**Gas Price Data Sources & Futures-Based Forecasting Guide**

**Recommended Data Sources for Gas Price Scraping**

**Official Government Sources (Highest Quality)**

**U.S. Energy Information Administration (EIA)**

* **URL**: https://www.eia.gov/opendata/
* **Coverage**: Comprehensive U.S. gasoline price data by region, state, and city
* **API Available**: Yes (free with registration)
* **Data Types**: Weekly retail prices, refinery prices, historical data
* **Update Frequency**: Weekly
* **Pros**: Most authoritative source, free, excellent historical data
* **Compliance**: Government data, fully public

**Commercial APIs (Paid but Comprehensive)**

**Xavvy Fuel Data**

* **Coverage**: 131,000+ gas stations in USA, 100,000+ in Europe
* **Data Types**: Regular, midgrade, premium, diesel prices by octane level
* **Update Frequency**: Multiple daily updates
* **Features**: Cash/credit price differentiation, real-time API
* **Best For**: Production applications requiring real-time data

**Barchart OnDemand - getFuelPrices API**

* **URL**: https://www.barchart.com/ondemand/api/getFuelPrices
* **Coverage**: Geographic coordinates, zip code radius searches
* **Features**: Filter by product names, county-level data
* **Best For**: Location-based searches with radius filtering

**Commodities API**

* **URL**: https://commodities-api.com/
* **Coverage**: Real-time and historical gasoline futures prices
* **Features**: 170+ currencies, Bitcoin support, time-series data
* **Update Frequency**: Up to every 60 seconds
* **Best For**: Futures data and price forecasting

**Public Websites (Free but Requires Scraping)**

**GasBuddy**

* **URL**: https://www.gasbuddy.com/
* **Coverage**: Crowd-sourced prices from millions of users
* **Data**: Station-level prices, user ratings, amenities
* **Compliance**: Check robots.txt carefully
* **Pros**: Most current retail prices, extensive coverage
* **Cons**: Inconsistent data quality, anti-scraping measures

**AAA Gas Prices**

* **URL**: https://gasprices.aaa.com/
* **Coverage**: State and metropolitan area averages
* **Update Frequency**: Daily
* **Pros**: Reliable averages, good for trend analysis
* **Cons**: No station-level detail

**Trading Economics**

* **URLs**:
  + Gasoline: https://tradingeconomics.com/commodity/gasoline
  + Natural Gas: https://tradingeconomics.com/commodity/natural-gas
* **Features**: Live prices, historical charts, forecasts
* **Best For**: Market analysis and futures data

**Specialized Data Providers**

**Tankerkönig (Germany-based)**

* **Coverage**: European fuel data
* **Best For**: International coverage

**CollectAPI Gas Prices**

* **URL**: https://collectapi.com/api/gasPrice/gas-prices-api
* **Features**: API marketplace with multiple gas price endpoints

**Compliance Considerations**

* **Always check robots.txt** for each website before scraping
* **Respect rate limits** - implement delays between requests (2-5 seconds minimum)
* **Use official APIs when available** - they're more reliable and legally compliant
* **Consider data licensing** for commercial use
* **Implement user-agent rotation** and respectful crawling practices

**Gas Futures Price Forecasting Methodology**

**Step 1: Data Collection and Preparation**

**Primary Data Sources**

- NYMEX RBOB Gasoline Futures (Chicago Mercantile Exchange)

- Brent and WTI Crude Oil Futures

- EIA Weekly Petroleum Status Report

- CFTC Commitments of Traders Reports

- Economic indicators (GDP, CPI, unemployment)

**Data Preparation Tasks**

* Collect daily closing prices for multiple contract months
* Calculate contango/backwardation relationships
* Gather volume and open interest data
* Compile inventory levels (crude oil, gasoline, distillate)
* Economic calendar events and seasonal patterns

**Step 2: Fundamental Analysis Framework**

**Supply-Side Factors**

* **Refinery Utilization Rates**: Higher utilization typically lowers prices
* **Refinery Maintenance Schedules**: Fall maintenance season creates supply tightness
* **Import/Export Levels**: U.S. product exports affect domestic supply
* **Inventory Levels**: EIA weekly inventory reports (gasoline and crude)
* **Production Capacity**: Refinery additions or closures

**Demand-Side Factors**

* **Driving Season Patterns**: Peak demand May-September
* **Economic Activity**: GDP growth correlates with fuel demand
* **Travel Patterns**: Holiday weekends, summer vacation season
* **Alternative Fuel Adoption**: EV penetration impact
* **Weather Patterns**: Hurricanes, extreme temperatures

**External Factors**

* **Crude Oil Prices**: Primary input cost (60-70% of gasoline price)
* **Geopolitical Events**: Middle East tensions, sanctions
* **Currency Fluctuations**: Dollar strength affects commodity prices
* **Regulatory Changes**: Environmental regulations, tax policies

**Step 3: Technical Analysis and Futures Curve Analysis**

**Futures Curve Interpretation**

# Example analysis framework

def analyze\_futures\_curve():

# Calculate curve slopes

front\_month\_price = get\_front\_month\_price()

back\_month\_price = get\_12\_month\_price()

# Contango vs Backwardation

curve\_slope = (back\_month\_price - front\_month\_price) / front\_month\_price

if curve\_slope > 0.02: # 2%+ contango

return "Oversupplied market - prices may decline"

elif curve\_slope < -0.02: # 2%+ backwardation

return "Tight market - prices may rise"

else:

return "Balanced market"

**Key Technical Indicators**

* **Moving Averages**: 20, 50, 200-day trends
* **RSI and MACD**: Momentum indicators
* **Bollinger Bands**: Volatility measurements
* **Support/Resistance Levels**: Historical price levels

**Step 4: Econometric Modeling**

**Time Series Models**

* **ARIMA Models**: Autoregressive Integrated Moving Average
* **VAR Models**: Vector Autoregression with multiple variables
* **Error Correction Models**: Long-term equilibrium relationships
* **Regime-Switching Models**: Account for structural breaks

**Multivariate Regression Framework**

# Example model structure

gasoline\_price = β0 + β1\*crude\_oil\_price + β2\*refinery\_utilization +

β3\*gasoline\_inventory + β4\*economic\_activity +

β5\*seasonal\_dummy + β6\*geopolitical\_risk + ε

**Machine Learning Approaches**

* **Random Forest**: Capture non-linear relationships
* **LSTM Networks**: Sequential pattern recognition
* **Ensemble Methods**: Combine multiple model predictions

