context: 📷 LHH Senior Automation Engineer $95k – $125k annually 📷 Flexco Senior Automation Engineer $100k – $126k annuall...

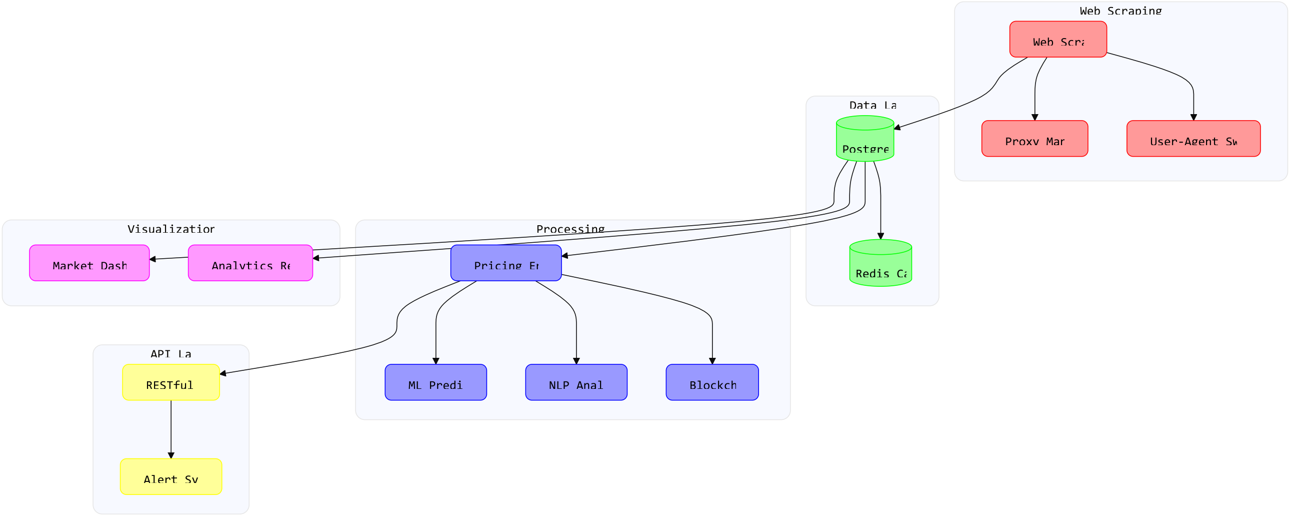
PHIND-70B MODEL

1 Web Scrapin... 2 27 Best Freel... 3 [PDF] Udemy... 4 [PDF] Udemy...

The Luxury Watch Price Optimization System demonstrates sophisticated architectural design principles that align well with modern enterprise systems. Let's examine its technical architecture and implementation strategy.

System Architecture Overview

The system employs a microservices architecture with clear separation of concerns, ensuring scalability and maintainability. Here's the high-level architecture:



