AI & UX

Technical Guide

Prepared by

Kadharmoideen Fadurudeen

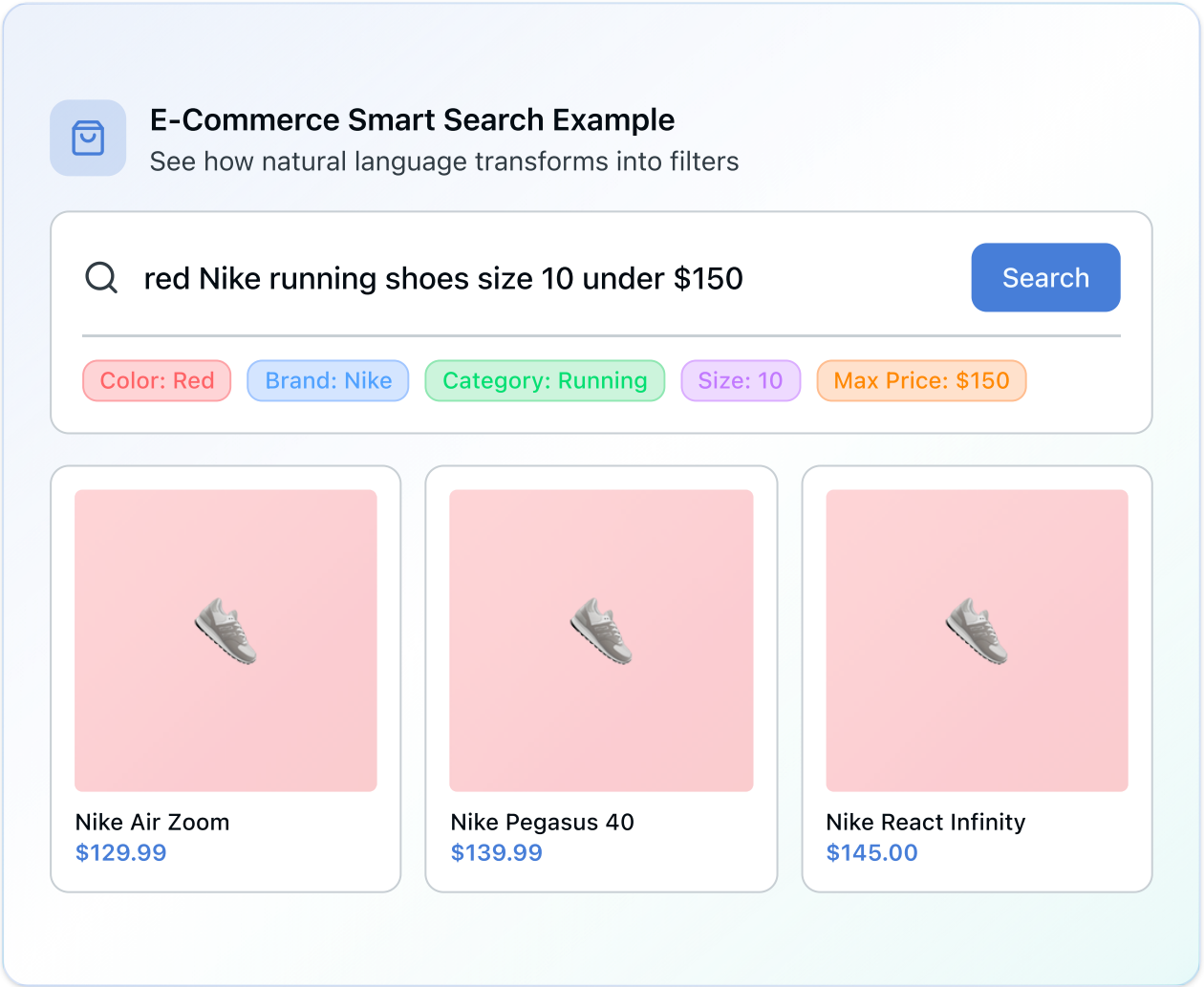
In this guide

1 article · ~16 min total reading

AI & UX October 16, 2025 16 min read

Let Users Speak Naturally — and Let AI Handle the Filters

Discover how AI-powered smart search filters transform user experience by allowing natural language queries. Learn to build intelligent search that understands user intent and extracts structured filters automatically.