**Step 5: Scenario Analysis and Risk Assessment**

**Base Case Scenario (60% probability)**

* Normal refinery operations
* Stable geopolitical environment
* Moderate economic growth
* Seasonal demand patterns

**Upside Scenarios (20% probability)**

* Geopolitical supply disruptions
* Unexpected refinery outages
* Stronger than expected economic growth
* Earlier/longer driving season

**Downside Scenarios (20% probability)**

* Economic recession reducing demand
* Significant increase in alternative fuel adoption
* New refinery capacity additions
* Warm winter reducing heating demand

**Step 6: 12-Month Projection Framework**

**Monthly Forecast Structure**

Month 1-3: Weight technical analysis heavily (70%)

Month 4-6: Balance technical and fundamental analysis (50/50)

Month 7-12: Emphasize fundamental analysis (70%)

**Seasonal Adjustment Factors**

* **Q1**: Typically lower demand, refinery maintenance
* **Q2**: Demand pickup, driving season preparation
* **Q3**: Peak driving season, highest prices
* **Q4**: Demand decline, fall maintenance season

**Price Target Methodology**

def calculate\_price\_targets():

base\_forecast = fundamental\_model\_prediction()

seasonal\_adjustment = calculate\_seasonal\_factors()

risk\_premium = assess\_geopolitical\_risk()

bull\_case = base\_forecast \* 1.15 + risk\_premium

base\_case = base\_forecast + seasonal\_adjustment

bear\_case = base\_forecast \* 0.85 - risk\_premium

return bull\_case, base\_case, bear\_case

**Step 7: Model Validation and Backtesting**

**Validation Techniques**

* **Out-of-sample testing**: Reserve 20% of data for validation
* **Rolling forecasts**: Test predictions against actual outcomes
* **Error metrics**: RMSE, MAPE, directional accuracy
* **Stress testing**: Model performance during volatile periods

**Continuous Model Improvement**

* Monthly model recalibration
* Parameter updates based on new data
* Incorporation of emerging factors (EV adoption, policy changes)
* Benchmark against professional forecasts (EIA, IEA projections)

**Current Market Context (July 2025)**

Based on recent data:

* **Current gasoline futures**: ~$2.19/gallon
* **12-month outlook**: Mixed signals with geopolitical risks
* **Key factors to monitor**: OPEC+ production decisions, U.S. refinery capacity, economic indicators

**Implementation Tools**

* **Python libraries**: pandas, numpy, sklearn, statsmodels
* **Data sources**: Bloomberg API, Alpha Vantage, EIA API
* **Visualization**: matplotlib, plotly for futures curve analysis

This framework provides a systematic approach to forecasting gasoline prices using futures markets while accounting for the complex interplay of supply, demand, and external factors that drive energy prices.

**Complete Gas Price Calculator Implementation Guide**

**Overview**

This guide details the step-by-step process to provide users with personalized gas prices for their vehicle based on zip code and octane requirements.

**Step 1: User Input Collection and Validation**

**Required User Inputs**

def collect\_user\_input():

"""Collect and validate user requirements"""

user\_data = {

'zip\_code': input("Enter your ZIP code: "),

'octane\_level': input("Select octane level (87/89/91/93+): "),

'search\_radius': input("Search radius in miles (default 10): ") or 10,

'max\_results': input("Max number of stations (default 20): ") or 20,

'price\_type': input("Cash or Credit prices (default both): ") or 'both'

}

return validate\_inputs(user\_data)

def validate\_inputs(data):

"""Validate user inputs"""

# ZIP code validation

if not re.match(r'^\d{5}(-\d{4})?$', data['zip\_code']):

raise ValueError("Invalid ZIP code format")

# Octane level validation

valid\_octanes = ['87', '89', '91', '93', '93+']

if data['octane\_level'] not in valid\_octanes:

raise ValueError("Invalid octane level")

# Radius validation

try:

data['search\_radius'] = float(data['search\_radius'])

if data['search\_radius'] <= 0 or data['search\_radius'] > 50:

raise ValueError("Radius must be between 1-50 miles")

except ValueError:

raise ValueError("Invalid radius value")

return data

**Step 2: Geographic Processing and Location Services**

**ZIP Code to Coordinates Conversion**

import requests

from geopy.distance import geodesic

import json

def zip\_to\_coordinates(zip\_code):

"""Convert ZIP code to latitude/longitude"""

# Option 1: Use Census Geocoding API (Free)

census\_url = f"https://geocoding.geo.census.gov/geocoder/locations/onelineaddress"

params = {

'address': zip\_code,

'benchmark': 'Public\_AR\_Current',

'format': 'json'

}

response = requests.get(census\_url, params=params)

if response.status\_code == 200:

data = response.json()

if data['result']['addressMatches']:

coords = data['result']['addressMatches'][0]['coordinates']

return {

'latitude': coords['y'],

'longitude': coords['x'],

'city': data['result']['addressMatches'][0]['addressComponents']['city'],

'state': data['result']['addressMatches'][0]['addressComponents']['state']

}

# Option 2: Fallback to ZIP code database lookup

return lookup\_zip\_database(zip\_code)

def calculate\_distance(lat1, lon1, lat2, lon2):

"""Calculate distance between two points in miles"""

point1 = (lat1, lon1)

point2 = (lat2, lon2)

return geodesic(point1, point2).miles

**Step 3: Data Source Integration and Price Collection**

**Multi-Source Data Collection**

class GasPriceAggregator:

def \_\_init\_\_(self):

self.sources = {

'eia': EIADataSource(),

'xavvy': XavvyDataSource(),

'gasbuddy': GasBuddyDataSource(),

'barchart': BarchartDataSource()

}

def get\_prices\_for\_location(self, user\_location, user\_preferences):

"""Aggregate prices from multiple sources"""

all\_stations = []

for source\_name, source in self.sources.items():

try:

stations = source.get\_stations\_near\_location(

latitude=user\_location['latitude'],

longitude=user\_location['longitude'],

radius=user\_preferences['search\_radius'],

octane=user\_preferences['octane\_level']

)

# Add source identifier

for station in stations:

station['data\_source'] = source\_name

station['timestamp'] = datetime.now()

all\_stations.extend(stations)

except Exception as e:

print(f"Error fetching from {source\_name}: {e}")

continue

return self.deduplicate\_stations(all\_stations)

