En modellagnostisk pipeline för strukturerad informationsutvinning från produktbilder med vision-språkmodeller

Examensarbete inom högskoleprogrammet Datateknik, högskoleingenjör

Marcus Karlsson

INSTITUTIONEN FÖR DATA. OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA

Göteborg 2026

www.chalmers.se

Examensarbete 2026

En modellagnostisk pipeline för strukturerad informationsutvinning från produktbilder med vision–språkmodeller

Marcus Karlsson

CHALMERS

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2026

En modellagnostisk pipeline för strukturerad informationsutvinning från produktbilder med vision-språkmodeller
Marcus Karlsson

Handledare: Hanna Ek, Doktorand, Data- och informationsteknik
Handledare: Viktor Jakobsson, Cr34tics AB
Examinator: Jonas Almström Duregård, Computer Science & Engineering

Examensarbete 2026
Data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Skriven i LaTeX
Göteborg 2026

En modellagnostisk pipeline för strukturerad informationsutvinning från produktbilder med vision–språkmodeller
Marcus Karlsson
Data- och informationsteknik
Chalmers Tekniska Högskola

Sammanfattning

Detta examensarbete behandlar utvecklingen av en tjänst för strukturerad informationsutvinning från produktbilder med hjälp av vision-språkmodeller. I detta arbete har tjänsten applicerats på textilprodukter, men den är designad för att vara generell och kan användas för olika typer av bildbaserad informationsutvinning. Syftet är att den extraherade informationen ska användas för att berika sökunderlaget i produkten Pivotise, ett system för sortimentshantering.

Metoden omfattade framtagning av en domänspecifik specifikation för textilprodukter, utveckling av en modulär extraktionspipeline samt utvärdering av olika kombinationer av pipeline- och modellkonfigurationer för att identifiera den mest ändamålsenliga konfigurationen för tjänsten.

Resultaten visar att återhämtningskomponenterna förbättrade stabiliteten och överensstämmelsen med den definierade specifikationen, medan ingen förbättring av informationsextraktionens korrekthet kunde påvisas. Qwen3 med JSON-mode i kombination med återhämtningskomponenterna framstår som den mest lämpliga konfigurationen.

Förord

Jag vill rikta ett särskilt tack till min handledare på Cr34tics, Viktor Jakobsson, för värdefull vägledning och stöd under arbetets gång. Jag vill även tacka övriga medarbetare på Cr34tics för en givande tid och för möjligheten att genomföra detta examensarbete hos dem.

Slutligen vill jag tacka min handledare på Chalmers tekniska högskola, Hanna Ek, för konstruktiv återkoppling och stöd genom hela processen.

Marcus Karlsson, Göteborg, Juni 2026

Innehåll

Figurer

Tabeller

1

Inledning

1.1 Bakgrund

För att upprätthålla en lönsam affärsmodell behöver företag på ett effektivt sätt kunna överblicka och hantera det egna sortimentet. Med målet att optimera denna process har företaget Creatics tagit fram produkten Pivotise, ett system för sortimentshantering. I Pivotse kan användaren på ett effektivt sätt filtrera, söka och jämföra produkter i det egna sortimentet. En begränsande faktor i produkten Pivotise är dock att sökbarheten är beroende av den data som kunden importerar till applikationen. Detta är i sig inte särskilt märkligt eftersom att kunden givetvis önskar nyttja Pivotise för att optimera hanteringen av sitt eget sortiment och behöver således ladda upp produktdata ifrån det egna företaget. Vad Creatics däremot har insett är att många av de produktbilder som kunderna laddar upp är visuellt informationsrika. Om denna information kunde extraheras och nyttjas för att berika sökunderlaget skulle användarupplevelsen troligtvis förbättras. Detta är åtminstone Creatics uppfattning, och det är i huvudsak detta som examensarbetet kommer att adressera.

1.2 Syfte och tillvägagångssätt

Syftet med projektet är att utveckla en tjänst som använder sig av för-tränade vision-språkmodeller (VLM:er) för informationsutvinning. Givet en produktbild och en specifikation av relevanta informationsfält, ska tjänsten använda sig av VLM:er för att extrahera strukturerad utdata i enlighet med specifikationen. Specifikationen fungerar som ett kontrakt mellan modellen och det övriga systemet, med definierade krav på både innehåll och form så att resultatet direkt kan användas av efterföljande system i Pivotise. I detta arbete används tjänsten specifikt för textilbilder, även om den är tänkt för produktbilder i stort.

1.3 Arbetets omfattning

Projektets genomförande är indelat i tre delar. Den första delen är i huvudsak utforskande och syftar till att skapa en förståelse för projektets centrala komponenter

(Avsnitt 4.1 och 4.2). Den andra delen fokuserar på att utforma tjänstens pipeline-struktur, baserat på de insikter som erhölls i den första delen (Avsnitt 4.3 och 4.4). Den tredje delen omfattar utvärdering av både pipeline-strukturens komponenter och tre utvalda VLM:er (Avsnitt 4.5 och 5).

1.4 Mål

- Designa och utveckla en tjänst som använder vision–språkmodeller för strukturerad informationsutvinning från produktbilder.
- Definiera en lämplig specifikation för informationsutvinning om textilprodukter.
- Testa och utvärdera tjänsten tillsammans med olika vision–språkmodeller för att identifiera den mest lämpliga lösningen.

1.5 Avgränsningar

För att tjänsten ska vara kompatibel med Pivotises system krävs att den utvecklas med specifika verktyg och inom en given arkitektur. Det finns därmed inget utrymme att påverka valet av tekniker som används för att konstruera tjänsten, eftersom dessa är förutbestämda.

Vidare ställs krav på vilka typer av vision–språkmodeller som får användas. Den data som tjänsten är tänkt att hanterar utgörs av känslig företagsinformation och måste därför hållas privat. Detta säkerställs genom lokal körning av vision–språkmodellerna via Ollama på Creatics egna servrar, vilket möjliggör full kontroll över datalagring och åtkomst. Följaktligen måste de modeller som kan vara aktuella för tjänsten kunna köras lokalt på Creatics infrastruktur.

2

Metod

Arbetet kommer att genomförs i en iterativ process och delas upp i tre övergripande delar: utforskning, implementering och utvärdering.

Under den utforskande fasen kombineras informationsinhämtning med praktiskt utforskande i syfte att etablera en förståelse för hur för-tränade vision–språkmodeller kan användas för informationsutvinning. Under implementeringsfasen utvecklas den faktiska tjänsten utifrån insikterna från föregående fas. Rent konkret ska tjänsten implementeras i TypeScript och Node.js (Sektion 3.3.1), med möjlighet att kommunicera via RabbitMQ. Under den avslutande delen sker en utvärdering av tjänsten baserat på jämförelse med referensdata för att bedöma kvalitet och tillförlitlighet i informationsutvinningen.

Det modeller som kommer att användas under projektets gång körs lokalt via Ollama (Sektion 3.2). Den specifikation som styr extraheringen är tänkt att implementeras med hjälp av Zod (Sektion 3.4).

3

Teknisk Bakgrund

I följande avsnitt ges en beskrivning av de centrala teorier och begrepp som ligger som till grunden för arbetet.

3.1 Vision-språkmodeller

I projektet används för-tränade vision–språkmodeller (VLM) för att möjliggöra strukturerad informationsutvinning från produktbilder. En VLM kan bearbeta både bild- och textinformation, vilket gör att den genererade beskrivningen baseras på båda typer av indata [1]. Vision–språkmodeller kan implementeras på olika sätt, men en vanlig arkitektur [2], som även ligger till grund för de modeller som utvärderas i detta projekt, består av tre huvudsakliga komponenter: en vision encoder (sv. bildkodare) som omvandlar indata-bilden till en vektorrepresentation, en projektor som mappar den visuella representationen till ett format som kan bearbetas av en stor språkmodell (LLM), samt en för-tränad LLM.

Språkmodellerna implementeras vanligtvis antingen som täta (dense) nätverk, där samtliga parametrar aktiveras vid generering av varje token, eller som Mixture-of-Experts (MoE)-nätverk, där endast en delmängd av parametrarna aktiveras för att effektivisera inferensen (användning av en tränad modell för att göra förutsägelser) [3].

Alla tre modeller([4],[5], [6]) som används i projektet är så kallade open-weights-modeller, vilket innebär att modellens färdigtränade parametrar är offentligt tillgängliga för lokal användning [7]. I projektet används Ollama för lokal körning av de tre open-weights-modellerna som utvärderas. Urvalet av modeller har gjorts i samråd med handledaren. Modellerna presenteras i Tabell 3.1.

Modell	Utvecklare	Arkitektur
Qwen3-VL	Alibaba	MoE
Gemma4	Google	Dense
Ministral-3	Mistral AI	Dense

Tabell 3.1: Modeller som används och utvärderas i projektet, dess utvecklare och arkitektur.

3.2 Ollama

Ollama [8] är ett verktyg framtaget för att underlätta implementering och användning av lokala VLM:er [9, Kap 3]. Många av de aspekter som är kopplade till att köra modeller lokal hanteras automatiskt av Ollama, däribland modellinläsning, resursallokering och modellinferens. Ollama tillhandahåller en mängd olika openweigths-modeller via dess öppna modellbibliotek [10] som kan laddas ner och köras lokalt på den dator där Ollama är installerat.

3.2.1 Modeller och tekniska egenskaper i Ollama

Varje modell förknippas med ett visst antal parametrar, till exempel 8B eller 32B där 'B' står för `billion` och motsvara en miljard parametrar. Parameterantalet syftar på de interna variabler, i huvudsak vikter och biaser, som optimerats då modellen tränats [11]. Varje parameter representeras av ett numeriskt värde med en viss precision, till exempel 16 eller 32 bitar. Mindre modeller är ofta snabbare och mer resurseffektiva än större modeller, men har samtidigt i regel en mer begränsad språkförståelse och resonemangsförmåga [9, Kap 3, s. 18]. I ollamas modellbibliotek finns det inom varje modellfamilj modeller av varierande storlek, till exempel `gemma4:31b` inom Gemma familjen som refererar till en modell med ungefär 31 miljarder parametrar.

