

Geautomatiseerde prijsvergelijking voor Belgische supermarktketens.

Een Proof of Concept-systeem voor de in Gent aanwezige supermarktketens, gebaseerd op openbaar toegankelijke data.

Aliaksandra Nemchynava.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Mevr. L. Vuyge

Co-promotor: Dhr. J. Pots

Academiejaar: 2025–2026

Eerste examenperiode

Departement IT en Digitale Innovatie .

**HO
GENT**

Woord vooraf

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Samenvatting

De stijgende voedselprijzen in België leggen een groeiende financiële druk op studenten met een beperkt budget en beperkte mobiliteit. Bestaande prijsvergelijkingstools houden onvoldoende rekening met praktische beperkingen van winkelen, zoals reisafstand en het aantal winkels dat een consument bereid is te bezoeken. Bovendien baseren veel tools zich op barcodevergelijking, waardoor huismerken vaak buiten de scope worden gelaten. Daarnaast staan veel van deze diensten of hun extra functies achter een betaalmuur.

Dit onderzoek richt zich op het ontwerpen van een systeem dat consumenten, met name studenten in Gent, ondersteunt om weloverwogen beslissingen te kunnen nemen over hun boodschappen. De centrale onderzoeksvraag luidt: “Hoe kan een transparant en schaalbaar systeem worden ontworpen en ontwikkeld om automatisch supermarktprijzen in België te verzamelen, matchen en vergelijken, rekening houdend met beperkingen, zoals afstand en het aantal te bezoeken winkels?”

Het voorgestelde prototype wordt ontwikkeld in Python en Django, waarbij gebruik wordt gemaakt van webscrapingstechnieken om prijsgegevens van Belgische supermarktwebsites te verzamelen. De gegevens worden opgeslagen in een PostgreSQL-database. Gebruikers kunnen een boodschappenlijst invoeren en een maximale reisafstand of een limiet voor het aantal winkels instellen. Vervolgens wordt het meest kostefficiënte aankoopplan door het systeem berekend. De gegenereerde voorstellen worden geëvalueerd door hun besparingen te vergelijken met een aankoop van alle boodschappen in één winkel, op basis van vooraf gedefinieerde boodschappenlijsten. De resultaten zijn merkbare besparingen over alle voorgestelde boodschappenlijsten.

Dit project draagt bij aan een realistisch en toegankelijk prijsvergelijkingssysteem gericht op studenten door technische efficiëntie te combineren met consument-gerichte beperkingen. Dit stelt studenten in staat bewuster en voordeliger te winkelen.

Inhoudsopgave

Lijst van figuren	vii
Lijst van tabellen	viii
Lijst van codefragmenten	ix
1 Inleiding	1
1.1 Probleemstelling	1
1.2 Onderzoeksvraag	1
1.3 Deelvragen	2
1.4 Onderzoeksdoelstelling	2
1.5 Opzet van deze bachelorproef	3
2 Stand van zaken	4
2.1 Context: voedselprijzen en studentendruk	4
2.2 Bestaande oplossingen	4
2.3 Supermarktdata: webscraping als praktische pijplijn	5
2.4 Juridische en ethische overwegingen voor scraping	5
2.5 Productmatching tussen retailers	5
2.6 Beslissingsondersteuning, vertrouwen en beperkingen in boodschap- penapps	6
3 Methodologie	7
3.1 Proof of concept	7
3.1.1 Systeemontwerp	7
3.1.2 Dataverzameling	7
3.1.3 Dataverwerking en productmatching	8
3.1.4 Interfaceontwikkeling en integratie	8
3.1.5 Evaluatie	8
4 Prototype	10
4.1 Keuze van programmeertaal en frameworks	10
4.2 Klassieke web scraping pijplijn	12
4.3 Gebruikersinterface	13
4.3.1 Rol van Django binnen het systeem	14
4.3.2 Data-architectuur en modellering	15
4.3.3 Zoekworkflow en asynchrone verwerking	15
4.3.4 Integratie met Kafka	15

4.3.5	Gebruikersfunctionaliteiten	16
4.3.6	Statusmodellering en gebruikersfeedback	16
4.3.7	Conclusie	16
4.4	Scrapinglaag	17
4.4.1	Algemene ontwikkelingsmethodologie van de spiders	17
4.4.2	Initiële aanpak en beperkingen	18
4.4.3	Overstap naar Scrapy en Playwright	18
4.4.4	Website-specifieke scrapingstrategieën	19
4.4.5	Orkestratie van scraping taken	19
4.4.6	Albert Heijn (AH)	20
4.4.7	Carrefour	22
4.4.8	Colruyt	23
4.4.9	Vergelijking van scraping-strategieën	29
4.5	Opslag van ruwe data	30
4.6	Transformatielaag	31
4.7	Gestructureerde opslag	31
4.8	Optimalisatie van productmatching	31
4.8.1	Probleemstelling	31
4.8.2	Methodologie: gelaagde similariteitsscore	31
4.8.3	Implementatie	32
4.8.4	Impact op systeemfunctionaliteiten	33
4.8.5	Conclusie	33
4.9	Geplande updates	33
4.10	Docker	34
4.11	Conclusie	34
5	Testen en Resultaten	35
6	Conclusie	36
A	Onderzoeksvoorstel	38
A.1	Introduction	38
A.2	Literature Review	39
A.2.1	Context: food prices and student pressure	39
A.2.2	Existing solutions	40
A.2.3	Supermarket data: web scraping as a practical pipeline	40
A.2.4	Legal and Ethical considerations for scraping	40
A.2.5	Product matching across retailers	41
A.2.6	Decision support, trust, and constraints in grocery apps	41
A.3	Methodology	41
A.3.1	System Design	41
A.3.2	Data Collection	42
A.3.3	Data Processing and Product Matching	42

A.3.4	System implementation	42
A.3.5	Evaluation	42
A.4	Expected results.	43

Bibliografie		44
---------------------	--	-----------

Lijst van figuren

4.1	Event-driven scraping pipeline: Django publiceert scrape-jobs naar Kafka; de scraper worker voert spiders uit; resultaten worden genormaliseerd en opgeslagen in de database.	13
4.2	Overzicht van de voorgestelde prototype-architectuur	14
4.3	Gedetecteerd met BeautifulSoup	19
4.4	Schermafbeelding browser	25
4.5	Schermafbeelding browser	26
4.6	Schermafbeelding browser van Colruyt met HTML element	26
4.7	Schermafbeelding browser van Colruyt met HTML element: button. . .	27
4.8	Colruyt scraping flow met Scrapy-Playwright: requests renderen in een headless browser waarna HTML wordt uitgelezen en geparsed.	28

Lijst van tabellen

2.1	Functionele vergelijking tussen bestaande tools en het voorgestelde prototype	6
4.1	Vergelijking van scraping-strategieën per supermarkt	29

Lijst van codefragmenten

4.1	Gelaagde productmatching met prioriteitsscores	32
-----	--	----

1

Inleiding

De voedselprijzen in België zijn de afgelopen jaren aanzienlijk gestegen, wat een groeiende financiële druk legt op studenten en andere budgetbewuste consumenten. Hoewel er verschillende prijsvergelijkingstools voor supermarkten bestaan, richten deze zich over het algemeen op het presenteren van de laagste prijzen zonder rekening te houden met praktische beperkingen, zoals de afstand die een consument bereid is af te leggen of het aantal winkels dat hij of zij redelijkerwijs kan bezoeken. Hierdoor bieden deze tools theoretisch optimale oplossingen die in de praktijk niet haalbaar zijn, vooral voor studenten met beperkte mobiliteit en een strak schema.

1.1. Probleemstelling

Studenten ondervinden vaak uitdagingen bij het vinden van de meest kostenefficiënte winkelopties. Beperkte budgetten, gecombineerd met tijd- en reisbeperkingen, bemoeilijken efficiënte prijsvergelijkingen tussen verschillende supermarkten. Bestaande tools houden zelden rekening met deze beperkingen, waardoor er een kloof ontstaat tussen beschikbare prijsinformatie en bruikbare, gebruikersgerichte inzichten. Dit onderzoek pakt deze kloof aan door zich te richten op de ontwikkeling van een systeem dat afgestemd is op de behoeften van studenten met een beperkt budget in Gent.

1.2. Onderzoeksvraag

Om een dergelijk systeem te ontwikkelen, moet de volgende hoofdonderzoeksvraag worden beantwoord: Hoe ontwerp en ontwikkel je een transparant en schaalbaar systeem dat automatisch supermarktprijzen in Gent kan verzamelen, matchen en vergelijken, rekening houdend met de afstand tot de consument en een door de gebruiker opgegeven limiet voor het aantal te bezoeken winkels?

1.3. Deelvragen

Verder moet onderzoek worden gedaan naar de programmatische en architecturale details van de beoogde oplossing en de succesfactoren ervan. Meer specifiek:

- Hoe kunnen supermarktprijsgegevens automatisch worden verzameld en gestructureerd?
- Welke benaderingen kunnen worden gebruikt om algemene productnamen te matchen en zo nauwkeurige vergelijkingen te maken?
- Hoe kan het systeem de meest kosteneffectieve combinaties van winkels binnen een door de gebruiker gedefinieerde afstand en een winkellimiet berekenen en aanbevelen?
- Hoe kan de systeemarchitectuur worden ontworpen om schaalbaarheid en transparantie van de data te ondersteunen?
- Aan welke criteria moet het prototype voldoen om als een geldig proof-of-concept te worden beschouwd?

Om deze vragen te beantwoorden en het onderzoek te sturen, is inzicht vereist in de doelgroep en de probleemcontext. Meer specifiek:

- Welke factoren beïnvloeden momenteel de consumptiegewoonten van studenten in België?
- Welke tools voor prijsvergelijking zijn er in België beschikbaar en welke tekortkomingen hebben ze voor studenten?
- Welke technische en praktische uitdagingen zijn er bij het verzamelen van prijsgegevens van Belgische supermarkten?

1.4. Onderzoeksdoelstelling

Het resultaat van dit onderzoek is een prototype, geïmplementeerd met Python en Django, dat prijsgegevens verzamelt via webscraping van Belgische supermarktwebsites. Gebruikers van dit prototype kunnen een algemene boodschappenlijst invoeren en een maximale reisafstand of een limiet voor het aantal winkels opgeven. Het systeem berekent vervolgens de meest kostenefficiënte combinatie van winkels op basis van deze beperkingen. Het systeem wordt geëvalueerd met behulp van een vooraf gedefinieerde standaardboodschappenlijst voor studenten om de totale kosten van een aankoop in één winkel te vergelijken met de door het systeem gegenereerde, geoptimaliseerde aanbeveling voor meerdere winkels.

Dit onderzoek draagt bij aan de ontwikkeling van realistische en toegankelijke tools voor supermarktprijsvergelijking die technische efficiëntie combineren met consumentgerichte beperkingen. Door zich te richten op de behoeften van studenten,

beoogt het systeem de prijstransparantie te vergroten en weloverwogen, budgetbewuste winkelbeslissingen te ondersteunen. Het biedt praktische inzichten die toekomstige consumentgerichte toepassingen kunnen inspireren.

1.5. Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 6, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2

Stand van zaken

In het vorige hoofdstuk is de probleemstelling geschetst: studenten in Gent hebben behoefte aan een prijsvergelijkingssysteem dat rekening houdt met praktische beperkingen zoals reisafstand en een beperkt budget. Dit hoofdstuk plaatst dat probleem in een bredere context door de stand van zaken in het onderzoeksdomein te beschrijven. Digitale prijsvergelijkingstools en geautomatiseerde dataverzameling worden steeds belangrijker in de voedselretail. Dit onderzoek situeert zich op het kruispunt van drie relevante domeinen: webgebaseerde dataverzameling, productnormalisatie- en prijsvergelijkingssystemen voor consumenten.

2.1. Context: voedselprijzen en studentendruk

Recente economische indicatoren van België wijzen op aanhoudende prijsdruk op voeding. Volgens het CPI-rapport van Statbel blijven de algemene en kerninflatie gedurende 2024-2025 hoog, met een kerninflatie van meer dan 2% in oktober 2025(Statbel, [2025a](#), [2025b](#)). Onafhankelijke tracking door Testaankoop/Testachats meldt eveneens een supermarktspecifieke inflatie van ongeveer 4% in 2025(Testaankoop, [2025a](#), [2025b](#)). Bredere macro-economische analyses(*OECD Economic Surveys: Belgium 2024*, [2024](#)) tonen de gedetailleerde impact van inflatie aan en bevestigen de prijsdruk op consumenten. Samen onderbouwen deze bronnen de relevantie van het probleem voor prijsgevoelige groepen zoals studenten.

2.2. Bestaande oplossingen

Er zijn verschillende specifieke Belgische tools beschikbaar om consumenten te helpen supermarktprijzen te vergelijken en betere producten te selecteren, zoals PingPrice (PingPrice, [2024](#)) en G4U (G4U, [2025](#)). Beide apps hebben echter hun beperkingen. PingPrice vergelijkt producten met behulp van barcodes, waardoor het geen effectieve vergelijking kan maken tussen huiskamerproducten of gene-

rieke producten die geen gestandaardiseerde identificatiecodes hebben. Hierdoor worden veel relevante artikelen uitgesloten van vergelijkingen.