**Individual Data Source Implementations**

**EIA Data Source**

class EIADataSource:

def \_\_init\_\_(self, api\_key):

self.api\_key = api\_key

self.base\_url = "https://api.eia.gov/v2"

def get\_regional\_averages(self, state, city=None):

"""Get EIA regional average prices"""

endpoint = f"{self.base\_url}/petroleum/pri/gnd/data/"

params = {

'api\_key': self.api\_key,

'frequency': 'weekly',

'data[0]': 'value',

'facets[product][]': 'EPMRU', # Regular unleaded

'facets[area][]': f'R{self.get\_region\_code(state)}',

'sort[0][column]': 'period',

'sort[0][direction]': 'desc',

'offset': 0,

'length': 1

}

response = requests.get(endpoint, params=params)

if response.status\_code == 200:

data = response.json()

return data['response']['data'][0]['value']

return None

**Xavvy API Integration**

class XavvyDataSource:

def \_\_init\_\_(self, api\_key):

self.api\_key = api\_key

self.base\_url = "https://api.xavvy.com/v1"

def get\_stations\_near\_location(self, latitude, longitude, radius, octane):

"""Get stations from Xavvy API"""

endpoint = f"{self.base\_url}/stations/search"

headers = {'Authorization': f'Bearer {self.api\_key}'}

params = {

'lat': latitude,

'lng': longitude,

'radius': radius,

'fuel\_type': self.map\_octane\_to\_fuel\_type(octane),

'limit': 100

}

response = requests.get(endpoint, headers=headers, params=params)

if response.status\_code == 200:

return self.parse\_xavvy\_response(response.json())

return []

def map\_octane\_to\_fuel\_type(self, octane):

"""Map octane levels to Xavvy fuel types"""

mapping = {

'87': 'regular',

'89': 'midgrade',

'91': 'premium',

'93': 'premium',

'93+': 'premium'

}

return mapping.get(octane, 'regular')

**Web Scraping Implementation (GasBuddy)**

class GasBuddyDataSource:

def \_\_init\_\_(self):

self.session = requests.Session()

self.session.headers.update({

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'

})

self.rate\_limiter = RateLimiter(max\_calls=10, period=60) # 10 calls per minute

def get\_stations\_near\_location(self, latitude, longitude, radius, octane):

"""Scrape GasBuddy for station data"""

self.rate\_limiter.wait()

# Search URL with coordinates

search\_url = f"https://www.gasbuddy.com/home?search={latitude},{longitude}&fuel=1"

response = self.session.get(search\_url)

if response.status\_code != 200:

return []

soup = BeautifulSoup(response.content, 'html.parser')

return self.parse\_gasbuddy\_stations(soup, octane)

def parse\_gasbuddy\_stations(self, soup, octane):

"""Parse GasBuddy HTML for station data"""

stations = []

station\_elements = soup.find\_all('div', class\_='StationDisplay-module\_\_container')

for element in station\_elements:

try:

station = self.extract\_station\_data(element, octane)

if station:

stations.append(station)

except Exception as e:

continue

return stations

**Step 4: Data Processing and Standardization**

**Station Data Standardization**

class StationDataProcessor:

def standardize\_station\_data(self, raw\_stations):

"""Standardize data from different sources"""

standardized = []

for station in raw\_stations:

std\_station = {

'id': self.generate\_station\_id(station),

'name': self.clean\_station\_name(station.get('name', '')),

'brand': self.standardize\_brand\_name(station.get('brand', '')),

'address': {

'street': station.get('street', ''),

'city': station.get('city', ''),

'state': station.get('state', ''),

'zip\_code': station.get('zip\_code', '')

},

'coordinates': {

'latitude': float(station.get('latitude', 0)),

'longitude': float(station.get('longitude', 0))

},

'prices': self.standardize\_prices(station.get('prices', {})),

'amenities': station.get('amenities', []),

'data\_source': station.get('data\_source', 'unknown'),

'last\_updated': station.get('timestamp', datetime.now())

}

standardized.append(std\_station)

return standardized

def standardize\_prices(self, prices):

"""Standardize price data format"""

std\_prices = {}

octane\_mapping = {

'regular': '87',

'midgrade': '89',

'mid': '89',

'plus': '89',

'premium': '91',

'super': '93',

'ultra': '93'

}

for fuel\_type, price\_data in prices.items():

octane = octane\_mapping.get(fuel\_type.lower(), fuel\_type)

std\_prices[octane] = {

'cash\_price': self.clean\_price(price\_data.get('cash', price\_data.get('price'))),

'credit\_price': self.clean\_price(price\_data.get('credit', price\_data.get('price')))

}

return std\_prices

**Duplicate Station Detection and Merging**

def deduplicate\_stations(self, stations):

"""Remove duplicate stations and merge price data"""

station\_groups = {}

for station in stations:

# Create location-based key for grouping

lat\_rounded = round(station['coordinates']['latitude'], 4)

lon\_rounded = round(station['coordinates']['longitude'], 4)

name\_cleaned = re.sub(r'[^a-zA-Z0-9]', '', station['name'].lower())

key = f"{lat\_rounded}\_{lon\_rounded}\_{name\_cleaned}"

if key not in station\_groups:

station\_groups[key] = []

station\_groups[key].append(station)

# Merge duplicate stations

deduplicated = []

for group in station\_groups.values():

merged\_station = self.merge\_station\_data(group)

deduplicated.append(merged\_station)

return deduplicated

def merge\_station\_data(self, station\_group):

"""Merge data from multiple sources for same station"""

if len(station\_group) == 1:

return station\_group[0]

# Use most recent data as base

base\_station = max(station\_group, key=lambda x: x['last\_updated'])

# Merge prices from all sources

merged\_prices = {}

for station in station\_group:

for octane, price\_data in station.get('prices', {}).items():

if octane not in merged\_prices:

merged\_prices[octane] = price\_data

else:

# Use most recent non-null price

if price\_data.get('cash\_price') and not merged\_prices[octane].get('cash\_price'):

merged\_prices[octane]['cash\_price'] = price\_data['cash\_price']

if price\_data.get('credit\_price') and not merged\_prices[octane].get('credit\_price'):

merged\_prices[octane]['credit\_price'] = price\_data['credit\_price']

base\_station['prices'] = merged\_prices

base\_station['data\_sources'] = [s['data\_source'] for s in station\_group]

return base\_station

**Step 5: Price Calculation and Filtering**

**Distance and Price Filtering**

def filter\_and\_rank\_stations(self, stations, user\_location, user\_preferences):