De flesta av de modeller som finns tillgängliga hos Ollama distribueras i kvantiserad form. Kvantisering kan beskrivas som en process där den precision som används för att representera en modells parametrar reduceras, från exempelvis 16bitar till 4bitar [9, Kap 3, s. 24]. Kvantiseringen resulterar dels i att modellen upptar mindre lagringsutrymme och arbetsminne vid körning, och dels i en precisionsförlust som försämrar modellens prestanda. Försämringen av prestanda sker dock sub-linjärt med reduceringen av modellparametrarna, och man anser därför att kvantisering i många fall är en rimlig åtgärd [9, Kap 3, s. 24].
Modellerna som går att finna i Ollamas bibliotek finns ofta tillgängliga i olika kvantiseringsalternativ. Kvantiseringen anges i en standardiserad notation, exempelvis `Q4` eller `Q8`, där siffran, grovt sett, representerar modellens precision i antal bitar.

Utöver skillnader i storlek varierar även de sätt som modellerna har blivit tränade på. En basmodell tränas till att förutsäga nästa token baserat på en viss prompt och tidigare genererade tokens [9, Kap 3, s. 2], vilket leder till en bred språklig förståelse men inte nödvändigtvis en förmåga att konsekvent följa givna instruktioner eller generera strukturerad utdata [12]. För att åstadkomma detta finjusteras modellerna genom att tränas på ytterligare data bestående av instruktion-svar-par, vilket lär modellen att tolka och besvara naturligt språk på ett mer förutsägbart sätt [12].

I Tabell 3.2 nedan visas en utökad version av Tabell 3.1 ovan, med tillägg av parameterantal, kvantisering och huruvida modellerna är instruktions justerad.

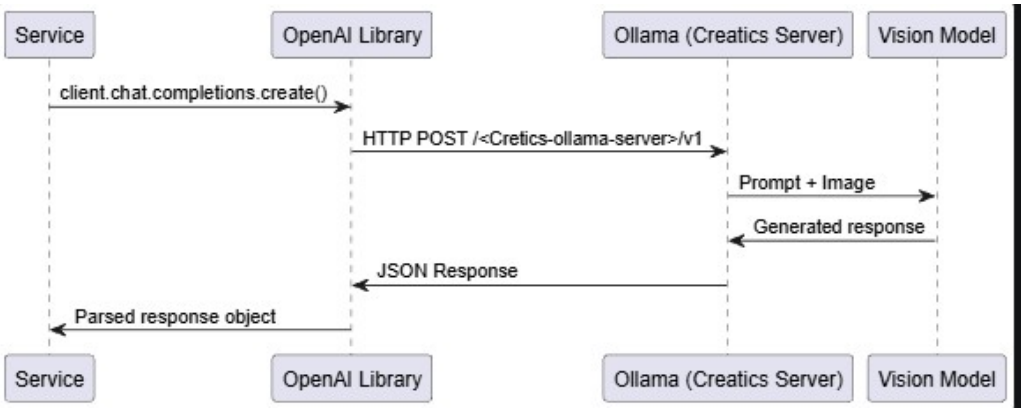
Modell	Utvecklare	Arkitektur	Parametrar	Kvantisering	Instr. justerad
Qwen3-VL	Alibaba	MoE	30B	Q4	✓
Gemma4	Google	Dense	27B	Q8	✓
Ministral-3	Mistral AI	Dense	14B	Q8	✓

Tabell 3.2: Tabellen bygger vidare på Tabell 3.1 med ytterligare kolumn för parametrar, kvantisering och instruktions justering

3.2.2 Modellinferens via Ollama

Tjänsten behöver kunna skicka produktbilder och instruktioner till modellerna och därefter hantera det genererade svaret. I detta projekt kommer denna kommunikation att ske via ett HTTP-baserat API, där den tilltänkta tjänsten agerar som en klient och modellerna hanteras separat via Ollama.

I Figur 3.1 nedan presenteras en översiktligt bild av kommunikationsflödet mellan tjänsten och de modeller som körs via Ollama. Samtliga komponenter i kommunikationsflödet kommer nu att beskrivas mer ingående i sektionen som följer.



Figur 3.1: Schematisk skiss över kommunikationsflöde mellan tjänst och VLM. Tjänsten bygger upp och skickar förfrågningar om att analysera bilder till den av Creatics servrar som kör Ollama. Förfrågan skickas via ett bibliotek från OpenAI [13] som automatiskt hanterar formatering och HTTP-kommunikation. Ollama vidarebefordrar därefter förfrågan till den angivna modellen som hanterar inputen och genererar ett svar. Responsen färdas tillbaka samma väg och konverteras av biblioteket till ett strukturerat objekt som tjänsten därefter kan hantera.

Ollama körs som en bakgrundsprocess på Creatics hårdvara och tar emot HTTP-förfrågningar för modellinferens [14]. I detta projekt kommunicerar tjänsten med VLM:erna via Ollamas OpenAI-kompatibla API, som är utformat för att vara förenligt med OpenAIs specifikation vad gäller meddelandestruktur och format [15]. Förfrågningarna från tjänsten följer därmed OpenAIs specifikation och hanteras av Ollamas kompatibilitetslager.

I praktiken sker kommunikationen med modellerna via OpenAIs officiella Typescript och Javascript bibliotek som distribueras via npm(Node Package Manager) under paketnamnet `openai` [13]. Biblioteket exporterar en klass vid namn OpenAI som instansieras med en URL, vilken anger vart förfrågan ska skickas samt en API-nyckel som är ett krav ifrån biblioteket men som ignoreras av Ollama, se Listing 3.1 nedan.

```
import OpenAI from "openai";

const client = new OpenAI({
  baseURL: "https://<Cretics-ollama-server>/v1",
  apiKey: "ollama", // required but ignored
});
```

Listing 3.1: Initialisering av OpenAI-klient för att möjliggöra API-anrop

Klientinstansen tillhandahåller funktioner som motsvarar OpenAIs API och hanterar automatiskt förfrågningsformatering, typkontroll och felhantering. Vilket innebär att behovet av att manuellt hantera HTTP-förfrågningar abstraheras bort.

En förfrågan för analys av en produktbild (Figur 3.2) skapas genom ett anrop till `client.chat.completions.create`, där modell, meddelandelista (inklusive text-prompt och bild) samt valfria modellparametrar anges. En fullständig förfrågan illustreras i Listing 3.2 nedan.

Figur 3.2: Exempelbild på en textilprodukt som ska analyseras. [16]

```
const response = await client.chat.completions.create({
    model: 'gemma3:27b-it-q8_0',
    messages: [
      {
        role: 'system',
        content: 'You are a helpful assistant.',
      },
      {
        role: 'user',
        content: [
          {
            type: 'text',
            text: 'Provide a concise description of this image in one
    ↪ sentence.',
          },
          {
            type: 'image_url',
            image_url: {
              url: `${base64EncodedImage}`,
            },
          },
        ],
      },
    ],
    // additional modelparameters
  })
  })
```

Listing 3.2: En förfrågan om att analsyera textilbilden i Figur 3.2 via klientinstansens. Fältet model specificerar vilken modell som ska hantera förfrågan. Meddelande-arrayen innehåller indata till modellen, där varje meddelande har en definierad roll och innehållstyp. I detta exempel inkluderas både text och bild i användarmeddelandet, vilka tillsammans utgör modellens input

Utöver innehållet i meddelande-arrayen ovan, är det möjligt att inkludera ett flertal olika parametrar [17] i förfrågan som påverkar hur modellen genererar sin output. I Tabell 3.3 nedan listas de modellparametrar som används och utvärderas i projektet.

Parametrar	Beskrivning
temperatur	Ett nummer mellan 0 och 2 som kontrollerar slumpmässighet i modellens output. Ett högre värd leder till en mer slumpartad output, och ett lägre värde till en mer fokuserad och deterministisk output.
seed	Genom att ge seed ett specifikt värde görs ett bästa möjlig försök till deterministisk sampling, så att upprepade förfrågningar med samma parametrar leder till samma output.
response_format	Ett objekt som specificerar vilket format modellen ska svara i enlighet med. För strukturerad output kan de två alternativen `json_schema` och `json_object` användas.
Tool calling	Tool calling möjliggör funktionsanrop från modellen med JSON-formaterade argument [18]. I detta projekt används detta enbart för strukturerad output

Tabell 3.3: Modellparametrar som används under projektet.

Svaret på förfrågan utgörs av ett strukturerat objekt innehållande modellens svar tillsammans med viss metadata (Listing 3.3).

```
{
  id: "chatcmpl-199",
  object: "chat.completion",
  created: 1775380195,
  model: "gemma3:27b-it-q8_0",
  system_fingerprint: "fp_ollama",
  choices: [
    {
      index: 0,
      message: {
        role: "assistant",
        content: "This image depicts a colorful, patterned floral
    ↪ arrangement with yellow and white flowers, green leaves, and a
    ↪ butterfly, set within a light gray border."
      },
      finish_reason: "stop"
    }
  ],
  usage: {
    prompt_tokens: 281,
    completion_tokens: 30,
    total_tokens: 311
  }
}
```

Listing 3.3: Det erhållna svaret ifrån förfrågan i Listing 3.2. Modellens output finns återgiven under `choices[0].message.content`. Objektet usage innehåller information om antalet tokens som behövde bearbetas ifrån användarprompten (prompt_tokens) och antalet tokens som krävdes för att generera ett svar (completions_tokens).

3.3 Utveklingsmiljö och teknologier

3.3.1 Node js, Typecript och Express

Projektet är utvecklat i Typescript och körs via Node.js. Node.js. är en plattformsoberoende open-source körmilj för Javascript och ett populärt val för att bland annat skapa servrar, webb applikationer och backend tjänster [19].

