Contents

[1. System Design Specifications 3](#_Toc201228438)

[2. Overview 3](#_Toc201228439)

[3. Architecture 3](#_Toc201228440)

[1. Frontend 4](#_Toc201228441)

[2. Backend 4](#_Toc201228442)

[3. Database 4](#_Toc201228443)

[4. Machine Learning Models 4](#_Toc201228444)

[5. Architecture Blueprint 4](#_Toc201228445)

[5.1 Key Users 4](#_Toc201228446)

[5.2 Deployment 4](#_Toc201228447)

[5.3 Key Tech Decisions**:** 4](#_Toc201228448)

[5.4 Key Assumptions and Dependencies 4](#_Toc201228449)

[6. Data Flow 3](#_Toc201228450)

[6.1 User Input: 4](#_Toc201228451)

[6.2 Search Process: 4](#_Toc201228452)

[6.3 Results Display: 4](#_Toc201228453)

[7. Deployment 3](#_Toc201228454)

[8. Security 3](#_Toc201228455)

[9. Scalability 3](#_Toc201228456)

[10. Technical Assumptions and Dependencies 3](#_Toc201228457)

[10.1 Assumptions 4](#_Toc201228458)

[10.2 Dependencies 4](#_Toc201228459)

[Libraries and Frameworks: 4](#_Toc201228460)

[10.3 Licensing: 4](#_Toc201228461)

[10.4 Infrastructure: 4](#_Toc201228462)

[11. Find My Part - Text Search 3](#_Toc201228463)

[11.1 Prepare Your Data 3](#_Toc201228464)

[11.2 Create Embeddings 3](#_Toc201228465)

[11.3 Select an Embedding Model 4](#_Toc201228466)

[11.4 Convert text fields to Vectors 4](#_Toc201228467)

[11.4 Store Vectors in Vector database 4](#_Toc201228468)

[11.3 Code - Generate Part Embeddings with python 3](#_Toc201228469)

[Option A: **Separate Vectors** Embed each field separately 4](#_Toc201228470)

[Option B: Concatenated String 4](#_Toc201228471)

[Option C: Both Vectors 4](#_Toc201228472)

[11.4 Set Up Query Pipeline 3](#_Toc201228473)

[User enters a query 3](#_Toc201228474)

[Code for User enters a query 3](#_Toc201228475)

[Code for SQL DB access using pyodbc 3](#_Toc201228476)

[11.5 Result Formatting 3](#_Toc201228477)

[12. Find My Part image search 3](#_Toc201228478)

[12.1 Approach : Visual Query to Part Search 3](#_Toc201228479)

[Image → Vector Representation 3](#_Toc201228480)

[Use the Image Embedding to Search 3](#_Toc201228481)

[Fallback to Image Captioning 4](#_Toc201228482)

[12.2 Implementing Image-based search 4](#_Toc201228483)

[Option A: Use CLIP for Cross-Modal Embedding with python code 4](#_Toc201228484)

[Preprocess image: 4](#_Toc201228485)

[Option B: Use BLIP or GPT4-Vision to Extract Text Description 4](#_Toc201228486)

[Option C: If You Already Have Images in Your DB… 4](#_Toc201228487)

[Option D : Combining These: 4](#_Toc201228488)

[12.3 Visual Query: User Experience Flow 4](#_Toc201228489)

[13. Conceptual mock-up of the **visual search schema** 4](#_Toc201228490)

[Visual Part Image - Search Schema 5](#_Toc201228491)

[**14. frontend wireframe concept** for the image-based part search feature. 5](#_Toc201228492)

[Frontend Prototype (Drag & Drop Upload) 5](#_Toc201228493)

[Frontend Elements to Implement: 5](#_Toc201228494)

# 1. System Design Specifications

# 2. Overview

The AI application for mechanical engineering designers is designed to provide a seamless search experience for parts using both text-based and visual inputs. The system integrates advanced machine learning models, a robust database architecture, and an intuitive frontend interface.