"""Filter stations by distance and rank by price/convenience"""

filtered\_stations = []

for station in stations:

# Calculate distance

distance = calculate\_distance(

user\_location['latitude'],

user\_location['longitude'],

station['coordinates']['latitude'],

station['coordinates']['longitude']

)

# Filter by radius

if distance <= user\_preferences['search\_radius']:

station['distance\_miles'] = round(distance, 2)

# Check if requested octane is available

octane = user\_preferences['octane\_level']

if octane in station.get('prices', {}):

station['requested\_price'] = self.get\_best\_price(

station['prices'][octane],

user\_preferences['price\_type']

)

filtered\_stations.append(station)

# Sort by price (ascending) then by distance

return sorted(filtered\_stations, key=lambda x: (

x.get('requested\_price', float('inf')),

x.get('distance\_miles', float('inf'))

))

def get\_best\_price(self, price\_data, price\_preference):

"""Get the appropriate price based on user preference"""

cash\_price = price\_data.get('cash\_price')

credit\_price = price\_data.get('credit\_price')

if price\_preference == 'cash' and cash\_price:

return cash\_price

elif price\_preference == 'credit' and credit\_price:

return credit\_price

elif price\_preference == 'both':

# Return the lower price if both available

if cash\_price and credit\_price:

return min(cash\_price, credit\_price)

return cash\_price or credit\_price

# Fallback to any available price

return cash\_price or credit\_price

**Price Analysis and Statistics**

def calculate\_price\_statistics(self, stations):

"""Calculate price statistics for the area"""

prices = [s.get('requested\_price') for s in stations if s.get('requested\_price')]

if not prices:

return {}

return {

'lowest\_price': min(prices),

'highest\_price': max(prices),

'average\_price': round(sum(prices) / len(prices), 3),

'median\_price': round(sorted(prices)[len(prices) // 2], 3),

'price\_range': round(max(prices) - min(prices), 3),

'stations\_found': len(prices)

}

**Step 6: Results Formatting and Presentation**

**User-Friendly Output Generation**

def format\_results\_for\_user(self, stations, statistics, user\_preferences):

"""Format results for user display"""

result = {

'search\_summary': {

'zip\_code': user\_preferences['zip\_code'],

'octane\_level': user\_preferences['octane\_level'],

'search\_radius': user\_preferences['search\_radius'],

'stations\_found': len(stations),

'search\_timestamp': datetime.now().isoformat()

},

'price\_summary': statistics,

'recommendations': {

'cheapest\_station': self.format\_station\_summary(stations[0]) if stations else None,

'closest\_station': self.get\_closest\_station(stations),

'best\_value': self.calculate\_best\_value(stations)

},

'all\_stations': [self.format\_station\_detail(s) for s in stations[:20]]

}

return result

def format\_station\_detail(self, station):

"""Format individual station for display"""

return {

'name': station['name'],

'brand': station['brand'],

'address': f"{station['address']['street']}, {station['address']['city']}, {station['address']['state']}",

'distance': f"{station['distance\_miles']} miles",

'price': f"${station['requested\_price']:.3f}/gal",

'cash\_price': f"${station['prices'][user\_octane]['cash\_price']:.3f}" if station['prices'][user\_octane].get('cash\_price') else 'N/A',

'credit\_price': f"${station['prices'][user\_octane]['credit\_price']:.3f}" if station['prices'][user\_octane].get('credit\_price') else 'N/A',

'amenities': station.get('amenities', []),

'last\_updated': station['last\_updated'].strftime('%Y-%m-%d %H:%M')

}

**Console Output Display**

def display\_results\_console(self, results):

"""Display results in console format"""

print("\n" + "="\*60)

print(f"GAS PRICE SEARCH RESULTS")

print("="\*60)

summary = results['search\_summary']

print(f"Search Area: {summary['zip\_code']}")

print(f"Octane Level: {summary['octane\_level']}")

print(f"Search Radius: {summary['search\_radius']} miles")

print(f"Stations Found: {summary['stations\_found']}")

if results['price\_summary']:

stats = results['price\_summary']

print(f"\nPRICE OVERVIEW:")

print(f"Lowest Price: ${stats['lowest\_price']:.3f}")

print(f"Average Price: ${stats['average\_price']:.3f}")

print(f"Highest Price: ${stats['highest\_price']:.3f}")

print(f"Price Range: ${stats['price\_range']:.3f}")

print(f"\nTOP RECOMMENDATIONS:")

if results['recommendations']['cheapest\_station']:

cheap = results['recommendations']['cheapest\_station']

print(f"💰 CHEAPEST: {cheap['name']} - ${cheap['price']:.3f} ({cheap['distance']} miles)")

print(f"\nALL STATIONS:")

print("-"\*60)

for i, station in enumerate(results['all\_stations'], 1):

print(f"{i:2}. {station['name']}")

print(f" Price: {station['price']} | Distance: {station['distance']}")

print(f" {station['address']}")

print()

**Step 7: Complete Integration and Main Function**

**Main Application Flow**

class GasPriceCalculator:

def \_\_init\_\_(self):

self.aggregator = GasPriceAggregator()

self.processor = StationDataProcessor()

def get\_gas\_prices\_for\_user(self, zip\_code, octane\_level,

search\_radius=10, max\_results=20,

price\_type='both'):

"""Main function to get gas prices for user"""

# Step 1: Validate inputs

user\_preferences = {

'zip\_code': zip\_code,

'octane\_level': octane\_level,

'search\_radius': search\_radius,

'max\_results': max\_results,

'price\_type': price\_type

}

# Step 2: Get location coordinates

user\_location = zip\_to\_coordinates(zip\_code)

if not user\_location:

raise ValueError(f"Could not find location for ZIP code: {zip\_code}")

# Step 3: Collect price data from multiple sources

raw\_stations = self.aggregator.get\_prices\_for\_location(

user\_location, user\_preferences

)

if not raw\_stations:

return {"error": "No gas stations found in the specified area"}

# Step 4: Process and standardize data

standardized\_stations = self.processor.standardize\_station\_data(raw\_stations)

deduplicated\_stations = self.processor.deduplicate\_stations(standardized\_stations)

