Automatiq.ai Product Recommendation Agent - Project Summary

Overview

In this project I implemented an AI-powered product recommendation system utilizing RAG, Semantic Search and Web Scraping.

I tried to design the system in a manner which reflects how I would like the experience of buying a laptop to ideally be.

Initially, I was faced with the challenge of limited text descriptions for my products and thus implemented a Web Scraping agent with the task of scraping NanoReview.net for additional information for the products.

This is as our "Knowledge Base" which serves 2 main purposes:

1- More relevant context for the LLM to base its recommendation upon

2- More info for the user once the product is recommended. When I buy a laptop I want all the information I can get and won't be satisfied with a 1 line text description.

Technology Stack

Frontend: - React 18 + TypeScript - Framer Motion (animations) - React Query (state management) - Tailwind CSS (styling) - Lucide Icons

Backend: - Python 3.11+ - FastAPI (async web framework) - Pydantic (data validation) - Google Generative AI (Gemini API) - NumPy (vector operations) - BeautifulSoup4 (web scraping) - HTTPX (async HTTP client)

AI/ML: - Google Gemini 2.5 Pro (main LLM) - Google Gemini 2.5 Flash (knowledge generation) - Google Embedding Model 001 (semantic embeddings)

Coding Assistants: - Claude Code, Codex

Architecture Overview

System Architecture

```
┌──────────────────────────────────────────────────────────────
──┐
│                          Frontend                       │
│ ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│ │ Chat View    │  │ Metrics View │  │ History View │      │
│ └──────────────┘  └──────────────┘  └──────────────┘         │
└──────────────────────────────┬───────────────────────────────
──┘
                         │ WebSocket / REST API
┌──────────────────────────────┴───────────────────────────────
──┐
│                     Backend (FastAPI)                   │
│ ┌─────────────────────────────────────────────────────┐
│ │                Chat Router (WebSocket + REST)      │  │
│ └────────────┬─────────────────────────────┬──────────┘
│              │                  │           │
│ ┌────────────▼───────────┐  ┌──────────────▼───────────┐
│ │    RAG Service      │  │  LLM Service        │  │
│ │ ┌──────────────────┐ │  │ ┌──────────────────┐  │  │
│ │ │Embedding Model │  │ │ │Gemini Provider │  │  │
│ │ │Vector Search   │  │ │ │(Gemini 2.5 Pro)│  │
│ │ │Hybrid Search   │  │ │ └──────────────────┘  │  │
│ │ │Knowledge Base  │  │ │ ┌──────────────────┐  │  │
│ │ └──────────────────┘ │ │ │ │Structured Parser│  │  │
│ └────────────────────────┘ │ │ └──────────────────┘  │  │
│                            └──────────────────────────┘ │
│ ┌───────────────────────────────────────────────────┐    │
│ │        Metrics Service                  │  │
│ │  - Conversation logging                 │  │
│ │  - Latency tracking                     │  │
│ │  - User feedback                        │  │
│ │  - Analytics aggregation                │  │
│ └───────────────────────────────────────────────────┘    │
│ ┌───────────────────────────────────────────────────┐
│ │      Scraping Service (Knowledge Builder)  │  │
│ │  - NanoReview web scraper               │  │
│ │  - LLM-based summarization              │  │
│ │  - Knowledge base caching               │  │
│ └───────────────────────────────────────────────────┘    │
└──────────────────────────────────────────────────────────────
──┘
                         │
┌──────────────────────────────┴───────────────────────────────
──┐
│                      Data Layer                         │
│ ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│ │ products.json│  │ embeddings   │  │ product_     │      │
│  │ (31 items)      │ │ .npy           │ │ knowledge      │      │
│  │                 │ │ (vectors)      │ │ .json          │      │
│  └───────────────────┘ └────────────────────┘ └────────────────────┘      │
└──────────────────────────────────────────────────────────────────────────
───┘
```

Technical Implementation

Implementation Details:

Semantic Embeddings

```
# Each product is converted to a rich text representation
def _product_text(self, product: Product) -> str:
    text = f"""
    SKU: {product.sku}
    Name: {product.name}
    Description: {product.description}
    CPU: {product.cpu}
    GPU: {product.gpu}
    ...
    Product Summary: {knowledge.summary}  # From knowledge base
    Strengths: {knowledge.strengths}
    Best for: {knowledge.use_cases}
    """
    return text
```

Hybrid Search Strategy

Combined two approaches for maximum retrieval quality:

1. **Semantic Search (80% weight):**
   - Embed query using same model as products - Google Embedding Model 001
   - Compute cosine similarity with pre-computed product embeddings
   - Normalized vectors for efficient computation
2. **Keyword Search (20% weight):**
   - Extract keywords from query and products
   - Match specific technical terms (GPU models, CPUs, etc.)
   - Boost results with exact keyword matches

```
combined_score = semantic_similarity + keyword_bonus
# keyword_bonus capped at 0.2 to avoid over-weighting
```

Why this method?

On the one hand, the knowledge base provides a rich semantic basis to base the LLMs answers upon and using Semantic Search can yield pretty good results. On the other hand, in the domain of laptops there is technical jargon that requires precision and the value of Keyword Search helps the LLM deal with this.

