AI Agent Architecture Document: PriceWarden

System Overview

PriceWarden is an agentic AI system that automates e-commerce price comparison through intelligent reasoning, planning, and execution. The agent processes multimodal inputs (text/images) to find the best prices across Amazon, Flipkart, and Myntra.

Core Components

1. PriceComparisonAgent (agent/core.py)

The central orchestrator implementing the agentic framework:

```python
class PriceComparisonAgent:
    def __init__(self):
        self.vision_model = ImageToTextModel()  # Fine-tuned BLIP
        self.scraper = WebScraperTool()
        self.memory = SQLiteMemory()
        self.confidence_scorer = ConfidenceScorer()
```

2. Vision Model (models/inference.py)

- **Base Model**: Salesforce/blip-image-captioning-base
- **Fine-tuning**: LoRA adapter for e-commerce specialization
- **Purpose**: Converts product images to searchable text queries

3. Web Scraper Tool (scrapers/web_scraper.py)

- **Dual-mode**: Requests for simple HTML, Selenium for JavaScript
- **Features**: Retry logic, rate limiting, fallback mechanisms
- **Sites**: Amazon, Flipkart, Myntra

4. Memory System (agent/memory.py)

- **Database**: SQLite for persistent storage
- **Tracks**: Search history, success rates, user preferences
- **Learning**: Improves confidence scores over time

Interaction Flow

```
User Input (Text/Image)
```

```
↓
1. REASONING PHASE
  - Input type detection
  - Image → Text conversion (if needed)
  - Query refinement (remove stopwords)
  ↓
2. PLANNING PHASE
  - Site selection from config
  - Priority ordering based on product type
  - Resource allocation
  ↓
3. EXECUTION PHASE
  - Parallel web scraping
  - Error handling & retries
  - Fallback to cache if needed
  ↓
4. LEARNING PHASE
  - Store results in memory
  - Update confidence scores
  - Pattern recognition
  ↓
5. SYNTHESIS PHASE
  - Relevance scoring
  - Price comparison
  - Recommendation generation
  ↓
Final Output (Sorted Results + Insights)
```

Detailed Component Interactions

Input Processing Pipeline

python

```python
def _reason(self, input_data):
    if isinstance(input_data, Image):
        # Use fine-tuned vision model
        query = self.vision_model.generate_caption(input_data)
        query = self._refine_search_query(query)
    else:
        # Direct text processing
        query = self._refine_search_query(input_data)
    return query
```

Parallel Execution Strategy

```python
def _execute(self, sites):
    with ThreadPoolExecutor(max_workers=3) as executor:
        futures = {
            executor.submit(self.scraper.scrape, site): site
            for site in sites
        }
        results = {}
        for future, site in futures.items():
            try:
                results[site] = future.result(timeout=10)
            except Exception as e:
                results[site] = self._get_fallback_data(site)
    return results
```

Learning Mechanism

```python
def _learn(self, query, results):
    # Update search history
    self.memory.add_search(query, results)

    # Calculate success metrics
    success_rate = len(results) / len(self.sites)
    avg_relevance = np.mean([r.relevance for r in results])

    # Update confidence
    self.confidence_scorer.update(success_rate, avg_relevance)

    # Pattern recognition
    if "kurta" in query.lower():
        self.memory.update_site_preference("myntra", +0.1)
```

Technology Choices & Rationale

Why BLIP + LoRA?

- **BLIP**: Strong baseline for image captioning
- **LoRA**: Enables fine-tuning on consumer GPU (8GB VRAM)
- **Result**: 73% accuracy with 99.75% fewer trainable parameters

Why Selenium + Requests?

- **Requests**: Fast for static content (Amazon product lists)
- **Selenium**: Necessary for JavaScript-rendered prices
- **Fallback**: Ensures reliability even when scraping fails

Why SQLite?

- **Lightweight**: No server setup required
- **Sufficient**: Handles our data volume easily
- **Portable**: Easy to deploy and backup

Why ThreadPoolExecutor?

- **Simplicity**: Easier than asyncio for this use case
- **Performance**: 3x speedup (15s → 5s)
- **Control**: Better error handling than multiprocessing

Configuration Management

All settings centralized in `config/config.yaml`:

```yaml
```

```yaml
agent:
  confidence_threshold: 0.7
  max_retries: 3
  timeout: 10

model:
  checkpoint_path: "./models/checkpoints/final_model"
  device: "cuda"
  max_length: 50

scraping:
  sites:
    - name: "amazon"
      url: "https://www.amazon.in/s?k="
      use_selenium: false
    - name: "flipkart"
      url: "https://www.flipkart.com/search?q="
      use_selenium: true
    - name: "myntra"
      url: "https://www.myntra.com/"
      use_selenium: true
      fallback_enabled: true

rate_limiting:
  requests_per_minute: 30
  delay_between_sites: 1
```

Error Handling Strategy

Multi-Layer Resilience

1. Try primary scraping method
2. On failure → Retry with exponential backoff
3. Still failing → Switch to Selenium
4. Persistent failure → Use cached fallback data
5. Log all failures for debugging

Example Implementation

```
python
```

```python
def scrape_with_resilience(self, url):
    strategies = [
        self._scrape_with_requests,
        self._scrape_with_selenium,
        self._get_cached_data
    ]

    for strategy in strategies:
        try:
            return strategy(url)
        except Exception as e:
            logger.warning(f"Strategy {strategy.__name__} failed: {e}")
            continue

    return []  # Empty results better than crashing
```

Performance Optimizations

Model Loading

- Lazy loading: Model loads only when needed
- Cached in memory after first load
- Quantized to INT8 for 2x faster inference

Scraping Optimization

- Connection pooling for HTTP requests
- Pre-compiled regex patterns
- CSS selector caching

Memory Management

- Limited to 100 products in memory
- Automatic cleanup of old searches (>24 hours)
- Efficient data structures (only store essentials)

Scalability Considerations

Horizontal Scaling Ready

- Stateless agent design
- Database supports concurrent access
- Could deploy multiple instances behind load balancer

Vertical Scaling Options

- Batch processing for multiple queries
- GPU inference for higher throughput
- Caching layer (Redis) for common searches

Security & Ethics

Web Scraping Ethics

- Respects robots.txt
- Rate limiting (30 requests/minute)
- User-agent identification
- No personal data collection

User Privacy

- All processing done locally
- No user data stored permanently
- Anonymous usage statistics only

System Requirements

Minimum Requirements

- Python 3.8+
- 8GB RAM
- 4GB GPU VRAM (or CPU mode)
- 2GB disk space

Recommended Setup

- Python 3.10
- 16GB RAM
- 8GB GPU VRAM
- SSD for database

Deployment Architecture

```
User Interface (Streamlit)
     ↓
  CostFilter Agent
  /    |    \
```

```
Vision  Scraper  Memory
Model    Tool    System
  |        |        |
 GPU   Internet  SQLite
```

Monitoring & Logging

Metrics Tracked

- Query response time
- Scraping success rate
- Model inference speed
- Memory usage
- Cache hit rate

Logging Levels

- INFO: Normal operations
- WARNING: Fallback activations
- ERROR: Scraping failures
- DEBUG: Detailed execution flow

Future Architecture Enhancements

Multi-Agent System

```
Orchestrator Agent
    ├── Search Agent (current functionality)
    ├── Price Tracking Agent (monitors changes)
    ├── Recommendation Agent (personalized suggestions)
    └── Negotiation Agent (finds coupons/deals)
```

Microservices Architecture

- Vision Service (Docker container)
- Scraping Service (distributed workers)
- API Gateway (FastAPI)
- Database Service (PostgreSQL upgrade)