G4U biedt daarentegen uitgebreide product- en promotie-informatie, maar werkt als een betaalde dienst, waardoor de toegankelijkheid beperkt is voor studenten die al met financiële beperkingen kampen. Daarom is er behoefte aan een gratis en transparant alternatief waarmee gebruikers generieke productcategorieën kunnen vergelijken in plaats van barcodes.

In Tabel 2.1 worden de belangrijkste kenmerken visueel representeert.

2.3. Supermarktdata: webscraping als praktische pijplijn

Omdat Belgische retailers zelden API's voor product-/prijsfeeds openbaar maken, is webscraping een pragmatische manier om gestructureerde prijsgegevens van openbare pagina's te verkrijgen. Hoewel (Logos et al., 2023) en (Brown et al., 2024) een ethische en methodologische benadering van webscraping beschrijven, stellen ze geen specifieke technische implementatie voor voor gevallen waarin openbare API's niet beschikbaar zijn.

Voortbouwend op hun aanbevelingen wordt in dit onderzoek het volgende proces voorgesteld: HTML-opvraging, parsing van de content, headless browser voor JavaScript-afhankelijke content en opslag van de prijsgegevens. Deze aanpak voor de specifieke Belgische markt is geïnspireerd op het (Ken Van Loon, 2018) artikel van Statbel.

2.4. Juridische en ethische overwegingen voor scraping

Scraping moet voldoen aan de servicevoorwaarden (ToS), intellectuele eigendomsrechten en beperkingen op het gebied van gegevensbescherming. Vergelijkende analyses van de ToS van websites laten zien dat veel platforms "robots/scrapers" expliciet reguleren, waardoor onderzoekers noodzakelijkheid, proportionaliteit en nalevingsmechanismen moeten afwegen (Fiesler et al., 2020). Recente overzichten stellen concrete checklists voor over legaliteit, ethiek en institutionele beoordeling: bijvoorbeeld het documenteren van het doel, snelheidslimieten, opslag en datadeeling (Brown et al., 2024; Logos et al., 2023). Deze kaders vormen de basis voor het beheer van het prototype.

2.5. Productmatching tussen retailers

Prijsvergelijking vereist het matchen van 'hetzelfde' artikel in alle winkels, ondanks verschillen in naamgeving/verpakkingsgrootte. Bestaande literatuur ondersteunt een tweefasenaanpak: 1. exacte identificatiegegevens (bijv. EAN/GTIN) indien beschikbaar; 2. benaderende/semantische matching met behulp van fuzzy similarity (Levenshtein/TF-IDF/cosinus) of ML-embeddings voor detectie van bijna-duplicaten (Kerek, 2020; Ning et al., 2022). Deze methoden koppelen de door de gebruiker op-

Tabel 2.1: Functionele vergelijking tussen bestaande tools en het voorgestelde prototype

Kenmerk	PingPrice	G4U	Prototype
Naam matching	-	?	+
Transparantie	-	-	+
Beperkingen (afstand/aantal winkels)	-	-	+
Gratis	+	-	+

+ = ondersteund, - = niet ondersteund, ? = gedeeltelijk/onduidelijk

gegeven productnaam direct aan het specifieke productaanbod van de winkel.

2.6. Beslissingsondersteuning, vertrouwen en beperkingen in boodschappenapps

Vertrouwen is een cruciale factor die de bereidheid van gebruikers om digitale boodschappentools te gebruiken beïnvloedt. (Chakraborty et al., 2024) benadrukt het belang van geloofwaardigheid van informatie, duidelijkheid en kwaliteit van de interactie om het vertrouwen van gebruikers in online boodschappenomgevingen te vergroten. Voortbouwend op dit perspectief benadrukt (DeZao, 2024) het vertrouwen in AI-gestuurde systemen. Door hun gegevensbronnen en tijdstempels te tonen, worden deze systemen transparanter en daardoor ook als betrouwbaarder en eerlijker ervaren door gebruikers. Bovendien beïnvloeden praktische beperkingen, zoals reisafstand en de mogelijkheid om slechts een bepaald aantal winkels te bezoeken, het nut van dergelijke tools. Integratie van deze beperkingen breidt de criteria voor beslissingsondersteuning verder uit en verbetert deze.

Samenvattend, de literatuur ondersteunt een pijplijn die webscraping, reproduceerbare matching (EAN-first + fuzzy/ML fallback) en transparante interfaces combineert, geëvalueerd op precisie/recall voor matches en realistische, op de student gerichte beperkingen (bijv. afstand, maximaal aantal winkels) voor kostenresultaten.

3

Methodologie

Dit proefschrift richt zich op het ontwerp en de implementatie van een functioneel prototype dat automatisch supermarktprijzen in Gent verzamelt, vergelijkt en matcht.

3.1. Proof of concept

De proof of concept opbouw bestaat uit vier opeenvolgende fasen: systeemontwerp, dataverzameling, dataverwerking en evaluatie.

3.1.1. Systeemontwerp

In de eerste fase werden de architectuur en datastroom van het systeem gedefinieerd met als doel modulariteit, schaalbaarheid en transparantie te garanderen. Het systeem is opgebouwd als een modulaire, event-gedreven pipeline bestaande uit vier lagen. De eerste laag is de scrapingslaag: product- en prijsinformatie wordt verzameld van de websites. De tweede laag is de transportlaag, verantwoordelijk voor publicatie van de ruwe data komende uit de scrapingslaag in Kafka. De derde laag is de verwerkingslaag, daar een Kafka-consumer verwerkt, normaliseert, matcht producten, en slaat de resultaten op in een PostgreSQL-database. De vierde laag is de presentatielaag: een Django-gebaseerde webinterface dat het mogelijk maakt om boodschappenlijsten en beperkingen in te voeren en berekent de goedkoopste winkelcombinatie.

3.1.2. Dataverzameling

De dataverzamelingsfase richt zich op het verzamelen van dagelijkse product- en prijsinformatie van geselecteerde Gentse supermarkten. Dit wordt uitgevoerd met behulp van webscrapingtechnieken, geïmplementeerd via Scrapy-spiders. Scrapy wordt gebruikt om HTTP-verzoeken te versturen en de onbewerkte HTML-inhoud

van webpagina's op te halen, waardoor toegang wordt verkregen tot publiek beschikbare informatie zonder een volledige browseromgeving.

De spiders extraheren relevante gegevens zoals productnamen, verpakkingsformaten, prijzen en merklables door specifieke HTML-elementen te parsen. Voor websites die gebruikmaken van JavaScript-gedreven dynamische inhoud wordt een browser-emulator ingezet, zodat pagina's eerst volledig kunnen laden voordat ze verwerkt worden.

In plaats van de gegevens onmiddellijk op te slaan of te verwerken, worden alle gescrapete productrecords als ruwe JSON-objecten gepubliceerd naar een Kafka-topic. Hierdoor wordt de dataverzameling losgekoppeld van de daaropvolgende verwerkingsstappen, wat de schaalbaarheid en fouttolerantie van het systeem verhoogt. Naast de productinformatie zelf wordt ook metadata zoals tijdstip en bronwinkel meegestuurd, waardoor transparantie en reproduceerbaarheid worden gegarandeerd.

3.1.3. Dataverwerking en productmatching

De dataverwerking gebeurt asynchroon in een aparte module die berichten uit Kafka consumeert. Omdat supermarkten verschillende productnamen en -formaten gebruiken, moeten de verzamelde gegevens worden voorbereid voordat ze kunnen worden vergeleken. Deze fase bestaat uit een aantal stappen: data cleaning, normalisatie en productmatching. De dataopschoningstap omvat het verwijderen van duplicaten en eenheidsnormalisatie (bijvoorbeeld prijs per kg of per liter). De matchingstap implementeert string-gelijkaardigheidssalgoritmen, zoals Levehnstein-afstand en cosinus-similariteit op TF-IDF-vectoren, om gelijkwaardige producten in verschillende winkels te matchen. De filterstap slaat de dichtstbijzijnde product-matches op om nauwkeurigheid in vergelijkingen te garanderen. Het resultaat van deze fase is een uniforme dataset waarin identieke of vergelijkbare producten uit verschillende winkels direct kunnen worden vergeleken.

3.1.4. Interfaceontwikkeling en integratie

De tool combineert alle componenten in één Django-gebaseerde webapplicatie. Daarnaast wordt er een vergelijking module binnen deze applicatie uitgewerkt die berekent voor een ingevoerd boodschappenlijstje de kostprijs als alle producten in één winkel worden gekocht en de minimale totale prijs bij een optimale winkelcombinatie, rekening houdend met maximale reisafstand en maximaal aantal winkels.

3.1.5. Evaluatie

De evaluatiefase beoordeelt de praktische bruikbaarheid van het systeem, met behulp van vooraf gedefinieerde winkelwagentjes voor studenten die realistische aankoopscenario's simuleren. Elk winkelwagentje wordt vanuit twee perspectieven geanalyseerd: winkelen in één winkel (alle artikelen in één supermarkt kopen) en ge-

optimaliseerd winkelen in meerdere winkels (alle artikelen kopen op basis van de aanbevelingen van het systeem).

Op basis van deze resultaten kan het prototype worden beschouwd als een succesvol proof-of-concept als het in staat is om kostenbesparingen te realiseren voor Gentse studenten met verschillende criteria, terwijl de transparantie in het besluitvormingsproces behouden blijft.

4

Prototype

Het prototype dat in deze bachelorproef werd ontwikkeld, is een proof-of-concept van een web scraping applicatie die bruikbaar is voor een breed publiek - met name studenten - dat niet beschikt over uitgebreide kennis van programmeren of softwareontwikkeling. De gebruiker heeft slechts een internetverbinding nodig en hoeft geen technische configuraties uit te voeren.

Deze doelstelling impliceert dat de technische complexiteit van web scraping wordt volledig afgeschermd van de eindgebruiker. Interacties met het systeem gebeuren via een eenvoudige en intuïtieve gebruikersinterface, terwijl alle processen zoals het ophalen van webpagina's, het verwerken van data, het omgaan met anti-scrapingsmechanismen en de opslag van gegevens, achter de schermen verlopen. Het prototype fungeert dus als een laag tussen de gebruiker en de onderliggende scrapingsinfrastructuur, waarbij gebruiksvriendelijkheid en betrouwbaarheid centraal staan.

4.1. Keuze van programmeertaal en frameworks

De keuze van programmeertaal en bijhorende frameworks vormt een essentiële ontwerpbeslissing binnen dit prototype. Literatuur toont aan dat verschillende programmeertalen geschikt zijn voor web scraping, waaronder Python, JavaScript en C#. Op basis van vergelijkende studies en best practices (Journal, 2024; Khder, 2021; Majebi, 2025; ProWebScraper, 2018) is Python gekozen als primaire programmeertaal voor het prototype.

Deze keuze voor Python kan gemotiveerd worden door meerdere factoren. Ten eerste beschikt Python over een uitgebreid ecosysteem van bibliotheken die specifiek ontworpen zijn voor interactie met de inhoud van websites, zoals *Requests*, *BeautifulSoup*, *Scrapy* en *Playwright*. Hierdoor kan snel en efficiënt worden ingespeeld op uiteenlopende scrapingsuitdagingen, variërend van eenvoudige HTML-pagina's

tot complexe, dynamisch gegenereerde webinhoud. Ten tweede is Python relatief eenvoudig aan te leren, wat het bijzonder geschikt maakt voor beginnende ontwikkelaars en studenten. De leesbaarheid van de syntaxis en de grote hoeveelheid beschikbare documentatie en voorbeelden verlagen de instapdrempel aanzienlijk. Dit sluit aan bij de doelstelling van dit prototype, namelijk het ontwikkelen van een systeem dat toegankelijk is voor gebruikers zonder diepgaande programmeerkenntnis. Tot slot biedt Python voldoende mogelijkheden voor toekomstige uitbreidingen, zoals grootschalige dataverwerking, data-analyse en integratie van machine learning technieken. Hierdoor is Python niet alleen geschikt voor het huidige prototype, maar ook toekomstbestendig.

Binnen het Python-ecosysteem bestaan meerdere bibliotheken voor web scraping. Na een vergelijking werd gekozen voor *Scrapy* als centraal scrapingsframework. Deze keuze kan onderbouwd worden door het werk van (Eyzenakh et al., 2021), waarin verschillende scrapingsoplossingen zijn vergeleken op vlak van prestatie, schaalbaarheid en architecturale opbouw. *Scrapy* onderscheidt zich van eenvoudigere oplossingen zoals *Requests* en *BeautifulSoup* door zijn asynchrone en event-gedreven architectuur. Dit maakt het mogelijk om meerdere webpagina's gelijktijdig te verwerken, wat resulteert in betere prestaties en efficiënter gebruik van systeembronnen. Daarnaast voorziet *Scrapy* ingebouwde ondersteuning voor request scheduling, foutafhandeling, throttling en uitbreidbare pipelines voor dataverwerking. Een bijkomend voordeel van *Scrapy* is de duidelijke scheiding tussen verschillende verantwoordelijkheden, zoals het ophalen van data, het parsen van responses en het verwerken van resultaten. Deze modulaire opbouw sluit nauw aan bij de architecturale principes die in de literatuur worden beschreven en verhoogt de onderhoudbaarheid van het systeem.

