbols in that community are also "positive"
- **Robustness:** Uses multiple training examples to make confident decisions

**F) Multi-Pass Refinement**

```
for pass_num in range(max_pass):
    # 1. Run community detection
    # 2. Identify "outlier" symbols (low confidence)
    # 3. Shift outliers toward their predicted class centroid
    # 4. Re-run community detection with adjusted positions
```

- **Purpose:** Iteratively improve classifications
- **Outlier shifting:** Move uncertain symbols closer to where they "should" be
- **Convergence:** Usually stabilizes after 2-3 passes

**Why This Approach Works:**

**Traditional approach problems:**

- Single legend symbol vs. many detected symbols (1-vs-N comparison)
- No context about symbol variations
- Sensitive to small differences

**Leiden ITC advantages:**

- **Augmentation:** Creates multiple training examples from single legend symbol
- **Graph structure:** Captures relationships between ALL symbols simultaneously
- **Community detection:** Finds natural groupings rather than forced classifications
- **Iterative refinement:** Improves results through multiple passes

**Real-World Example:**

**Input:**

- Legend symbol: A clean pipe symbol "|"
- Page detections: 50 symbol candidates (pipes, text, other symbols)

**Process:**

a. **Augmentation:** Generate 60 variations of the pipe symbol (rotated, flipped, etc.)

b. **Filtering:** Keep 30 candidates with similarity ≥ 0.445 to original pipe

c. **Graph:** Build connections between all 90 symbols (60 training + 30 test)

d. **Communities:** Algorithm finds 3 groups:

   - Community 1: Most training pipes + 15 test pipes → Label as "pipe"
   - Community 2: Some training pipes + 8 test symbols → Label as "pipe"
   - Community 3: Few training pipes + 7 test symbols → Label as "not pipe"

e. **Result:** 23 symbols classified as pipes, 7 as non-pipes

**Output:** List of classified symbols with confidence scores for integration back into the main system.

This approach achieves much higher accuracy than simple similarity thresholding, especially when dealing with symbol variations, rotations, and partial occlusions common in real engineering drawings.

#### Template Matching Detection

```
def template_detection(request: DetectRequest):
    """
    Precise matching using cross-correlation with multi-angle support.
    Pipeline: Template matching (0°/90°/180°/270°) → NMS → Siamese similarity
    """
    img = preprocess_with_template(request.image, request.region)
    legend_symbols = fetch_symbols_or_use_provided_template(request)

    matches = []
    for symbol_id, symbol_img in legend_symbols.items():
        # Two-stage correlation matching with rotation support
        stage1_matches = parallel_template_matching_at_angles(img, symbol_img, [0,90,180,270])
        stage2_matches = refine_top_matches(stage1_matches, img, symbol_img)
        nms_filtered = apply_non_maximum_suppression(stage1_matches + stage2_matches)
        siamese_verified = verify_matches_with_neural_network(nms_filtered)
        matches.extend(siamese_verified)

    return template_matches_with_rotation_info(matches, request.page_id)
```

#### Distinct Legend Symbol (Classes) Detection

```
def distinct_detection(requests: List[DetectRequest]):
    """
    Automatic symbol discovery across multiple pages using unsupervised learning.
    Pipeline: Multi-page RF-DETR → Crop aggregation → OCR classification → HDBSCAN clustering
    """
    # Phase 1: Aggregate detections from all pages
    all_crops = []
    for req in requests:
        img = preprocess_with_template(req.image, req.region)
        page_detections = run_model_inference(img, thresholds...)
        page_crops = extract_and_binarize_crops(img, page_detections, req.page_threshold)
```

```
        all_crops.extend(page_crops_with_metadata(page_crops, req.page_id))

    # Phase 2: Clustering pipeline
    cluster_representatives, labels, indices = ocr_and_hdbscan_clustering(all_crops)

    # Phase 3: Generate new legend symbols
    new_symbols = []
    for rep_index, cluster_label in zip(indices, labels):
        if cluster_label == TEXT_CLUSTER:
            class_id = f"TEXT_{run_multistage_ocr(all_crops[rep_index])}"
        else:
            class_id = f"SYMBOL_{rep_index}"
        new_symbols.append(create_legend_symbol(class_id, all_crops[rep_index]))

    return new_legend_symbols_with_locations(new_symbols)
```

**Distinct Classes Algorithm: File: `group_rep.py`**

**What is "Distinct" Detection?**

- Automatically discover **all unique symbol types** across multiple pages without any prior knowledge. Think "show me every different symbol that exists in this document."
- **Challenge:** You don't know what symbols exist, how many types there are, or what they look like. The algorithm must figure this out from scratch.

**The Complete Pipeline Explained:**

**Phase 1: OCR-Based Partitioning - The Text/Symbol Separator**

**Multi-Grid OCR Processing:**

- PaddleOCR works better on full sized documents instead of multiple tiny crops. Our solution is to arrange symbol crops in a 2x2 grid to synthetically generate a “mini document” for the OCR engine, this format helps OCR recognize characters more accurately.

```
def paddleocr_ocr(pil_img, grid=(2,2), padding=23, thumb_size=(192,192)):
    # Create thumbnail grid for better OCR accuracy
    thumbs = [img.resize(thumb_size)]
    if height > 3*width:  # Tall symbols might be rotated text
        thumbs.append(img.rotate(-90).resize(thumb_size))

    montage = create_thumbnail_grid(thumbs, grid, padding)
    ocr_results = paddle_ocr_with_boxes(montage)


def partition_images_by_ocr(images):
    # Calculate median area of all symbols
    median_area = np.median([w*h for w,h in image_sizes])

    for img, ocr_text, area in ocr_results:
        # Filter out obvious false positives
        if not ocr_text or (ocr_text == "1" and area <= 300):
```

```
            unrecognized.append(img)  # Send to HDBSCAN
        else:
            recognized[ocr_text].append(img)  # Group by text

    # Apply size-based filtering
    for text_group, images in recognized.items():
        if len(images) == 1 and text_group != "1":
            # Singleton text groups need size validation
            if images[0].area <= median_area - 10:
                unrecognized.extend(images)  # Too small, probably noise
```

