TATA Nano RAG Diagnostics Chatbot

Technical Documentation

Presented by IntelliPredikt

Document Overview

- **Introduction:** TATA Nano RAG Diagnostics Chatbot
- **Tech Stack:** Core technologies used
- **Architecture:** High-level and detailed system design
- **System Flow:** Step-by-step process of query handling
- **Implementation Approach:** Key technical strategies
- **Data Structures & APIs:** How data is organized and exposed
- **Frontend & Security:** User interface and protection measures
- **Performance & Testing:** Optimizations and validation
- **Deployment & Decisions:** Infrastructure and rationale behind choices

🏗️ Technology Stack

Backend Framework

- Flask 3.0.0: Lightweight web framework
- Flask-CORS 4.0.0: Cross-Origin support
- Gunicorn 21.2.0: WSGI HTTP server

AI & Machine Learning

- Anthropic Claude API (Sonnet 4.5): LLM
- Sentence Transformers 2.2.2: Embeddings
- scikit-learn 1.3.2: Cosine similarity
- NumPy 1.24.3: Numerical operations

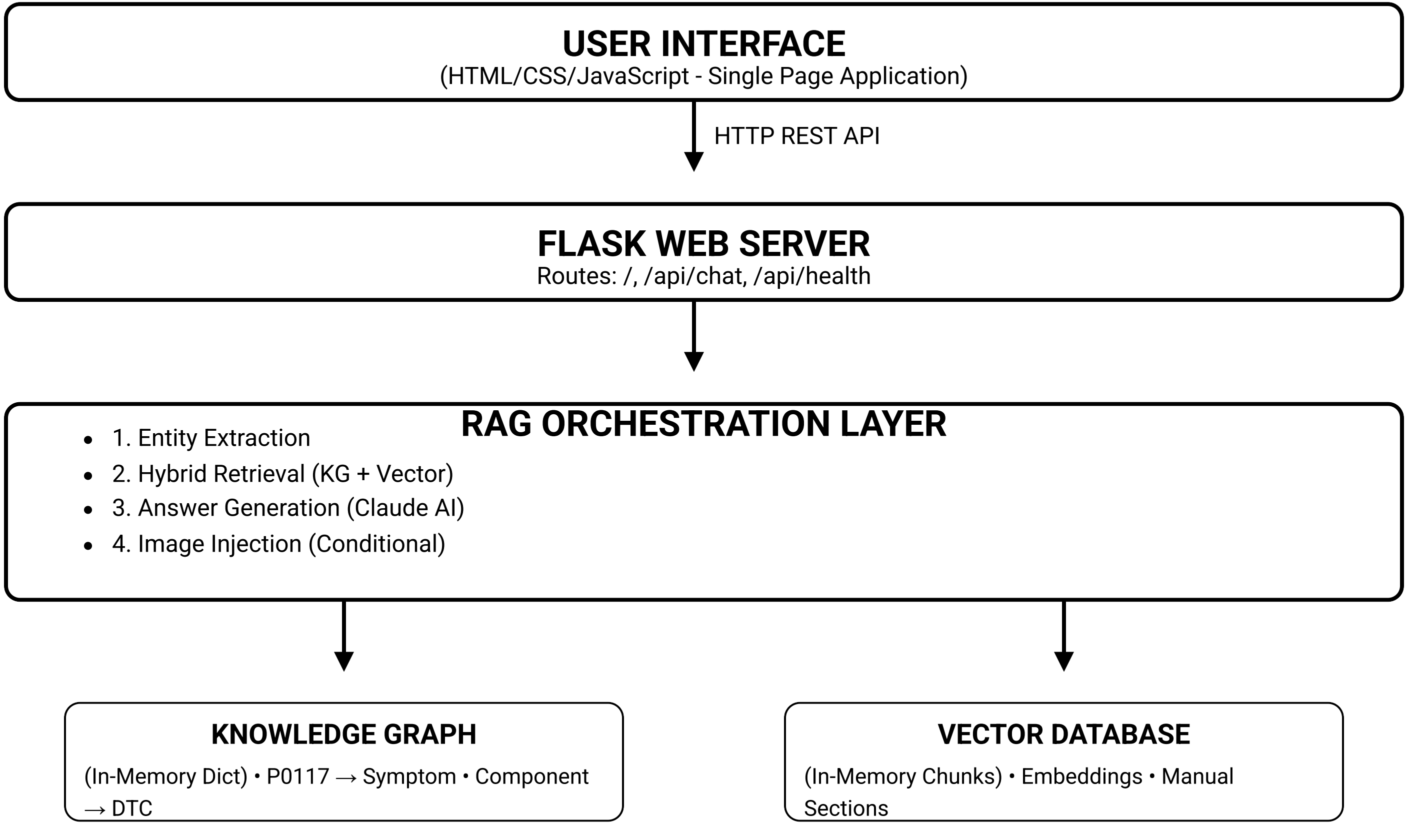
Containerization

- Docker: Container platform
- Docker Compose: Orchestration
- Python 3.11-slim: Base image

Development Tools

- Python 3.11: Core language
- pip: Package management

🎯 High-Level Architecture



🔄 System Flow Diagram (Part 1)

User Query

1. QUERY ANALYSIS & ENTITY EXTRACTION

```
Input: "P0117 showing, show details"
Output: { dtc_codes: ["P0117"], query_type: "explanation" }
```

2. PARALLEL RETRIEVAL

Knowledge Graph Retrieval

Query by: DTC codes, Components, Symptoms

Returns: Triples (max 10)

Vector Database Search

1. Encode query 2. Filter chunks 3. Cosine sim 4. Rank results

Returns: Chunks (top 5)

3. HYBRID FUSION (RRF)

KG Score (40%) + Vector Score (60%) Combined ranking

🔄 System Flow Diagram (Part 2)

4. CONTEXT PREPARATION

• Format KG triples • Format vector chunks • Build system prompt

↓

5. CLAUDE AI GENERATION

API Call: • Model: claude-sonnet-4-20250514
• System prompt + context • User query • Max tokens: 2048
Returns: Structured HTML response

↓

6. POST-PROCESSING

IF query_type == "image_request": → Inject SVG diagrams
ELSE: → Return as-is

↓

7. RESPONSE DELIVERY

`JSON: { answer: "...", sources: {...} }`

🧩 Implementation Approach

1. Hybrid RAG Architecture

Why Hybrid?