Hoewel *Scrapy* krachtig is voor het verwerken van statische HTML-pagina's, volstaat het niet in alle situaties. Steeds meer websites maken gebruik van JavaScript om inhoud dynamisch te genereren. Om deze pagina's correct te kunnen verwerken, werd *Playwright* geïntegreerd in het scrapingsproces. *Playwright* maakt het mogelijk om een echte browseromgeving te simuleren en JavaScript-code uit te voeren alvorens de HTML wordt geëxtraheerd. Hierdoor kunnen ook websites met complexe client-side logica succesvol gescrapet worden. De combinatie van *Scrapy* en *Playwright* zorgt voor een flexibele aanpak waarbij per website de meest geschikte scrapingstrategie kan worden toegepast.

Om de verschillende lagen binnen het systeem van elkaar te ontkoppelen, werd gekozen voor Apache Kafka als verbindende component tussen de scrapingslaag en verdere verwerking van data. Kafka fungeert hierbij als een message broker die ruwe scrapingsresultaten doorstuurt naar volgende verwerkingsstappen. De keuze voor Kafka is geïnspireerd door moderne, event-gedreven architecturen zoals beschreven in (Alexander, 2015). Door gebruik te maken van een message broker wordt de afhankelijkheid tussen scraping en verwerking verminderd. Dit ver-

hoogt de schaalbaarheid en fouttolerantie van het systeem en maakt het mogelijk om data opnieuw te verwerken zonder de scrapingsfase te herhalen. Hoewel Kafka in dit prototype niet noodzakelijk op industriële schaal wordt ingezet, biedt het conceptueel een duidelijke meerwaarde en vormt het een solide basis voor toekomstige uitbreidingen.

Voor de backend van het prototype werd gekozen voor het Django-framework. Django sluit naadloos aan bij Python en biedt uitgebreide ondersteuning voor webapplicaties, databankintegratie en gebruikersinteractie. Een belangrijk voordeel van Django is de eenvoudige koppeling met Python-gebaseerde scrapingscomponenten. Hierdoor kunnen scrapingsprocessen, dataverwerking en gebruikersinterface binnen één technologisch ecosysteem worden geïntegreerd. Daarnaast voorziet Django standaardfunctionaliteiten zoals ORM (Object-Relational Mapping), authenticatie en administratie, wat de ontwikkeltijd aanzienlijk verkort.

Om de reproduceerbaarheid en consistentie van het systeem op verschillende machines te garanderen, werd gebruikgemaakt van Docker. Docker maakt het mogelijk om de volledige applicatie, inclusief afhankelijkheden, configuraties en services, te verpakken in containers. Hierdoor kan het prototype zonder bijkomende installatieproblemen uitgevoerd worden op verschillende systemen, wat zowel de ontwikkeling als de evaluatie vereenvoudigt.

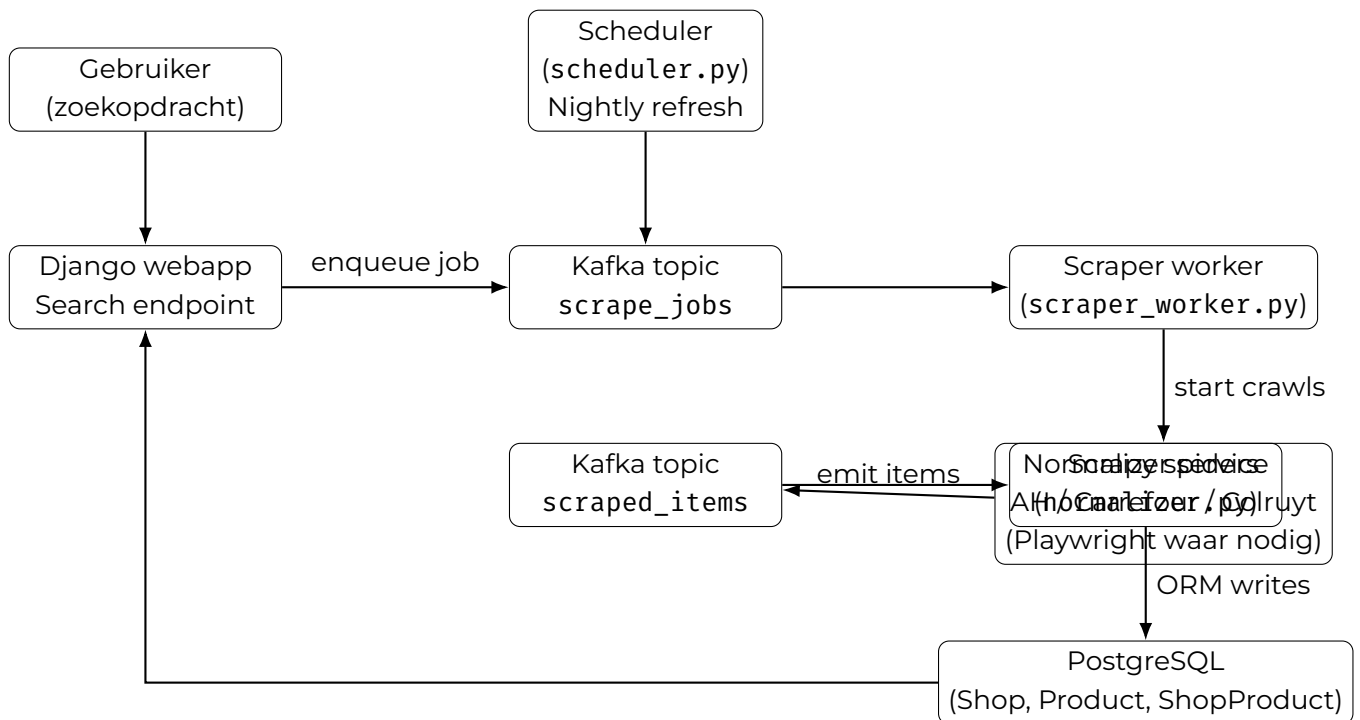
De combinatie van Python, Scrapy, Kafka, Django en Docker resulteert in een coherente en uitbreidbare architectuur waarin elke technologie een duidelijk afgebakende rol vervult.

4.2. Klassieke web scraping pijplijn

De structuur van web scraping systemen wordt in de literatuur vaak beschreven als een opeenvolging van afzonderlijke verwerkingsstappen. (Laender et al., 2002) en (Khder, 2021) beschrijven web data extractie als een pijplijn bestaande uit meerdere logisch gescheiden fasen:

- selectie van databronnen;
- extractie van ruwe data;
- transformatie en normalisatie;
- integratie en opslag.

Deze architecturale scheiding verhoogt de onderhoudbaarheid van het systeem en maakt hergebruik en schaalbaarheid mogelijk. Het prototype volgt deze pijplijnstructuur expliciet. Op basis van de overleg met de co-promotor en de besproken state-of-the-art werd een modulaire architectuur voorgesteld. Deze architectuur is weergegeven in Figuur 4.1.



Figuur 4.1: Event-driven scraping pipeline: Django publiceert scrape-jobs naar Kafka; de scraper worker voert spiders uit; resultaten worden genormaliseerd en opgeslagen in de database.

De architectuur bestaat uit vijf hoofdlagen:

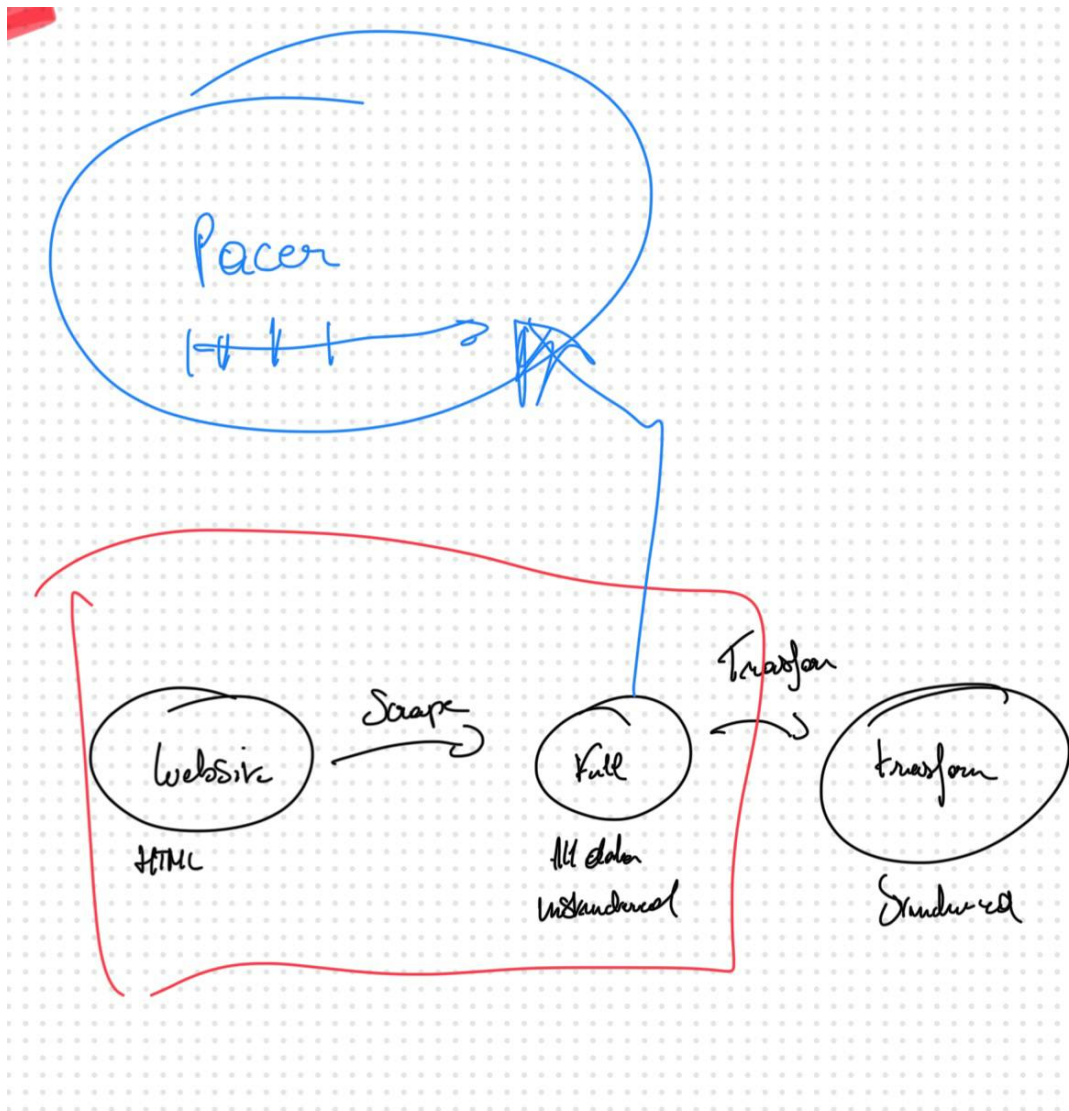
1. **Gebruikersinterface (Django)**
2. **Scrapinglaag (Scrapy spiders)**
3. **Opslag van ruwe data (Kafka topics)**
4. **Transformatielaag (Normalizer)**
5. **Gestructureerde databank (PostgreSQL)**

Figuur 4.2.

4.3. Gebruikersinterface

De gebruikersinterface vormt het enige directe contactpunt tussen de gebruiker en het systeem. Via deze interface kan de gebruiker producten opzoeken of een lijst van producten invoeren die vergeleken moeten worden over meerdere supermarkten. De gebruiker heeft geen zicht op, noch controle over, de onderliggende scraping- en verwerkingslogica; deze wordt volledig door het systeem afgehandeld.

De implementatie van de gebruikersinterface is gebaseerd op het Django-framework, dat fungeert als centrale coördinatielaag tussen de frontend, de databank en de asynchrone scraping-infrastructuur. Django beslist op basis van bestaande data of



Figuur 4.2: Overzicht van de voorgestelde prototype-architectuur

een scraping-opdracht noodzakelijk is en fungeert daarbij als Kafka-producer. Indien data ontbreekt of verouderd is, publiceert Django een bericht naar het Kafka-topic `scrape_jobs`. De verdere verwerking verloopt volledig asynchroon en zonder blokkering van de webserver.

4.3.1. Rol van Django binnen het systeem

Binnen dit project vervult Django een dubbele rol. Enerzijds verzorgt het framework de presentatie van gegevens via HTML-pagina's en API-endpoints, anderzijds fungeert het als orkestrator die scraping-opdrachten initieert en de levenscyclus van zoekopdrachten bewaakt. Django vormt hiermee het centrale knooppunt van het systeem.

4.3.2. Data-architectuur en modellering

Om frequente prijsupdates te ondersteunen en tegelijkertijd een duidelijke relatie tussen producten en winkels te behouden, werd de databank opgebouwd rond een genormaliseerd datamodel:

- **Shop:** representeert een retailer (bv. Albert Heijn, Carrefour, Colruyt).
- **Product:** stelt een generiek product voor, los van een specifieke winkel (bv. “volle melk”).
- **ShopProduct:** vormt de verbindende entiteit tussen een Shop en een Product. Hier worden de effectieve prijsgegevens opgeslagen, waaronder `price_amount`, `price_per_unit`, `source_url` en `scraped_at`.