# 3. Architecture

### 1. Frontend

* **Framework**: React
* **Key Features**:
  + Drag-and-drop image upload
  + Optional text input for semantic search
  + Results display with images, descriptions, and links

### 2. Backend

* **Framework**: FastAPI
* **Key Features**:
  + API endpoints for text-based and visual search
  + Integration with SQL and vector databases
  + Image and text embedding generation

### 3. Database

* **SQL Database**: PostgreSQL for storing part metadata
* **Vector Database**: FAISS or Qdrant for fast similarity search

### 4. Machine Learning Models

* **Text Embedding**: SentenceTransformer (e.g., BAAI/bge-small-en)
* **Image Embedding**: CLIP (e.g., openai/clip-vit-base-patch32)

## 5. Architecture Blueprint

User Interface (Web or Desktop App)

↓

Query Processor (Backend API)

↓

[1] Semantic Vectorizer (Embeddings)

[2] SQL Part Database

[3] Vector Store (e.g., FAISS / Qdrant / Azure AI Search)

↓

Ranking + Response Formatter

↓

Results Returned to UI

### **5.1 Key Users**

* **Parts Designers - Mechanical and others**
* **Design Lead - Approval/Archival**
* **Purchase Lead - For Sourcing**

### **5.2 Deployment**

* **Tool will be deployed in their local corporate network and integrated with Enterprise application for secure access. Tool will not access anything outside the local network and only the approved systems/applications using APIs.**
* **Accessed only by design personnel and others authenticated**
* **All data will remain secured within their databases and other vaults and their access managed by org administrator**

### **5.3 Key Tech Decisions:**

* Vector Store: Use **FAISS** (local, fast), **Qdrant** (cloud), or **Pinecone** (if integrating with large-scale APIs).
* Hosting: Azure, AWS, or on-prem if needed for data privacy.
* UI Layer: Focus on ease of use—mechanical engineers often prefer clear filters and preview graphics.

### 5.4 Key Assumptions and Dependencies

* Tools/libraries/models: Only open-source free for commercial use
* APIs available for integrating with enterprise system, drawing vault etc
* Post Installation Training and fine tuning will be done within the network

# 6. Data Flow

### 6.1 User Input:

* + Text input is processed through the text embedding model to generate a query vector.
  + Image input is processed through the image embedding model to generate a query vector.

### 6.2 Search Process:

* + Query vector is matched against the vector database to retrieve top-k similar parts.
  + Part IDs from the vector database are used to fetch metadata from the SQL database.

### 6.3 Results Display:

* + Results are ranked and displayed on the frontend with images, descriptions, and links.

# 7. Deployment

* **Containerization**: Docker
* **Orchestration**: Kubernetes (optional for scaling)
* **Hosting**: AWS or Azure

# 8. Security

* **Authentication**: OAuth2 for secure API access
* **Data Encryption**: SSL/TLS for data in transit
* **Access Control**: Role-based access control for database and API

# 9. Scalability

* **Horizontal Scaling**: Add more instances of the backend and database services
* **Caching**: Use Redis or Memcached for frequently accessed data
* **Load Balancing**: Distribute traffic across multiple backend instances

# 10. Technical Assumptions and Dependencies

## 10.1 Assumptions

* The SQL database contains clean and normalized data for parts metadata.
* The vector database is pre-populated with embeddings for all parts.
* The application will be deployed within a secure corporate network.
* Users will have authenticated access to the application.
* Open-source libraries and models used are compatible with the project's licensing requirements.

## 10.2 Dependencies

### Libraries and Frameworks:

* + SentenceTransformer for text embeddings
  + CLIP for image embeddings
  + FastAPI for backend development
  + React for frontend development
  + FAISS or Qdrant for vector database
  + PostgreSQL for SQL database

### 10.3 Licensing:

* + Ensure all open-source libraries and models used are free for commercial use.
  + Verify compatibility with the organization's licensing policies.