Typescript är ett statiskt typat programmeringsspråk som bygger på Javascript [20]. Typescript kod kan inte exekveras direkt i Node.js utan måste först översättas till Javascript. Genom att använda Typescript kan vissa typer av fel identifieras under kompilering snarare än vid körning.

Express är ramverk för webbapplikationer i Node.js som erbjuder diverse verktyg för routing och HTTP-hantering [21]. I detta projekt används express för att skapa en enklare server under genomförandets första fas för att etablera kommunikation mellan prototyp och användargränssnitt (Sektion 4.1.2).

3.3.2 React och Vite

React är ett komponentbaserat Javascript bibliotek som kan användas för att bygga användargränssnitt [22]. Vite är ett verktyg för frontend-utveckling som tillhandahåller utvecklingsserver och paketering för produktion [23]. I detta projekt används React tillsammans med Vite för att skapa upp användargränssnittet under genomförandets inledande fas (Sektion 4.1.2).

3.3.3 Strukturerad output och validering via Zod

Standardformatet för den output som returneras ifrån många modeller är fri text. Detta format lämpar sig väl då det ska tolkas av en människa, men om outputen däremot ska hanteras av ett annat system kan de finnas specifika krav på både format och förutsägbarhet. För att styra en modell till att följa en viss strukturer kan den uppmanas att generera ett svar i enlighet med ett fördefinierat schema/specifikation.

Specifikationen som ligger till grund för den strukturerade outputen kan konstrueras i olika format, men i detta projekt kommer Zod [24] att användas. Zod är ett Typescript-validerings bibliotek som kan användas för att både utforma specifikationer och validera data. Listing 3.4 visar ett förenklat exempel som illustrerar hur

funktionaliteten används i projektet. Ett och samma Zod-objekt kan agerar som både källa till det schema som modellen förses med och förväntas följa, och till att validera den data som modellen returnerar.

Det API som används under projektet för att kommunicera med modellerna förväntar sig dock att det schema som skickas med i anropet anges i JSON format. Zod-objekten som används för att skapa upp scheman behöver således kunna omvandlas till JSON fomrat, vilket görs via biblioteket `zod-to-json-schema` [25].

```
import { z } from 'zod';
import { zodToJsonSchema } from 'zod-to-json-schema';

const DesignStyleEnum = z.enum(['illustrated', 'realistic', 'abstract'])

const TextileSchema = z.object({
  designStyle: DesignStyleEnum.describe(
    'The dominant visual style of the textile design.',
  ),
  motifs: z
    .array(z.string())
    .min(1)
    .max(5)
    .describe('Recognizable motifs depicted in the textile.'),
})

// jsonSchema is then passed to the model
const jsonSchema = zodToJsonSchema(TextileSchema)

// pretended incorrect model output
const modelOutput = {
  designStyle: 'geometric',
  motifs: [],
}

// Validate the modelouput
const result = TextileSchema.safeParse(modelOutput)

// Zod provides detailed validation errors
if (!result.success) {
  for (const issue of result.error.issues) {
    console.log(issue.path, issue.message)
    // [ 'designStyle' ]  Invalid enum value. Expected 'illustrated' |
    ↪ 'realistic' | 'abstract', received 'geometric'
    // [ 'motifs' ] Array must contain at least 1 element(s)
  }
}
```

Listing 3.4: Exempel på hur Zod kan används för att skapa upp ett objekt/schema som sedan kan användas som både input till modell och för att validera modellens output. I exemplet ovan skapas schemat TextileSchema upp med enum-fältet designStyle och array-fältet motifs. Datatyper och valideringlogik anges via specifika funktioner som till exempel z.string() eller .min(1). Om outputen inte följer de angivna schemat visar parsingresultatets issues vilket fält som avviker och på vilket sätt.

3.3.4 Bildbehandling via Sharp

Sharp [26] är en bildbehandlings-modul utformad för Node.js, som implementeras ovanpå C-biblioteket libvips [27]. Sharp tillhandahåller en mängd olika operationer som kan utföras på bilder via kedjade anrop. Efter en genomförd bearbetning kan

bilden antingen lagras i en utdatabuffert eller i en fil.

De två huvudsakliga Sharp-operationer som används under projekt är Normalisering och Tiling. Normalisering används för att förstärka en bildens kontraster och bygger på ett histogram-baserat tillvägagångssätt [26]. Tiling syftar på att dela upp en bild i ett regelbundet rutnät av mindre bilder, för att sedan låta varje mindre bild analyserad separat av VLM:en. Antalet rutor kan varieras efter behov. En 2x2 tiling, skapar exempelvis fyra delbilder som tillsammans omfattar den ursprungliga bilden. I listing 3.5 nedan ges ett generellt exempel på hur detta används i projektet.

```
import sharp from 'sharp';

// Normalization
const normalizedImage: Buffer = await sharp('textileImage.jpg')
  .normalize()
  .toBuffer();

// Tiling
const topLeftTile: Buffer = await sharp('textileImage.jpg')
  .extract({ left: 0, top: 0, width: 256, height: 256 })
  .toBuffer();
```

Listing 3.5: Exempel på hur Sharp kan användas för normalisering och tiling. I detta exempel extraheras en bildruta av storlek 256x256 pixlar som utgår ifrån bildens övre vänstra hörn (0,0).

3.4 Metriker

I denna sektion introduceras de tre metriker [28] som används för att utvärdera modellernas genererade output mot grundsanningar: accuracy (sv. noggranhet), Jaccard similarity (sv. Jaccard likhet) och $F1$-mått.

3.4.1 Accuracy

Accuracy definieras som andelen korrekta observationer.

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}[\hat{y}_i = y_i]$$

Där N är antalet utvärderade exemplar, $\hat{y}_i$ är modellens genererade svar och y_i är det korrekt värdet. Ett Accuracy-värde av 1 signalerar att samtliga observationer var korrekt och ett värde av 0 att inga observationer var korrekta.

3.4.2 Jaccard Similarity

Jaccardi similarity är ett mått på likheten mellan två olika mängder $(\hat{A}, A)$ och beräknas genom att dela antalet gemensamma element med det totala antalet unika element:

$J(\hat{A}, A) = \frac{|\hat{A} \cap A|}{|\hat{A} \cup A|}$.

Ett Jaccardi-värde av 1 signalerar att mängderna är identiska och ett värde av 0 att det är disjunkta.

3.4.3 F1

$F1$ måttet är användbart för att mäta effektivitet i obalanserade dataset. Måttet baseras på precision och sensitivity (sv. sensitivitet) och beräknas på följande vis:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \quad \text{Precision} = \frac{TP}{TP+FP}, \quad \text{Recall} = \frac{TP}{TP+FN}$$

Där där TP avser sanna positiva, FP falska positiva, TN sanna negativa och FN falska negativa. F_1 värdet varierar mellan 0-1, och ett högre värdet indikerar bättre prestanda.

4

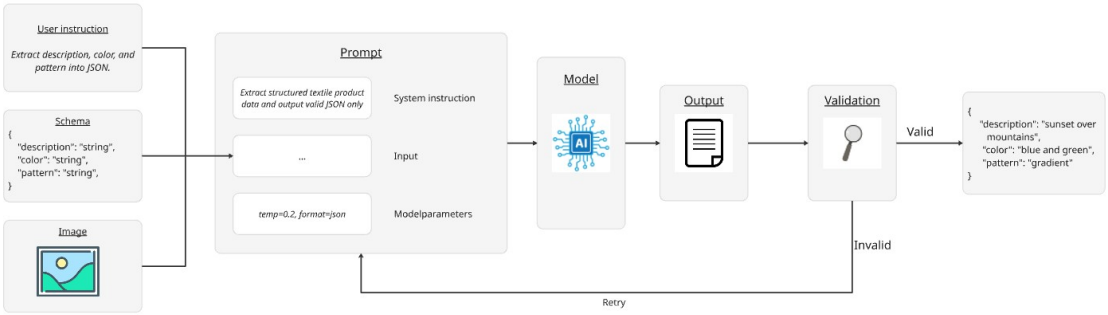
Genomförande

4.1 Prototyputveckling

Uppstarten av projektet skedde i samverkan med den externa handledaren och bestod av ett flertal vägledande genomgångar som syftade till att introducera projektet och dess relation till Pivotise. För att underlätta projektets genomförande togs en översiktlig tidsplan fram, över projektets mest väsentliga inslag. Dessutom sattes utvecklingsmiljön upp i Visual Studio Code med Typescript och Node.js tillsammans med nödvändiga beroenden och konfigurationsfiler. Ollama fanns sedan tidigare installerat på Creatics server, vilket gjorde att modellerna kunde användas direkt via det av handledare rekommenderade API:et (Sektion 3.2), utan behov av ytterligare åtgärder. För att bekanta sig med ovan nämnda teknologier avsattes en första period till att praktiskt och teoretiskt utforska dessa områden. Nedan presenteras de centrala arbetsmoment som genomfördes under denna inledande period, samt vilket lärdomar arbetet gav upphov till .

För att tjänsten skulle vara kompatibel med Pivotise befintliga system var det nödvändigt att den utvecklades i Node.js och med Typescript som programmeringsspråk. En grundläggande förutsättning för projektets genomförande innebar alltså att först skaffa sig god kännedom om dessa teknologier. Båda var sedan tidigare ytligt bekanta, men det krävdes ytterligare fördjupning för att underlätta det kommande arbetet.