2. LLM Integration Architecture

Design Philosophy: Abstract LLM provider behind interface for easy switching.

Prompt Engineering:

The system uses a sophisticated multi-part prompt:

```
system_prompt = """
You are a knowledgeable laptop shopping assistant...

CONVERSATION APPROACH:
- Ask 2-3 clarifying questions before recommending
- Understand use case, budget, priorities
- Be conversational and helpful

RECOMMENDATION FORMAT:
When ready, provide recommendations in this EXACT format:
[REASONING]
Your thought process...
[/REASONING]

[RECOMMENDATIONS]
SKU: <product-sku>
Name: <product-name>
Rationale: <why this product fits>
Confidence: <0.0-1.0>
[/RECOMMENDATIONS]
"""
```

Why this approach? - Clear separation between reasoning and recommendations - Structured format enables reliable parsing - Confidence scores help with ranking - Natural conversation flow before recommendations

3. Knowledge Base Enrichment

Problem: Raw product specs aren't enough for quality recommendations.

Solution: Built an automated knowledge base builder that scrapes reviews and generates summaries. Add products to products.json and upon building the knowledge base these products will be scraped for. NO HARD CODING.

NanoReview Scraper

```
class NanoReviewScraper:
    async def scrape_nanoreview(self, product: Product) -> Optional[str]:
        # Try multiple URL patterns
        potential_urls = [
            f"https://nanoreview.net/en/laptop/{vendor}-{normalized_name}",
            f"https://nanoreview.net/en/laptop/{normalized_name}",
        ]

        # Extract pros/cons, descriptions, reviews
        # LLM generates 2-paragraph summary from scraped content
```

Knowledge Structure:

```
class ProductKnowledge(BaseModel):
    sku: str
    summary: str  # 2-paragraph LLM-generated overview
    strengths: List[str]  # Key advantages
    weaknesses: List[str]  # Limitations
    use_cases: List[str]  # Best scenarios
    last_updated: datetime  # For cache invalidation
```

4. Metrics System

Tracking Layers:

1. **Per-Message Metrics:**
   - Timestamp
   - Role (user/assistant)
   - Content
   - Message ID for feedback
2. **Per-Session Metrics:**

```
class SessionMetrics:
    session_id: str
    turn_count: int
    retrieval_latency_ms: float
    llm_latency_ms: float
    recommended_products: List[str]
    user_feedback: Dict[str, str]  # positive/negative
    started_at: datetime
    updated_at: datetime
```

3. **Aggregate Metrics:**
   - Total sessions
   - Average conversation length
   - Average latencies
   - Most recommended products
   - Positive feedback ratio

Persistence: All metrics stored as JSON files with thread-safe locking.

5. Frontend Architecture

Three Main Views:

1. **Chat Interface:**
   - Real-time message streaming
   - Product cards with expandable details
   - Knowledge summaries
   - Reasoning display
   - Feedback buttons
2. **Metrics Dashboard:**
   - Aggregate statistics
   - Session performance
   - Product recommendation frequency
   - Latency trends
   - CSV export
3. **Conversations History:**
   - Past conversation list
   - Feedback status
   - Conversation preview
   - Rating display
   - Re-open conversations

Core Features & Innovations

1. Intelligent Budget Extraction

```
def _extract_price_from_query(query: str) -> Optional[float]:
    """Extract maximum price constraint from natural language query."""
    patterns = [

r"(?:under|below|max|maximum|up\s+to)\s*\$?\s*([0-9,]+(?:\.[0-9]{2})?)",
        r"\$?\s*([0-9,]+(?:\.[0-9]{2})?)\s*(?:or\s+)?(?:less|under|below)",
    ]
    # Automatically applies price filters without explicit user input
```

Impact: Users can naturally mention budget constraints in conversation.

2. Conversation Memory

- Full conversation history passed to LLM
- Configurable history window (default: 6 turns)

3. Product Knowledge Enrichment

- Automated web scraping of reviews
- LLM-generated product summaries

- Strengths/weaknesses analysis

Result: Recommendations are based on deep product understanding, not just specs.

4. *Hybrid Search Innovation*

- Combined semantic + keyword search
- Keyword index for exact matches
- Handles both natural queries and technical terms

Example: - Query: "gaming laptop with RTX 4090" - Semantic: Understands "gaming" context - Keyword: Exact match on "RTX 4090" - Combined: Perfect results

5. *Performance Optimization*

Several optimizations for production-grade performance:

```
# Pre-computed normalized embeddings
self._normalized_embeddings = self._normalize_embeddings(embedding_matrix)

# Efficient similarity with single matrix multiplication
similarities = self._normalized_embeddings @ query_embedding

# Cached product text representations
self._product_text_cache: Dict[str, str] = {}

# Keyword index for O(1) lookups
self._keyword_index = self._build_keyword_index(self.products)
```

Latency Results: - RAG retrieval: ~100-300ms - LLM streaming: First token ~500-800ms - Total time to first recommendation: <2 seconds

6. *Conversation Quality Features*

Reasoning Display: - Shows LLM's internal thought process - Explains why products were recommended - Builds user trust in recommendations

Challenges & Solutions

Challenge 1: LLM Response Parsing

Problem: LLMs generate free-form text. Extracting structured recommendations reliably was difficult.