# Step 5: Filter and rank stations

filtered\_stations = self.filter\_and\_rank\_stations(

deduplicated\_stations, user\_location, user\_preferences

)

# Step 6: Calculate statistics

statistics = self.calculate\_price\_statistics(filtered\_stations)

# Step 7: Format results

results = self.format\_results\_for\_user(

filtered\_stations[:max\_results], statistics, user\_preferences

)

return results

# Usage Example

def main():

calculator = GasPriceCalculator()

try:

results = calculator.get\_gas\_prices\_for\_user(

zip\_code="90210",

octane\_level="91",

search\_radius=15,

max\_results=10,

price\_type="both"

)

calculator.display\_results\_console(results)

# Return structured data for API/web interface

return results

except Exception as e:

print(f"Error: {e}")

return {"error": str(e)}

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Step 8: Performance Optimization and Caching**

**Caching Strategy**

import redis

import json

from datetime import timedelta

class CacheManager:

def \_\_init\_\_(self):

self.redis\_client = redis.Redis(host='localhost', port=6379, db=0)

self.cache\_ttl = {

'station\_data': 1800, # 30 minutes

'zip\_coordinates': 86400, # 24 hours

'regional\_averages': 3600 # 1 hour

}

def get\_cached\_stations(self, cache\_key):

"""Retrieve cached station data"""

cached\_data = self.redis\_client.get(cache\_key)

if cached\_data:

return json.loads(cached\_data)

return None

def cache\_stations(self, cache\_key, stations):

"""Cache station data"""

self.redis\_client.setex(

cache\_key,

self.cache\_ttl['station\_data'],

json.dumps(stations, default=str)

)

This comprehensive implementation provides users with accurate, real-time gas prices tailored to their specific vehicle requirements and location, aggregating data from multiple sources while handling edge cases and optimizing for performance.

**Monthly Gas Spending Forecast Implementation Guide**

**Overview**

This guide outlines how to forecast a user's monthly gas spending over 12 months by combining vehicle consumption data, driving patterns, and price forecasting models.

**Step 1: User Data Collection and Vehicle Profile**

**Required User Inputs**

def collect\_user\_profile():

"""Collect comprehensive user data for spending forecasts"""

user\_profile = {

# Vehicle Information

'vehicle': {

'year': input("Vehicle year: "),

'make': input("Vehicle make: "),

'model': input("Vehicle model: "),

'engine\_size': input("Engine size (L) or 'auto-detect': "),

'fuel\_type': input("Required octane (87/89/91/93+): "),

'mpg\_city': input("City MPG (or 'auto-detect'): "),

'mpg\_highway': input("Highway MPG (or 'auto-detect'): "),

'tank\_capacity': input("Tank capacity in gallons (optional): ")

},

# Driving Patterns

'driving\_patterns': {

'annual\_miles': input("Estimated annual miles: "),

'city\_highway\_split': input("City driving % (0-100): "),

'commute\_miles\_weekday': input("Daily commute miles: "),

'weekend\_driving': input("Weekend miles per week: "),

'seasonal\_variation': input("Do you drive more in summer? (y/n): ")

},

# Financial Preferences

'preferences': {

'budget\_conscious': input("Price-sensitive? (y/n): "),

'station\_loyalty': input("Preferred station brand (or 'any'): "),

'payment\_method': input("Preferred payment (cash/credit/both): "),

'max\_detour': input("Max detour for cheaper gas (miles): ")

},

# Location Data

'location': {

'home\_zip': input("Home ZIP code: "),

'work\_zip': input("Work ZIP code (if different): "),

'frequent\_areas': input("Other frequent ZIP codes (comma-separated): ")

}

}

return validate\_and\_enhance\_profile(user\_profile)

def validate\_and\_enhance\_profile(profile):

"""Validate inputs and auto-detect missing vehicle data"""

# Auto-detect vehicle specifications

if profile['vehicle']['mpg\_city'] == 'auto-detect':

vehicle\_specs = get\_vehicle\_specifications(

profile['vehicle']['year'],

profile['vehicle']['make'],

profile['vehicle']['model']

)

profile['vehicle'].update(vehicle\_specs)

# Calculate derived metrics

city\_pct = float(profile['driving\_patterns']['city\_highway\_split']) / 100

highway\_pct = 1 - city\_pct

profile['calculated'] = {

'combined\_mpg': (

float(profile['vehicle']['mpg\_city']) \* city\_pct +

float(profile['vehicle']['mpg\_highway']) \* highway\_pct

),

'monthly\_miles': float(profile['driving\_patterns']['annual\_miles']) / 12,

'gallons\_per\_month': None # Calculated below

}

profile['calculated']['gallons\_per\_month'] = (

profile['calculated']['monthly\_miles'] / profile['calculated']['combined\_mpg']

)

return profile

**Vehicle Specification Database Integration**

class VehicleSpecsAPI:

"""Interface to get vehicle fuel efficiency data"""

def \_\_init\_\_(self):

# EPA Fuel Economy API or NHTSA database

self.epa\_api\_key = "your\_epa\_api\_key"

self.base\_url = "https://www.fueleconomy.gov/feg/ws/rest/"

def get\_vehicle\_specifications(self, year, make, model):

"""Fetch official EPA ratings"""

# Search for vehicle

search\_url = f"{self.base\_url}vehicle/menu/options"

params = {

'year': year,

'make': make,

'model': model

}

response = requests.get(search\_url, params=params)

if response.status\_code == 200:

vehicles = self.parse\_epa\_response(response.text)

if vehicles:

return vehicles[0] # Return first match

# Fallback to manual database

return self.lookup\_vehicle\_database(year, make, model)

def lookup\_vehicle\_database(self, year, make, model):

"""Fallback vehicle database lookup"""

vehicle\_db = {

# Example entries - would be comprehensive database

('2023', 'toyota', 'camry'): {

'mpg\_city': 28,

'mpg\_highway': 39,

'tank\_capacity': 14.3,

'fuel\_type': '87'

},

('2023', 'bmw', '330i'): {

'mpg\_city': 26,

'mpg\_highway': 36,

'tank\_capacity': 15.6,

'fuel\_type': '91'

}

}

key = (str(year), make.lower(), model.lower())

return vehicle\_db.get(key, {

'mpg\_city': 25, # Conservative defaults

'mpg\_highway': 35,

'tank\_capacity': 15,

'fuel\_type': '87'

})