### 10.4 Infrastructure:

* + Docker for containerization
  + Kubernetes for orchestration (if required)
  + Cloud hosting (AWS, Azure) or on-premise deployment

This revised system design specification incorporates the technical assumptions, dependencies, and licensing aspects to ensure a comprehensive and robust application.

# 11. Find My Part - Text Search

## 11.1 Prepare Your Data

* **Extract parts info** from the SQL database.
* **From your SQL DB, pull and prepare fields like:**
  + Part Name
  + Part Number (keep as-is or use tokenization if it encodes structure)
  + Description
  + Specifications or Features
* **Clean text:** 
  + Strip special characters, units (standardize), and irrelevant formatting.
  + Normalize synonyms if needed: e.g., “OD” vs “Outer Diameter”
  + Normalize text (remove HTML, correct typos, handle units).
* **Optional (but useful):**
  + Add metadata like category, manufacturer, and CAD preview links if available.

## 11.2 Create Embeddings

### 11.3 Select an Embedding Model

* + Choose a model suited for engineering or technical language. Options:
* BGE Small or Large: Compact and fast, good for edge deployment.
* Sentence-BERT: Flexible, fine-tunable, widely used.
* OpenAI’s text-embedding-3-small: Works well if you're building on Azure OpenAI.

### 11.4 Convert text fields to Vectors

* + Convert text fields in your parts database into *numerical vectors* that capture meaning, so similar concepts are close together even if different words are used.
* For each part, embed:
  + Name
  + Description
  + Number (as-is or tokenized)
  + Concatenated version for fallback matching

### 11.4 Store Vectors in Vector database

* + Store these vectors in a vector database like **FAISS** or **Qdrant**.

## 11.3 Code - Generate Part Embeddings with python

from sentence\_transformers import SentenceTransformer

import pandas as pd

import numpy as np

# **Load embedding model**

model = SentenceTransformer("BAAI/bge-small-en")

# **Example: Part descriptions from your SQL DB**

df = pd.DataFrame({

'part\_id': [101, 102, 103],

'name': ['Hex Bolt', 'Socket Screw', 'Carriage Bolt'],

'description': [

'M10 x 50mm grade 8.8 zinc-plated hex head bolt',

'M8 x 40mm alloy steel socket cap screw',

'M10 x 60mm mild steel round head carriage bolt'

]

})

**# Combine fields for better semantic context**

df['text\_for\_embedding'] = df['name'] + ": " + df['description']

# **Generate embeddings**

embeddings = model.encode(df['text\_for\_embedding'].tolist(), normalize\_embeddings=True)

# **Save for later use (e.g. insert into FAISS, Qdrant, etc.)**

df['embedding'] = embeddings.tolist()

* This gives you a vector per part, capturing its **semantic fingerprint**.
* Once stored in a vector database, these embeddings let you retrieve the most relevant parts—even if the user’s query is vague or uses different terminology.

### Option A: **Separate Vectors** Embed each field separately

### Option B: Concatenated String

"Bolt, Grade 8.8, M10x50mm, Zinc Plated" Embed this as a *combined item fingerprint* for faster retrieval.

### Option C: Both Vectors

* You can even store both: use detailed vectors for re-ranking, fast fingerprint vector for initial search.

import faiss

import numpy as np

# Assuming embeddings is a NumPy array of shape (N, D)

embedding\_matrix = np.array(df['embedding'].to\_list()).astype('float32')

# Create FAISS index

dimension = embedding\_matrix.shape[1]

index = faiss.IndexFlatL2(dimension) # Use cosine similarity with normalized vectors

index.add(embedding\_matrix)

If you want to retrieve metadata (like part IDs), wrap it in an IndexIDMap.

python

index = faiss.IndexIDMap(faiss.IndexFlatL2(dimension))

index.add\_with\_ids(embedding\_matrix, df['part\_id'].values)

## 11.4 Set Up Query Pipeline

### User enters a query

* Embed the query.
* Search vector DB for top N matches using cosine similarity.
* Optionally rerank with keyword filters or metadata (e.g., part category or tolerance).