Solution: Implemented robust format:

```
[REASONING]
...
[/REASONING]

[RECOMMENDATIONS]
SKU: XPS-9340
```

```
Name: XPS 13
Rationale: ...
[/RECOMMENDATIONS]
```

Challenge 2: Knowledge Base Quality

Problem: Product specs alone don't provide enough context for good recommendations.

Also, user needs more information to A- understand what product he is buying. B- gain confidence the AI hes interacting with is grounded and not bullshitting.

Solution: NanoReview Focused Scraper (nanoreview_scraper.py):

Final Solution:

```python
# Try scraping, but always generate summary
scraped_content = await self.scrape_nanoreview(product)
summary = await self.generate_summary_with_llm(product, scraped_content)

# If no content, LLM generates from specs
if not scraped_content:
    return self._generate_fallback_knowledge(product)
```

Result: 100% knowledge base coverage, with varying levels of detail.

Challenge 3: Product Comparison

Problem: I tried to implement a product comparison card where the user gets multiple options and Pros and Cons of each alternative.

Decision:

This made my system crash and the latency added from this wasn't worth the benefit at this stage. In the future this feature is cool and can be added.

Challenge 4 : Conversation Context Management

Problem: How much history to pass to LLM?

Too Little: - Loses context - Repetitive questions - Poor user experience

Too Much: - Token costs increase - Slower responses - Context window limits

Current Solution: 6 turns (12 messages) provides good balance for laptop shopping conversations.

Future: Optimize this further. Still not working optimally and the LLM sometimes loses context even in the 6 turns window.

Challenge 5: How to quantify technical specs?

Problem: Technical specifications like "32GB DDR5" vs "64GB DDR4" or "RTX 4090 Laptop" vs "RTX 5000 Ada" cannot be directly compared numerically, and each component type (CPU/GPU/RAM/Storage) has different generations, variants, and performance characteristics that make simple string matching inadequate.

Solution: The system relies entirely on AI understanding rather than explicit normalization - the semantic embedding model naturally learned hardware hierarchies during training, while Gemini LLM has basic technical understanding which is enhanced by knowledge base with additional context from NanoReview.net

Challenge 6 : Conversation Context Management

Problem: How much history to pass to LLM?

Too Little: - Loses context - Repetitive questions - Poor user experience

Too Much: - Token costs increase - Slower responses - Context window limits

Current Solution: 6 turns (12 messages) provides good balance for laptop shopping conversations.

Future: Optimize this further. Still not working optimally and the LLM sometimes loses context even in the 6 turns window.

Next Steps & Improvements

1. Enable Product Comparison

Problem: Comparison generation adds latency and crashes system.

Value: As a buyer, I think in terms of comparisons and want to weigh my alternatives. I believe this feature has a lot of value for potential customers.

2. Add Vector Database (Qdrant/Chroma)

Current: NumPy in-memory vectors.

Limitation: - No persistence across restarts - Limited to small datasets - No advanced filtering.

Benefits: - Persistent vector storage - Scales to millions of products - Advanced filtering capabilities - Distributed deployment ready

3. Multi-Agent Architecture

Solution: Replace the single conversational agent with specialized agents - a router agent to classify intent, a product expert for recommendations, a technical expert for spec questions, and a price comparison agent for deals.

Improvement: Each agent becomes highly specialized in its domain, leading to more accurate recommendations and faster responses since you can use lighter models (Gemini Flash) for simple queries and reserve Pro for complex recommendations.

4. User Preference Learning

Solution: Track user interactions (products viewed, liked, conversation patterns) and build a user preference vector that gets blended with their query embeddings - essentially personalizing the semantic search based on past behavior.

Improvement: Reduces conversation length for repeat users since the system already knows their preferences (e.g., always prefers Lenovo, values battery life over performance)

5. Real-Time Inventory Integration

Solution: Integrate with retailer APIs or web scrapers to check real-time product availability and current prices, filtering out-of-stock items and showing live pricing with direct purchase links.

Improvement: Transforms the system from an advisory tool into an actionable shopping assistant

Technical Metrics Summary

Code Statistics: - Backend: ~3,000 lines of Python - Frontend: ~2,000 lines of TypeScript/React - Total: ~5,000 lines of production code - Test coverage: Basic manual testing (automated tests needed)

Performance: - Average response latency: 3-5 seconds - RAG retrieval: 100-300ms - First token latency: 500-800ms - Supported concurrent users: 10-20 (single instance)

Scalability: - Products supported: 31 (tested) - Can scale to: 1,000+ products with current architecture - Recommended for: 10,000+ products switch to vector database

Cost Efficiency: - Average tokens per conversation: 3,000-5,000 - Cost per conversation (Gemini): ~$0.01-0.02 - Monthly costs (1,000 conversations): ~$10-20