**Smart filtering logic:**

- **"1" character:** Often OCR misreads symbols as "1" - filter by size
- **Singleton groups:** Single occurrence text might be OCR error
- **Size validation:** Tiny symbols with text are probably noise
- **Median area threshold:** Uses document context to set size expectations

### Phase 2: HDBSCAN Clustering - The Symbol Grouper

Dual-Pass Clustering Strategy:

```
def run_hdbscan_pipeline(unrecognized_images):
    # Primary clustering - conservative settings
    clusterer = hdbscan.HDBSCAN(
        min_cluster_size=2,              # At least 2 symbols per type
        cluster_selection_epsilon=0.50,   # Moderate density requirement
        cluster_selection_epsilon_max=0.90
    )
    labels = clusterer.fit_predict(features)

    # Secondary clustering on "noise" points
    unclustered_indices = np.where(labels == -1)[0]
    if len(unclustered_indices) > 0:
        # More aggressive clustering on leftovers
        secondary_clusterer = hdbscan.HDBSCAN(
            min_cluster_size=2,
            cluster_selection_epsilon_max=0.80  # Lower threshold
        )
        new_labels = secondary_clusterer.fit_predict(features[unclustered_indices])
        # Merge results with offset to avoid label conflicts
```

Real World Example:

```
Input: 42 graphical symbols (no readable text)
Primary HDBSCAN:
├── Cluster 0: 8 pipe junction symbols
├── Cluster 1: 6 valve symbols
├── Cluster 2: 4 pump symbols
```

```
├── Cluster 3: 5 pressure gauge symbols
└── Noise (-1): 19 symbols (various one-offs)

Secondary HDBSCAN on the 19 "noise" symbols:
├── Cluster 4: 3 temperature sensor symbols (found!)
├── Cluster 5: 2 flow meter symbols (found!)
└── Noise (-1): 14 truly unique symbols
```

#### Feature Extraction with MobileNetV4

```python
# Extract high-quality embeddings for clustering
model = timm.create_model("mobilenetv4_conv_medium", num_classes=0)
model.load_state_dict(checkpoint["student_model_state_dict"])

for img in unrecognized_images:
    features = model(transform(img))
    normalized_features = features / (norm(features) + 1e-7)
```

- **Why MobileNetV4:** Good balance of accuracy and speed for symbol recognition
- **L2 normalization:** Ensures distance measurements focus on shape, not magnitude
- **Student model:** Distilled from larger model for efficiency

### Phase 3: Representative Selection - The Quality Filter

#### Majority Voting for Large Groups

```python
def prune_final_output(groups_dict):
    for group_name, symbol_list in groups_dict.items():
        if len(symbol_list) >= 3:
            # Build similarity graph within the group
            similarities = compute_pairwise_cosine_similarity(symbol_list)
            adjacency = similarities > 0.8  # High similarity threshold

            # Find largest connected component (majority consensus)
            components = find_connected_components(adjacency)
            majority_component = max(components, key=len)

            # Select medoid (most representative symbol)
            medoid_idx = find_medoid(majority_component, similarities)
            representatives[group_name] = symbol_list[medoid_idx]
```

#### Why this approach?

- **Problem:** HDBSCAN clusters might contain some misclassified symbols
- **Solution:** Within each cluster, find the largest group of mutually similar symbols
- **Medoid selection:** Choose the symbol most similar to all others (not just the average)

### Perceptual Hash Deduplication

```
def deduplicate_with_phash(representative_images):
    hashes = [imagehash.phash(img) for img in representative_images]

    for i in range(len(hashes)):
        for j in range(i+1, len(hashes)):
            hamming_distance = hashes[i] - hashes[j]
            if hamming_distance < 3:  # Very similar images
                remove_indices.add(j)  # Keep first occurrence
```

- **pHash:** Perceptual hash - robust to small rotations, scaling, compression
- **Hamming distance < 3:** Very strict threshold - only removes near-identical symbols
- **Purpose:** Prevent duplicate legend symbols from different pages/locations

**Complete Example Workflow:**

**Input:** 200 symbol crops from a 5-page engineering PDF

**Phase 1 Results:**

- OCR recognizes: 45 text symbols → 8 groups ("V", "P", "T", "M", "F", "H", "C", "1")
- OCR fails: 155 graphical symbols → send to HDBSCAN

**Phase 2 Results:**

- Primary HDBSCAN: 12 clusters (2-15 symbols each) + 67 noise
- Secondary HDBSCAN: 4 more clusters (2-4 symbols each) + 51 noise
- Total: 16 symbol clusters + 51 unique symbols

**Phase 3 Results:**

- Majority voting: 16 clusters → 16 representative symbols
- OCR groups: 8 text groups → 8 representative symbols
- Deduplication: 24 symbols → 22 symbols (removed 2 near-duplicates)

**Final Output:** 22 unique legend symbols automatically discovered and ready for user review

**OCR Text Search Detection**

```
def text_search_detection(requests: List[DetectRequest]):
    """
    OCR-based text pattern search with intelligent caching.
    Pipeline: Cache check → RF-DETR detection → OCR processing → Regex filtering
    """
    search_pattern = compile_regex(requests[0].symbol_to_search_for)
    project_id = requests[0].project_id

    # Check OCR cache first
    if ocr_cache.has_results(project_id):
        cached_ocr_results = ocr_cache.get(project_id)
        return filter_by_pattern_and_region(cached_ocr_results, search_pattern)
```

```
    # Build OCR cache
    ocr_results_by_page = {}
    for req in requests:
        img = preprocess(req.image, req.region)
        detections = run_model_inference(img, thresholds...)
        raw_ocr = run_multistage_ocr_pipeline(img, detections)
        corrected_ocr = fix_common_ocr_errors(raw_ocr)  # '0'→'O', '1'→'I', etc.
        ocr_results_by_page[req.page_id] = corrected_ocr

    ocr_cache.store(project_id, ocr_results_by_page)
    return filter_by_pattern_and_region(ocr_results_by_page, search_pattern)
```

**What is "Text Search" Detection?**

- Find specific text patterns (like "L14" or "R4B") across document pages using advanced OCR preprocessing to maximize text recognition accuracy.
- **Challenge:** Engineering drawings have poor text quality - small fonts, rotated text, mixed with graphics, low contrast.