### Code for User enters a query

query = "Mild steel round head bolt for structural applications"

query\_embedding = model.encode([query], normalize\_embeddings=True).astype('float32')

# Search FAISS index

top\_k = 5

distances, ids = index.search(query\_embedding, top\_k)

# Fetch part metadata from SQL using the returned IDs

similar\_parts = index = faiss.IndexIDMap(faiss.IndexFlatL2(dimension))

index.add\_with\_ids(embedding\_matrix, df['part\_id'].values)

df[df['part\_id'].isin(ids[0])]

* Once you retrieve the part\_ids from FAISS, use them to query your existing SQL database for detailed specs, drawings, pricing, or CAD links.

### Code for SQL DB access using pyodbc

If your SQL DB is connected via something like SQLAlchemy or pyodbc, you can easily run:

import pyodbc

conn = pyodbc.connect('your-connection-string')

query = f"SELECT \* FROM parts\_table WHERE part\_id IN ({','.join(map(str, ids[0]))})"

results = pd.read\_sql(query, conn)

## Result Formatting

* Present matches with:
  + Part name + number
  + Key specs snippet
  + Confidence score or similarity (optional)
  + “View More” link to full database entry or 3D preview

# 12. Find My Part image search

## 12.1 Approach : Visual Query to Part Search

### Image → Vector Representation

Use a **Visual Encoder** that can translate the image into an embedding:

* **CLIP (Contrastive Language-Image Pretraining)**: Excellent for general-purpose visual semantics. It encodes images and text into the *same embedding space*, allowing you to match a user-supplied image directly against text-based part records.
* **BLIP or GIT**: These models can generate a natural language description *from* the image, which you can then use for semantic search as if the user had typed it.

### Use the Image Embedding to Search

* Two paths here:

1. **Cross-modal matching (CLIP-style):**
   * Encode the image
   * Compare it with the *text embeddings* of parts you’ve already created (from names, descriptions, etc.)
2. **Image-to-image similarity (if you have image records in the DB):**
   * Embed every part image and store those in a vector DB
   * Then use *image-to-image* similarity to find the closest matches visually

### Fallback to Image Captioning

If embedding alone is too fuzzy:

* Use a captioning model (like BLIP) to create a rich **textual prompt** from the image
* Then run the regular **text→text embedding search** pipeline

## 12.2 Implementing Image-based search

### Option A: Use CLIP for Cross-Modal Embedding with python code

**Tools:**

* **CLIP** (by OpenAI) aligns text and images in the same embedding space
* Hugging Face’s openai/clip-vit-base-patch32
* Python packages: transformers, torch, PIL, faiss

### Preprocess image:

python

from PIL import Image

from transformers import CLIPProcessor, CLIPModel

import torch

model = CLIPModel.from\_pretrained("openai/clip-vit-base-patch32")

processor = CLIPProcessor.from\_pretrained("openai/clip-vit-base-patch32")

image = Image.open("user\_uploaded.jpg")

inputs = processor(images=image, return\_tensors="pt")

image\_embedding = model.get\_image\_features(\*\*inputs)

1. **Normalize & search vector DB** with this image embedding against your **text-based part embeddings** (from Step 2 earlier).

### Option B: Use BLIP or GPT4-Vision to Extract Text Description

If your user uploads an image without any accompanying text:

* Use **BLIP** to automatically generate a caption like: *“Hex-head bolt, partially threaded, zinc-plated.”*

**Tools:**

* BLIP captioning model
* Pipe the generated caption into your **existing semantic search pipeline**—as if the user typed that description.