**Step 2: Driving Pattern Analysis and Seasonal Modeling**

**Monthly Driving Pattern Calculation**

class DrivingPatternAnalyzer:

def \_\_init\_\_(self, user\_profile):

self.profile = user\_profile

self.seasonal\_factors = self.get\_seasonal\_factors()

def calculate\_monthly\_consumption(self):

"""Calculate fuel consumption for each month"""

base\_monthly\_gallons = self.profile['calculated']['gallons\_per\_month']

monthly\_consumption = {}

for month in range(1, 13):

# Apply seasonal variations

seasonal\_multiplier = self.seasonal\_factors[month]

# Account for vacation patterns, weather, etc.

consumption = base\_monthly\_gallons \* seasonal\_multiplier

monthly\_consumption[month] = {

'estimated\_miles': self.profile['calculated']['monthly\_miles'] \* seasonal\_multiplier,

'estimated\_gallons': consumption,

'driving\_pattern': self.get\_month\_driving\_pattern(month)

}

return monthly\_consumption

def get\_seasonal\_factors(self):

"""Define seasonal driving multipliers"""

# Based on DOT transportation statistics

base\_factors = {

1: 0.85, # January - reduced winter driving

2: 0.88, # February

3: 0.95, # March - spring pickup

4: 1.00, # April - baseline

5: 1.05, # May - increased travel

6: 1.15, # June - vacation season

7: 1.20, # July - peak travel

8: 1.18, # August - continued summer travel

9: 1.02, # September - back to normal

10: 1.00, # October - baseline

11: 1.08, # November - holiday travel

12: 0.95 # December - weather impact

}

# Adjust based on user preferences

if self.profile['driving\_patterns']['seasonal\_variation'].lower() == 'n':

# Flatten seasonal variation for non-seasonal drivers

avg\_factor = sum(base\_factors.values()) / len(base\_factors)

return {month: avg\_factor for month in range(1, 13)}

return base\_factors

def get\_month\_driving\_pattern(self, month):

"""Determine driving patterns for specific month"""

patterns = {

'winter\_months': [12, 1, 2],

'spring\_months': [3, 4, 5],

'summer\_months': [6, 7, 8],

'fall\_months': [9, 10, 11]

}

if month in patterns['winter\_months']:

return {

'city\_pct': 0.7, # More city driving in winter

'highway\_pct': 0.3,

'efficiency\_factor': 0.9 # Cold weather impact

}

elif month in patterns['summer\_months']:

return {

'city\_pct': 0.5, # More highway travel

'highway\_pct': 0.5,

'efficiency\_factor': 1.0 # Optimal conditions

}

else:

return {

'city\_pct': 0.6,

'highway\_pct': 0.4,

'efficiency\_factor': 0.95

}

**Step 3: Price Forecasting Integration**

**Monthly Price Prediction Model**

class MonthlyGasPriceForecast:

def \_\_init\_\_(self, user\_location):

self.location = user\_location

self.futures\_analyzer = GasolineFuturesAnalyzer()

self.regional\_adjustments = self.get\_regional\_factors()

def generate\_12\_month\_price\_forecast(self, octane\_level):

"""Generate monthly price forecasts for user's location and fuel type"""

# Get base futures prices

futures\_curve = self.futures\_analyzer.get\_futures\_curve()

# Apply regional adjustments

regional\_premium = self.regional\_adjustments.get(

self.location['state'], 0.0

)

# Apply octane premium

octane\_premium = self.get\_octane\_premium(octane\_level)

monthly\_forecasts = {}

for month in range(1, 13):

# Get futures price for that month

futures\_price = self.interpolate\_futures\_price(futures\_curve, month)

# Apply adjustments

retail\_margin = 0.35 # Average retail margin

taxes = self.get\_state\_tax\_rate(self.location['state'])

seasonal\_adjustment = self.get\_seasonal\_price\_adjustment(month)

predicted\_price = (

futures\_price +

retail\_margin +

taxes +

regional\_premium +

octane\_premium +

seasonal\_adjustment

)

monthly\_forecasts[month] = {

'base\_futures\_price': futures\_price,

'predicted\_retail\_price': round(predicted\_price, 3),

'confidence\_interval': self.calculate\_confidence\_interval(month),

'price\_components': {

'futures': futures\_price,

'retail\_margin': retail\_margin,

'taxes': taxes,

'regional\_premium': regional\_premium,

'octane\_premium': octane\_premium,

'seasonal\_adjustment': seasonal\_adjustment

}

}

return monthly\_forecasts

def get\_octane\_premium(self, octane\_level):

"""Calculate premium for higher octane fuels"""

premiums = {

'87': 0.00, # Regular baseline

'89': 0.15, # Mid-grade premium

'91': 0.25, # Premium

'93': 0.35, # Super premium

'93+': 0.40 # Ultra premium

}

return premiums.get(octane\_level, 0.0)

def get\_seasonal\_price\_adjustment(self, month):

"""Seasonal price adjustments based on demand patterns"""

adjustments = {

1: -0.05, # Winter blend savings

2: -0.03,

3: 0.02, # Refinery maintenance impact

4: 0.05, # Summer blend transition

5: 0.08, # Driving season begins

6: 0.12, # Peak driving season

7: 0.15, # Summer peak

8: 0.10, # Late summer

9: 0.02, # Seasonal transition

10: -0.02, # Fall driving decline

11: 0.00, # Baseline

12: -0.05 # Winter conditions

}

return adjustments.get(month, 0.0)

**Step 4: Monthly Spending Calculation**

**Core Spending Calculation Engine**

class MonthlySpendingCalculator:

def \_\_init\_\_(self, user\_profile, consumption\_data, price\_forecasts):

self.profile = user\_profile

self.consumption = consumption\_data

self.prices = price\_forecasts

def calculate\_monthly\_spending\_forecast(self):

"""Calculate expected spending for each month"""

spending\_forecast = {}

for month in range(1, 13):

month\_consumption = self.consumption[month]

month\_prices = self.prices[month]

# Base calculation

base\_spending = (

month\_consumption['estimated\_gallons'] \*

month\_prices['predicted\_retail\_price']

)