**The Complete Pipeline Explained:**

**The Multi-Stage Fallback Strategy**

Stage-by-Stage Processing

```
def try_ocr_multistage(crop):
    # Stage 1: Raw crop (commented out - often fails)
    # result = run_paddle_ocr(crop)

    # Stage 2: Padded crop
    padded = pad_image(crop, padding_ratio=0.1)
    result = run_paddle_ocr(padded)
    if result and result[0].strip():
        return result[0]  # Success!

    # Stage 3: Scaled crop
    scaled = scale_image(crop, scale_factor=2.0)
    result = run_paddle_ocr(scaled)
    if result and result[0].strip():
        return result[0]  # Success!

    # Stage 4: Rotation voting (last resort)
    return try_rotated_ocr(scaled)
```

**Why Each Stage Works:**

**Stage 2: Padding (10% border)**

```
def pad_image(img, padding_ratio=0.1):
    w, h = img.size
    pad_x, pad_y = int(w * 0.1), int(h * 0.1)
```

```
    return ImageOps.expand(img, border=(pad_x, pad_y), fill='white')
```

- **Problem:** Text touching crop boundaries confuses OCR
- **Solution:** Add white border so text appears "floating" in space

**Stage 3: 2x Scaling with LANCZOS**

```
def scale_image(img, scale_factor=2.0):
    new_size = (int(w * 2), int(h * 2))
    return img.resize(new_size, Image.Resampling.LANCZOS)
```

- **Problem:** Text too small for OCR engine's optimal resolution
- **Solution:** High-quality up sampling makes tiny text readable
- **LANCZOS:** Best resampling for preserving text sharpness

**Stage 4: Rotation Voting**

```
def try_rotated_ocr(crop, angles=[-60, -45, -30, -15, 0, 15, 30, 45, 60]):
    best_label, best_confidence = "", 0.0

    for angle in angles:
        rotated = crop.rotate(angle, expand=True, fillcolor='white')
        result = run_paddle_ocr(rotated)
        if result and result[-1] > best_confidence:
            best_label = result[0]
            best_confidence = result[-1]

    return best_label
```

- **Problem:** Text is rotated/skewed in engineering drawings
- **Solution:** Try multiple angles, pick result with highest confidence
- **Angle range:** ±60° covers most real-world text orientations

**Radial Blur Enhancement**

**The OCR Problem:**

- Engineering drawings mix text with graphics
- OCR gets confused by nearby lines, symbols, arrows
- Text recognition drops when visual "noise" is present

**The Radial Solution:**

- **Assumption:** Important text is usually centered in detection boxes
- **Strategy:** Keep text sharp, blur everything else into background
- **Effect:** OCR focuses on the text, ignores graphical distractions

```
def apply_radial_blur(crop, blur_strength=13, center_ratio=0.80):
```

```
        # Calculate distance from center
        Y, X = np.ogrid[:H, :W]
        cx, cy = W // 2, H // 2
        dist = np.sqrt((X - cx)**2 + (Y - cy)**2)
        radius = min(H, W) * center_ratio / 2

        # Binary mask: 1 in center, 0 at edges
        center_mask = (dist <= radius).astype(np.float32)

        # Strong blur for edges
        blurred = cv2.GaussianBlur(crop, (27, 27), 0)  # blur_strength=13 → kernel=27

        # Blend: sharp center + blurred edges
        result = crop * center_mask + blurred * (1 - center_mask)
```

**Integration with RF-DETR Detections**

```
    def ocr_from_yolo(image, yolo_preds, ocr_engine):
        crops = extract_symbol_crops(image, yolo_preds)
        final_preds = []

        for i, crop in enumerate(crops):
            # Apply multi-stage OCR enhancement
            ocr_label = try_ocr_multistage(crop, ocr_engine)

            # Preserve YOLO spatial information
            pred = yolo_preds[i][:]  # Copy original prediction
            if ocr_label:
                pred[0] = ocr_label  # Replace label with OCR result
            final_preds.append(pred)

        return final_preds
```

**Smart Integration:**

- **Preserve coordinates:** Keep RF-DETR's accurate bounding boxes
- **Enhance labels:** Replace generic labels with specific OCR text
- **Fallback gracefully:** If OCR fails, keep original RF-DETR prediction
- **Maintain format:** Output still compatible with rest of pipeline

Real-World Performance Example:

**Input:** Page with 50 text elements detected by RF-DETR

**Stage-by-Stage Results:**

- Raw OCR: 12 successful reads (24%)
  - Padding: 8 more successful (40% total)
  - Scaling: 12 more successful (70% total)
  - Rotation: 10 more successful (94% total)

- Final: 42/50 text elements successfully read

**Typical improvements:**

- "VALVE" (tiny text) → readable after 2x scaling
- "P-101" (rotated 30°) → readable after rotation voting
- "TEMP" (near pipe line) → readable after radial blur
- "Flow→" (partial crop) → readable after padding

**Integration with Regex Search:**

```
# User searches for pattern: "VALVE-\d+"
regex_pattern = re.compile(r"VALVE-\d+")

# OCR found: ["VALVE-001", "PUMP-23", "VALVE-105", "TEMP-A"]
matches = [text for text in ocr_results if regex_pattern.match(text)]
# Result: ["VALVE-001", "VALVE-105"]
```

This multi-stage approach achieves 80%+ text recognition accuracy on challenging engineering drawings, compared to ~40% with basic OCR alone.