Parallellt med detta inleddes ett arbete för att sätta sig in i hur kommunikationen mellan tjänsten och de visionmodeller som fanns nedladdade på Creatics server fungerade. Detta mynnade så småningom ut i en prototyp som successivt utvecklades i takt med att förståelsen för både projektet och tekniken växte. På en övergripande nivå använde sig prototypen av OpenAI:s bibliotek för att kommunicera med modellerna (på samma sätt som beskrivs i Listings 3.2 och 3.3) enklare promptar som uppmanade modellen att analyser textilbilderna utifrån ett givet Zod-schema, Zod-baserad validering (Sektion 3.3.3), och slutligen en enklare logik för att hantera upprepande försök då en output inte överensstämde med det angiva schemats format (på inrådan av handledare). En schematisk skiss av prototypen visas i Figur 4.1 nedan.



Figur 4.1: Schematisk skiss över prototyp. En prompt innehållande instruktioner för extrahering, schema, bild och modellparametrar skickas till modellen. Responsen valideras med hjälp av Zod och baserat på resultatet genomförs eventuellt ett nytt försök till analys.

4.1.1 Olika metoder för strukturerar output

Ett av de krav som fanns på projektet var att tjänsten skulle returnera strukturerad output. Detta var en av det första aspekterna som utforskades och det visade sig i teorin finnas flera olika tillvägagångssätt för att åstadkomma detta. I det API som används under projekt fanns ett inbyggt stöd för två olika varianter [29]. Den första varianten benämns i dokumentationen som Structured Outputs och garanterar i princip att modellen följer det angivna schemat genom en teknik som kallas Constrained decoding [30, 31].Tekniken bygger på att tokens som inte följer det angivna schemat blockeras och därför inte kan genereras av modellen. Ett exempel på hur Structured Outputs kan används i praktiken ges i Listing 4.1 nedan.

```
response_format: {
     type: 'json_schema',
     json_schema: {
       name: 'TextileProductSchema',
       schema: zodToJsonSchema(ProductSchema)
     },
   }
```

Listing 4.1: Exempel på användning av OpenAIs Structured Outputs. Genom att i förfrågan förse modellparametern response_format med typen 'json_schema', ett valfritt namn och ett schema som modellen förväntas följa, kan strukturerad output erhållas. Funktionen zodToJsonSchema används för att omvandla Zod-schemat till JSON format

Den andra varianten kallas JSON Mode och säkerställer att modellen genererar giltig JSON. Det är däremot en svagare typ av strukturerad output eftersom den inte kommer med några garantier på att outputen följer det angivna schemat, som i denna variant inte kan anges i direkt i modellparametern utan måste anges inbyggt i användarprompten i enlighet med Listing 4.2 nedan.

```
response_format: { type: 'json_object' },
messages: [
  {
    role: 'user',
    content: [
      {
        type: 'text',
        text: `Extract information about the textileimage according to this
    ↪ provided JSON schema: ${zodToJsonSchema(ProductSchema)}`,
      },
    ],
  },
],
```

Listing 4.2: Exempel på användning av OpenAIs JSON Mode. Genom att i förfrågan till modellen förse API-parametern response_format med ett object innehållande typen 'json_object', garanteras att outputen är giltig JSON. För att instruera modellen om vilka fält som ska extraheras behöver schemat/specifikationen inkluderas i användarmeddelandet

Det går även att förse modellen med en list av tools för att på så vis erhålla strukturerad output (Tabell 3.3). Och som ett sista alternativ för att generera strukturerad output kan schemat helt enkelt inkluderas i användarprompten utan att förlita sig på någon av det ovan nämnda parametrarna Structured Outputs, JSON Mode eller tools. Då detta testades gjordes iakttagelsen att modellerna inte alltid returnerade giltig JSON även om de uppmuntrades till detta via prompten. I vissa fall var outputen omsluten av ett kodblock som avgränsades med backticks, föregicks av en beskrivande text, eller innehöll mindre syntax fel, se exempel på detta i Listing 4.3 nedan. Att direkt försöka parsa en sådan sträng leder till SyntaxError eftersom att utdatan i sin helhet är inte följer JSON format. För att undvika detta krävs det att den JSON-formaterade strängen som utgör det faktiska svaret, först extraheras ur den övriga utdatan innan dess att den parsas.

```
"Here is the extracted metadata for the product:

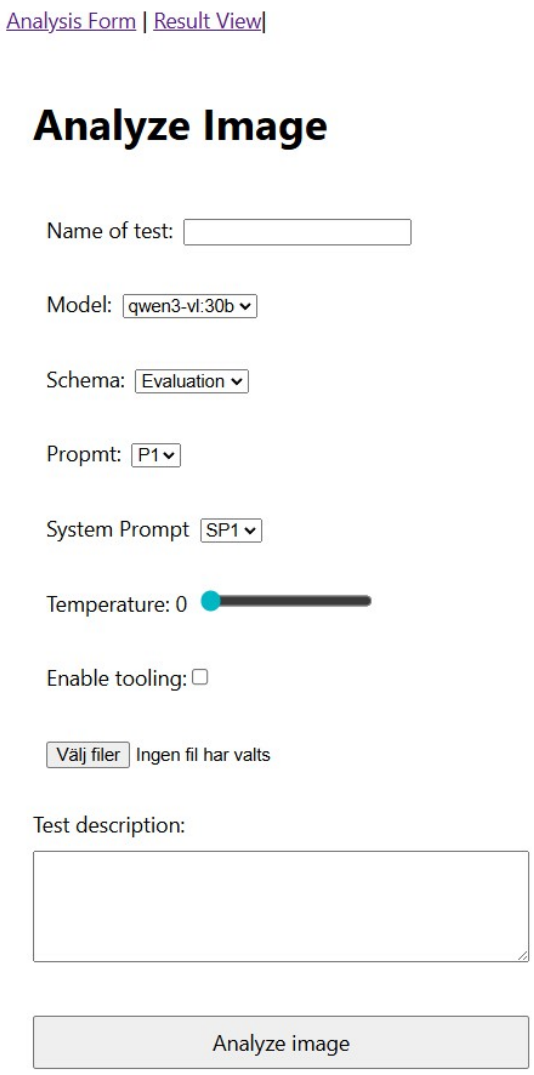
```json
{
  "designStyle": "illustrated",
  "motifs": ["buildings","swan"]
}
```
"
```

Listing 4.3: Exempel på ogiltig JSON-text som returneras ifrån modell då endast användarprompten används för styra modellen till att generera strukturerar ouput. Exemplet innehåller både ett kodblock som anvgränsas med hjälp av backticks och en föregåes av en beskrivande text.

4.1.2 Gränssnitt för visualisering och API-kommunikation

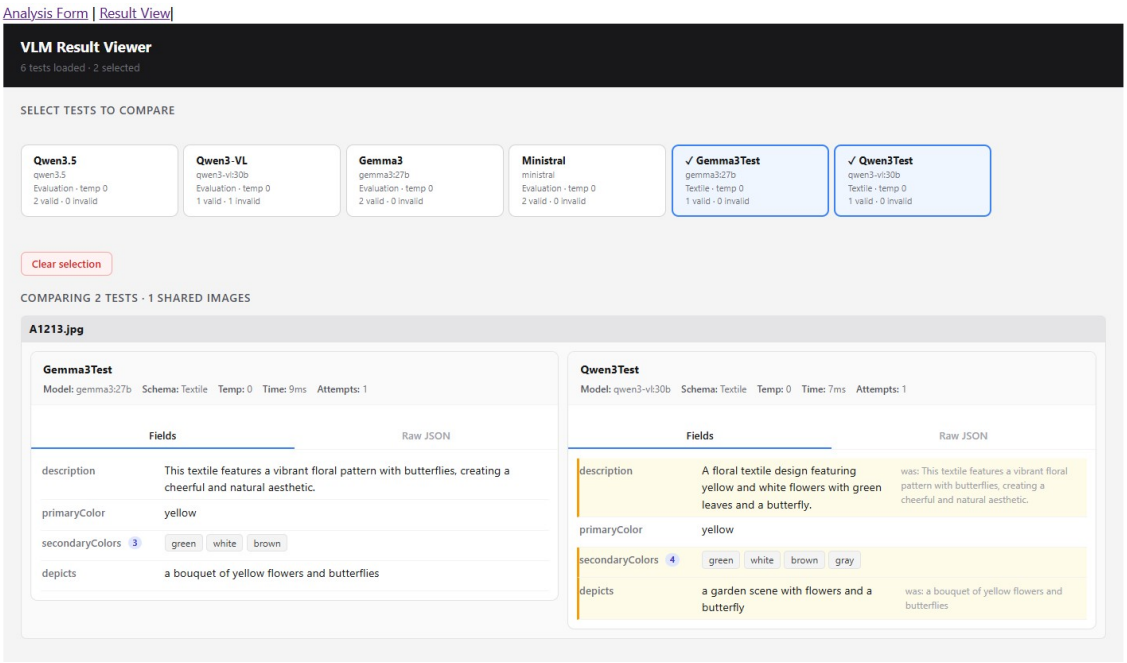
För att enklare kunna testa modellerna emot varandra och därigenom få en känsla för hur de presterade, utvecklades ett enklare gränssnitt med hjälp av Vite och React (Sektion 3.3). Gränssnitten var tänkt att användas både för att underlätta testandet av olika modeller och strategier genom att förenkla konstruktionen av förfrågningar, samt för att underlätta visualisering och jämförelser av modellernas output. För att möjliggöra kommunikationen mellan vyerna i gränssnittet och prototypen, utökades prototypen med en enkel Express server (Sektion 3.3). Utvecklande av gränssnittet kom dessvärre att avbrytas i förtid på grund av att det blev för tidskrävande och endast delvis gick i linje med projektets mål. De ofullständiga vyerna som togs fram under denna period redovisas trotts detta i texten som följer eftersom att det i viss mån bidrog till lärorika erfarenheter och för att de var en del av genomförandet.

Den ena av gränssnittets två vyer utformades som ett formulär och var tänkta att spegla samtliga relevanta delar av API:et. Kommunikationen med modellerna kunde således skötas direkt ifrån gränssnittet utan att aktivt behöva ändra i koden. Formuläret visas i Figur 4.2 nedan.