Om dataconsistentie te garanderen werd een uniciteitsconstraint toegepast op de combinatie van product en winkel, zodat per winkel maximaal één actuele prijs per product wordt bijgehouden.

Daarnaast werden de modellen `SearchQueryRun` en `SearchResult` geïntroduceerd om zoekopdrachten en bijhorende resultaten historisch bij te houden. Deze modellen maken het mogelijk om herhaalde zoekopdrachten binnen een bepaalde tijdspanne te herkennen en onnodige scraping te vermijden.

4.3.3. Zoekworkflow en asynchrone verwerking

Wanneer een gebruiker een zoekopdracht uitvoert via de interface, wordt een vaste beslissingslogica toegepast:

1. Eerst wordt de databank geraadpleegd om bestaande resultaten te vinden die overeenkomen met de zoekterm.
2. Indien resultaten beschikbaar zijn, wordt gecontroleerd of de bijhorende `scraped_at`-timestamp recenter is dan een vooraf gedefinieerde drempel (24 uur).
3. Indien geen resultaten beschikbaar zijn, of indien de data als verouderd wordt beschouwd, publiceert Django automatisch een scraping-opdracht naar Kafka.

Deze aanpak zorgt ervoor dat gebruikers altijd snel een antwoord krijgen, terwijl het systeem op de achtergrond zorgt voor het verversen van data. De webserver blokkeert hierbij niet op trage scraping-taken, maar blijft responsief.

4.3.4. Integratie met Kafka

De koppeling tussen de synchrone webinterface en de asynchrone scraping-infrastructuur wordt gerealiseerd via Kafka. Django fungeert hierbij als producer en publiceert berichten naar het `scrape_jobs`-topic. Deze berichten bevatten onder andere de zoekterm en de reden van de scraping-aanvraag.

Achtergrondprocessen, zoals de scraper worker en de normalizer service, consumeren deze berichten en voeren respectievelijk scraping en datanormalisatie uit. Zodra nieuwe data beschikbaar is, wordt de databank geüpdatet zonder tussenkomst van de gebruiker.

Deze ontkoppeling verhoogt de schaalbaarheid en betrouwbaarheid van het systeem, aangezien langdurige scraping-taken de gebruikersinterface niet beïnvloeden.

4.3.5. Gebruikersfunctionaliteiten

De gebruikersinterface biedt meerdere kernfunctionaliteiten:

Productzoekfunctie

Via een zoekpagina kan de gebruiker individuele producten opzoeken. De resultaten tonen prijzen van verschillende winkels, gesorteerd op de laagste prijs per eenheid. Indien data nog niet beschikbaar of verouderd is, wordt automatisch een achtergrondtaak gestart.

Slim winkelmandje

De interface ondersteunt ook het invoeren van meerdere producten tegelijk, gescheiden door komma's of nieuwe regels. Voor elk product wordt de goedkoopste winkel bepaald. Ontbrekende producten worden automatisch ingepland voor scraping, zonder dat de gebruiker expliciet actie moet ondernemen.

REST API

Naast HTML-pagina's voorziet de applicatie een REST API waarmee de frontend (of externe clients) de status van een zoekopdracht kunnen opvragen. Dit maakt het mogelijk om de interface dynamisch te updaten zonder volledige paginaherlading.

4.3.6. Statusmodellering en gebruikersfeedback

Om transparantie te bieden aan de gebruiker maakt de interface gebruik van expliciete statussen:

Status	Betekenis	Systeemactie
ok	Data is actueel	Resultaten worden direct weergegeven
queued	Data ontbreekt of is verouderd	Scraping-opdracht wordt gestart

4.3.7. Conclusie

De Django-gebaseerde gebruikersinterface functioneert als een toestandsgestuurde coördinatielaag die de volledige levenscyclus van een zoekopdracht beheert. Door

het combineren van caching, asynchrone verwerking en duidelijke gebruikersfeedback slaagt het systeem erin om zowel actuele prijsinformatie als een snelle en responsieve gebruikerservaring te bieden.

4.4. Scrapinglaag

4.4.1. Algemene ontwikkelingsmethodologie van de spiders

De ontwikkeling van elke scraper binnen dit project volgde een systematische en herhaalbare analysemethode, ongeacht de specifieke technologie van de doelwebsite. Deze methode bestaat uit het analyseren van HTML-structuren, het inspecteren van netwerkverkeer en het identificeren van geschikte data-ingangspunten. Door deze gestructureerde aanpak kon per supermarkt de meest geschikte scrapingtechniek worden gekozen.

Analyse van HTML-structuren

De eerste stap in de ontwikkeling van elke spider bestond uit het analyseren van de HTML-structuur van de zoekresultaatpagina's. Met behulp van browserontwikkeltools werden DOM-elementen geïnspecteerd om te bepalen waar productinformatie zich bevindt en of deze server-side dan wel client-side wordt gerenderd. Hierbij werd specifiek gekeken naar herhaalbare HTML-blokken (zoals productkaarten), consistente CSS-klassen en data-attributen die gebruikt worden voor identificatie en tracking.

Volgens Mitchell (2018) vormt DOM-analyse de fundamentele basis van HTML-gebaseerde web scraping, omdat stabiele en semantisch betekenisvolle structuren doorgaans minder frequent wijzigen dan visuele layout-elementen (**mitchell2018webscraping**). Deze observatie werd bevestigd bij de implementatie van de Carrefour spider, waar server-side gerenderde HTML voldoende stabiel bleek voor klassieke parsing met CSS-selectors.

Inspectie van netwerkverkeer

Wanneer HTML-analyse wees op beperkte of ontbrekende data in de initiële DOM, werd de aandacht verlegd naar netwerkverkeer. Via het Network-tabblad van browserontwikkeltools werden HTTP-requests geanalyseerd die uitgevoerd worden bij het laden van zoekresultaten of bij interacties zoals paginatie en scrollen.

Deze stap maakte het mogelijk om verborgen API-aanroepen te identificeren, zoals JSON- of GraphQL-endpoints die productdata rechtstreeks leveren. Dit bleek cruciaal bij de ontwikkeling van de Albert Heijn spider, waar productinformatie niet primair in HTML aanwezig was, maar via een GraphQL-endpoint werd aangeleverd. Het benutten van dergelijke endpoints sluit aan bij aanbevelingen uit de literatuur, waar wordt gesteld dat directe communicatie met backend-API's robuuster en efficiënter is dan HTML-parsing indien beschikbaar (**almeida2020api**).

Analyse van request- en responsepatronen

Naast het identificeren van endpoints werd ook aandacht besteed aan request headers, HTTP-methodes en payloadstructuren. Door deze parameters correct te repliceren (zoals Content-Type, Origin en Referer) konden requests succesvol nagebootst worden als regulier browserverkeer. Dit verkleint de kans op detectie door eenvoudige anti-botmechanismen en verhoogt de betrouwbaarheid van de scraper (**zhao2021antibot**).

Detectie van client-side rendering

Wanneer noch HTML-parsing noch directe API-aanroepen voldoende data opleverden, werd vastgesteld dat de website afhankelijk is van client-side JavaScript-rendering. In dergelijke gevallen werd een headless browser ingezet om de pagina volledig te renderen alvorens data-extractie plaatsvond. Dit scenario deed zich voor bij Colruyt, waar productkaarten pas na uitvoering van JavaScript en interne API-calls beschikbaar waren.

Onderzoek toont aan dat moderne e-commerceplatformen steeds vaker client-side rendering combineren met anti-botmaatregelen, waardoor headless browsers zoals Playwright of Puppeteer noodzakelijk worden voor betrouwbare scraping (**li2022dynamic**).

Iteratieve validatie en verfijning

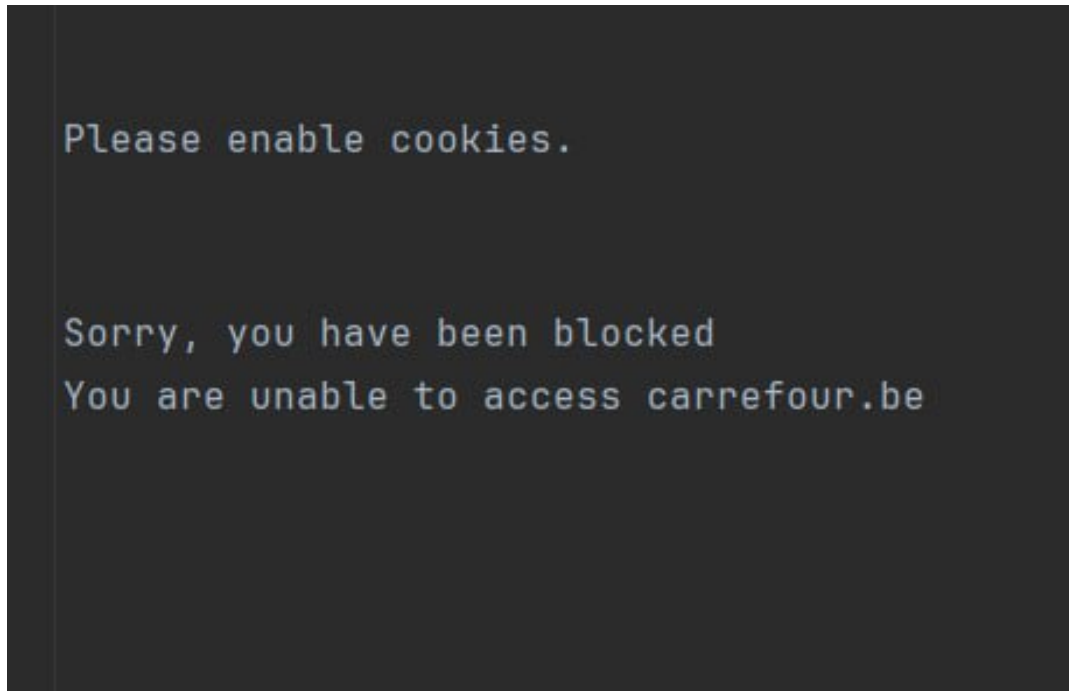
De analysefase werd iteratief herhaald tijdens de ontwikkeling. Wijzigingen in frontendgedrag, time-outs of blokkades leidden tot hernieuwde inspectie van netwerkverkeer en DOM-structuren. Deze feedbackloop maakte het mogelijk om scraping-logica incrementeel te verbeteren en aan te passen aan veranderende omstandigheden, wat essentieel is voor scraping in productieomgevingen (**olston2010web**).

4.4.2. Initiële aanpak en beperkingen

In een eerste experimentele fase werd gebruikgemaakt van de combinatie *Requests* en *BeautifulSoup*. Hoewel deze aanpak eenvoudig te implementeren is, bleek ze in de praktijk onvoldoende robuust. Tijdens tests werden meerdere HTTP-foutcodes waargenomen, waaronder: 403 (Forbidden), 456 (Quota exceeded). Bovendien werd vastgesteld dat bepaalde websites, zoals Carrefour, gebruikmaken van Cloudflare om niet-menselijk verkeer te detecteren en te blokkeren [Figuur 4.3](#).

4.4.3. Overstap naar Scrapy en Playwright

Op advies van de co-promotor werd overgestapt naar het Scrapy-framework. Scrapy biedt uitgebreide mogelijkheden voor request scheduling, foutafhandeling en data-extractie, en is gebaseerd op een asynchrone, event-gedreven architectuur. Door het scrapingproces te mogen isoleren en als een afzonderlijke service of proces uit te voeren, wordt het systeem beter bestand tegen fouten en netwerkproblemen. Hierdoor blijft de rest van de applicatie operationeel, zelfs wanneer scraping taken



Figuur 4.3: Gedetecteerd met BeautifulSoup

falen of tijdelijk onderbroken worden. Daarnast om detectie te vermijden, wordt gebruikgemaakt van *scrapy-impersonate*, waarmee browserheaders en gebruikersgedrag geïmiteerd kunnen worden. Voor websites met dynamische content werd Scrapy gecombineerd met Playwright, zodat JavaScript-elementen correct kunnen worden geladen.

4.4.4. Website-specifieke scrapingstrategieën

Voor de verschillende webshops werden specifieke scrapingstrategieën toegepast:

- **Albert Heijn:** data wordt opgehaald via de GraphQL-API, waarbij JSON-responses rechtstreeks worden verwerkt (JaapWestera, [z.d.](#)).
- **Carrefour:** HTML-scraping met Scrapy en browser-imitatie.
- **Colruyt:** scraping met Scrapy in combinatie met Playwright om JavaScript-gegenereerde content te laden.

4.4.5. Orkestratie van scraping taken

`scraper_worker.py` — Orchestrator

Het script `scraper_worker.py` fungeert als centrale orkestrator van scraping-opdrachten. Dit proces luistert continu naar het Kafka-topic `scrape_jobs` en verwerkt telkens één job tegelijk.

Bij ontvangst van een job:

1. wordt de zoekterm gevalideerd;

2. wordt gecontroleerd of de job recent reeds werd uitgevoerd;
3. wordt de scraping-opdracht gestart via `run_all(query)`.