#### Shared Infrastructure

Each algorithm leverages this shared infrastructure while implementing specialized logic for its particular detection methodology, ensuring consistent preprocessing and efficient resource utilization across all detection types.

```
# Common preprocessing pipeline
def preprocess(image, region, template=None):
    return apply_region_mask(image, region) or apply_template_bounds(image, template)

# Unified YOLO inference with caching
def run_model_inference(img, conf_threshold, iou_threshold, temp_dir):
    cache_key = compute_cache_key(img, conf_threshold, iou_threshold)
    if cached_result := detection_cache.get(cache_key):
        return cached_result

    sliced_predictions = slice_and_predict(img, CROP_SIZE, ROBOFLOW_MODEL)
    nms_filtered = apply_non_maximum_suppression(sliced_predictions, iou_threshold)
    detection_cache.store(cache_key, nms_filtered)
    return nms_filtered
```

### (5) Inference Server Result Transmission

Once ML processing completes, the inference server sends results back to the main server using standardized HTTP endpoints:

```
# Result formatting and transmission
def send_results_to_main_server(detections, request):
    formatted_results = format_as_detection_objects(detections)
```

```
        response = requests.post(request.response_url, json=formatted_results)
        logger.info(f"Sent {len(formatted_results)} results to main server")

    def send_results_to_main_server(detections, request):
        """Send processed results back to main server for integration"""
        endpoint = f"{MAIN_SERVER_URL}/provide-inference/..."

        formatted_results = format_detection_results(detections, request)
        response = requests.post(endpoint, json=formatted_results)
        logger.info(f"Sent {len(formatted_results)} results to main server")
```

### (6) Main Server Result Processing

The main server receives inference results through the `/provide-inference` endpoint and processes them differently based on detection type:

**Result Reception Endpoint:**

```
@app.post("/provide-
inference/{project_id}/{page_id}/{detection_type}/{full_page_detection}/{iou_threshold}/{conf_threshold}")
async def receive_inference(
    project_id: str,
    page_id: str,
    detection_type: DetectionType,
    full_page_detection: bool,
    detection_result: List[Detection] | DetectError
):
```

**LEGEND Detection Results:**

- Creates new LegendSymbol candidates from raw RF-DETR detections
- Generates unique identifiers and visual properties for each symbol
- Saves cropped symbol images for manual classification
- Does not create DetectedSymbol instances (legend symbols are templates, not detections)

**PAGE and TEMPLATE Detection Results:**

- Groups new detections by class_id for batch processing
- Merges with existing symbols using Non-Maximum Suppression (NMS threshold: 0.2)
- Updates legend symbol counts across all pages
- Maintains detection confidence and similarity scores

**DISTINCT Detection Results:**

- Creates new LegendSymbol entries for each discovered cluster representative
- Generates symbol crops and saves to legend_symbols directory
- Assigns unique colors and group IDs for visual organization
- Sets `popup=True` to trigger user review interface
- Distinguishes text symbols (class_id starts with "TEXT_") from graphical symbols

**TEXT_SEARCH Detection Results:**

- Groups detections by OCR label (class_id)
- Creates one LegendSymbol per unique text pattern found
- Generates DetectedSymbol instances for each match location
- Updates symbol counts and creates visual crops from first detection
- Sets `pending_text_search_popup=True` for user review interface

**State Management and Finalization**

```
save_project(project_id)
add_current_state_to_history(project_id)
tracker.end_inference(project_id)

if not tracker.is_inference_running(project_id):
    project_data.status = ProjectStatus.COUNTING
    save_project(project_id)

mark_to_be_updated(project_id)
```

**Thread Safety:** All result processing occurs within `data_lock` context to prevent race conditions during concurrent operations.

**User Interface Updates:** The `mark_to_be_updated(project_id)` call triggers frontend refresh to display new results immediately.

## Algorithm Characteristics

| Algorithm | Input | Core Process | Output | Main Server Pr |
|---|---|---|---|---|
| **Legend** | Region selection | RF-DETR detection | Raw legend symbol candidates | Creates new Le crops, no Dete |
| **Page (Leiden)** | Legend symbols | RF-DETR + ML similarity | Classified symbol instances | Groups by clas updates symbo |
| **Template** | Legend symbols / box template | Cross-correlation + NMS | Classified symbol instances | Groups by clas updates symbo |
| **Distinct** | Full page(s) | RF-DETR + OCR + clustering | Raw legend symbol candidates | Creates Legen cluster |
| **Text Search** | Regex Search pattern | RF-DETR + OCR + regex | Text pattern matches | Groups by OCF corresponding |

# Server - Authentication

## Overview

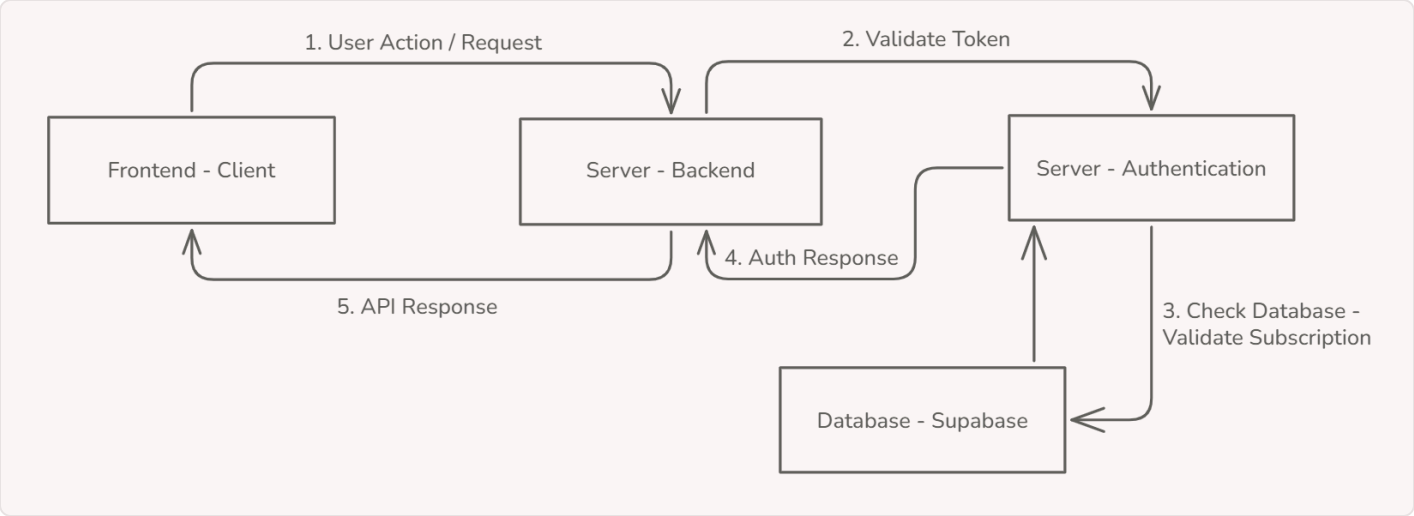
The Authentication Server is a FastAPI application that serves as the central hub for user authentication and subscription management. It handles:

- **JWT Token Validation**: Verifying user sessions
- **Stripe Payment Processing**: Managing subscriptions and billing
- **Subscription State Management**: Tracking user access permissions
- **Webhook Processing**: Handling real-time updates from Stripe

**Key Technologies:**

- FastAPI (Python web framework)
- Supabase (Database and authentication)
- Stripe (Payment processing)
- JWT (JSON Web Tokens)

**General Architecture** (What happens when a user makes a request):




## Stripe

Stripe Docs | Stripe Dashboard

Stripe serves as our payment processing and subscription management platform, chosen for it's robust and easy to use API and comprehensive webhook system. Stripe handles all of the payment processing, subscription lifecycle management, billing portal and customer management. We leverage Stripe's subscription model with automatic recurring billing, trial periods, and webhook events to maintain real-time synchronization between payment state and user access permissions. The integration includes checkout sessions for seamless payment flows and a customer portal for self-service subscription management.

Login: To Login to the Stripe dashboard, Owen must add you to the Team (Setting > Team and Security > Team) as an Administrator/Developer, you should then be able to access the Dashboard and make any necessary modifications.

## Current Implementation:

- **Subscription Plans**: Monthly and Annual pricing tiers with automatic renewal (modifiable in Product Catalog).

- **Trial System**: 14-day free trials for new users - one trail per user per lifetime. We enforce and track this via our database (Supabase).
- **Webhook Integration**: Real-time updates for subscription changes, payments and cancellations.
- **Customer Portal**: Self-service billing management for users to update payment methods, view invoices, and cancel subscriptions.
- **Price IDs**: `price_1RBPVY2VzTOO0oEwgZNr2nya` (Monthly), `price_1RBPVY2VzTOO0oEwJzYct8QR` (Annual)

**Architecture:** Our authentication server acts as a middleware between the main application and Stripe, processing webhooks to maintain subscription state in Supabase and validating user access based on subscription status. This design separates payment concerns from core application logic while ensuring data consistency.

At the time of writing, we're on Stripe's standard pricing (2.9% + 30¢ per transaction) which scales with our revenue. Stripe's webhook system ensures reliable payment state synchronization, and their billing portal reduces customer support overhead by enabling self-service subscription management.

## Database Schema

### Subscriptions Table (Supabase)

| Field | Type | Description | Example |
|---|---|---|---|
| id | UUID | User ID (Primary Key) | 1beded1e-c7ef-4433-ad40-87e9435bbd1c |
| is_subscribed | Boolean | Current subscription status | TRUE |
| stripe_subscription_id | String | String subscription identifier | sub_1RTAON2VzTOO0oEwCYdzAkyh |
| plan | String | Plan name (e.g., “Monthly”) | Yearly |
| stripe_price_id | String | Stripe price identifier | price_1RBPVY2VzTOO0oEwJzYct8QR |
| current_period_start | Timestamp | Current billing period start | 2025-07-16 18:42:46+00 |
| current_period_end | Timestamp | Current billing period end | 2026-07-16 18:42:46+00 |
| cancel_at_period_end | Boolean | Scheduled cancellation flag | FALSE |
| trial_start | Timestamp | Trial period start date | 2025-07-02 18:42:46+00 |
| trial_end | Timestamp | Trial period end date | 2025-07-16 18:42:46+00 |
| plan_status | String | Stripe status (active, trialing, etc.) | active |

## API Endpoints

### Authentication Endpoints:

`POST /validate_token`

> **Purpose**: Validate JWT token and check subscription status. This is the primary gatekeeper for premium features - it ensures users are both authenticated AND have an active subscription or trial. Used by the backend to protect subscription-required functionality like all of our inference tools/pipelines, and essentially all interactions with the Canvas.

`POST /validate_token_no_subscription`

> **Purpose**: Validate JWT token without subscription check for basic application functionality. Allows authenticated users to access core features (viewing previously created documents if they were on a subscription previously and account management) regardless of current subscription status.

### Stripe Payment Endpoints

`POST /checkout-session`

**Purpose**: Creates a Stripe checkout session for subscription purchase. Handles trial eligibility logic - new users get 14-day trials, returning users get immediate paid subscriptions. Creates or retrieves Stripe customers and returns a secure checkout URL that redirects to our Stripe hosted payment page.

`POST /manage-subscription-billing-portal`

**Purpose**: Generates a Stripe billing portal session for self-service subscription management. Allows users to update payment methods, view billing history, download invoices, and cancel subscriptions.

### Webhook Endpoint

`POST /stripe/webhook`

**Purpose**: Maintains real-time synchronization between Stripe payments, Supabase and application access control. Processes payment events as they happen to ensure users gain/lose access immediately.