Figur 4.2: Konfigurationsformulär för API-anrop. Inför varje test görs ett val av modell, schema, prompt, parametrar och vilka bilder som önskas analyseras. Förfrågan skickas därefter till express-servern som vidarebefordrar den till prototypen som utför analys av de valda textil-bilderna. Notera att formuläret inte är komplett.

Gränssnittets andra vy syftade till att presentera och ställa modellernas svar emot varandra för att enklare kunna avgöra hur olika ändringar i diverse parametrar/strategier påverkade outputen. I Figur 4.3 nedan visas denna resultats-vyn då två olika tester jämförs.



Figur 4.3: Resultats-vyn ifrån gränssnitt. Efter att en förfrågan om att analysera en textilbild är upprättad, genom att fylla i formuläret i Figur 4.2, dyker testerna upp i resultat-vyn som små kort, på övre halvan av figuren, innehållande relevant information för det specifika testet. Genom att klicka på ett eller flera test kan modellernas output studeras och jämföras. I exemplet ovan har samma textilbild (A1213.jpg, samma som visas i Figur 3.2) analyserats utifrån samma schema (Textile) av två olika modeller i testerna Gemma3Test och Qwen3Test. Outputen för respektive modell visas i den undre halvan av vyn tillsammans med övrig information, som exempelvis tidsåtgång för analys. De gulmarkerade fälten indikerar att olika svar angivits för samma fält och hur de skiljer sig åt.

4.1.3 Insikter och lärdomar

Genom att utveckla prototypen och gränssnittet byggdes en grundförståelse upp kring hur modellerna påverkades av olika inputs, hur strukturerad output kan erhållas på olika sätt för olika modeller och hur vanligt förekommande problem kunde tas itu med vid extrahering av metadata. Framför allt blev det tydligt att många av de faktorer som tycktes påverkade tillförlitligheten och kvaliteten på outputen var kopplade till variabler som låg utanför själva modellen. Faktorer så som konstruktionen av promptar, hanteringen av felaktigt formaterad output och felbaserad återkoppling.

Andra frågor som också dök upp i samband med utvecklandet av prototypen var relaterade till hur tätt den slutgiltiga tjänsten skulle vara kopplad till en viss specifik modell. Under tiden som arbetet med prototypen fortgick, släpptes 3 nya visionmodeller hos Ollama. Samtliga modellerna framställdes som överlägenas sina före-

gångare och båda kunde vara potentiella alternativ för tjänsten att använda för att analysera textilbilder. Nya och mer kraftfulla modeller kommer med största sannolikhet fortsätta att regelbundet lanseras, vilket kan få tidigare relevanta modellerna att bli inaktuella. Att låta tjänsten bli allt för beroende av en specifik modell, riskerar alltså att leda till en inlåsningseffekt. Samma resonemang kan även illustreras av att en specifik modells prestanda i hög grad kopplad till dess träningsunderlag. Modeller som tränats på olika datamängder kan därmed prestera olika bra på olika områden. Den modell som är bäst lämpad för att extrahera metadata ifrån en viss typ av produktbilder behöver alltså inte vara lika kapabel vid analys av en annan typ av produkt. Om tjänsten ska kunna användas för att analysera andra typer av produkter vore det återigen olämpligt om den vore allt för starkt bunden till en specifik modell. Så för att framtidssäkra tjänsten utifrån dessa två aspekter, skulle det vara fördelaktigt om det enkelt gick att växla mellan olika modeller.

Dessa insikter bidrog till ett viktigt skifte av fokus i projektet. Infrastrukturen kring modellen visade sig alltså vara viktigare än vad som tidigare antagits när det kom till att påverka modellernas output. Och för att framtidssäkra tjänsten vore de även lämpligt om infrastrukturen utformades på ett modell-agnostiskt vis(icke låste till en specifik modell). Lärdomarna ifrån utvecklande av prototypen öppnade på så sätt upp för att ge infrastrukturen en allt mer framträdande roll i både utvärderingsfasen och i den slutgiltiga tjänsten. Detta banade även väg för frågor om vilka övriga externa faktorer och strategier som kunde inkluderas i infrastrukturen kring modellen för att förbättra extraheringen av metadata, och hur dessa på ett systematisk sätt kunde utvärderas tillsammans med modellerna själva.

Därmed blev det även tydligt att den nuvarande designen av prototypen behövde förändras. Eftersom att det var just en prototyp, hade den inte konstruerats i enlighet med programmeringskonstens alla regler. Att lägga till eller ta bort komponenter krävde således stora förändringar i kodbasen på grund av att system var för tätt kopplat. Prototypen hade därmed tjänat sitt syfte som ett verktyg för lärande och ett beslut fattades om att bygga om system från grunden utifrån det insikter som erhållits. Mot denna bakgrund började arbetet med att ta fram en mer välstrukturerad, modulär och modell-agnostisk systemdesign, som var tänkte att kunna användas både under den systematiska utvärderingsfasen och i den slutgiltiga tjänsten. Detta blev startpunkten för den Pipeline-design som kom att bli gällande för tjänsten och vars tillkomst beskrivs i Sektion 4.3.

Vid det här laget hade merparten av uppmärksamheten riktats emot att praktiskt och teoretiskt utforska olika processer relaterade till modelinferens via Ollama. Vad som däremot ännu inte hade adresserats, men som lik väl var viktig för att uppnå projektets mål, var vilken typ av information som var relevant för tjänsten att extrahera, och på vilket sätt output ifrån modellerna kunde utvärderas och jämföras på ett systematiskt vis. En vidare diskussion om detta förs i Sektion 4.2 nedan.

4.2 Identifiering och strukturering av extraherad information

Det schema/specifikation som tjänsten förses med är alltså tänkt att avgöra vilken information som ska extraheras. Den strukturerade utdatan blir följaktligen helt och hållet beroende av schemats innehåll eftersom att det är detta VLM:en extraherar information i enlighet med. För att kunna ta fram ett relevant schema för textilprodukter krävdes först en kartläggning av vilken data som var viktigt för att underlätta navigeringen i Pivotise. Denna process behandlas i Sektion 4.2.1 nedan. Därefter behandlas frågor om på vilket sätt modellers output kunde utvärderas och jämföras på ett kvantitativt sätt i Sektion 4.2.2 Att ta fram ett ändamålsenligt schema blev alltså centralt för att uppnå projekts mål och i sektionerna som följer kommer denna processen att beskrivas i närmare detalj.

4.2.1 Förberedelser

Arbetet inleddes med att klargöra vilken typ av data som var önskvärd att extrahera, det vill säga vilken typ av information som skulle kunna göra sökprocessen mer flexibel i Pivotise, och därigenom förbättra användarupplevelsen. Den vägledande frågan var vilken typ av sökning en användare av applikationen kunde tänkas vilja göra för att hitta en specifik produkt? Många av de textilprodukter som tjänsten hade till uppgift att analysera var både innehållsrika och detaljerade, vilket innebar att mängden information som i teorin gick att söka på var stor och behövdes därför på något vis avgränsas. I Figur 4.4 nedan ett visas ett exempel på två olika textilbilder.

Figur 4.4: Två textilbilder som utgör exempel på bilder som kan behandlas av tjänsten. I figuren till vänster visas en handduk (35x50cm) i 100% bomull tillhörande kollektionen STAD. Till höger visas en bordslöpare (35x120cm) även denna i 100% bomull tillhörande kollektionen SOMMAR. [16].

Genom att undersöka de givna datasetet med produktbilder kunde vissa övergripande attribut så som motiv, layout, design och färger identifieras som relevant för någon som önskar navigera i ett stort sortiment av textilprodukter. Dessa ansågs alltså utgöra en sorts grund till hur olika textilprodukter på ett meningsfullt sätt kunde skiljas ifrån varandra, och blev således relevanta som underlag för specifikationens fält och struktur.

Utöver de kategorier som identifierades på egen hand, förknippade textilföretaget sedan tidigare viss data med varje produkt. Bland annat hade de information om varje produkts typ, material, och vilken kollektion den tillhörde. Denna typ av information var inte nödvändigtvis relevant för den slutgiltiga tjänsten att extrahera eftersom den redan fanns till hands. Men om företagets redan befintliga produktdata inkluderades i schemat kunde detta utgöra en bra grund för jämförelse mellan modellernas output och företagets egen produktbeskrivning. Dessutom var tanken att dessa kategorier kunde fungera som en utgångspunkt som allt eftersom kunde utvecklas till mer nyanserande typer av extrahering.

Genom denna process identifierades total nio olika attribut som kunde ligga till grund för det blivande schemat. Attributen ansågs användbara för en användare som önskar söka eller filtrerar bland en stor mängd textilprodukter. Attributen listas nedan tillsammans med en kort beskrivning av dess innebörd.

- **Motiv**: Igenkännbara motiv
- **Design**: Övergripande visuell stil
- **Layout**: Hur mönstret är uppbyggt/fördelat över ytan
- **Kollektion**: Säsongs- eller temabaserad tillhörighet
- **Produkttyp**: Typ av textilprodukt
- **Färger**: Huvudsakliga färger
- **Kantmönster**: Eventuellt förekommande dekorativ inramning
- **Textinnehåll**: Eventuellt förekommande textinslag
- **Upprepande mönster**: Huruvida designmönstret upprepas eller ej

4.2.2 Utvärdering av output

Kvar återstod bara att utifrån det ovan identifierade attributen skapa upp det schema som tjänsten skulle extrahera information i enlighet med. I detta skede uppstod emellertid vissa frågetecken kring schemats utformning. För att kunna utvärdera olika modeller och dess kringliggande infrastruktur på ett kvantitativt sätt behövdes någon form av konkret referens. Eftersom att attributen togs fram specifikt för de textilprodukter som tjänsten skulle analysera, fanns det inget sedan tidigare existerande referensmaterial tillgängligt. Referensmaterialet behövdes således tas fram manuellt genom att på egen hand annotera en delmängd av textilbilderna.