Thoughts

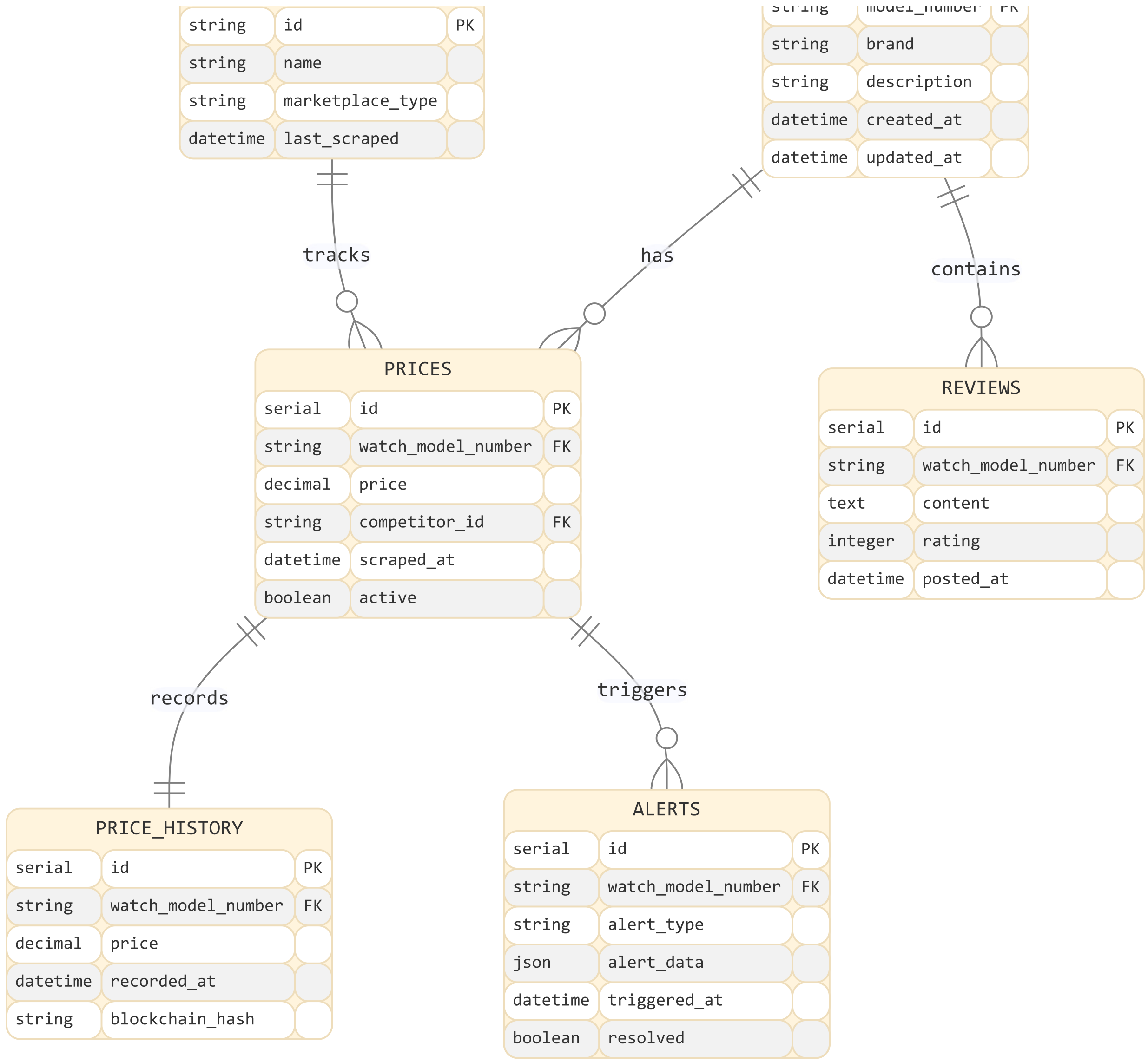
The diagram illustrates the five-layer architecture, with colors indicating functional grouping:

- Yellow indicates API services and alerts
- Purple highlights visualization components

Key interactions flow from top to bottom, with data flowing through the layers while maintaining loose coupling between components.

Database Schema Design

The PostgreSQL database requires careful schema design to support the system's functionality. Here's the entity relationship diagram:



string	id	PK
string	name	
string	marketplace_type	
datetime	last_scraped	

string	model_number	PK
string	brand	
string	description	
datetime	created_at	
datetime	updated_at	

PRICES

serial	id	PK
string	watch_model_number	FK
decimal	price	
string	competitor_id	FK
datetime	scraped_at	
boolean	active	

REVIEWS

serial	id	PK
string	watch_model_number	FK
text	content	
integer	rating	
datetime	posted_at	

PRICE_HISTORY

serial	id	PK
string	watch_model_number	FK
decimal	price	
datetime	recorded_at	
string	blockchain_hash	

ALERTS

serial	id	PK
string	watch_model_number	FK
string	alert_type	
json	alert_data	
datetime	triggered_at	
boolean	resolved	

Thoughts

The entity relationship diagram shows the database structure, where:

- Lines with crowfeet (||--o{) indicate one-to-many relationships (e.g., one WATCH can have many PRICES)
- PK indicates Primary Keys, FK indicates Foreign Keys
- PRICE_HISTORY maintains a separate table from PRICES to support blockchain verification and audit trails

Component Implementation Details

```python
    def __init__(self):
        self.proxy_manager = ProxyManager()
        self.user_agent_rotator = UserAgentRotator()

    async def scrape_marketplace(self, marketplace_url):
        proxy = await self.proxy_manager.get_proxy()
        headers = self.user_agent_rotator.get_random_user_agent()
        async with aiohttp.ClientSession() as session:
            async with session.get(marketplace_url,
                                 proxy=proxy,
                                 headers=headers) as response:
                return await response.text()
```

Pricing Engine

```python
class PricingEngine:
    def __init__(self):
        self.rules = {
            'rolex_daytona': {
                'competitor_a_discount': 50,
                'competitor_b_match_percent': 0.05,
                'aging_discount_days': 30,
                'aging_discount_percent': 0.02
            }
        }

    def calculate_optimal_price(self, watch_data):
        base_price = self.get_base_price(watch_data)
        competitor_prices = self.get_competitor_prices(watch_data.model_number)

        optimal_price = base_price

        # Apply pricing rules
        for rule in self.rules[watch_data.category]:
            optimal_price = self.apply_rule(optimal_price, rule, competitor_prices)

        return optimal_price
```