Door deze centrale coördinatie wordt vermeden dat meerdere workers dezelfde scraping-taak gelijktijdig uitvoeren.

subsubsection `run_spiders.py` — Procesisolatie

Scrapy maakt gebruik van de Twisted reactor, die niet opnieuw kan worden gestart binnen hetzelfde proces. Om dit probleem te vermijden lanceert `run_spiders.py` alle spiders in een afzonderlijk multiprocessing-proces.

Deze aanpak:

- voorkomt reactor-gerelateerde crashes;
- verhoogt de stabiliteit van de worker;
- laat toe om falende spiders te isoleren.

4.4.6. Albert Heijn (AH)

Implementatie van een GraphQL-gebaseerde scraper

In tegenstelling tot webshops die voornamelijk HTML-rendering gebruiken, biedt Albert Heijn productzoekresultaten aan via een GraphQL-laag. Om deze data op een stabiele en performante manier te verzamelen, werd een Scrapy spider geïmplementeerd die rechtstreeks communiceert met het GraphQL-endpoint van AH via HTTP POST requests.

GraphQL-query als contract voor data-extractie

De spider maakt gebruik van een vaste GraphQL-query (`PRODUCT_SEARCH`) die in een hulpfunctiebestand werd ondergebracht. Deze query definieert expliciet welke velden nodig zijn voor het prijsvergelijkingsplatform, waaronder paginatie-informatie en essentiële productvelden zoals titel, merk, prijs en eenheidsprijs. De query werd als string-template geïmplementeerd, waarbij de gebruikerszoekterm dynamisch ingevuld wordt ("%s"). Dit maakt het mogelijk om met dezelfde query-structuur verschillende zoekopdrachten uit te voeren. (Zie `scripts/util.py`.)

Request-opbouw en communicatie met het endpoint

De effectieve scraping gebeurt door het versturen van een POST request naar het GraphQL-endpoint van AH, met als body een JSON payload waarin de query-string wordt meegegeven. De spider voorziet daarnaast specifieke HTTP headers zoals `Content-Type`, `Accept`, `Origin` en `Referer`. Dit zorgt ervoor dat het request voldoende lijkt op normaal browsergedrag en verhoogt de slaagkans in een productieomgeving. (Zie `scripts/ah_spider.py`.)

Parsing en normalisatie van het GraphQL-responseformaat

De response van het endpoint bevat JSON-data met een vaste structuur: `data.productSearch.page` bevat paginatievelden en `data.productSearch.products` bevat de productlijst. Per product worden prijsvelden uitgenest uit `price.now` (actuele prijs) en `price.unitInfo.price` (eenheidsprijs). De spider yieldt vervolgens per product een gestandaardiseerd record met onder meer:

- `id`, `brand`, `title`, `summary`
- `price_amount` (huidige prijs)
- `price_per_unit` (eenheidsprijs)
- `unit` (beschrijving van de eenheid)
- `price_scraped_at` (timestamp)

Deze velden vormen de basis voor verdere normalisatie en opslag downstream in de pipeline. (Zie `scripts/ah_spider.py`.)

Waarom GraphQL (en hoe het schema werd bepaald)

Een praktisch voordeel van GraphQL is dat de client precies kan opvragen welke velden nodig zijn, zonder bijkomende HTML-parsing of meerdere REST-calls. Om de beschikbare queries en datavelden van AH te achterhalen, werd gebruikgemaakt van schema-visualisatie via GraphQL Voyager (**AHGraphQLSchema**) gevonden in de repository (JaapWestera, [z.d.](#)) Daarnaast werd het reverse-engineering proces ondersteund door bestaande documentatie en voorbeeldimplementaties uit een open-source repository die het AH GraphQL-schema via introspection beschrijft en illustreert. `:contentReference[oaicite:4]index=4`

Robuustheid en anti-bot overwegingen

Hoewel deze spider geen headless browser nodig heeft, werd wel rekening gehouden met anti-botmaatregelen. Daarom gebruikt de spider Scrapy-instellingen die typische browser fingerprinting beperken, zoals het gebruik van `scrapy_impersonate` middleware en het uitschakelen van een vaste `USER_AGENT`. Dit verhoogt de betrouwbaarheid van requests naar het endpoint. (Zie `scripts/ah_spider.py`.)

Conclusie

Door rechtstreeks met het GraphQL-endpoint te communiceren, vermijdt de AH scraper de fragiliteit van HTML-scraping en kan productdata efficiënt en consistent verzameld worden. De combinatie van een vaste query-structuur (`PRODUCT_SEARCH`) en gecontroleerde request-headers levert een stabiele basis voor verdere verwerking in de event-driven scraping pipeline.

4.4.7. Carrefour

HTML-gebaseerde scraping met dynamische paginatie

In tegenstelling tot Albert Heijn, waar productgegevens via een GraphQL-endpoint beschikbaar zijn, presenteert Carrefour zijn zoekresultaten hoofdzakelijk als server-side gerenderde HTML-pagina's. Om deze data betrouwbaar te extraheren werd een Scrapy spider ontwikkeld die gebruikmaakt van klassieke HTML-parsing via CSS-selectors, aangevuld met browser-impersonatie om anti-botmaatregelen te omzeilen.

De implementatie van deze spider bevindt zich in `carrefour_spider.py`. :contentReference[oaicite:0]index=0

Zoekgebaseerde URL-opbouw en paginatie

De scraper werkt op basis van een zoekterm die dynamisch wordt meegegeven bij het initialiseren van de spider. De zoekterm wordt geïnjecteerd in de basis-URL van Carrefour, waarna paginatie gebeurt door incrementeel een paginaparameter (`p`) toe te voegen aan de querystring.

De spider start op pagina 1 en blijft nieuwe pagina's opvragen zolang er producten worden aangetroffen. Dit mechanisme maakt het mogelijk om volledige zoekresultaten te scrapen zonder vooraf kennis te hebben van het totaal aantal pagina's.

Detectie van einde van zoekresultaten

Om te vermijden dat onnodige requests worden uitgevoerd, controleert de spider of alle producten reeds zijn weergegeven. Dit gebeurt via een HTML-element dat door Carrefour gebruikt wordt als voortgangsindicator (`div.progress`). De attributen `data-shown-items` en `data-total-items` worden met elkaar vergeleken om te bepalen of verdere paginatie nodig is.

Wanneer het aantal getoonde items gelijk is aan het totaal aantal beschikbare items, stopt de spider automatisch met verdere requests. Deze logica voorkomt overbodige netwerkbelasting en verhoogt de efficiëntie van het scrapingproces.

Browser-impersonatie en anti-bot mitigatie

Carrefour implementeert basisvormen van botdetectie op basis van HTTP headers en browserkenmerken. Om dit te omzeilen maakt de spider gebruik van de `scrapy_impersonate` download handler en middleware.

Concreet:

- worden requests uitgevoerd met een willekeurige, realistische browserfingerprint;
- wordt geen vaste `USER_AGENT` gebruikt;
- wordt het `Twisted AsyncioSelectorReactor` expliciet geconfigureerd om compatibiliteit met moderne asynchrone netwerkstacken te garanderen.

Deze aanpak verhoogt de betrouwbaarheid van de scraper aanzienlijk zonder de overhead van een volledige headless browser, zoals bij Playwright.

Extractie en normalisatie van productgegevens

Per productkaart worden relevante velden geëxtraheerd via CSS-selectors, waaronder:

- merk en producttitel;
- actuele prijs;
- eenheidsprijs (prijs per kg/l);
- eenheidsbeschrijving;
- bron-URL en timestamp.

De eenheidsprijs wordt bijkomend genormaliseerd via een hulpfunctie (`price_per_kg_extractor`), die numerieke waarden extraheert uit tekstuele prijsstrings. Numerieke conversies worden afgehandeld via een robuuste `parse_decimal`-functie om fouten door locale-specifieke notaties (komma versus punt) te vermijden. (Zie `util.py`.) `:contentReference[oaicite:1]index=1`

Fouttolerantie en logging

Na het parsen van elke pagina wordt gelogd hoeveel producten succesvol werden geëxtraheerd. Indien een pagina geen producten bevat, stopt de paginatie automatisch. Deze eenvoudige maar effectieve controle voorkomt oneindige loops en vergemakkelijkt debugging tijdens runtime.

Conclusie

De Carrefour scraper illustreert een klassieke maar efficiënte HTML-gebaseerde scraping-aanpak, waarbij performantie en stabiliteit centraal staan. Door het combineren van server-side HTML parsing, browser-impersonatie en slimme paginatiecontrole kan productdata betrouwbaar verzameld worden zonder de complexiteit van een headless browser.

In combinatie met de GraphQL-gebaseerde AH scraper en de Playwright-gedreven Colruyt scraper toont deze implementatie aan dat de scraping-architectuur flexibel genoeg is om verschillende technische ecosystemen binnen één uniform verwerkingsmodel te ondersteunen.

4.4.8. Colruyt

Ontwikkeling en optimalisatie van een robuuste Colruyt webscraper

Tijdens de ontwikkeling van het prijsvergelijkingsplatform bleek het scrapen van productgegevens van de Colruyt-website uitzonderlijk complex. In tegenstelling

tot andere supermarkten maakt Colruyt intensief gebruik van client-side rendering, trage backend-API's en geavanceerde anti-botmaatregelen. Deze combinatie leidde in de initiële implementatie tot frequente `TimeoutError`-exceptions, blokkades en onbetrouwbare dataverzameling.

Om deze problemen structureel op te lossen werd de oorspronkelijke Scrapy-Playwright spider iteratief herwerkt tot een robuuste en fouttolerante scraping-oplossing. De uiteindelijke optimalisaties focussen op navigatiestrategie, browser stealth, resourcebeheer en state handling.

Initiële implementatie

De oorspronkelijke Colruyt-scraper werd geïmplementeerd als een standaard Scrapy spider met ondersteuning van `scrapy-playwright`. De configuratie maakte gebruik van de standaard Playwright-navigatiestrategie waarbij gewacht wordt tot het `load`-event wordt getriggerd, wat impliceert dat alle pagina-assets volledig geladen zijn.

Daarnaast:

- werd de standaard navigatie-time-out van 60 seconden gebruikt;
- waren stealthmaatregelen beperkt tot het overschrijven van `navigator.webdriver`;
- werden alle resources (afbeeldingen, fonts, trackers en scripts) zonder filtering geladen;
- werd elke paginatie-aanvraag behandeld als een nieuwe browsercontext zonder expliciete state-afhandeling.

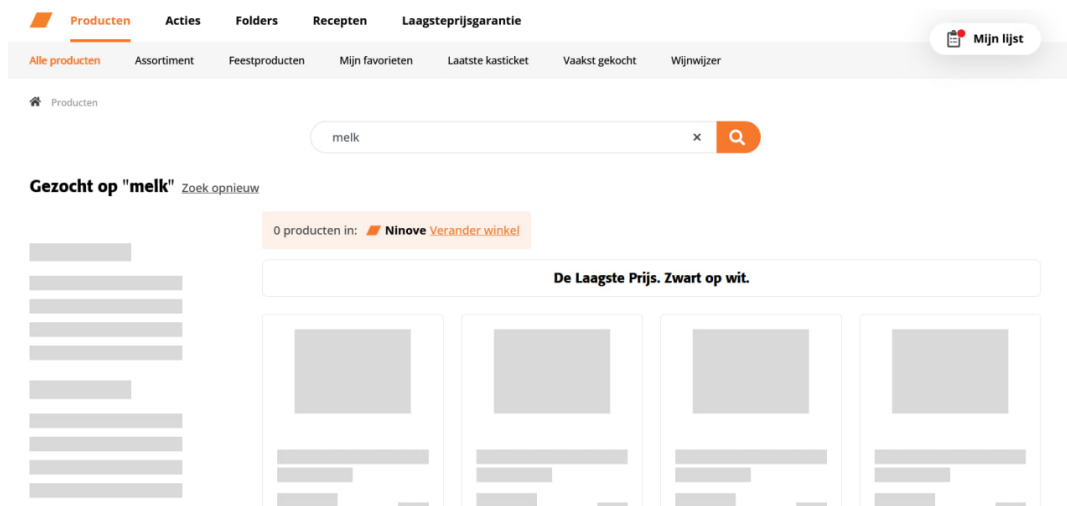
Hoewel deze aanpak functioneel correct was, bleek ze ongeschikt voor stabiele scraping in een productieomgeving.

Geïdentificeerde problemen

Tijdens testen en deployment werden meerdere structurele problemen vastgesteld:

Timeout

Tijdens scraping van Colruyt trad sporadisch een `playwright._impl._errors.TimeoutError` bij de initeele scrape of op bij paginatie (bv. pagina 2 voor zoekterm "ijs"). Dit werd veroorzaakt doordat de Colruyt-website achtergrondprocessen en tracking-API's laadt die traag reageren of soms nooit afronden, waardoor het `load`-event niet wordt bereikt. Daarnaast de fout ontstond in de Playwright-integratie wanneer expliciet werd gewacht tot de productkaarten *zichtbaar* waren: `Page.wait_for_selector("a.card-article[data-tms-product-id]")` met een time-out van 45 seconden. Wat was in de browser te zien is 4.4. In dergelijke gevallen is de pagina wel geladen, maar blijven productkaarten onzichtbaar door trage JavaScript-rendering, overlays (cookie banner) of anti-bot interventies, waardoor de zichtbaarheid-conditie niet wordt voldaan.