Detta föranledde framtagandet av ett första grundläggande schema, som avsiktligt begränsades till att innehålla fält av typen enum och boolean. Båda dessa är entydiga och direkt jämförbara med egen-annoterade grundsanningar, vilket möjliggör en standardiserad utvärdering via diverse etablerade utvärderingsmått(Sektion 3.4). Fritext-fält utelämnades helt och hållet eftersom att dessa krävde mer sofistikerade utvärderingsmetoder på grund av det semantiska variationer som det medför. Att uteslutande låta grundschemat bestå av enum-fält bidrar alltså till att lättare kunna jämföra och utvärdera olika konfigurationer. Men det begränsar samtidigt modellernas förmåga att uttrycka sig fritt, och därmed uteblir mer detaljerade och nyanserade beskrivningar, som vore önskvärde i det slutgiltiga schemat.

Den ursprungliga planen om att direkt skapa upp ett komplett och slutgiltigt schemat som kunde användas till både utvärdering och i den slutgiltiga tjänsten justerade alltså i detta skede. Istället utvecklades en första grundläggande version av schemat, som var tänkt att fungera som en slags utgångspunkt för att etablera mätbara prestandajämförelser och där komplexiteten och nyanserna i vad som efterfrågades av modellerna sedan kunde byggas vidare på i efterföljande moment, genom att exempelvis introducera fritext fält.

De fördefinierade värdena togs fram via en iterativ manuell process genom att systematiskt gå igenom textilbilderna i datasetet. För exempelvis fältet motiv, an-

tecknades först alla de huvudsakliga motiv som framgick på de olika textilbilderna. Därefter grupperades motiven in olika kategorier som var tillräckligt omfattande för att täcka den bredd av motiv som förekom, samtidigt som de var tillräckligt tydligt separerade ifrån varandra. Dessa kategorier fick sedan utgöra det möjliga värden som enumfältet motiv kunde ta (Tabell 4.2). Målsättningen var att alltså inkludera värden som kunde användas för att beskriva samtliga produktbilder på ett sätt som fångade dess grundläggande drag och samtidigt enkelt gick att verifiera. I Figur 4.1 nedan visas de fält och fördefinierade värden som motsvarade redan befintlig företagsdata, och i den efterföljande Figur 4.2 visas de egenkonstruerade fälten och dess värden. För varje fält inkluderas även en beskrivning som anger fältets funktion och värdemängd, på samma vis som illustreras Listing 3.4. Dessa har dock utelämnats ur tabellerna nedan i syfte att bibehålla tabellens läsbarhet.

Tabell 4.1: Tabellen visar det av grundschemats fält som tagits fram utifrån textilföretagets redan befintliga produktdata. Kolumnen Fält anger schemats fält, Typ dess datatyp och Värde dess fördefinierade värden.

Fält	**Typ**	**Värde**
`product_type`	enum	`towel`, `tablecloth`, `table_runner`, `placemat`, `plaid`, `pillowcase`, `napkin`, `kids_blanket`, `accessories`
`collection`	enum	`spring`, `easter`, `summer`, `autumn`, `christmas_and_winter`, `swedish_design`, `maritime`, `cities`, `none`

Tabell 4.2: Tabellen visar det av grundschemats fält som tagits fram på egen hand. Kolumnen Fält anger schemats fält, Typ dess datatyp, Villkor om fältet är villkorat och Värde dess fördefinierade värden.

Fält	**Typ**	**Villkor**	**Värde / notiser**
`design_style`	enum		`illustrated`, `realistic`, `abstract`
`layout`	enum		`structured`, `all_over`, `focal`, `scenic`, `other`
`primary_colors`	enum[]	min:1, max:5	`beige`, `black`, `blue`, `green`, `grey`, `red`, `white`, `yellow`, ...
`motifs`	enum[]	min:1, max:3	`animals`, `people`, `plants`, `food`, `buildings_or_structures`, `household_objects`, `boats_or_vehicles`, `symbols`, `natural_scenery`, `none`
`has_border`	boolean		`true` / `false`
`border_color`	enum	Nullable	`beige`, `black`, `blue`, `green`, `grey`, `red`, `white`, `yellow`, ... `null` when `has_border` is `false`.
`text_visible`	boolean		`true` / `false`
`text_content`	string[]	Nullable	Transcribed text strings visible in the design. `null` when `text_visible` is `false`.
`pattern_repeat`	boolean		`true` / `false`

Med basschemat på plats var nästa steg att ta fram grundsanningar, det vill säga korrekta svar om vad bilderna som skulle analyseras faktiskt föreställde. Utan en på förhand given grundsanning, fanns det som tidigare nämnt ingen möjlighet att kvantitativt avgöra huruvida modellerna var korrekta i sina beskrivningar och inte heller någon möjlighet att jämföra dem sinsemellan på ett rimligt sätt. Grundsanningarna för de två olika grupperna i basschemat togs fram ifrån två olika källor. För fälten i Tabell 4.1 fanns produktdatan som tidigare nämnt redan tillgänglig hos textilföretaget, så dessa kunde hämtas därifrån utan att bekymra sig över subjektiviteten som introduceras vid egen annotering. För fälten i Tabell 4.2 däremot, fanns inga korrekta svar att hämta eftersom att dessa fält representerade ny metadata som tjänsten hade för avsikt att bidra med. För dessa fält skapades egna grundsanningar genom att annotera en slumpvis utvald delmängd av bilderna.

Under annoterings-processen granskades varje bild i datasetet, och fälten i basschemat tilldelades ett eller flera av de fördefinierade värden utifrån vad som bäst stämde överens med vad produktbilden faktiskt föreställde. Samtliga bilder annoterades på liknande vis för uppnå en enhetlig annotering. En delmängd av de annoterade urvalet om-annoterades även, utan hänsyn till tidigare klassificering, för att verifiera att processen gav upphov till överensstämmande stabila resultat. Ett exempel på en annotering av den vänstra textilbilden i Figur 4.4 ovan visas i Listing 4.4 nedan.

```
"A1706.jpg": {
   "image_id": "A1706.jpg",
   "annotated_at": "2024-05-08",
   "complexity": "complex",
   "annotation": {
     "product_type": "towel",
     "collection": "cities",
     "designStyle": "realistic",
     "layout": "scenic",
     "motifs": ["buildings_or_structures", "people", "natural_scenery"],
     "primary_colors": ["orange", "yellow", "green", "blue", "grey"],
     "has_border": true,
     "border_colour": "grey",
     "text_visible": true,
     "text_content": ["HAPARANDA"],
     "pattern_repeat": false
   }
 },
```

Listing 4.4: Exempel på en manuell annotering av textilbild utifrån det framtagna grundschemat.

4.3 Pipelinen och dess arkitektur

4.3.1 Designprinciper

Den grundläggande iden var att tjänsten skulle designas som en modulär pipeline(en strukturerad process i flera steg) där visionmodellen utgjorde en fast, men utbytbar roll. Istället för att bygga tjänsten utifrån en specifik modell, skulle alltså modellen behandlas som en av pipelines komponenter som enkelt gick att ersätta utan att resterande kodbas påverkades. Denna ansats speglar de lärdomar som erhölls under konstruktionen av prototypen. Det vill säga att det stödjande processer som omger modellen spelar en central roll för resultatet och att tjänsten bör designas på ett modellagnostiskt vis för att på så sätt enkelt tillåta tjänsten att byta till nyare eller mer lämpade visionmodeller.

För att inte begränsa tjänsten till att uteslutande extrahera information om just textilbilder, var ambitionen att helt och hållet låta det schema som pipelinen förses med avgöra vilken typ av information som skall extraheras, för att på så vis öppna upp för en produktoberoende informationsutvinning. Olika produkttyper förutsätter olika scheman som är anpassade till produktens specifika egenskaper. Ett schema som tagits fram för att extrahera information om textilbilder kommer exempelvis se annorlunda ut än ett som är utformat för teknikprodukter, men icke desto mindre ska tjänsten kunna användas för att extrahera strukturerad information i båda fallen. Det fält, datatyper och villkor som utgöra schemats olika komponenter blir alltså helt avgörande för påverka vilken data som återfinns i den strukturerade outputen som produceras av pipelinen.

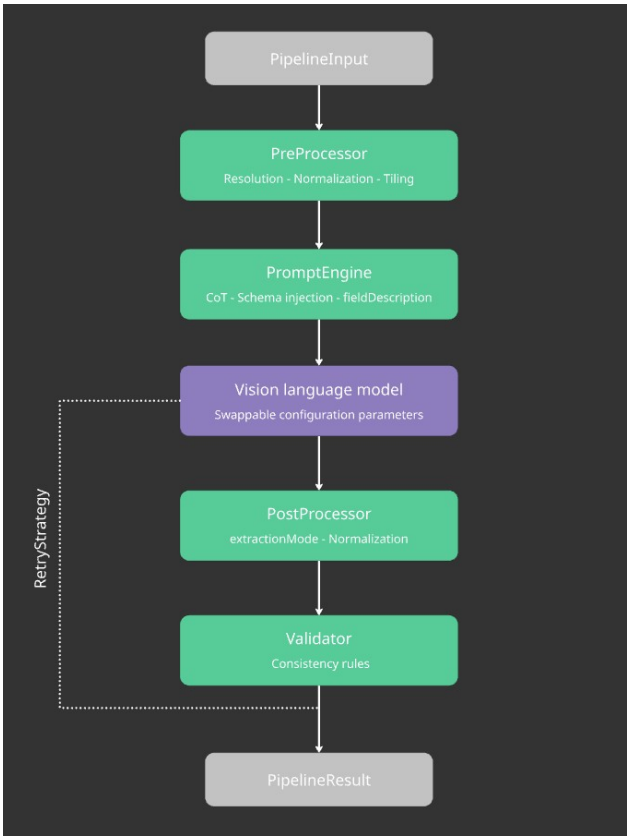
I ett försök att påverka mer än bara formatet, var planen att introducera ett domänspecifikt konfigurations objekt som även detta kunde skickas med då pipelinen skapades. Det domänspecifika objektet var tänkt komplettera schemat genom att förse modellen med kontextuella hjälpregler för att vägleda dess tolkning. Reglernas innehåll och utformning definieras, precis som för schemat, av användaren utifrån den produkt som skall analyseras. Det två funktionerna `aliasMapping` och `consistency rules` som kan aktiveras via detta objekt beskrivs i mer detalj i Sektion 4.4.3 och 4.4.4 nedan.