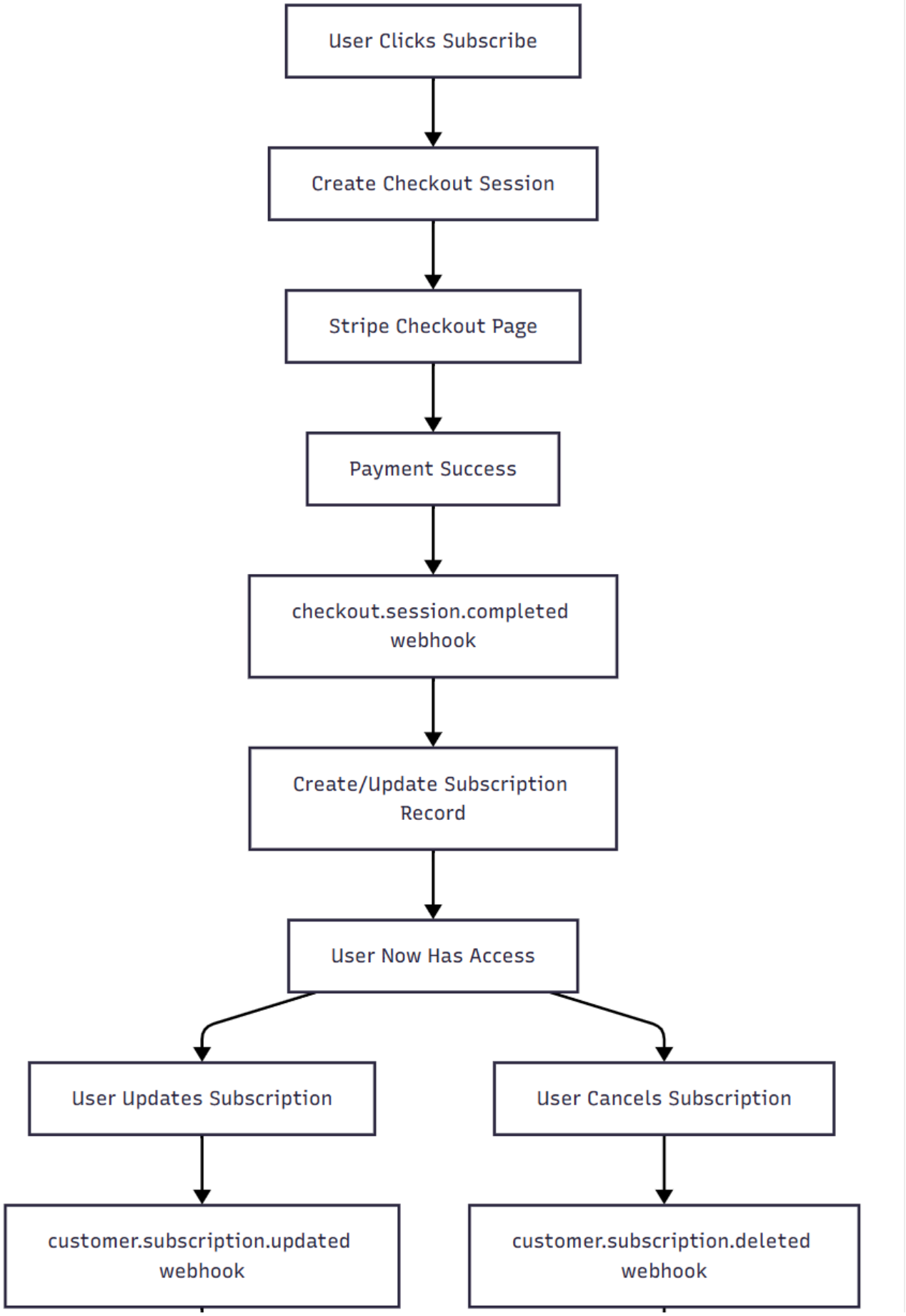
**Events Handled**:

- `checkout.session.completed:` Creates subscription record after successful payment, granting immediate access
- `customer.subscription.updated:` Syncs subscription changes like renewals, plan changes, and cancellations
- `customer.subscription.deleted:` Revokes access for cancelled or expired subscriptions while preserving historical data

## Stripe Integration

### Subscription Lifecycle

The following outlines the basic process of what happens when a user subscribes for the first time.


User Clicks Subscribe
Create Checkout Session
Stripe Checkout Page
Payment Success
checkout.session.completed webhook
Create/Update Subscription Record
User Now Has Access
User Updates Subscription
User Cancels Subscription
customer.subscription.updated webhook
customer.subscription.deleted webhook


## Trial Logic

**Key Points:**

- Each user gets ONE trial period per lifetime
- Trial status is tracked via `trial_start` field in Supabase subscriptions table
- Trial periods are 14 days
- Missing payment method during trial cancels subscription

```
# Trial eligibility check in checkout-session endpoint
sub_res = supabase_client.table("subscriptions") \
    .select("trial_start") \
    .eq("id", parsed_user_id) \
    .maybe_single() \
    .execute()

sub_data = sub_res.data if sub_res and hasattr(sub_res, "data") else None

# If user never had a trial, grant 14-day trial
if not sub_data or not sub_data.get("trial_start"):
    subscription_data_args = {
        "trial_period_days": 14,
        "trial_settings": {"end_behavior": {"missing_payment_method": "cancel"}},
    }
else:
    # User already had trial, no trial for new subscription
    subscription_data_args = {}
```

## Webhook Event Processing

### checkout.session.completed

**Triggered**: When user completes payment checkout

**Purpose**: Processes successful payments to immediately grant user access. This is the critical function that transforms a Stripe payment into application permissions.

```
async def handle_checkout_session_completed(event: dict):
    # 1. Extract user_id from session metadata
    session = event["data"]["object"]
    user_id = parse_raw_user_id(session["metadata"]["user_id"])

    # 2. Get full subscription data from Stripe
    stripe_subscription_id = session.get("subscription")
    subscription_data = stripe.Subscription.retrieve(stripe_subscription_id)

    # 3. Build update data with all subscription details
    update_data = build_subscription_update_data(
        user_id, subscription_data, stripe_subscription_id
    )
```

```
    # 4. Preserve existing trial fields (don't overwrite)
    update_data = preserve_existing_trial_fields(user_id, update_data)

    # 5. Upsert to database
    supabase_client.table("subscriptions").upsert(update_data).execute()
```