Figuur 4.4: Schermafbeelding browser

Om dit robuuster te maken werd de readiness-conditie aangepast: in plaats van te wachten op `visible` werd gewacht op `attached` (aanwezig in de DOM). Dit reduceert false negatives op dynamische pagina's en laat toe om nadien expliciet scrollen wachtacties toe te passen indien lazy-loading vereist is (zie implementatie in de Colruyt spider).

Anti-bot detectie

De website detecteerde de browser als geautomatiseerd verkeer, wat resulteerde in HTTP 456-responses of redirects naar blokkadepagina's. De beperkte stealthmaatregelen waren onvoldoende tegen moderne browser fingerprinting.

Cookie banner-interferentie

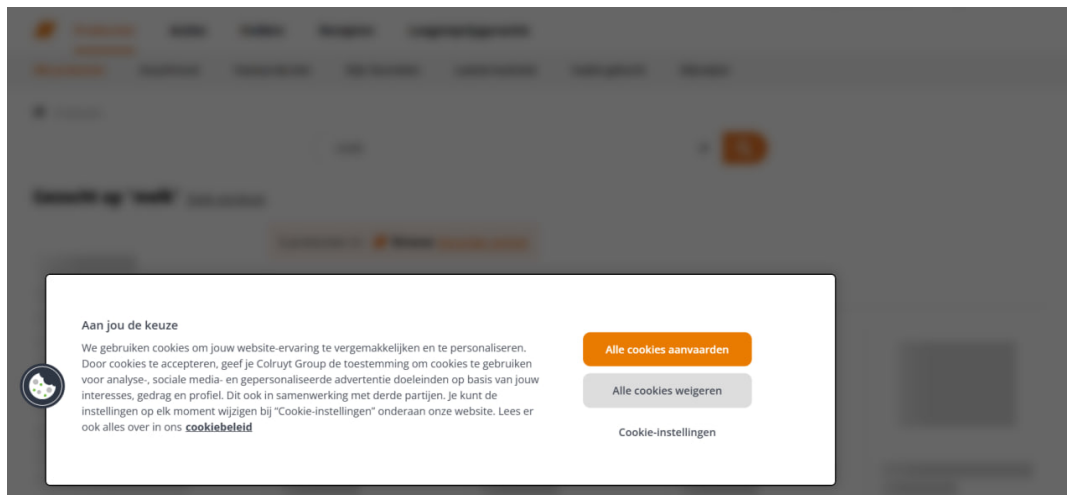
Bij elke nieuwe pagina verscheen een cookie consent overlay 4.5. Omdat Scrapy-Playwright vaak nieuwe browsercontexten gebruikt, bedekte deze banner de productkaarten bij paginatie, waardoor `wait_for_selector`-logica faalde.

Onvolledige rendering

In sommige gevallen rapporteerde Playwright dat de pagina geladen was, terwijl productkaarten slechts als skeleton placeholders aanwezig waren omdat interne JavaScript-logica nog niet voltooid was.

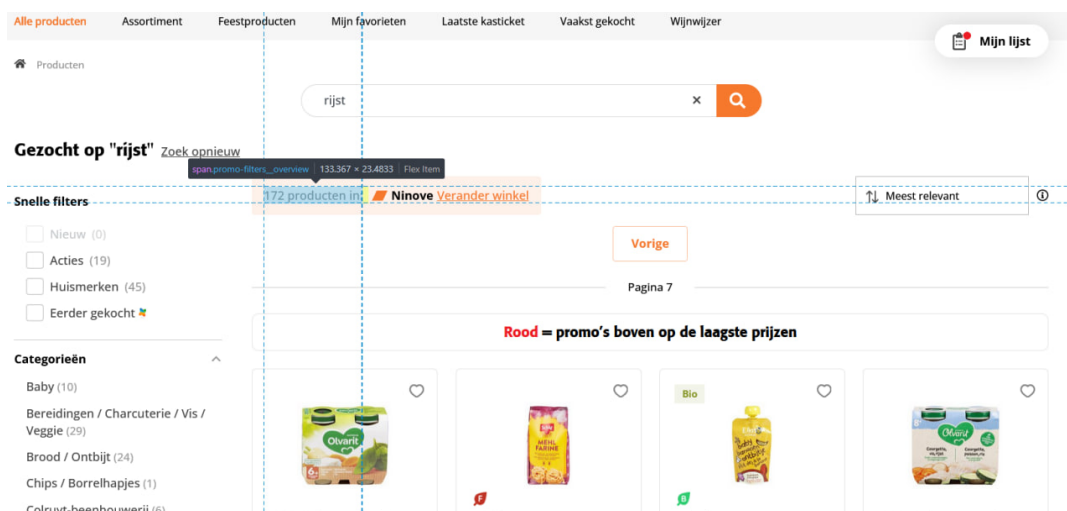
Volgende pagina scraping

Om alle gevonden producten te kunnen scrapen moeten er alle paginas geladen en gescraped worden. Om dat te kunnen doen worden er twee benaderingen gebruikt: 1. Er wordt de aantal van gevonden producten gelezen uit HTML blok `promo_filters__overview` die op 4.6 wordt getoond. 2. Er wordt gekeken of er een knop "Meer bekijken" staat die te zien is op 4.7. Als die niet gevonden is, dan betekent



Figuur 4.5: Schermafbeelding browser

het voor Colruyt spider en voor een gebruiker dat het de laatste pagina met de producten gefilterd met gevraagde product naam.



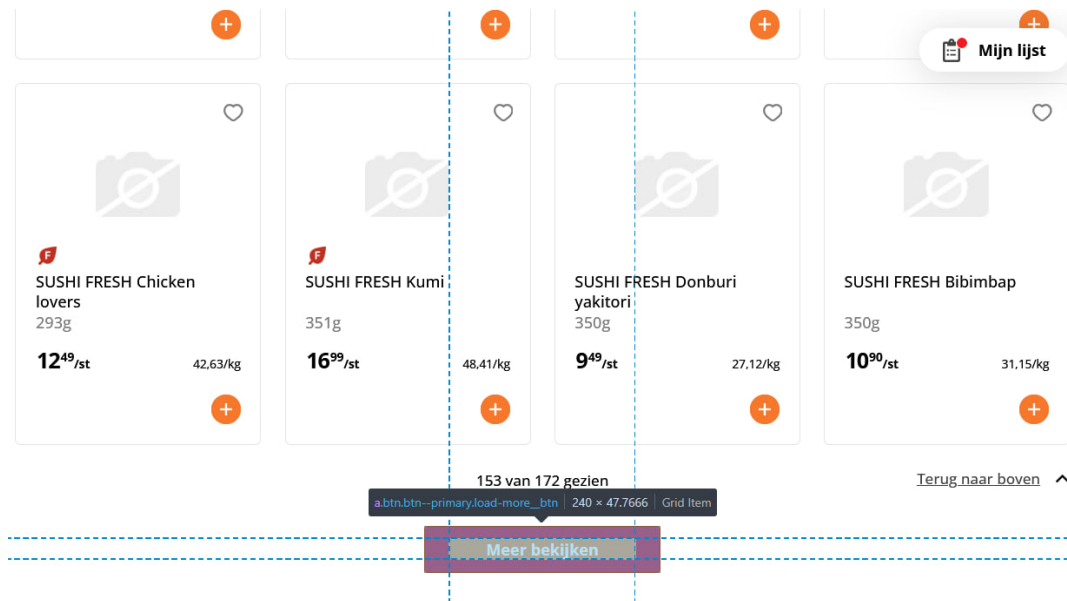
Figuur 4.6: Schermafbeelding browser van Colruyt met HTML element

Geïmplementeerde optimalisaties

1. Geoptimaliseerde navigatiestrategie

De navigatiestrategie werd aangepast door expliciet te wachten op domcontent-loaded in plaats van het load-event. Dit werd gerealiseerd via een aangepaste Playwright-configuratie in de spider (custom_settings):

- `PLAYWRIGHT_DEFAULT_NAVIGATION_TIMEOUT` werd verhoogd tot 90 seconden;
- `wait_for_selector` werd gebruikt met een verhoogde time-out van 45 seconden om te wachten op volledig gerenderde productkaarten (`a.card-article[data-tms-product-id]`).



Figuur 4.7: Schermafbeelding browser van Colruyt met HTML element: button.

Deze aanpak laat toe om de HTML-structuur vroegtijdig te verwerken zonder te blokkeren op niet-essentiële assets.

2. Geavanceerde browser stealth

Om anti-bot detectie te minimaliseren werd een multi-layered stealth-aanpak geïmplementeerd via `PageMethod("add_init_script", ...)`. Hierbij werden onder andere de volgende eigenschappen overschreven vóór uitvoering van pagina-scripts:

- `navigator.webdriver`
- `navigator.languages`
- `navigator.plugins`

Daarnaast werden Chromium launch-argumenten toegevoegd zoals `--disable-blink-features=AutomationControlled`. Hierdoor gedraagt de browser zich vrijwel identiek aan een reguliere Chrome-browser, wat fingerprinting aanzienlijk bemoeilijkt.

3. Selectieve resource filtering

Om laadtijden te reduceren en time-outs te vermijden werd resource filtering toegepast. Niet-essentiële assets zoals afbeeldingen, fonts en externe tracking scripts (bijv. Google Analytics en Facebook trackers) werden geblokkeerd.

CSS-bestanden werden expliciet toegestaan, aangezien bepaalde JavaScript-componenten afhankelijk zijn van layout- en zichtbaarheidberekeningen voor correcte rendering van productkaarten. Deze keuze bleek cruciaal voor betrouwbare dataverzameling.

4. Automatische UI-state afhandeling

Omdat cookie banners bij elke nieuwe browsercontext opnieuw verschijnen, werd een herhaald script geïnjecteerd dat automatisch de “Accept All Cookies”-knop detecteert en aanklikt. Deze logica werd geïntegreerd via `add_init_script` en wordt periodiek uitgevoerd zolang de pagina actief is.

Dit garandeert dat productkaarten steeds zichtbaar blijven, zowel bij initiële pagina's als bij paginatie.

5. Stabiliteit en resourcebeheer

Om geheugenlekken en ophoping van browserprocessen te vermijden werd de parsing-logica omhuld met een `try ... finally`-constructie waarin de Playwright `page` expliciet wordt afgesloten. Dit is essentieel voor langdurige uitvoering binnen een containerized omgeving.

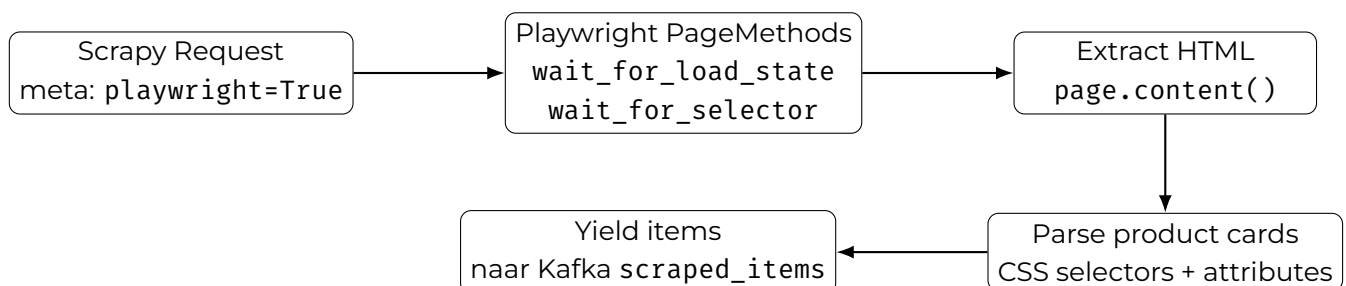
Daarnaast werden expliciete controles toegevoegd voor HTTP 456-responses en geblokkeerde URL's, zodat failures correct gelogd worden in plaats van te resulteren in stille time-outs.

Resultaat

Door deze iteratieve optimalisaties werd de Colruyt-scrapers getransformeerd van een fragiele proof-of-concept naar een stabiele en productieklare scrapingcomponent. De scraper is nu in staat om:

- complexe JavaScript-gedreven pagina's betrouwbaar te verwerken;
- anti-botmaatregelen grotendeels te omzeilen;
- consistente productdata te extraheren;
- langdurig te draaien zonder resource-uitputting.

Deze aanpak minimaliseert blokkades en verhoogt de betrouwbaarheid van het scrapingproces aanzienlijk. Eindarchitectuur is te zien in figuur 4.8.



Figuur 4.8: Colruyt scraping flow met Scrapy-Playwright: requests renderen in een headless browser waarna HTML wordt uitgelezen en geparsed.