Själva designen var avsedd att följa en objektorienterad struktur och skulle vägledas av ett antal principer [32]. För det första: löst kopplade komponenter (`lose coupling`). Varje komponent i pipelinen bör vara beroende av väldefinierade gränssnitt istället för konkreta implementationer av övriga komponenter. På så vis kan komponenterna enklare bytas ut eller skrivas om utan att påverka det övriga systemet i allt för stor utsträckning. För det andra: Varje komponent bör ha ansvar för och fokusera på en specifik uppgift (`Single Responsibility Principle`). Ett system med en tydlig ansvarsfördelning blir lättare att underhålla. Och för det tredje: en enhetlig design som kunde användas under både utvärderingsfasen och i den slutgiltiga tjänsten. Genom att göra det enskilda komponenter i pipelinen utbytbara och dess funktionalitet selektivt aktiverbara, kan varje komponents påverkan på

modellens output undersökas separat under utvärderingsfasen, samt att det öppnar upp för en flexibilitet i den slutgiltiga produkten.

4.3.2 Strukturell överblick

Pipelinen som togs fram följer en sekventiell struktur som inleds med en `PipelineInput`, innehållande en referens till den bild som ska analyseras, som därefter genomgår en serie av valfri processer innan den extraherade metadata returneras i form av ett `PipelineResult`-objekt. Valet av komponenter och dess funktionalitet har dels gjorts med stöd i relevant litteratur om hur man åstadkommer goda resultat vid användning av VLM:er för strukturerad output, och dels baserat på de lärdomar som framkom under arbetet med prototypen. Pipelinens strukturen visas i den schematiska skissen i Figur 4.5 nedan.



Figur 4.5: En schematisk skiss över pipelinens struktur. Pipelinen tar emot en bild via `PipelineInput` som därefter stegvis behandlas av olika komponenter för att slutligen returnera ett `PipelineResult` innehållande metadatan om bilden och ytterligare relevant information. Varje komponent (`Preprocessor`, `PromptEngine`, `PostProcessor`, `Validator` och `RetryStrategy`) representerar en valfri och självständigt konfigurerbar process i den sekventiella designen. De specifika funktioner som varje komponent erbjuder anges i mindre text under respektive komponentnamn, för en fullständig beskrivning av dessa se Sektion 4.4.

Varje komponent utgör en valfri och justerbar process. Om en eller flera av komponenterna väljs bort, applicerar pipelinen ett lämpligt standardbeteende. Om exempelvis `PreProcessor` utesluts från pipelinen kommer bildsökvägen, i obearbetad form, att vidarebefordras direkt till modellen. Eller om `Validator` exkluderas, så behandlas all output som modellen returnerar som giltig oavsett om den är strukturerad och följer det angivna schemat eller ej. Denna design bidrar således till att pipelinen är fullt funktionell oavsett om samtliga komponenter är inkluderade eller om endast VLM:en används för sig själv.

4.3.3 Builder Pattern

Varje Pipeline representeras som ett objekt av typen `VisionPipeline`. För att enklare kunna skapa upp olika representationer av VisionPipeline-objekt, det vill säga med olika komponenter och funktioner, används det etablerade designmönstret `Builder pattern` [33], som implementeras via klassen `PipelineBuilder`. Builderklassen tillhandahåller ett gränssnitt med metodkejdning där komponenter och diverse konfigurationsparametrar stegvis kan byggas upp till en pipeline av önskad slag. I Listing 4.5 nedan skapas ett pipeline-objekt upp innehållande samtliga tillgängliga komponenter.

```
const pipeline = new PipelineBuilder(baseConfig, modelClient)
  .withPreprocessor(new Preprocessor({ normalize: true, tiling: false }))
  .withPromptEngine(Prompnew tEngine({ chainOfThought: false, bareFieldList:
    ↪ false, fieldDescriptions: true,}),)
  .withPostProcessor(new PostProcessor({ extractionMode: 'fence',
    ↪ normalization:true, }, baseConfig.domainConfig,))
  .withValidator(new Validator({ consistencyRules: true }))
  .withRetryStrategy(new RetryStrategy(baseConfig.retryOptions))
  .build()
```

Listing 4.5: Exempel på hur ett pipeline-objekt kan skapas upp via designmönstret Builder pattern. Builder-klassen tilldelas två grundläggande objekt via dess konstruktur. Ett konfigurations-objekt (baseConfig) innehållande bland annat domänspecifik information, det schema och den modell som önskas användas, samt ett objekt som fungerar som kommunikationslager mot modellerna (modelClient). Dessa två objekt utgör den minsta möjliga uppsättnig komponenter i pipelinen. En mer komplex pipeline kan därefter byggas upp via kedjade metodanrop där den komponent som önskas adderas instantieras med diverse parametrar i dess konstruktor. Slutligen skapas själva VisionPipeline objektet upp via metoden build().

För att sedan analysera en bild anropas `VisionPipeline`-objektets `run`-metod som förses med ett `PipelineInput` objekt enligt Listing 4.6

```
const result = await pipeline.run({
    imageId: imageID,
    imagePath: imagePath,
  })
```

Listing 4.6: Exempel på hur klassen VisionPipelines körmetoden används för initiera bildanalys. Körmetoden förses med ett objekt av typen PipelineInput som innehåller bildens sökväg och ett för bilden unikt id.

4.4 Pipelinens komponenter

Varje Pipeline kan konfigureras med upp till fem komponenter som var och en ansvara för en specifikt process i den sekventiella extraheringsprocessen. Varje komponent initieras med ett unikt konfigurationsobjekt som påverkar vilken/vilka av dess funktioner som ska vara aktiverade för en viss pipeline, se exempel på detta i Listing 4.5 ovan. I denna sektion kommer komponenterna att beskrivas i mer detalj i den turordning som de körs, se Figur 4.5 för en schematisk skiss över pipelinens struktur.

4.4.1 PreProcessor: Förbehandling av bilder

`PreProcessor` är pipelinens första komponent och har till uppgift att förbehandla textilbilderna innan de når visionmodellen. Det finns stöd för tre olika typer av bearbetning, som samtliga sker med hjälp av Sharp (Sektion 3.3.4). I Listing 4.7 nedan visas den typ som definierar konfigurationsobjektet.

```
type PreProcessorConfig = {
  width?: number
  height?: number
  normalize?: boolean
  tiling?: boolean
}
```

Listing 4.7: Konfigurationobjekt av typen PreProcessorConfig skickas med då PreProcessor instantieras för att påverka val av bildbearbetning. De olika fälten width, heigth, normalize och tiling motsvara olika sätt att bearbeta en bild innan analys.

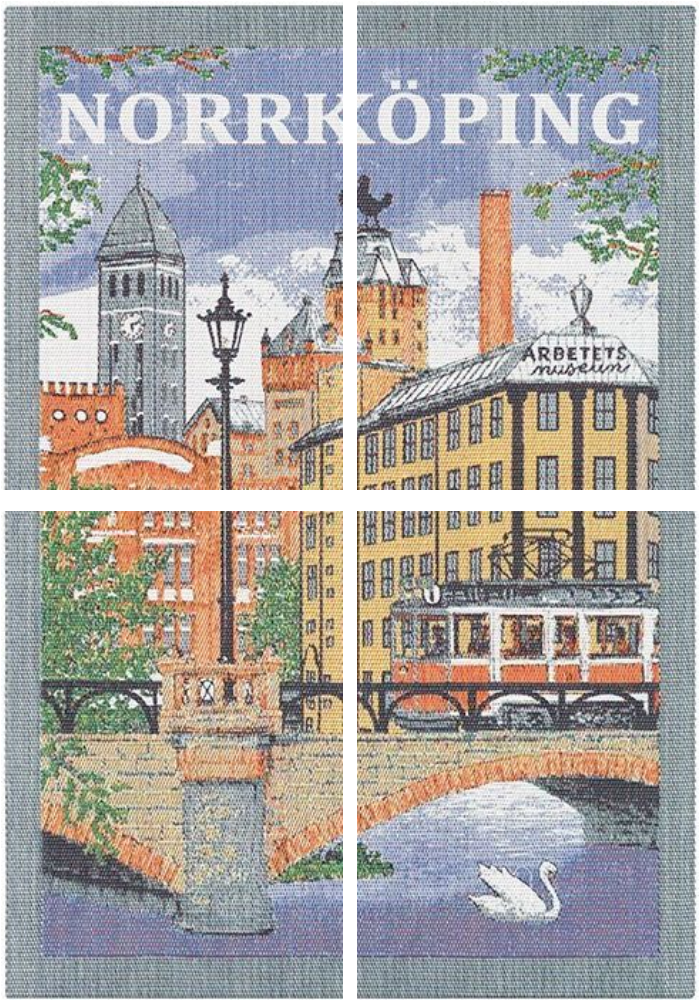
Den första typen(`width` och `height`) avser möjligheten att skala textilbilderna med bibehållna proportioner. Om inga dimensioner anges i konfigurationsobjektet, bibehåller textilbilden sin ursprungliga storlek. Den andra typen av bearbetning är normalisering (Sektion 3.3.4). Genom att skapa tydligare kontraster i textilbilderna är förhoppningen att modellerna bättre ska kunna fånga upp former och strukturer. I Figur 4.6 nedan visas ett exempel på en och samma textilbild där den vänstra bilden är i originalversion och den högra har normaliserats. Skillnaderna är subtila men ändå märkbara.