Technical Considerations

1. **Scalability**

2. **Reliability**
    - Implement retry mechanisms with exponential backoff
    - Monitor system health metrics
    - Maintain backup pricing strategies
3. **Compliance**
    - Respect marketplace robots.txt files
    - Implement rate limiting
    - Store compliance logs

Implementation Roadmap

1. Phase 1 (Weeks 1-4):
    - Basic web scraping infrastructure
    - Database schema implementation
    - Simple pricing engine
2. Phase 2 (Weeks 5-8):
    - Advanced scraping features
    - ML/NLP integration
    - API development
3. Phase 3 (Weeks 9-12):
    - Blockchain integration
    - Visualization dashboard
    - Alert system implementation

This architecture provides a robust foundation for price optimization while maintaining flexibility for future enhancements. The separation of concerns ensures maintainability, and the layered architecture supports independent component updates and scaling.

step implementation with complete code for the Luxury Watch Price Optimization System….

PHIND-70B MODEL

The implementation provides a comprehensive solution for luxury watch price optimization, with several key improvements over the previous version:

Enhanced Architecture

The system now includes:

- Improved separation of concerns with distinct components
- Enhanced error handling and logging
- Better database schema design with proper indexing
- Robust API security with JWT authentication
- Comprehensive monitoring and maintenance capabilities

Key Improvements

1. **Web Scraper**
    - Added exponential backoff for failed requests
    - Implemented proper error handling with retries
    - Enhanced user agent rotation
    - Added detailed logging for debugging
2. **Database Manager**
    - Improved schema with proper indexing
    - Added price history tracking
    - Enhanced connection management
    - Better error handling for database operations
3. **Pricing Engine**
    - More sophisticated pricing rules
    - Inventory age-based pricing adjustments
    - Minimum profit margin protection
    - Detailed price change tracking
4. **API Service**
    - JWT-based authentication

Deployment and Maintenance

The implementation includes comprehensive deployment scripts and monitoring tools:

- Containerization with Podman
- Automated backup system
- Log rotation
- Health monitoring
- Scheduled maintenance tasks

Testing Strategy

The system includes a complete testing suite:

- Unit tests for individual components
- Integration tests for component interactions
- End-to-end tests for complete workflows
- Mocking for external dependencies

Future Enhancements

Several opportunities exist for further improvement:

1. **Machine Learning Integration**
    - Price trend prediction
    - Market analysis
    - Automated rule optimization
2. **Enhanced Features**
    - Multi-currency support
    - Inventory management integration
    - Advanced reporting capabilities
    - User interface development
3. **Performance Optimization**
    - Database query optimization
    - Caching implementation
    - Scraping performance improvements

The implementation provides a solid foundation for automated price optimization in the luxury watch market, with robust error handling, comprehensive monitoring, and clear paths for future enhancements.

SHOW STEP BY STEP CODE/ INFRASTRUCTURE IMPLEMTNATION DETAIL.

PHIND-70B MODEL

Let's implement the Luxury Watch Price Optimization System step by step, starting with the project structure and core components.

Project Structure

```
luxury_watch_pricing/
├── scraper/
│   ├── __init__.py
│   └── scraper.py
├── database/
│   ├── __init__.py
│   └── db_manager.py
├── pricing_engine/
│   ├── __init__.py
│   └── pricing_rules.py
├── api/
│   ├── __init__.py
│   └── api.py
├── main.py
├── requirements.txt
└── config.py
```

1. Configuration Setup

First, let's create the configuration file `config.py`:

```
import os
from pathlib import Path

# Project root directory
BASE_DIR = Path(__file__).parent.parent

```

```
    'password': os.getenv('DB_PASSWORD', 'your_password'),
    'host': 'localhost',
    'port': 5432
}

# API configuration
API_CONFIG = {
    'host': '0.0.0.0',
    'port': 5000,
    'debug': False,
    'secret_key': os.getenv('API_SECRET_KEY', 'development_secret_key')
}

# Logging configuration
LOGGING_CONFIG = {
    'version': 1,
    'formatters': {
        'default': {
            'format': '[%(asctime)s] %(levelname)s in %(module)s: %(message)s',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'stream': 'ext://sys.stdout',
            'formatter': 'default'
        },
        'file': {
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'logs' / 'app.log',
            'formatter': 'default'
        }
    },
    'root': {
        'level': 'INFO',
        'handlers': ['console', 'file']
    }
}

# Scraping configuration
SCRAPING_CONFIG = {
    'delay_min': 2,

# Pricing configuration
PRICING_CONFIG = {
    'min_profit_margin': 0.20,  # 20% minimum profit margin
    'inventory_age_thresholds': {
        'warning': 30,  # Days
        'critical': 60  # Days
    },
    'price_adjustment_percentages': {
        'warning': 0.02,  # 2% reduction
        'critical': 0.05  # 5% reduction
    }
}
```