4.4.9. Vergelijking van scraping-strategieën

Kenmerk	Albert Heijn	Carrefour	Colruyt
Databron	GraphQL API	Server-side HTML	Client-side JavaScript
Scraping-techniek	HTTP POST (JSON)	Scrapy HTML parsing	Scrapy + Playwright
Rendering nodig	Nee	Nee	Ja (headless browser)
Anti-bot complexiteit	Laag	Gemiddeld	Hoog
Stealthmaatregelen	Headers + impersonatie	Browser impersonatie	Multi-layer stealth + fingerprint spoofing
Paginatie	GraphQL response metadata	URL-parameter (p)	Query-parameter + JS-rendering
Performantie	Zeer hoog	Hoog	Lager
Foutgevoeligheid	Laag	Gemiddeld	Hoog (zonder mitigaties)
Onderhoudbaarheid	Zeer goed	Goed	Complex
Gebruik Playwright	Nee	Nee	Ja

Tabel 4.1: Vergelijking van scraping-strategieën per supermarkt

Analyse en motivatie van de gekozen aanpak

Tabel 4.1 toont duidelijk aan dat voor elke supermarkt een verschillende scraping-strategie werd toegepast, afgestemd op de onderliggende technische architectuur van de website. Deze bron-specifieke benadering was noodzakelijk om zowel performantie als betrouwbaarheid te maximaliseren.

Albert Heijn

Albert Heijn biedt productdata aan via een GraphQL-endpoint, wat directe en gestructureerde toegang tot prijs- en productinformatie mogelijk maakt. Hierdoor is geen HTML-parsing of JavaScript-rendering vereist. De scraper communiceert rechtstreeks met het backend-systeem via HTTP POST requests en ontvangt JSON-responses met een voorspelbare structuur. Deze aanpak resulteert in een zeer hoge performantie en lage foutgevoeligheid, aangezien wijzigingen in de frontend

geen impact hebben op de scraping-logica.

Carrefour

Carrefour presenteert zijn zoekresultaten hoofdzakelijk als server-side gerenderde HTML-pagina's. Hierdoor volstaat klassieke HTML-parsing met CSS-selectors. Omdat er geen complexe client-side rendering nodig is, kan scraping gebeuren zonder headless browser, wat de performantie ten goede komt. Wel zijn beperkte anti-botmaatregelen aanwezig, waardoor browser-impersonatie werd ingezet om requests te laten lijken op regulier gebruikersverkeer.

Colruyt

Colruyt vormt het meest complexe scraping-scenario. De website is sterk afhankelijk van client-side JavaScript, lazy-loading en interne API-calls, gecombineerd met actieve anti-botdetectie. Hierdoor is het gebruik van een headless browser (Playwright) noodzakelijk om de pagina correct te renderen voordat data geëxtraheerd kan worden. Deze aanpak is intrinsiek trager en foutgevoeliger, maar onvermijdelijk om betrouwbare data te verkrijgen. Om dit te mitigeren werden geavanceerde stealthmaatregelen, aangepaste navigatiestrategieën en expliciet resourcebeheer geïmplementeerd.

Overkoepelende conclusie

De vergelijking toont aan dat een uniforme scraping-oplossing onvoldoende zou zijn voor dit project. Door per databron de meest geschikte techniek te kiezen — GraphQL waar mogelijk, HTML-parsing waar voldoende, en headless rendering waar noodzakelijk — wordt een optimale balans bereikt tussen performantie, stabiliteit en onderhoudbaarheid. Deze gelaagde aanpak onderstreept de flexibiliteit en schaalbaarheid van de gekozen scraping-architectuur.

4.5. Opslag van ruwe data

De spiders sturen hun resultaten niet rechtstreeks naar de databank, maar publiceren ruwe JSON-berichten naar het Kafka-topic `scraped_items`. Kafka fungeert hierbij als buffer en ontkoppelingslaag tussen scraping en dataverwerking. Deze keuze biedt meerdere voordelen. Ten eerste kan data opnieuw verwerkt worden zonder opnieuw te scrapen. Ten tweede blijft de oorspronkelijke brondata beschikbaar voor validatie en foutanalyse. Verder maakt deze tussenlaag het systeem tolerant voor tijdelijke fouten en piekbelasting.

Deze stap sluit conceptueel aan bij event-gedreven architecturen waarbij data eerst als onbewerkte events wordt opgeslagen.

4.6. Transformatielaag

De normalizer consumeert berichten van het `scraped_items`-topic en is verantwoordelijk voor:

- productnamen opgeschoond;
- prijsnotaties genormaliseerd;
- eenheden geharmoniseerd;
- irrelevante HTML-elementen verwijderd.

Via `update_or_create` worden bestaande prijsrecords geüpdatet, wat voorkomt dat duplicaten ontstaan bij herhaalde scraping. Door transformatie los te koppelen van scraping wordt het systeem beter onderhoudbaar en uitbreidbaar.

4.7. Gestructureerde opslag

Na normalisatie wordt de data opgeslagen in een PostgreSQL-databank via Django ORM. De databank bevat onder meer: Shop, Product, ShopProduct. Deze structuur laat toe om prijzen per winkel te vergelijken en historische updates bij te houden.

4.8. Optimalisatie van productmatching

4.8.1. Probleemstelling

In de initiële implementatie van de zoek- en winkelmandfunctionaliteit werd productmatching uitgevoerd via een eenvoudige substring-vergelijking op databank-niveau, gebruikmakend van SQL LIKE-operatoren (of Django's `icontains`-filter). Hoewel deze aanpak computationeel efficiënt is en een hoge recall oplevert, bleek de precisie onvoldoende voor betrouwbare prijsvergelijking.

Een typisch probleem hierbij is het zogenoemde *substring collision*-effect. Zo resulteerde een zoekopdracht naar “ijs” (ice cream) ook in producten zoals “rijst”, omdat de substring “ijs” voorkomt in het woord “rijst”. Aangezien het systeem initieel de goedkoopste match selecteerde, leidde dit tot incorrecte resultaten waarbij irrelevante producten als beste optie werden voorgesteld. Dit had een negatieve impact op zowel de nauwkeurigheid van de vergelijking als de gebruikerservaring.

4.8.2. Methodologie: gelaagde similariteitsscore

Om dit probleem te verhelpen werd een aangepaste matching- en rangschikkingslogica ontwikkeld op applicatieniveau. In plaats van een binaire match/no-match-benadering werd een meerlagig scoresysteem geïntroduceerd dat producten rangschikt op basis van hun linguïstische en semantische overeenkomst met de zoekterm.

De matchinglogica bestaat uit drie opeenvolgende lagen, waarbij elke laag een hogere tolerantiegraad heeft maar een lagere prioriteit krijgt in de rangschikking.

Exacte overeenkomst (Laag 0)

De hoogste prioriteit wordt toegekend aan een exacte, hoofdletterongevoelige overeenkomst tussen de productnaam en de zoekterm. Indien beide strings volledig overeenkomen, krijgt het product de laagst mogelijke score (0.0). Dit garandeert dat perfecte matches altijd bovenaan de resultaten verschijnen.

Volledig-woordmatching via reguliere expressies (Laag 1)

Indien geen exacte overeenkomst wordt gevonden, controleert het systeem of de zoekterm voorkomt als een afzonderlijk woord binnen de productnaam. Dit wordt gerealiseerd via reguliere expressies met woordgrenzen (\b).

Deze aanpak vermijdt substringfouten en onderscheidt bijvoorbeeld correct “ijs” in “vanille ijs” van “rijst”, waar “ijs” slechts deel uitmaakt van een groter woord. Bij meerdere geldige matches wordt een lichte voorkeur gegeven aan kortere productnamen, aangezien deze doorgaans generieker en relevanter zijn.

Fuzzy matching met Levenshtein-afstand (Laag 2)

Als laatste vangnet wordt fuzzy matching toegepast op basis van de Levenshtein-afstand. Dit algoritme berekent het minimale aantal karakterbewerkingen (invoegingen, verwijderingen of vervangingen) dat nodig is om de productnaam om te zetten in de zoekterm.

Om te garanderen dat fuzzy matches altijd lager gerangschikt worden dan exacte of volledig-woordmatches, wordt de afstand verhoogd met een vaste offset. Hierdoor blijven zelfs zeer goede fuzzy matches ondergeschikt aan semantisch sterkere overeenkomsten.

4.8.3. Implementatie

De matchinglogica werd geïmplementeerd binnen de Django-applicatielaag, aangezien dergelijke gelaagde ranking moeilijk efficiënt uit te drukken is in standaard SQL-query's. Listing 4.1 toont de kern van het scoringsmechanisme.

Listing 4.1: Gelaagde productmatching met prioriteitsscores

```
def score_match(name: str, query: str) -> float:
    name, query = name.lower(), query.lower()

    # Layer 0: Exact match
    if name == query:
        return 0.0

    # Layer 1: Whole-word match
    if re.search(rf"\b{re.escape(query)}\b", name):
```

```
return 1.0 + (len(name) - len(query)) / 100.0
```

```
# Layer 2: Fuzzy match (Levenshtein)
```

```
return float(levenshtein(name, query) + 2)
```

4.8.4. Impact op systeemfunctionaliteiten

De introductie van het scoresysteem had een directe impact op meerdere onderdelen van de applicatie:

Herordening van zoekresultaten

De zoekpagina haalt nu eerst een beperkte set kandidaatproducten op via `icontains`. Vervolgens wordt voor elk product een similariteitsscore berekend en worden de resultaten gesorteerd op (`score`, `prijs_per_eenheid`). Hierdoor verschijnen de meest relevante producten bovenaan, zelfs indien zij niet de absoluut laagste prijs hebben.

Intelligente winkelmandberekening

Bij het automatisch samenstellen van een goedkoopste winkelmand wordt niet langer blind het goedkoopste product geselecteerd. In plaats daarvan wordt eerst bepaald welke producten tot de beste matchcategorie behoren. Indien enkel lage-kwaliteit fuzzy matches beschikbaar zijn, wordt dit beschouwd als een lage betrouwbaarheidssituatie en wordt automatisch een nieuwe, gerichte scraping-opdracht ingepland.

—

4.8.5. Conclusie

Door over te stappen van een eenvoudige substringvergelijking naar een gelaagde similariteitsscore werd het aantal foutieve productmatches aanzienlijk verminderd. De combinatie van volledig-woorddetectie en Levenshtein-afstand biedt een evenwicht tussen strikte nauwkeurigheid en flexibiliteit voor variabele productbenamingen. Deze aanpak verhoogt zowel de betrouwbaarheid van prijsvergelijkingen als de kwaliteit van de gebruikerservaring.

4.9. Geplande updates

Naast gebruikersgestuurde scraping wordt ook proactieve data-verversing voorzien via `scheduler.py`. Dit script publiceert op vaste tijdstippen (dagelijks om 02:00) scraping-opdrachten voor een set basisproducten.

Hierdoor bevat het systeem steeds recente data, zelfs zonder actieve gebruikers.

4.10. Docker

Alle componenten van het systeem draaien als afzonderlijke Docker-containers:

- Django webapplicatie
- PostgreSQL databank
- Kafka broker
- Scraper worker
- Normalizer
- Scheduler

Via `docker-compose` worden deze containers samen opgestart en via een gedeeld netwerk met elkaar verbonden. Omgevingsvariabelen (zoals databank- en Kafka-configuratie) worden centraal beheerd via een `.env`-bestand.

Deze aanpak garandeert reproduceerbaarheid, vereenvoudigt deployment en maakt horizontale schaalvergroting mogelijk door meerdere workers te starten.

4.11. Conclusie

Het prototype combineert een gebruiksvriendelijke interface met een modulaire en schaalbare backend-architectuur. Door de scheiding tussen scraping, opslag en transformatie sluit het ontwerp nauw aan bij zowel academische literatuur als hedendaagse best practices.

Deze architectuur biedt een solide basis voor verdere uitbreidingen, zoals automatische planning van scrapingtaken, ondersteuning voor bijkomende websites en grootschalige data-analyse.

5

Testen en Resultaten

Om de effectiviteit van het prototype en het onderliggende algoritme te evalueren, werd gekozen voor een praktijkgerichte testmethode gebaseerd op het vergelijken van winkelmanden. Deze aanpak sluit nauw aan bij het beoogde gebruiksscenario van de applicatie, namelijk het vergelijken van productprijzen tussen verschillende winkels.

Een winkelmand wordt hierbij gedefinieerd als een verzameling producten die door een gebruiker geselecteerd worden. Voor elke winkelmand werd het scraping- en verwerkingsproces volledig doorlopen, waarna de totale prijs per webshop werd berekend op basis van de beschikbare productdata.

6

Conclusie

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetur libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem. Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.



Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1. Introduction

In recent years, food prices in Belgium have risen significantly, placing increasing financial pressure on students and other budget-conscious consumers. While several supermarket price comparison tools exist, they generally focus on presenting the lowest prices without considering practical limitations, such as the distance a consumer is willing to travel or the number of stores they can reasonably visit. As a result, these tools provide theoretically optimal solutions that are difficult to implement in real-world scenarios, particularly for students with limited mobility and tight schedules.

Students frequently face challenges in identifying the most cost-effective shopping options. Limited budgets, combined with time and travel constraints, complicate efficient price comparisons across different supermarkets. Existing tools rarely incorporate these constraints, creating a gap between available price information and actionable, user-centered insights. This research addresses this gap by focusing on the development of a system tailored to the needs of students in Ghent.

To develop such a system, the following main research question needs to be answered: How to design and develop a transparent and scalable system, capable of automatically collecting, matching, and comparing supermarket prices in Ghent, that takes into account consumers' distance and limits the amount of shops visited?