# Apply user behavior adjustments

adjusted\_spending = self.apply\_user\_behavior\_adjustments(

base\_spending, month, month\_prices

)

spending\_forecast[month] = {

'base\_spending': round(base\_spending, 2),

'adjusted\_spending': round(adjusted\_spending, 2),

'gallons\_consumed': round(month\_consumption['estimated\_gallons'], 1),

'average\_price\_paid': round(month\_prices['predicted\_retail\_price'], 3),

'miles\_driven': round(month\_consumption['estimated\_miles'], 0),

'fill\_ups\_per\_month': self.calculate\_fill\_ups(month\_consumption),

'spending\_breakdown': self.get\_spending\_breakdown(month, adjusted\_spending),

'savings\_opportunities': self.identify\_savings\_opportunities(month, month\_prices)

}

# Add annual summary

spending\_forecast['annual\_summary'] = self.calculate\_annual\_summary(spending\_forecast)

return spending\_forecast

def apply\_user\_behavior\_adjustments(self, base\_spending, month, price\_data):

"""Adjust spending based on user behavior patterns"""

adjusted\_spending = base\_spending

# Price sensitivity adjustment

if self.profile['preferences']['budget\_conscious'].lower() == 'y':

# Budget-conscious users may drive less when prices are high

price\_elasticity = -0.15 # 15% reduction for 10% price increase

avg\_price = 3.50 # National average baseline

price\_ratio = price\_data['predicted\_retail\_price'] / avg\_price

if price\_ratio > 1.1: # Prices 10%+ above average

demand\_reduction = price\_elasticity \* (price\_ratio - 1)

adjusted\_spending \*= (1 + demand\_reduction)

# Detour behavior for cheaper gas

max\_detour = float(self.profile['preferences'].get('max\_detour', 0))

if max\_detour > 0:

potential\_savings = self.calculate\_detour\_savings(max\_detour, price\_data)

adjusted\_spending -= potential\_savings

# Payment method discounts

payment\_method = self.profile['preferences']['payment\_method']

if payment\_method == 'cash':

adjusted\_spending \*= 0.97 # 3% cash discount average

return adjusted\_spending

def calculate\_fill\_ups(self, consumption\_data):

"""Calculate number of fill-ups per month"""

tank\_capacity = float(self.profile['vehicle'].get('tank\_capacity', 15))

gallons\_per\_month = consumption\_data['estimated\_gallons']

return round(gallons\_per\_month / tank\_capacity, 1)

def identify\_savings\_opportunities(self, month, price\_data):

"""Identify potential savings for the user"""

opportunities = []

# Cash vs credit savings

cash\_savings = price\_data['predicted\_retail\_price'] \* 0.03 # 3% average

opportunities.append({

'type': 'payment\_method',

'description': 'Pay with cash instead of credit',

'monthly\_savings': round(cash\_savings \* self.consumption[month]['estimated\_gallons'], 2)

})

# Station loyalty programs

loyalty\_savings = price\_data['predicted\_retail\_price'] \* 0.02 # 2% average

opportunities.append({

'type': 'loyalty\_program',

'description': 'Join station loyalty program',

'monthly\_savings': round(loyalty\_savings \* self.consumption[month]['estimated\_gallons'], 2)

})

# Warehouse club membership

warehouse\_savings = price\_data['predicted\_retail\_price'] \* 0.05 # 5% average

opportunities.append({

'type': 'warehouse\_club',

'description': 'Use warehouse club gas station',

'monthly\_savings': round(warehouse\_savings \* self.consumption[month]['estimated\_gallons'], 2)

})

return opportunities

**Step 5: Scenario Modeling and Uncertainty Analysis**

**Multiple Scenario Generation**

class ScenarioAnalyzer:

def \_\_init\_\_(self, base\_forecast):

self.base\_forecast = base\_forecast

def generate\_spending\_scenarios(self):

"""Generate optimistic, pessimistic, and realistic scenarios"""

scenarios = {}

# Realistic Scenario (Base Case)

scenarios['realistic'] = {

'description': 'Most likely outcome based on current trends',

'probability': 60,

'monthly\_spending': self.base\_forecast,

'assumptions': [

'Normal economic conditions',

'Stable geopolitical environment',

'Typical weather patterns',

'Current driving habits maintained'

]

}

# Optimistic Scenario (Lower Spending)

scenarios['optimistic'] = {

'description': 'Best case - lower prices and efficient driving',

'probability': 20,

'monthly\_spending': self.apply\_scenario\_adjustments(

self.base\_forecast,

price\_adjustment=-0.15, # 15% lower prices

consumption\_adjustment=-0.10 # 10% better efficiency

),

'assumptions': [

'Economic growth reduces oil prices',

'No major geopolitical disruptions',

'Improved driving efficiency',

'More strategic fuel purchasing'

]

}

# Pessimistic Scenario (Higher Spending)

scenarios['pessimistic'] = {

'description': 'Worst case - higher prices and increased consumption',

'probability': 20,

'monthly\_spending': self.apply\_scenario\_adjustments(

self.base\_forecast,

price\_adjustment=0.25, # 25% higher prices

consumption\_adjustment=0.15 # 15% more driving

),

'assumptions': [

'Geopolitical tensions raise oil prices',

'Supply disruptions occur',

'Increased commuting or travel',

'Economic inflation affects fuel costs'

]

}

return scenarios

def apply\_scenario\_adjustments(self, base\_forecast, price\_adjustment, consumption\_adjustment):

"""Apply scenario-specific adjustments to base forecast"""

adjusted\_forecast = {}

for month in range(1, 13):

base\_month = base\_forecast[month]

adjusted\_gallons = base\_month['gallons\_consumed'] \* (1 + consumption\_adjustment)

adjusted\_price = base\_month['average\_price\_paid'] \* (1 + price\_adjustment)

adjusted\_spending = adjusted\_gallons \* adjusted\_price

adjusted\_forecast[month] = {

\*\*base\_month,

'adjusted\_spending': round(adjusted\_spending, 2),

'gallons\_consumed': round(adjusted\_gallons, 1),

'average\_price\_paid': round(adjusted\_price, 3),

'scenario\_adjustments': {

'price\_change': f"{price\_adjustment:+.1%}",

'consumption\_change': f"{consumption\_adjustment:+.1%}"

}

}

return adjusted\_forecast

**Step 6: Results Presentation and User Interface**

**Comprehensive Results Formatting**

class SpendingForecastPresenter:

def \_\_init\_\_(self, scenarios, user\_profile):

self.scenarios = scenarios

self.profile = user\_profile

def generate\_user\_report(self):