The Problem with Traditional Search

Traditional search interfaces force users to think like databases. They need to select the right dropdowns, check the correct boxes, and adjust sliders to find what they want. This creates friction and cognitive load.

What if users could just type what they want in plain English, and your application would understand their intent and apply the right filters automatically?

Traditional Search

- ✕ Multiple dropdown menus
- ✕ Complex filter panels
- ✕ Users must know exact categories
- ✕ High cognitive load
- ✕ Mobile-unfriendly

AI-Powered Smart Search

- ✓ Natural language input
- ✓ AI extracts filters automatically
- ✓ Understands user intent
- ✓ Minimal cognitive load
- ✓ Works great on mobile

Real-World Examples

Here are some examples of how users naturally express their search intent, and how AI can extract structured filters:

"3 bedroom apartments under $2000 near downtown with parking"

bedrooms: 3 | max_price: $2000 | location: downtown | amenities: parking

"senior software engineer jobs in San Francisco with remote option"

title: Senior Software Engineer | location: San Francisco | remote: true

"red Nike running shoes size 10 under $150 with free shipping"

color: red | brand: Nike | category: running | size: 10 | max_price: $150 | free_shipping: true

How It Works

The magic happens in three steps: the user types naturally, AI extracts structured data, and your application applies the filters.

1

User Input

User types their query in natural language

2

AI Processing

AI extracts structured filters from the query

3

Apply Filters

Application searches with extracted filters

Implementation with AI SDK

Let's build a smart search filter using the Vercel AI SDK. We'll create a function that takes natural language input and returns structured filter data.

Step 1: Define Your Filter Schema

First, define what filters your application supports using Zod schema:

```
import { z } from 'zod'

// Define your filter schema
const PropertySearchFilters = z.object({
  bedrooms: z.number().optional().describe('Number of bedrooms'),
  bathrooms: z.number().optional().describe('Number of bathrooms'),
  minPrice: z.number().optional().describe('Minimum price in dollars'),
  maxPrice: z.number().optional().describe('Maximum price in dollars'),
  location: z.string().optional().describe('Location or neighborhood'),
  propertyType: z.enum(['apartment', 'house', 'condo', 'townhouse']).op
  amenities: z.array(z.string()).optional().describe('Required amenitie
  petFriendly: z.boolean().optional().describe('Pet-friendly property'
  parking: z.boolean().optional().describe('Parking available'),
})

type PropertyFilters = z.infer<typeof PropertySearchFilters>
```

Step 2: Create the AI Parser Function

Use the AI SDK to extract structured data from natural language:

```
import { generateObject } from 'ai'

export async function parseSearchQuery(query: string) {
  const { object } = await generateObject({
    model: 'openai/gpt-4o-mini',
    schema: PropertySearchFilters,
    prompt: `Extract search filters from this natural language query: '

    Guidelines:
    - Extract all relevant filter values
    - Convert price mentions to numbers (e.g., "$2000" -> 2000)
    - Identify location names
    - Detect amenities mentioned (parking, pool, gym, etc.)
    - Infer property type if mentioned
    - Set boolean flags for pet-friendly, parking, etc.

    If a filter is not mentioned, omit it from the response.`,
  })

  return object
}
```

Step 3: Build the Search Component

Create a React component that uses the AI parser:

```
'use client'

import { useState } from 'react'
import { Search } from 'lucide-react'

export function SmartSearchBar() {
  const [query, setQuery] = useState('')
  const [filters, setFilters] = useState(null)
  const [loading, setLoading] = useState(false)

  const handleSearch = async () => {
    setLoading(true)
    try {
      const response = await fetch('/api/parse-search', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ query }),
      })

      const data = await response.json()
      setFilters(data.filters)

      // Now use these filters to search your database
      await searchProperties(data.filters)
    } catch (error) {
      console.error('Search failed:', error)
    } finally {
      setLoading(false)
    }
  }

  return (
    <div className="space-y-4">
      <div className="flex gap-2">
        <input
          type="text"
          value={query}
          onChange={(e) => setQuery(e.target.value)}
          placeholder="Try: 2 bedroom apartment under $2000 near downto
          className="flex-1 px-4 py-2 border rounded-lg"
          onKeyDown={(e) => e.key === 'Enter' && handleSearch()}
        />
        <button
          onClick={handleSearch}
          disabled={loading}
          className="px-6 py-2 bg-primary text-white rounded-lg"
        >
          <Search className="w-5 h-5" />
```

```
        </button>
      </div>

      {filters && (
        <div className="flex flex-wrap gap-2">
          {Object.entries(filters).map(([key, value]) => (
            <span key={key} className="px-3 py-1 bg-primary/20 rounded-
              {key}: {JSON.stringify(value)}
            </span>
          ))}
        </div>
      )}
    </div>
  )
}
```