To support answering this question and guide the research, an improved understanding of the target audience and the problem context is required. More specifi-

cally:

- What factors currently influence students' consumption habits in Belgium?
- What kind of tools for price comparisons are available in Belgium, and what shortcomings do they have for students?
- What technical and practical challenges are there for collecting price data from Belgian supermarkets?

Furthermore, research into programmatic and architectural details of the candidate solution, and its success factors, needs to be conducted. More precisely:

- How can supermarket price data be automatically collected and structured?
- What approaches can be used to match general product names to achieve accurate comparisons?
- How can the system calculate and recommend the most cost-effective combinations of stores within a user-defined distance and a store limit?
- How can the system architecture be designed to support scalability and transparency of the data?
- What criteria must the prototype meet in order to be considered a valid proof-of-concept?

The result of this research is a prototype, implemented using Python and Django, that collects price data through web scraping from Belgian supermarket websites. Users of this prototype will be able to input a general shopping list and specify a maximum travel distance or a limit on the number of stores. The system will then calculate the most cost-effective combination of stores based on these constraints. The system will be evaluated using a predefined "student shopping cart" to compare the total cost of shopping at a single store versus the optimized multi-store recommendation generated by the system.

This research contributes to the development of a realistic and accessible supermarket price comparison tools that integrate technical efficiency with consumer-centered constraints. By focusing on students' needs, the system aims to enhance price transparency and support informed, budget-conscious shopping decisions, offering practical insights that could inform future consumer-oriented applications.

A.2. Literature Review

A.2.1. Context: food prices and student pressure

Recent Belgian indicators show persistent price pressure on food. According to Statbel's CPI' report, the headline and core inflation remain elevated throughout

2024-2025, with core inflation above 2% in October 2025 (Statbel, [2025a](#), [2025b](#)). Independent tracking by Testaankoop/Testachats likewise reports supermarket specific inflation

around 4% in 2025 (Testaankoop, [2025a](#), [2025b](#)). Broader macro assessments (*OECD Economic Surveys: Belgium 2024*, [2024](#)) show the detailed impact of inflation, confirming price pressure on consumers. Together, these sources substantiate the problem relevance for the price-sensitive groups such as students.

A.2.2. Existing solutions

There are several Belgian specific tools available to help consumers compare supermarket prices and select better products such as PingPrice (PingPrice, [2024](#)) and G4U (G4U, [2025](#)). However, both apps have their limitations. PingPrice compares products using barcodes, which prevents it from effectively comparing store-brand or generic products that lack standardized identifiers. As a result, many relevant items are excluded from comparisons.

G4U, on the other hand, offers extensive product and promotion information, but operates as a paid service, limiting its accessibility for students who already face financial constraints. Consequently, there is a need for a free and transparent alternative that allows users to compare generic product categories, rather than barcodes.

A.2.3. Supermarket data: web scraping as a practical pipeline

Because Belgian retailers rarely expose APIs for product/price feeds to the public, web scraping is a pragmatic way of obtaining structured price data from public pages. While (Logos et al., [2023](#)) and (Brown et al., [2024](#)) guide through ethical and methodological approach to web scraping, they do not propose specific technical implementation for cases where public APIs are not available.

Building on their recommendations, the following process is proposed in this paper: HTML retrieval, parsing of the content, automated browser rendering for JavaScript-dependent content, and storing the price data. This approach to Belgian specific market is inspired by Statbel (Ken Van Loon, [2018](#)) paper.

A.2.4. Legal and Ethical considerations for scraping

Scraping must respect terms of service (ToS), IP, and data-protection constraints. Comparative analyses of website ToS show many platforms explicitly regulate "robots/scrapers", requiring researchers to weigh necessity, proportionality, and compliance mechanisms (Fiesler et al., [2020](#)). Recent overviews propose concrete checklists on legality, ethics, and institutional review: for example, documenting purpose, rate limits, storage, data sharing (Brown et al., [2024](#); Logos et al., [2023](#)). These frameworks guide the governance of the prototype.

A.2.5. Product matching across retailers

Price comparison requires aligning "the same" item across stores despite naming/pack size differences. Literature supports a two-stage approach: 1. exact identifiers (e.g., EAN/GTIN) where available; 2. approximate/semantic matching using fuzzy similarity (Levehnstein/TF-IDF/cosine) or ML embeddings for near-duplicate detection (Kerek, 2020; Ning et al., 2022). These methods map directly from user-supplied product name to store's specific unit.

A.2.6. Decision support, trust, and constraints in grocery apps

Trust is a critical factor influencing user's willingness to adopt digital grocery tools. (Chakraborty et al., 2024) highlights the importance of information credibility, clarity and quality of interaction for building user's trust in online grocery environments. Building on this perspective, (DeZao, 2024) emphasizes trust in AI-powered systems. By showing their data sources and timestamps, these systems become more transparent and consequently are perceived as more reliable and fair by users. Moreover, real-world constraints such as travel distance and ability to visit certain amount of stores affect the usefulness of such tools. Integration of these constraints expands and improves decision support criteria.

In summary, the literature supports a pipeline combining web scraping, reproducible matching (EAN-first + fuzzy/ML fallback), and transparent interfaces that expose source and recency, evaluated on precision/recall for matches and realistic student-centric constraints (e.g. distance, maximum amount of stores) for cost outcomes.

A.3. Methodology

This thesis focuses on the design and implementation of a functional prototype that automatically collects, matches, and compares supermarket prices in Ghent. The research process is divided into five main phases: system design, data collection, data processing and product matching, system implementation and evaluation.

A.3.1. System Design

In the first phase, the system's architecture and data flow were defined to ensure modularity, scalability, and transparency. The architecture consists of three distinct layers. The first layer is called the data layer and manages the storage of product and price information in a PostgreSQL relational database. The second layer is the processing layer and handles web scraping, data cleaning, and product matching logic, implemented in Python. The third layer is the presentation layer, which provides a user interface through a Django web application, allowing users to input shopping lists and define constraints such as travel distance and the number of stores. These design choices were made to support future scalability and the integration of new supermarket and product data.

A.3.2. Data Collection

The data collection phase focuses on gathering daily product and price information from selected Ghent supermarkets. This is achieved using web scraping techniques, using Python libraries such as Requests, BeautifulSoup, and Selenium. Requests is used to send HTTP requests and retrieve the raw HTML content of web pages, giving access to publicly available information without a browser. BeautifulSoup is employed to parse and extract specific elements from the HTML content obtained through Requests. Lastly, Selenium is used for scraping dynamic websites that load data through JavaScript after initial page request.

Each scraper retrieves product names and prices. All data is stored along with additional metadata such as timestamps and data sources to maintain transparency and traceability.

A.3.3. Data Processing and Product Matching

Since supermarkets use different product names and formats, the collected data requires preprocessing before comparison. This phase happens in a number of steps: data cleaning, normalization and product matching. The data cleaning step includes the removal of duplicates and unit normalization (e.g. price per kg or per liter). The matching step implements string similarity algorithms, such as Levenshtein distance and cosine similarity on TF-IDF vectors, to match equivalent products across stores. The filtering step stores the closest product matches to ensure accuracy in comparisons. The result of this phase is a unified dataset where identical or similar products from different stores can be compared directly.

A.3.4. System implementation

The tool integrates following components into a single Django-based web application. Data Scraper runs scheduled scraping jobs and updates the database. Data Processor performs data cleaning and product matching. Comparison Engine calculates the most cost-effective store combinations based on user's constraints. User Interface allows users to input shopping lists and define parameters such as maximum distance and store count. The system is designed for local deployment during testing but can be extended for public use.

A.3.5. Evaluation

The evaluation phase assesses the practical usefulness of the system, using predefined student shopping carts that simulate realistic purchase scenarios. Each cart is analyzed from two perspectives: Single-store shopping (purchasing all items in one supermarket) and Optimized multi-store shopping (purchasing all items using the system's recommendation).

Based on these results, the prototype can be considered a successful proof-of-concept if it is capable of producing cost reductions for Ghent students with various criteria,

while maintaining transparency in its decision process.

A.4. Expected results

The main expected outcome of this research is a functional prototype that demonstrates the feasibility of a system for collecting, matching and comparing product prices. This includes a working scraping module that collects price and data from multiple stores' websites, and a matching module for corresponding items across different shops with a high level of accuracy. After evaluation using predefined student carts, the prototype is expected to demonstrate measurable price difference when compared to shopping in a single supermarket. These results should confirm added value of the system in terms of affordability.

Bibliografie

- Alexander, S. (2015). Distributed Frontera: Web crawling at scale. Zyte. <https://www.zyte.com/blog/distributed-frontera-web-crawling-at-large-scale>
- Brown, M. A., Gruen, A., Maldoff, G., Messing, S., Sanderson, Z., & Zimmer, M. (2024). Web Scraping for Research: Legal, Ethical, Institutional, and Scientific Considerations. <https://doi.org/10.48550/ARXIV.2410.23432>
- Chakraborty, D., Kumar Kar, A., Patre, S., & Gupta, S. (2024). Enhancing trust in online grocery shopping through generative AI chatbots. *Journal of Business Research*, 180, 114737. <https://doi.org/10.1016/j.jbusres.2024.114737>
- DeZao, T. (2024). Enhancing transparency in AI-powered customer engagement. *Journal of AI, Robotics & Workplace Automation Volume 3 Number 2, 2024*. <https://doi.org/10.48550/ARXIV.2410.01809>
- Eyzenakh, D., Rameykov, A., & Nikiforov, I. (2021). High Performance Distributed Web-Scraper. *Proceedings of the Institute for System Programming of the RAS*, 33(3), 87–100. [https://doi.org/10.15514/ispras-2021-33\(3\)-7](https://doi.org/10.15514/ispras-2021-33(3)-7)
- Fiesler, C., Beard, N., & Keegan, B. C. (2020). No Robots, Spiders, or Scrapers: Legal and Ethical Regulation of Data Collection Methods in Social Media Terms of Service. *Proceedings of the International AAAI Conference on Web and Social Media*, 14, 187–196. <https://doi.org/10.1609/icwsm.v14i1.7290>
- G4U. (2025). G4U. <https://g4u-app.com/>
- JaapWestera. (z.d.). Appy. <https://github.com/JaapWestera/albert-heijn-graphql-api/tree/main?tab=readme-ov-file>
- Journal, D. (2024). Javascript vs. Python for Web Scraping. *Data Journal*. <https://medium.com/@datajournal/javascript-vs-python-for-web-scraping-dee9102e56cf>
- Ken Van Loon, D. R. (2018). Webscraping, de verzameling en verwerking van online data voor de consumptieprijnsindex. https://statbel.fgov.be/sites/default/files/files/documents/Analyse/NL/webscraping_nl.pdf
- Kerek, H. (2020). *Product Similarity Matching for Food Retail using Machine Learning* (**publication** Nr. 2020:067) [masterscriptie, KTH, Mathematical Statistics].
- Khder, M. A. (2021). Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application. <https://www.i-csrs.org/Volumes/ijasca/2021.3.11.pdf>
- Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., & Teixeira, J. S. (2002). A brief survey of web data extraction tools. *ACM SIGMOD Record*, 31(2), 84–93. <https://doi.org/10.1145/565117.565137>

- Logos, K., Brewer, R., Langos, C., & Westlake, B. (2023). Establishing a framework for the ethical and legal use of web scrapers by cybercrime and cybersecurity researchers: learnings from a systematic review of Australian research. *International Journal of Law and Information Technology*, 31(3), 186–212. <https://doi.org/10.1093/ijlit/eaad023>
- Majebi, I. (2025). Best Practices for Web Scraping in 2025. <https://www.scrapaperapi.com/web-scraping/best-practices/>
- Ning, W., Cheng, R., Shen, J., Haldar, N. A. H., Kao, B., Yan, X., Huo, N., Lam, W. K., Li, T., & Tang, B. (2022). Automatic Meta-Path Discovery for Effective Graph-Based Recommendation. *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 1563–1572. <https://doi.org/10.1145/3511808.3557244>
- OECD Economic Surveys: Belgium 2024. (2024, september). OECD Publishing. <https://doi.org/10.1787/c671124e-en>
- PingPrice. (2024). PingPrice. <https://www.pingprice.be/en/>
- ProWebScraper. (2018). The 5 Best Programming Languages for Web Scraping. <https://prowebscraper.com/blog/best-programming-language-for-web-scraping/>
- Statbel. (2025a). Harmonised index of consumer prices - December 2024. <https://statbel.fgov.be/en/news/harmonised-index-consumer-prices-december-2024>
- Statbel. (2025b). Consumer price index - September 2025. <https://statbel.fgov.be/en/themes/consumer-prices/consumer-price-index>
- Testaankoop. (2025a). Rundsvlees wordt duur: biefstuk 18% duurder dan vorig jaar, american natuur 17% duurder. <https://www.test-aankoop.be/familie-prive/supermarkten/pers/inflatie-april-2025>
- Testaankoop. (2025b). Inflatie in de supermarkt: net onder 4 % in september. *Heleena Coupette*. <https://www.test-aankoop.be/familie-prive/supermarkten/nieuws/maandelijkse-inflatie-supermarkt>