"""Generate comprehensive spending forecast report"""

report = {

'user\_summary': self.create\_user\_summary(),

'annual\_overview': self.create\_annual\_overview(),

'monthly\_breakdown': self.create\_monthly\_breakdown(),

'scenarios': self.scenarios,

'recommendations': self.generate\_recommendations(),

'savings\_analysis': self.analyze\_savings\_opportunities(),

'budget\_planning': self.create\_budget\_recommendations()

}

return report

def create\_annual\_overview(self):

"""Create annual spending overview"""

realistic = self.scenarios['realistic']['monthly\_spending']

optimistic = self.scenarios['optimistic']['monthly\_spending']

pessimistic = self.scenarios['pessimistic']['monthly\_spending']

return {

'realistic\_scenario': {

'total\_annual\_spending': sum(realistic[m]['adjusted\_spending'] for m in range(1, 13)),

'average\_monthly\_spending': round(sum(realistic[m]['adjusted\_spending'] for m in range(1, 13)) / 12, 2),

'total\_gallons': sum(realistic[m]['gallons\_consumed'] for m in range(1, 13)),

'average\_price': round(sum(realistic[m]['average\_price\_paid'] for m in range(1, 13)) / 12, 3)

},

'spending\_range': {

'best\_case': sum(optimistic[m]['adjusted\_spending'] for m in range(1, 13)),

'worst\_case': sum(pessimistic[m]['adjusted\_spending'] for m in range(1, 13)),

'range\_width': sum(pessimistic[m]['adjusted\_spending'] for m in range(1, 13)) -

sum(optimistic[m]['adjusted\_spending'] for m in range(1, 13))

}

}

def generate\_recommendations(self):

"""Generate personalized recommendations"""

recommendations = []

# Budget planning recommendation

annual\_spending = self.scenarios['realistic']['monthly\_spending']

total\_annual = sum(annual\_spending[m]['adjusted\_spending'] for m in range(1, 13))

monthly\_budget = round(total\_annual / 12 \* 1.1, 2) # 10% buffer

recommendations.append({

'category': 'budgeting',

'title': 'Monthly Gas Budget',

'description': f"Set aside ${monthly\_budget}/month for gas expenses",

'impact': 'Helps avoid budget surprises',

'priority': 'high'

})

# High spending months warning

high\_months = [m for m in range(1, 13)

if annual\_spending[m]['adjusted\_spending'] > monthly\_budget]

if high\_months:

month\_names = [calendar.month\_name[m] for m in high\_months]

recommendations.append({

'category': 'planning',

'title': 'High Spending Months',

'description': f"Expect higher gas costs in: {', '.join(month\_names)}",

'impact': 'Plan for seasonal variations',

'priority': 'medium'

})

# Efficiency recommendations

current\_mpg = self.profile['calculated']['combined\_mpg']

if current\_mpg < 30:

recommendations.append({

'category': 'efficiency',

'title': 'Improve Fuel Efficiency',

'description': 'Consider fuel-efficient driving techniques or vehicle upgrade',

'impact': f'Could save ${round(total\_annual \* 0.15, 2)}/year with 15% improvement',

'priority': 'medium'

})

return recommendations

def display\_console\_summary(self):

"""Display summary in console format"""

print("\n" + "="\*80)

print("12-MONTH GAS SPENDING FORECAST")

print("="\*80)

# User info

print(f"Vehicle: {self.profile['vehicle']['year']} {self.profile['vehicle']['make']} {self.profile['vehicle']['model']}")

print(f"Fuel Type: {self.profile['vehicle']['fuel\_type']} octane")

print(f"Estimated Annual Miles: {self.profile['driving\_patterns']['annual\_miles']}")

print(f"Combined MPG: {self.profile['calculated']['combined\_mpg']:.1f}")

# Annual overview

overview = self.create\_annual\_overview()

realistic = overview['realistic\_scenario']

range\_data = overview['spending\_range']

print(f"\nANNUAL SPENDING FORECAST:")

print(f"Most Likely: ${realistic['total\_annual\_spending']:,.2f}")

print(f"Best Case: ${range\_data['best\_case']:,.2f}")

print(f"Worst Case: ${range\_data['worst\_case']:,.2f}")

print(f"Monthly Budget: ${realistic['average\_monthly\_spending']:,.2f}")

# Monthly breakdown

print(f"\nMONTHLY BREAKDOWN (Realistic Scenario):")

print("-"\*80)

print(f"{'Month':<10} {'Spending':<12} {'Gallons':<10} {'Avg Price':<12} {'Miles':<8}")

print("-"\*80)

realistic\_monthly = self.scenarios['realistic']['monthly\_spending']

for month in range(1, 13):

month\_data = realistic\_monthly[month]

month\_name = calendar.month\_abbr[month]

print(f"{month\_name:<10} ${month\_data['adjusted\_spending']:<11.2f} "

f"{month\_data['gallons\_consumed']:<9.1f} "

f"${month\_data['average\_price\_paid']:<11.3f} "

f"{month\_data['miles\_driven']:<8.0f}")

print("\n" + "="\*80)

# Usage Example

def main():

# Collect user data

user\_profile = collect\_user\_profile()

# Analyze driving patterns

pattern\_analyzer = DrivingPatternAnalyzer(user\_profile)

monthly\_consumption = pattern\_analyzer.calculate\_monthly\_consumption()

# Generate price forecasts

price\_forecaster = MonthlyGasPriceForecast(user\_profile['location'])

price\_forecasts = price\_forecaster.generate\_12\_month\_price\_forecast(

user\_profile['vehicle']['fuel\_type']

)

# Calculate spending forecasts

spending\_calculator = MonthlySpendingCalculator(

user\_profile, monthly\_consumption, price\_forecasts

)

base\_forecast = spending\_calculator.calculate\_monthly\_spending\_forecast()

# Generate scenarios

scenario\_analyzer = ScenarioAnalyzer(base\_forecast)

scenarios = scenario\_analyzer.generate\_spending\_scenarios()

# Present results

presenter = SpendingForecastPresenter(scenarios, user\_profile)

presenter.display\_console\_summary()

return presenter.generate\_user\_report()

if \_\_name\_\_ == "\_\_main\_\_":

forecast\_report = main()

This comprehensive implementation provides users with detailed 12-month gas spending forecasts, incorporating vehicle efficiency, driving patterns, seasonal variations, price predictions, and multiple scenarios to help with budget planning and decision-making.