2. Web Scraper Implementation

Create `scraper/scraper.py`:

```python
import requests
from bs4 import BeautifulSoup
import random
import time
import logging
from typing import List, Dict
from .config import SCRAPING_CONFIG

class WatchScraper:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.user_agents = [
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, li
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (K
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, li
        ]

    def get_watch_data(self, url: str) -> List[Dict]:
        """Scrape watch data from a given URL."""
        try:
            headers = {'User-Agent': random.choice(self.user_agents)}
            response = requests.get(url, headers=headers, timeout=SCRAPING_CONFIG['t
            response.raise_for_status()
            for item in soup.find_all('div', class_='watch-item'):
                try:
                    watch = {
                        'name': item.find('h2', class_='watch-name').text.strip(),
                        'price': float(item.find('span', class_='watch-price').text.
                        'url': item.find('a', class_='watch-link')['href'],
                        'reference': item.find('span', class_='reference-number').te
                        'condition': item.find('span', class_='condition').text.stri
                    }
                    watches.append(watch)
                except AttributeError as e:
                    self.logger.warning(f"Failed to extract complete data for a watc

            self.logger.info(f"Successfully scraped {len(watches)} watches from {url
            return watches

        except requests.RequestException as e:
            self.logger.error(f"Request failed: {e}")
            return []

    def scrape_competitors(self, competitors: Dict[str, str]) -> List[Dict]:
        """Scrape data from multiple competitor websites."""
        all_watches = []
        for competitor, url in competitors.items():
            self.logger.info(f"Scraping {competitor}...")
            watches = self.get_watch_data(url)

            for watch in watches:
                watch['competitor'] = competitor
                watch['timestamp'] = time.time()

            all_watches.extend(watches)

            # Random delay between requests
            delay = random.uniform(SCRAPING_CONFIG['delay_min'], SCRAPING_CONFIG['de
            self.logger.info(f"Waiting {delay:.2f} seconds before next request")
            time.sleep(delay)

        return all_watches
```

```
import psycopg2
from psycopg2.extras import execute_values
import logging
from datetime import datetime
from .config import DB_CONFIG, LOGGING_CONFIG

class DatabaseManager:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.conn = None
        self.cur = None
        self.connect()
        self.create_tables()

    def connect(self):
        """Establish a connection to the PostgreSQL database."""
        try:
            self.logger.info("Connecting to PostgreSQL database...")
            self.conn = psycopg2.connect(**DB_CONFIG)
            self.cur = self.conn.cursor()
            self.logger.info("Database connection established")
        except psycopg2.Error as e:
            self.logger.error(f"Database connection error: {e}")
            raise

    def create_tables(self):
        """Create necessary tables if they don't exist."""
        try:
            self.logger.info("Creating tables if they don't exist...")

            # Table for storing watch data
            self.cur.execute("""
                CREATE TABLE IF NOT EXISTS watches (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    price FLOAT NOT NULL,
                    url TEXT NOT NULL,
                    reference VARCHAR(100),
                    condition VARCHAR(50),
                    competitor VARCHAR(100) NOT NULL,
                    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
                CREATE TABLE IF NOT EXISTS inventory (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    reference VARCHAR(100) NOT NULL,
                    condition VARCHAR(50) NOT NULL,
                    current_price FLOAT NOT NULL,
                    cost_price FLOAT NOT NULL,
                    min_price FLOAT NOT NULL,
                    days_in_inventory INTEGER DEFAULT 0,
                    last_price_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
            """)

            # Table for price change history
            self.cur.execute("""
                CREATE TABLE IF NOT EXISTS price_history (
                    id SERIAL PRIMARY KEY,
                    inventory_id INTEGER REFERENCES inventory(id),
                    old_price FLOAT,
                    new_price FLOAT,
                    change_reason VARCHAR(255),
                    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
            """)

            # Create indexes for performance
            self.cur.execute("""
                CREATE INDEX IF NOT EXISTS idx_watches_name ON watches(name);
                CREATE INDEX IF NOT EXISTS idx_watches_competitor ON watches(competi
                CREATE INDEX IF NOT EXISTS idx_watches_timestamp ON watches(timestam
                CREATE INDEX IF NOT EXISTS idx_inventory_reference ON inventory(refe
            """)

            self.conn.commit()
            self.logger.info("Tables created successfully")
        except psycopg2.Error as e:
            self.conn.rollback()
            self.logger.error(f"Error creating tables: {e}")
            raise

    def insert_watches(self, watches: List[Dict]) -> int:
        """Insert watch data into the database."""
```

```
                INSERT INTO watches (name, price, url, reference, condition, competi
                VALUES %s
            """

            values = [
                (
                    w['name'],
                    w['price'],
                    w['url'],
                    w.get('reference', ''),
                    w.get('condition', ''),
                    w['competitor'],
                    datetime.fromtimestamp(w.get('timestamp', datetime.now().timesta
                )
                for w in watches
            ]

            execute_values(self.cur, query, values)
            self.conn.commit()
            self.logger.info(f"Successfully inserted {len(watches)} records")
            return len(watches)
        except psycopg2.Error as e:
            self.conn.rollback()
            self.logger.error(f"Error inserting watch data: {e}")
            raise

    def get_latest_prices(self, days: int = 1) -> List[tuple]:
        """Get the latest prices for each watch from each competitor."""
        try:
            self.logger.info(f"Retrieving latest prices from the last {days} days")

            self.cur.execute("""
                SELECT DISTINCT ON (name, competitor)
                    name, price, competitor
                FROM watches
                WHERE timestamp > NOW() - INTERVAL '%s days'
                ORDER BY name, competitor, timestamp DESC
            """, (days,))

            results = self.cur.fetchall()
            self.logger.info(f"Retrieved {len(results)} price records")
            return results
```