Step 4: Create the API Route

Set up a Next.js API route to handle the parsing:

```
// app/api/parse-search/route.ts
import { NextResponse } from 'next/server'
import { parseSearchQuery } from '@/lib/search-parser'

export async function POST(request: Request) {
  try {
    const { query } = await request.json()

    if (!query) {
      return NextResponse.json(
        { error: 'Query is required' },
        { status: 400 }
      )
    }

    const filters = await parseSearchQuery(query)

    return NextResponse.json({ filters })
  } catch (error) {
    console.error('Parse error:', error)
    return NextResponse.json(
      { error: 'Failed to parse query' },
      { status: 500 }
    )
  }
}
```

Real-World Use Cases

Smart search filters can transform user experience across many domains:

E-Commerce

"wireless headphones under $100 with noise cancellation"

category: headphones | max_price: $100 | features: noise cancellation

Job Search

"remote frontend developer jobs with React experience"

role: frontend developer | remote: true | skills: React

Travel Booking

"hotels in Paris with breakfast included under €150 per night"

location: Paris | amenities: breakfast | max_price: €150

Restaurant Discovery

"Italian restaurants near me with outdoor seating"

cuisine: Italian | location: nearby | features: outdoor seating

Best Practices

Show Extracted Filters

Display the extracted filters as badges or chips so users can see what the AI understood. This builds trust and allows users to verify the interpretation.

Allow Manual Editing

Let users click on extracted filters to modify them. The AI might not always get it right, so provide an easy way to adjust.

Provide Examples

Show example queries as placeholder text or suggestions to help users understand what they can ask for.

Handle Ambiguity Gracefully

When the AI is uncertain, ask clarifying questions or show multiple interpretations for the user to choose from.

Cache Common Queries

Cache the results of common queries to reduce API calls and improve response time. Many users search for similar things.

Advanced Features

Multi-Turn Conversations

Take it further by allowing users to refine their search through conversation:

User: "Show me apartments in downtown"

AI: "Found 45 apartments. What's your budget?"

User: "Under $2000"

AI: "Narrowed to 12 apartments. Any other requirements?"

User: "Must have parking"

Fuzzy Matching

Handle typos and variations in user input gracefully:

```
// The AI can understand variations:
"appartment" → "apartment"
"2br" → "2 bedrooms"
"pet ok" → "pet friendly"
"dwntwn" → "downtown"
"w/ parking" → "with parking"
```

Context-Aware Suggestions

Use the user's search history and preferences to provide better suggestions:

```
const { object } = await generateObject({
  model: 'openai/gpt-4o-mini',
  schema: PropertySearchFilters,
  prompt: `Extract filters from: "${query}"

  User context:
  - Previously searched for: ${previousSearches.join(', ')}
  - Preferred locations: ${preferredLocations.join(', ')}
  - Budget range: ${budgetRange}

  Use this context to better understand the query.`,
})
```

Conclusion

AI-powered smart search filters represent a paradigm shift in how users interact with search interfaces. By allowing natural language input and automatically extracting structured filters, you can dramatically improve user experience, reduce friction, and increase conversion rates.

The implementation is straightforward with modern AI SDKs, and the benefits are immediate. Users no longer need to think like databases—they can just speak naturally, and your application will understand.

Ready to Transform Your Search Experience?

Start by identifying the filters your application uses, define a schema, and integrate the AI SDK. Your users will thank you for the improved experience.