### customer.subscription.updated

**Triggered**: When subscription changes (renewal, plan change, cancellation scheduled)

**Purpose**: Keeps our database synchronized with Stripe as subscriptions evolve over time. Handles both automatic changes (renewals) and user-initiated changes (plan upgrades, cancellations).

```
async def handle_subscription_updated(event: dict):
    # 1. Find user by Stripe subscription ID
    subscription_id = event["data"]["object"]["id"]
    user_lookup = supabase_client.table("subscriptions")\
        .select("id")\
        .eq("stripe_subscription_id", subscription_id)\
        .execute()

    # 2. Update subscription status and billing dates
    # 3. Preserve trial information
    # 4. Handle cancellation scheduling
```

### customer.subscription.deleted

**Triggered**: When subscription is immediately cancelled or expires

**Purpose**: Revokes user access when subscriptions end while maintaining historical data for analytics and potential reactivation in the future.

```
async def handle_subscription_deleted(event: dict):
    # Force unsubscribe but preserve historical data
    update_data = build_subscription_update_data(
        user_id, subscription_data, subscription_id,
        force_unsubscribe=True  # <- Key parameter
    )
```

## Authentication Flow

### JWT Token Validation Process

**Purpose**: Verifies JWT token authenticity and extracts user information. Uses Supabase's JWT secret to ensure tokens weren't tampered with and haven't expired.

```
def decode_token_payload(token: str) -> dict:
    try:
        decoded = jwt.decode(
```

```
            token,
            SUPABASE_JWT_SECRET,      # Secret key
            algorithms=[JWT_ALGORITHM], # HS256
            audience=JWT_AUDIENCE,      # "authenticated"
        )
        return decoded
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token expired")
    except jwt.JWTError as e:
        raise HTTPException(status_code=401, detail=f"Invalid token: {str(e)}")
```

### Subscription Status Check

**Purpose**: Determines if an authenticated user has permission to access premium features. Checks both active subscriptions / trial periods to grant appropriate access levels.

```
def check_subscription_status(user_id: str) -> None:
    # Query subscription table
    res = supabase_client.table("subscriptions")\
        .select("*")\
        .eq("id", user_id)\
        .maybe_single()\
        .execute()

    subscription = res.data

    # Check if user has valid access
    has_valid_subscription = (
        subscription is not None and (
            subscription.get("is_subscribed", False) or
            subscription.get("plan_status") == "trialing"
        )
    )

    if not has_valid_subscription:
        raise HTTPException(status_code=401, detail="Subscription required")
```

### Existing User Access Check

```
1. User makes API request → Backend
2. Backend extracts JWT token → Auth Server (/validate_token)
3. Auth Server validates token → JWT validation
4. Auth Server checks subscription → Supabase query
5. Auth Server returns user data → Backend
6. Backend processes request → Response
```

## Development Tips (Keep in mind when making changes / modifications)

- **Use Stripe test mode during development** - Always use test API keys (`sk_test_...` and `pk_test_...`) to avoid processing real payments. Test mode provides identical functionality to live mode but with fake payment processing, allowing you to test subscription flows without financial consequences.
- **Test webhook events** using Stripe CLI - This forwards live webhook events from Stripe directly to your local development server, allowing you to test webhook processing in real-time as you trigger events in the Stripe dashboard or through API calls.

```
stripe listen --forward-to localhost:8000/stripe/webhook
```

- **Always preserve trial fields when updating subscriptions** - The `preserve_existing_trial_fields()` function prevents accidentally overwriting trial history when processing subscription updates. This ensures users don't lose their trial eligibility tracking or get multiple trials.
- **Test edge cases** like users switching between plans, canceling during trial periods, failed payment renewals, and reactivating cancelled subscriptions to ensure your webhook handlers work correctly in all scenarios.
- **Log webhook events extensively** during development to debug subscription state synchronization:

```
logger.info(f"Processing {event['type']} for user {user_id}")
```

- **Verify webhook endpoints locally** via testing before deploying to ensure your authentication server can properly receive and process Stripe events.

# Desktop App Deployment

## Overview

- The purpose of having a desktop app is so that we don't have to scale our servers up as our client pool grows
- We expect that the majority of our clients will already have computers powerful enough to run inference, so it makes sense to just give them an app they can run locally
- The desktop app is built on Electron JS, which is very commonly used to convert JavaScript web apps into desktop apps
- To package all of our python backend code, we use PyInstaller to create a self contained .exe, this way our python code can run locally even if the client's computer does not have python installed
- To package everything up into a distributable desktop app, we use electron forge

## Code Structure

- Listed below are some of the main components that go into the electron app
- **electron/main.js**
  - This file serves as the main entry-point for the electron app
  - It launches all 3 servers on dynamically generated ports to avoid collision with background processes
  - Depending on which command is run, it will either start the app in development mode or production mode, **for testing purposes you should only be running development mode**, production mode is automatically run when you launch the app executable
    - **Development mode:**
      - Can be run with **"npm run electron-dev"**
      - Requires that the python executable is built already with the command **"make electron-dev"**
      - This mode is great for testing that the python exe is correctly built
      - **I find a common issue is the .exe environment will sometimes miss required packages/binaries**
        - To fix this you can usually adjust the .spec file to that these required packages/binaries are included
    - **Production mode:**
      - This mode is automatically chosen when you launch the fully built app
      - Requires that the full application is built with the command **"make electron"**
      - In this mode, all the assets, front-end code, and models are fully self contained
      - **If something is working in the development mode but not production mode:**
        - launch the executable through the command line to see error messages
        - check common deployment bugs
        - check that any additional resources the app needs are either downloaded upon launch, or included in the extraResource list (in forge.config.js)
- **everything.py**
  - This is the file that we build the python executable off of
  - It serves as a singular entry-point that downloads model files, installs large packages, and launches the backend and inference servers on separate ports
- **everything.spec**
  - This file is used to specify hidden packages and binaries that are needed for the python executable to run properly

    - Uses Python syntax
- **forge.config.js**
    - This file configures the settings for electron forge
    - We can specify what files to leave out of the final package
    - We also need to specify what files we need as an extra resource, files specified here will be copied over in full to a self contained extra resource folder
    - All other files will be compressed into an .asar archive
- **package.json**
    - This file specifies the metadata for the application, most importantly the app version
    - You can also specify custom commands here