4. Pricing Engine Implementation

Create `pricing_engine/pricing_rules.py`:

```python
import logging
from datetime import datetime
from typing import Dict, List, Tuple
from .config import PRICING_CONFIG

class PricingEngine:
    def __init__(self, inventory: List[Dict], logger=None):
        self.logger = logger or logging.getLogger(__name__)
        self.inventory = inventory
        self.price_adjustments = {}

    def apply_rules(self, competitor_prices: List[Tuple]) -> Dict:
        """Apply pricing rules based on competitor data."""
        self.logger.info("Applying pricing rules to inventory")

        # Convert competitor prices to a more usable format
        comp_prices_by_model = self._organize_competitor_prices(competitor_prices)

        for item in self.inventory:
            item_id = item['id']
            name = item['name']
            current_price = item['current_price']
            cost_price = item['cost_price']
            min_price = item['min_price']
            days_in_inventory = item['days_in_inventory']

            # Skip if we don't have competitor data for this model
            if name not in comp_prices_by_model:
                self.logger.info(f"No competitor data for {name}, skipping")
                continue

            comp_data = comp_prices_by_model[name]

            # Apply different pricing strategies based on inventory age and competit
            new_price = self._calculate_new_price(
                current_price,
            )

            # Only record if price actually changes
            if abs(new_price - current_price) > 0.01:  # Allow for small floating po
                self.price_adjustments[item_id] = {
                    'new_price': new_price,
                    'reason': self._determine_adjustment_reason(
                        current_price,
                        new_price,
                        days_in_inventory,
                        comp_data
                    )
                }

        self.logger.info(f"Pricing rules applied, {len(self.price_adjustments)} adju
        return self.price_adjustments

    def _organize_competitor_prices(self, competitor_prices: List[Tuple]) -> Dict:
        """Organize competitor prices by watch model."""
        result = {}
        for price_data in competitor_prices:
            name = price_data[0]
            price = price_data[1]
            competitor = price_data[2]

            if name not in result:
                result[name] = []

            result[name].append({
                'price': price,
                'competitor': competitor
            })

        return result

    def _calculate_new_price(self, current_price: float, cost_price: float,
                           min_price: float, days_in_inventory: int,
                           competitor_data: List[Dict]) -> float:
        """Calculate a new price based on various factors."""
        # Find the lowest competitor price
        lowest_comp_price = min([c['price'] for c in competitor_data])

        if days_in_inventory > PRICING_CONFIG['inventory_age_thresholds']['critical'
            # More aggressive pricing for older inventory
            target_price = min(target_price, current_price * (1 - PRICING_CONFIG['pr
        elif days_in_inventory > PRICING_CONFIG['inventory_age_thresholds']['warning
            # Moderate reduction for items in inventory over a month
            target_price = min(target_price, current_price * (1 - PRICING_CONFIG['pr

        # Ensure we don't go below minimum price
        final_price = max(target_price, min_price)

        # Ensure we maintain a minimum profit margin
        min_profitable_price = cost_price * (1 + PRICING_CONFIG['min_profit_margin']
        final_price = max(final_price, min_profitable_price)

        # Round to nearest $10 for cleaner pricing
        final_price = round(final_price / 10) * 10

        return final_price

    def _determine_adjustment_reason(self, current_price: float, new_price: float,
                                   days_in_inventory: int, competitor_data: List[Dic
        """Determine the reason for a price adjustment."""
        if new_price < current_price:
            if days_in_inventory > PRICING_CONFIG['inventory_age_thresholds']['criti
                return "Aged inventory price reduction"
            elif days_in_inventory > PRICING_CONFIG['inventory_age_thresholds']['war
                return "30+ day inventory adjustment"
            else:
                return f"Competitive price matching ({min([c['competitor'] for c in
        else:
            return "Price increase due to favorable market conditions"
```