- **Knowledge Graph:** Structured relationships (DTC → Symptom → Component)
- **Vector Database:** Semantic understanding of natural language
- **Combination:** Best of both worlds - precision + comprehension

Implementation: In-memory KG (dict) and Vector DB (chunks with embeddings).

2. Query Type Detection

Smart Context-Aware Responses: Query Analysis → Query Type → Response Strategy

- "P0117 details" → `explanation`
- "Show me picture" → `image_request`
- "How to fix P0117" → `repair`
- "Fan always running" → `general`

Implementation: Keyword-based entity extraction function `extract_entities(query)`.

🧩 Implementation Approach (Cont.)

⌛ 3. Lazy Loading Pattern

Problem: Pre-computing embeddings causes memory issues in Docker.
Solution: Compute embeddings on first use for filtered chunks.
Benefits: Reduces startup memory, avoids crashes, caches embeddings.

🗨 4. Prompt Engineering Strategy

Context-Aware System Prompts: Dynamic prompts based on `query_type`.
Example: `explanation` prompt includes symptoms/causes, excludes repair steps.
Example: `image_request` prompt focuses on location/connector details.

◯ 5. Embedding Model Selection

Model:`all-MiniLM-L6-v2`
Why: ✅ Lightweight (80MB), ✅ Fast (~0.01s/encoding), ✅ Balanced accuracy for automotive, ✅ 384 dimensions.
Alternatives:`all-mpnet-base-v2` (larger), `distilbert-base-nli` (lower quality).

🧩 Implementation Approach (Cont.)

6. Docker Optimization

Single Worker Architecture:

```
CMD ["gunicorn", "-w", "1", "--threads", "4", "-b", "0.0.0.0:5001", "--timeout", "120", "--preload", "chatbot_backend:app"]
```

Why: ❌ Multiple workers = SIGSEGV (model instances). ✅ Single worker + threads = shared memory. ✅ `--preload` forking.

7. Image Handling Strategy

Inline SVG Data URIs:`COOLANT_SENSOR_IMG = ''''''`

Why: ✅ No external files, ✅ Fast rendering, ✅ Scalable (vector), ✅ Small size (~2KB).

Alternatives Rejected: ❌ External image files, ❌ Base64 PNGs (larger).

📊 Data Structures & 🔧 API Endpoints

Knowledge Graph Structure

{ "entity_id": { "type": "DTC|Component|Symptom", "relationships": [...], "attributes": {...} } }
Example: "P0117": { "type": "DTC", "fault_cause": "Short Circuit to Ground", "symptoms": [...] }

Vector Chunk Structure

{ "id": "chunk_1", "text": "DTC P0117 indicates...", "dtc": "P0117", "component": "Coolant Sensor", "embedding": np.array([...]) }

API Endpoints

- **GET /:** Serves frontend HTML (SPA with chat interface)
- **POST /api/chat:** Handles user queries, returns structured JSON response
- **GET /api/health:** Provides system status and configuration details

🎨 Frontend Architecture & 🔐 Security Considerations

Frontend Architecture

Technology Stack:

- HTML5: Structure
- CSS3: Styling (Gradients, animations, flexbox)
- Vanilla JavaScript: No frameworks (lightweight)

Design Patterns: Async message handling with fetch API.

UI Features:

- Gradient backgrounds
- Slide-in animations
- Responsive design
- Auto-scroll messages
- Loading states

Security Considerations

1. Environment Variables: API keys loaded securely (fail-fast if missing).

2. CORS Configuration:`CORS(app)` to allow cross-origin requests.

3. Input Validation: Query length and content checked to prevent abuse.

4. No External Dependencies:

- ✅ SVG images embedded
- ✅ No CDN dependencies
- ✅ Self-contained application

📈 Performance, 🧪 Testing & 🚀 Deployment

Performance Optimizations

- Lazy Embedding Computation: Fast subsequent queries.
- In-Memory Storage: O(1) lookup, no disk I/O.
- Efficient Similarity Calculation: NumPy vectorized cosine similarity.
- Limited Context Window: Max 10 triples, 5 chunks, 2048 tokens.

Testing Approach

Query Categories Tested:

- Explanation ("P0117 details")
- Image Request ("Show me coolant sensor")
- Repair ("How to fix P0117")
- Symptom ("Fan always running")

Test Commands:`curl` examples for API endpoints.

Deployment Architecture & Resources

Container Structure: Gunicorn WSGI (1 worker, 4 threads, 120s timeout), Flask App, ML Models (SentenceTransformer, Embeddings cache), Data (KG & Vector chunks in-memory).

Resource Requirements:

- Memory: 2GB min (4GB recommended)
- CPU: 1 core min (2 cores recommended)
- Disk: 500MB
- Network: Internet for first startup (model download)