### Option C: If You Already Have Images in Your DB…

You can:

1. Precompute **image embeddings** for your entire parts catalog using CLIP.
2. Then compare the **user's image → image-to-image** similarity using cosine distance.

This avoids needing any text—but is better when your part images are clean, standardized, and well-structured.

### Option D : Combining These:

You could even offer a hybrid:

* Use BLIP → text search (fast + high recall)
* Then rerank based on CLIP image-to-image match (precision)

## 12.3 Visual Query: User Experience Flow

1. **User uploads an image** (photo, drawing, or screenshot of a part)
2. App runs:
   * **BLIP** to generate a *caption or semantic description*, *or*
   * **CLIP** to create **a *visual embedding***
3. Use the output to query:
   * Your **existing** **text embedding DB** (BLIP path), or
   * Your **image embedding DB** (CLIP path, if your SQL parts have images too)
4. **Match results returned** with part names, numbers, specs, and previews

You can also show side-by-side results: *“Here’s what you searched for vs what we found.”*

# 13. Conceptual mock-up of the **visual search schema**

Visual Part Image - Search Schema

┌───────────────────────────────┐

│ User Uploads an Image │ ← (CAD image, photo, screenshot)

└───────────────────────────────┘

↓

┌───────────────────────────────┐

│ Image Understanding Module │

│ • CLIP (Image → Embedding) │

│ • or BLIP (Image → Caption) │

└───────────────────────────────┘

↓

┌──────────────────────┐ ┌────────────────────────┐

│ Image Embedding │──────▶│ Image Vector DB (opt) │

└──────────────────────┘ └────────────────────────┘

│

▼

┌────────────────────────────────┐

│ Text Caption (from BLIP/GPT-4V)│

└────────────────────────────────┘

↓

┌────────────────────┐

│ Text Vector Search │ ← searches precomputed part vectors

└────────────────────┘

↓

┌─────────────────────────┐

│ SQL DB Part Metadata │ ← (Part ID, Name, Specs, CAD Link)

└─────────────────────────┘

↓

┌─────────────────────────┐

│ Visual + Ranked Results│ ← (Top matches with image & info)

└─────────────────────────┘

This schema gives you two key paths:

**Pure vision-to-vision**: CLIP for comparing uploaded image with existing part images.

1. **Vision-to-text**: BLIP generates a semantic caption, and you use your **existing text search pipeline**.

# **14. frontend wireframe concept** for the image-based part search feature.

Implement using a modern stack like React + Flask/FastAPI or Node.js..

## Frontend Prototype (Drag & Drop Upload)

┌──────────────────────────────────────────┐

│ 🔍 Find a Mechanical Part │

│------------------------------------------│

│ │

│ 📂 Drag & drop an image here │

│ or click to upload │

│ │

│ [ Optional Text Description ] │

│ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ │

│ | "Bolt with flanged hex head..." | │

│ |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_| │

│ │

│ [ SEARCH SIMILAR PARTS ] │

└──────────────────────────────────────────┘

⬇

┌──────────────────────────────────────────┐

│ 🔎 Results: │

│------------------------------------------│

│ ✅ M10 x 60mm Flanged Hex Bolt │

│ • Grade 8.8 Zinc Plated Steel │

│ • Part No: BOLT-1088-Z │

│ • Confidence: 92% │

│ • [🖼️ View Image] [🔗 CAD File] │

│ │

│ ✅ M10 Hex Head Structural Bolt │

│ • Grade 10.9 Black Oxide │

│ • Part No: STR-M10-109B │

│ • Confidence: 87% │

│ │

└──────────────────────────────────────────┘

## Frontend Elements to Implement:

* **Dropzone + File Picker** (e.g. react-dropzone)
* **Optional Text Field** (semantic fallback)
* **Preview uploaded image**
* **Progress bar or loading spinner**
* **Results panel** with cards showing:
  + Image
  + Description
  + Part number
  + Confidence score or rank
  + Link to SQL details or CAD
  + \