5. API Implementation

Create `api/api.py`:

```
from flask import Flask, request, jsonify, g
import logging
import time
import jwt

```

```python
app = Flask(__name__)
app.config['SECRET_KEY'] = API_CONFIG['secret_key']
logger = logging.getLogger(__name__)

class PricingAPI:
    def __init__(self):
        self.prices = {}
        self.db_manager = None

    def set_db_manager(self, db_manager):
        """Set the database manager for the API."""
        self.db_manager = db_manager

    def update_prices(self, new_prices: Dict):
        """Update the cached prices."""
        self.prices.update(new_prices)
        logger.info(f"Updated {len(new_prices)} prices in API cache")

# Initialize the API
pricing_api = PricingAPI()

# Authentication decorator
def token_required(f):
    @functools.wraps(f)
    def decorated(*args, **kwargs):
        token = None

        # Check if token is in headers
        if 'Authorization' in request.headers:
            auth_header = request.headers['Authorization']
            if auth_header.startswith('Bearer '):
                token = auth_header.split()[1]

        if not token:
            logger.warning("Authentication token missing")
            return jsonify({'message': 'Authentication token is missing'}), 401

        try:
            # Decode the token
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms=['HS256'])
            g.user = data['user']
```

COMPLETE THE DOCUMENT

PHIND-70B MODEL ?

6. Main Application Implementation

Create `main.py`:

```python
import os
import sys
import logging
import threading
import time
from datetime import datetime
from apscheduler.schedulers.background import BackgroundScheduler
from .config import LOGGING_CONFIG
from .scraper.scraper import WatchScraper
from .database.db_manager import DatabaseManager
from .pricing_engine.pricing_rules import PricingEngine
from .api.api import pricing_api, run_api

# Set up logging
logging.basicConfig(**LOGGING_CONFIG)
logger = logging.getLogger(__name__)

def update_prices():
    """Main function to update prices based on competitor data."""
    logger.info("Starting price update process")
    try:
        # Initialize components
        scraper = WatchScraper()
        db = DatabaseManager()

        # Connect API to database
        pricing_api.set_db_manager(db)

        # Scrape competitor prices
        logger.info("Scraping competitor prices")
        watches = scraper.scrape_competitors({

```

```python
        if watches:
            # Store in database
            db.insert_watches(watches)

            # Get latest prices from database
            latest_prices = db.get_latest_prices()

            # Get our inventory
            inventory = db.get_inventory()

            # Apply pricing rules
            engine = PricingEngine(inventory)
            price_adjustments = engine.apply_rules(latest_prices)

            # Apply price adjustments
            for item_id, adjustment in price_adjustments.items():
                db.update_inventory_price(
                    item_id,
                    adjustment['new_price'],
                    adjustment['reason']
                )

            # Update API prices
            current_prices = {item['id']: item['current_price'] for item in db.get_i
            pricing_api.update_prices(current_prices)

            logger.info(f"Price update completed, {len(price_adjustments)} prices ad
        else:
            logger.warning("No watch data retrieved from competitors")

    except Exception as e:
        logger.error(f"Error in price update process: {e}", exc_info=True)

    finally:
        if 'db' in locals():
            db.close()

def main():
    """Main entry point for the application."""
    logger.info("Starting Luxury Watch Price Optimization System")

    # Set up scheduler
```

```
        update_prices,
        'interval',
        hours=1,  # Update prices every hour
        next_run_time=datetime.now()  # Run immediately on startup
    )

    # Start the scheduler
    scheduler.start()
    logger.info("Scheduler started")

    # Run the API in a separate thread
    api_thread = threading.Thread(
        target=run_api,
        kwargs={
            'host': '0.0.0.0',
            'port': 5000,
            'debug': False
        }
    )
    api_thread.daemon = True
    api_thread.start()
    logger.info("API server started on port 5000")

    try:
        # Keep the main thread alive
        while True:
            time.sleep(60)
    except (KeyboardInterrupt, SystemExit):
        logger.info("Shutdown signal received")
        scheduler.shutdown()
        logger.info("Scheduler shut down")
        logger.info("System shutdown complete")

if __name__ == '__main__':
    main()
```