## Release Process

- Pull in new changes from main, resolve any conflicts that come up
- **If there are new models/data being used:**
    - Upload them to the fileserver computer at 10.0.0.5 at the desired location
    - Adjust the bootstrapping code located in everything.py
    - Make sure the new paths are being set as an env variable in electron/main.js
- To test the electron app without building the full thing, you can run “make electron-dev”
- This will create a python executable at “dist/everything/everything.exe
- Then run “npm run electron-dev” to test the app in development mode
- **TO DISTRIBUTE**
    - To update the version of the app, change the “version” variable in the root package.json
    - Run “**make electron**”
    - The .exe file will be contained in OUTPUT_DIRECTORY/make/squirrel.windows/x64
        - OUTPUT_DIRECTORY is specified in forge.config.json
    - To distribute the setup file, zip up the x64 folder, and send it to the users
    - If this is their first time installing the app, the bootup process can take about 10 - 15 min to install torch, if they already had a previous version installed, this process is skipped
        - all persistent packages should be found in **C:/Users/YOUR_USERNAME/AppData/Roaming/metrized-symbol-counter/packages**

# Common Deployment Bugs

### Squirrel creates “dummy update.exe”

- This happens when the app is too large for squirrel to package up
- This usually happens when a new large python package is added, these large packages need to be bootstrapped to save space
- WinDirStat can be used to check dist/everything/_internal to see what packages are taking up the most space

### Inference models aren’t downloading properly

- You can check this by running “metrized-symbol-counter.exe” from the terminal, the error looks like this:
  - ```
    Backend Error: Traceback (most recent call last):
      File "everything.py", line 111, in <module>

    Backend Error:   File "everything.py", line 83, in download_inference_models
      File "everything.py", line 45, in download_with_auth
      File "urllib\request.py", line 216, in urlopen
      File "urllib\request.py", line 525, in open
      File "urllib\request.py", line 634, in http_response
      File "urllib\request.py", line 563, in error
      File "urllib\request.py", line 496, in _call_chain
      File "urllib\request.py", line 643, in http_error_default

    Backend Error: urllib.error.HTTPError: HTTP Error 404: Not Found
    [PYI-1784:ERROR] Failed to execute script 'everything' due to unhandled exception!

    Backend: Backend port: 49257 | Inference port: 49256
    Installing torch packages...
    Torch packages already installed
    nvidia packages already installed
    paddle packages already installed
    Siamese weights not found at C:\Users\josephd\AppData\Local\metrized_symbol_counter\app-1.0.2\resources\siamese_symbol-small.pt
    , downloading from fileserver
    ```
- This will usually be because an inference model has been renamed or swapped out
- You will need to edit the model names in the download_inference_models() function in everything.py

### Next js client gets “hanged” when main.js tries to load the client URL:

- This issue can be caused when certain packages are deleted client/package.json
- This change caused the error
    ```

- 

```
@@ -9,7 +9,6 @@
    "lint": "next lint"
  },
  "dependencies": {
-   "-": "^0.0.1",
    "@dnd-kit/core": "^6.3.1",
    "@dnd-kit/modifiers": "^9.0.0",
    "@dnd-kit/sortable": "^10.0.0",
@@ -33,46 +32,32 @@
    "@radix-ui/react-tabs": "^1.0.4",
    "@radix-ui/react-toast": "^1.1.5",
    "@radix-ui/react-tooltip": "^1.0.7",
-   "@react-oauth/google": "^0.12.1",
-   "@supabase/ssr": "^0.5.2",
+   "@supabase/ssr": "^0.1.0",
    "@supabase/supabase-js": "^2.43.4",
    "@tanstack/react-table": "^8.17.3",
-   "axios": "^1.7.2",
-   "chart.js": "^4.4.3",
    "class-variance-authority": "^0.7.0",
    "clsx": "^2.1.1",
-   "core-js": "^3.37.1",
    "fabric": "^5.3.0",
-   "jsonwebtoken": "^9.0.2",
    "lucide-react": "^0.319.0",
    "moment": "^2.30.1",
-   "moment-timezone": "^0.5.45",
    "next": "^14.2.3",
    "react": "^18.3.1",
-   "react-chartjs-2": "^5.2.0",
    "react-dom": "^18.3.1",
    "react-hook-form": "^7.51.5",
    "react-icons": "^5.2.1",
    "react-resizable": "^3.0.5",
    "react-rnd": "^10.4.14",
    "react-selectable-box": "^1.9.4",
-   "react-selectable-fast": "^3.4.0",
    "tailwind-merge": "^2.3.0",
    "tailwindcss-animate": "^1.0.7",
    "zod": "^3.23.8"
  },
  "devDependencies": {
    "@types/fabric": "^5.3.7",
-   "@types/jsonwebtoken": "^9.0.6",
-   "@types/node": "^20.14.1",
    "@types/react": "^18.3.3",
    "@types/react-dom": "^18.3.0",
-   "@types/react-resizable": "^3.0.8",
-   "autoprefixer": "^10.4.19",
    "eslint": "^8.57.0",
    "eslint-config-next": "14.0.3",
-   "postcss": "^8.4.38",
-   "supabase": "^1.172.2",
    "tailwindcss": "^3.4.3",
    "typescript": "^5.4.5"
  }
```

### Model file could not be found:

- This can happen for a variety of reasons
  - check that all required model files are being downloaded properly (in everything.py)
  - check that the .env variables for the model paths are properly updated (in electron/main.js)
  - check that the model path is being properly loaded from the .env file in python, lines that load model paths should look like this:
    - `SIAMESE_PATH = os.environ.get("SIAMESE_PATH", "data_inference/siamese-4096-epoch=99-val_acc=~0.89.ckpt")`
    - This way the code will look in the .env variable called “SIAMESE_PATH” first. This variable should be specified if running in a production environment. If it is not, we are in development and we can just use the regular model path

### Transformers missing pyc file:

```
FileNotFoundError: [WinError 3] The system cannot find the path specified:
transformers\\models\\__init__.pyc'
```

- Some versions of transformers may cause this error
- Can be fixed by downgrading to transformers==4.51.3