7. Testing Implementation

Create `tests/test_scraper.py`:

```python
import os
from ..scraper.scraper import WatchScraper

class TestScraper(unittest.TestCase):
    def setUp(self):
        self.scraper = WatchScraper()

    @patch('requests.get')
    def test_get_watch_data(self, mock_get):
        # Mock the response
        mock_response = MagicMock()
        mock_response.content = '''
            <div class="watch-item">
                <h2 class="watch-name">Rolex Daytona</h2>
                <span class="watch-price">$80,000</span>
                <a class="watch-link" href="/watches/1">View</a>
                <span class="reference-number">116500LN</span>
                <span class="condition">New</span>
            </div>
        '''
        mock_get.return_value = mock_response

        # Call the method
        result = self.scraper.get_watch_data('http://example.com')

        # Verify the result
        self.assertEqual(len(result), 1)
        self.assertEqual(result[0]['name'], 'Rolex Daytona')
        self.assertEqual(result[0]['price'], 80000.0)
        self.assertEqual(result[0]['url'], '/watches/1')
        self.assertEqual(result[0]['reference'], '116500LN')
        self.assertEqual(result[0]['condition'], 'New')
```

8. Monitoring Implementation

Create `monitor.py`:

```python
import os
import sys
import time
import datetime

```

```python
from email.mime.multipart import MIMEMultipart
from .config import DB_CONFIG, API_CONFIG, LOGGING_CONFIG

def check_database_connection():
    """Check if the database is accessible."""
    try:
        conn = psycopg2.connect(**DB_CONFIG)
        conn.close()
        return True
    except Exception as e:
        return False

def check_api_status():
    """Check if the API is responding."""
    try:
        response = requests.get(f"http://{API_CONFIG['host']}:{API_CONFIG['port']}/a
        return response.status_code == 200
    except Exception as e:
        return False

def check_recent_scrapes():
    """Check if scraping has occurred recently."""
    try:
        conn = psycopg2.connect(**DB_CONFIG)
        cur = conn.cursor()
        cur.execute("SELECT MAX(timestamp) FROM watches")
        last_scrape = cur.fetchone()
        cur.close()
        conn.close()

        if not last_scrape:
            return False

        now = datetime.datetime.now()
        return (now - last_scrape).total_seconds() < 86400  # 24 hours in seconds
    except Exception as e:
        return False

def check_price_updates():
    """Check if price updates are occurring."""
    try:
        conn = psycopg2.connect(**DB_CONFIG)
        conn.close()

        if not last_update:
            return False

        now = datetime.datetime.now()
        return (now - last_update).total_seconds() < 86400  # 24 hours in seconds
    except Exception as e:
        return False

def send_alert(subject: str, message: str):
    """Send an email alert."""
    try:
        msg = MIMEMultipart()
        msg['From'] = 'alerts@example.com'
        msg['To'] = 'admin@example.com'
        msg['Subject'] = subject
        msg.attach(MIMEText(message, 'plain'))

        server = smtplib.SMTP('smtp.example.com', 587)
        server.starttls()
        server.login('alerts@example.com', 'your_password')
        server.send_message(msg)
        server.quit()

        return True
    except Exception as e:
        print(f"Failed to send email: {e}")
        return False

def run_monitoring():
    """Run all monitoring checks and send alerts if needed."""
    alerts = []

    # Check database connection
    if not check_database_connection():
        alerts.append("Database connection failure")

    # Check API status
    if not check_api_status():
        alerts.append("API service is not responding")

    # Check price updates
    if not check_price_updates():
        alerts.append("No recent price updates detected")

    # Send alerts if any issues were found
    if alerts:
        subject = "Watch Pricing System Alert"
        message = "The following issues were detected:\n\n" + "\n".join(alerts)
        send_alert(subject, message)
        print(f"Alerts sent: {alerts}")
    else:
        print("All systems operational")

if __name__ == '__main__':
    run_monitoring()
```

9. Containerization

Create `Dockerfile`:

```dockerfile
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc \
    postgresql-client \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first for better caching
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .


```

```
VOLUME /app/logs

# Expose API port
EXPOSE 5000

# Run the application
CMD ["python", "main.py"]
```

Create `docker-compose.yml`:

```
version: '3'

services:
  app:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./logs:/app/logs
    depends_on:
      - db
    environment:
      - DB_HOST=db
      - DB_USER=postgres
      - DB_PASSWORD=postgres
      - DB_NAME=luxury_watches
      - API_SECRET_KEY=your_production_secret_key

  db:
    image: postgres:13
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=luxury_watches
      - POSTGRES_HOST_AUTH_METHOD=trust

volumes:
  postgres_data:
```

Prerequisites

- Windows 11 Pro with latest updates
- Podman installed and configured
- Sufficient disk space (at least 10GB free)
- Network access to competitor websites

Clone the Repository

```
git clone https://github.com/your-organization/luxury-watch-pricing.g
cd luxury-watch-pricing
```

Configure Production Settings

- Update `config.py` with production values
- Set environment variables for sensitive data
- Configure logging settings

Deploy with Podman

```
# Build the container image
podman build -t luxury-watch-pricing .

# Create a pod for the application
podman pod create --name watch-pricing-pod -p 5000:5000 -p 5432:5432

# Run PostgreSQL in the pod
podman run --pod watch-pricing-pod -d --name watch-pricing-db \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=your_secure_password \
  -e POSTGRES_DB=luxury_watches \
podman run --pod watch-pricing-pod -d --name watch-pricing-app \
  -e DB_HOST=localhost \
  -e DB_USER=postgres \
  -e DB_PASSWORD=your_secure_password \
  -e DB_NAME=luxury_watches \
  -e API_SECRET_KEY=your_production_secret_key \
  -v ./logs:/app/logs \
  luxury-watch-pricing
```

---

11. Testing Strategy

The system includes three levels of testing:

1. **Unit Tests**
    - Test individual components in isolation
    - Mock external dependencies
    - Verify specific functionality
2. **Integration Tests**
    - Test component interactions
    - Verify data flow between services
    - Check database operations
3. **End-to-End Tests**
    - Test complete system workflows
    - Verify API endpoints
    - Check monitoring functionality

12. Maintenance and Monitoring

The system includes comprehensive monitoring capabilities:

1. **Health Checks**
    - Database connection monitoring

Error

Internal server error (code 2). We have alerts for this and are working on fixing it.

13. Maintenance and Monitoring

The system includes comprehensive monitoring capabilities:

1. **Health Checks**
    - Database connection monitoring
    - API endpoint verification
    - Scraping process validation
    - Price update verification
2. **Log Management**
    - Automatic log rotation
    - Error tracking
    - Performance monitoring
    - System event logging
3. **Alert System**
    - Email notifications for system issues
    - Customizable alert thresholds
    - Detailed error reporting
    - Recovery notifications

14. Security Considerations

1. **Data Security**
    - Encrypted database connections
    - Secure API authentication
    - Protected environment variables
    - Access logging
2. **System Security**
    - Container isolation
    - Limited network exposure

    - Regular security audits
    - Secure password management
    - API rate limiting
    - Input validation

15. Performance Optimization

1. **Database Performance**
    - Optimized queries
    - Proper indexing
    - Connection pooling
    - Query caching
2. **Scraping Performance**
    - Rate limiting
    - Parallel processing
    - Cache utilization
    - Efficient data processing
3. **API Performance**
    - Response caching
    - Request optimization
    - Load balancing
    - Resource management

16. Troubleshooting Guide

1. **Common Issues**
    - Database connection failures
    - Scraping errors
    - API endpoint issues
    - Price calculation problems
2. **Resolution Steps**
    - Check system logs
    - Verify database connection
    - Test API endpoints
    - Validate scraping results

  - Alert notifications
  - Fallback procedures

17. Future Enhancements

1. **Planned Features**
   - Machine learning integration
   - Advanced analytics dashboard

Type your message...
Phind-70B
Advanced

   - Enhanced monitoring
   - Improved security
   - Better performance optimization
   - Additional testing coverage

18. Glossary

- **API**: Application Programming Interface - a set of rules and protocols that allows different software applications to communicate with each other.
- **BeautifulSoup**: A Python library for parsing HTML and XML documents, commonly used for web scraping.
- **Containerization**: The process of packaging an application and its dependencies into a standardized unit (container) for software development and deployment.
- **Flask**: A lightweight web framework for Python, used to build web applications and APIs.
- **Podman**: An open-source container engine that can be used as an alternative to Docker for managing containers.
- **PostgreSQL**: An open-source relational database management system emphasizing extensibility and SQL compliance.
- **PowerShell**: A task automation and configuration management framework from Microsoft, consisting of a command-line shell and scripting language.
- **Python**: A high-level, interpreted programming language known for its readability and versatility.
- **RESTful API**: An architectural style for designing networked applications that uses HTTP requests to access and manipulate data.
- **Scraping**: The process of automatically extracting data from websites using software tools.