Figur 4.6: Exempel på användning av normalisering via Sharp.Den vänstra bilden visar textilbilden i originalversion, medan den högra har normaliserats. [16].

Den tredje typen av bearbetning som erbjuds via `PreProcessor` är `tiling` (Sektion 3.3.4). Många av det textilbilder som ska analyseras av tjänsten är väldigt detaljrika, vilket Figur 4.6 ovan är ett exempel på. Genom att dela upp en komplex produktbild i ett rutnät av mindre delbilder och sedan låta modellen analysera dessa var för sig, är förhoppningen att detaljer bättre ska kunna fångas upp. Tillvägagångssätten är inspirerat av [34] men med en förenklad implementation. I Figur 4.7 nedan visas ett exempel på hur bild-`tiling` används i projektet för att dela upp en textilbild i fyra delbilder. För att ge visionmodellen en sammanhängande representation förses den med kontext via prompten enligt Listing 4.8 nedan.

```
1 `You are analyzing a single textile product shown in 5 images.`
2  `Image 1: Full view - use for overall structure and general
     ↪ characteristics.`
3 `Image 2: top_left - use for fine detail in that region.`,
4 `Image 3: top_right - use for fine detail in that region.`,
5 `Image 4: bottom_left - use for fine detail in that region.`,
6 `Image 5: bottom_right - use for fine detail in that region.`,
7 'All images show the same product. Use them together to fill the fields
     ↪ below.',
```

Listing 4.8: Exempel på hur användarprompten formuleras för att ge visionmodell kontext vid användning av tiling. Total fem bilder skickas med i samma förfrågan. Den första bilden(Image 1) motsvara hela textilbilden och det efterföljande bilderna (Image 2-5) motsvara det delbilderna i rutnätet.



Figur 4.7: Exempel på en användning av bild-`tiling`. Textilbilden delas upp i fyra delbilder (top_left, top_right, bottom_left, bottom_rigth) i ett 2x2 rutnät, som därefter analyseras separat av visionmodellen. [16].

4.4.2 PromptEngine: Konstruktion av promptar

`PromptEngine` ansvara för konstruktion av de promptar som skickas till modellen vid varje bildanalys. Samtliga prompt-alternativ har som gemensamt mål att vägleda modellen till att generera strukturerar output i enlighet med ett angivet schema. Det som skiljer dem åt är mängden information och den strukturen som inkluderas i respektive alternativ. `PromptEngine` erbjuder tre olika promptvarianter som kan väljas via konfigurationsobjektet då pipelinen skapas, se Listing 4.9 nedan.

```
type PromptEngineConfig = {
  chainOfThought?: boolean
  fieldDescriptions?: boolean
  rawSchemaInjection?: boolean
}
```

Listing 4.9: Konfigurationobjekt av typen PromptEngineConfig skickas med då PromptEngine instantieras för att påverka val av promptvariant

En prompt inkluderar, som tidigare visats i Listing 3.2, en systemdel och en användardel. Den systemprompten som används under arbetet består av fasta grundläggande instruktioner som inkluderas vid varje anrop oavsett vilken propmtstrategi som valts. Den innehåller generella regler kring hur extraheringen av metadata ska gå till och krav på utdatans format. Se exempel på detta i Listing 4.10 nedan.

```
'You are a metadata extraction assistant. Analyze the provided image and
extract structured data according to the schema.'

    'Rules:',
    '- Return ONLY a valid JSON object. No markdown, no explanation.',
    '- Include every field defined in the schema.',
    '- Base values on what is directly observable in the image.',
    '- Use null only where the schema permits it; otherwise provide your
    ↪ best estimate based on visible evidence.',
```

Listing 4.10: Exempel på hur den statiska systemprompten är utformad. Samma systemprompt skickas med i samtliga förfrågningar oavsett val av propmtstrategi.

Användarprompten är däremot dynamiskt och kan konstrueras på olika sätt. `Chain-of-though` är ett av de alternativ som kan väljas och innebär att modellen instrueras att första beskriva textilbilden som ska analyseras i övergripande ordalag, för att sedan utifrån denna beskrivning returnera ett svar i enlighet med schemat. Detta är en väletablerad strategi som rapporteras kunna förbättra en modells prestation, framförallt då den ställs inför mer komplexa uppgifter [35]. `Chain-of-though` används i detta projekt som ett förberedande steg och ger i sig ingen information om det faktiska schemat som modellen ska följa. Vilket innebär att den måste kombineras med någon av de övriga strategierna för att åstadkomma önskad effekt. I Listing 4.11 nedan visas den sträng som infogas allra först i användarprompten om `Chain-of-thought` aktiveras i konfigurationsobjektet.

```
'Step 1: Carefully observe the image.',
      'Describe what you see in 2-3 sentences - colors, patterns, shapes,
    ↪ themes, and any visible text.',
      '',
'Step 2: Using your observation, extract the metadata fields defined below.',
```

Listing 4.11: Exempel på den sträng som infogas i början av användarprompt om chain-of-though är aktiv. Modellen upmanas i ett första steg att beskriva bilden i allmäna ordalag. Baserat på denna beskrivning instrueras modellen därefter till extrahera metadata i enligehet med det givna schemat.

Det övriga två varianter av användarpromptar representerar olika sätt som schemat

kan presenteras för modellen, och precis som med `chain-of-thought` kan dessa aktiveras och avaktiveras via konfigurationsobjektet i Listing 4.9.

Genom att aktivera `fieldDescriptions` inkluderas en strukturerad och mer läsbar variant av schemat i prompten genom introspektion av zod-schemat. För varje fält beskrivs dess namn, dess typ (samt giltiga alternativ om typen är enum), eventuella villkor och fältbeskrivningar. Denna promptstrategi är tänkt att ge modellen tydliga instruktioner genom att endast plockat ut schemats relevanta delar att utelämna centrala detaljer.

`rawSchemaInjection` är det sista alternativet som finns tillgänglig i `PromptEngine`, och här inkluderas hela schemat direkt in i användarprompten. Denna representation av schemat bearbetas alltså inte alls utan hela schemat så som det är skickas med i användarprompten.

I Figur 4.8 nedan illustreras hur det olika varianterna tar sig uttryck i praktiken. Det textilschema som visas i Listing 3.4 har används som grund och de tre olika exemplen nedan motsvara alltså den sträng som skulle inkluderats i användarprompten för respektive variant.

fieldDescriptions

```
'Fields to extract:

  'designStyle': 'illustrated' | 'realistic' | 'abstract'
  The dominant visual style of the textile design.

  'motifs': string[]  (1-5 items)
  Recognizable motifs depicted in the textile.

  Now extract the metadata from the image and return a valid JSON
    ↪ object.'
```

Listing 4.12: Exempel på användning av promptvarianten fieldDescription. Fält, datatyper, villkor och beskrivningar extraheras ifrån det givna schemat och presenteras för modellen på en mer kompakt form. Promptens tokenmängd: 496

rawSchemaInjection

```
 "Extract the metadata fields defined in the schema below.

{
  "type": "object",
  "properties": {
    "designStyle": {
      "type": "string",
      "enum": ["illustrated", "realistic", "abstract"],
      "description": "The dominant visual style of the textile design."
    },
    "motifs": {
      "type": "array",
      "items": { "type": "string" },
      "minItems": 1,
      "maxItems": 5,
      "description": "Recognizable motifs depicted in the textile."
    }
  },
  "required": ["designStyle", "motifs"],
  "additionalProperties": false
}
Now extract the metadata from the image and return a valid JSON object."
```

Listing 4.13: Exempel på användning av promptvarianten rawSchemaInjection. Hela det schema som som modellen förväntas extrahera information i enlighet med inkluderas i prompten. Promptens tokenmängd: 607

Figur 4.8: En jämförelse mellan det olika promptvarianterna `fieldDescription` och `rawSchemaInjection` baserat på schemat `TextileSchema` i Listing 3.4. Under respektive Listing återges även promptens tokenmängd

Det är även värt att notera att ju utförligare användarprompten är, desto fler tokens krävs för att modellen ska kunna bearbeta inputen. Det textilschema som ligger till grund för det olika promptalternativ som visas i Figur 4.8 ovan, representerar en avskalad version. I praktiken vill man ofta extrahera betydligt mer information vilket leder till att schemats omfång ökar avsevärt. Förhoppningen är att via en jämförelse av modellernas output och token-kostnaden för respektive promptvariant kunna avgöra om den extra kostnaden för en mer informativ prompt leder till ett bättre resultat eller ej.

4.4.3 PostProcessor: Efterbearbetning av utdata

`PostProcessor` ansvarar för att efterbearbeta den del av modellresponsen som innehåller det genererade svaret (Listing 3.3). `PostProcessor` tillhandahåller fyra olika funktioner som samtliga syftar till att justera modellens utdata före validering. Den typ som definierar konfigurationsobjekt för denna komponent visas i Listing 4.14 nedan och kan användas för att styra vilken typ av efterbehandling som utförs.

```
type PostProcessorConfig = {
  extractionMode: 'none' | 'fence' | 'fence_and_repair'
  normalization?: boolean
  aliasMapping?: boolean
}
```

Listing 4.14: Konfigurationobjekt av typen PostProcessorConfig skickas med då PostProcessor instantieras för att påverka efterbehandligen av output före validering

Den första funktionen som går att påverka benämns som `extractionMode` i Listing 4.14 ovan och har att göra med konverteringen av modellens output ifrån en sträng till ett objekt. Under bygget av prototypen gjordes en iakttagelse att om varken API-parametern `Structured Outputs` eller `JSON Mode` inkluderades vid extrahering, kunde modellerna ibland returnera strängar som inte följde JSON-formatet (Sektion 4.1 och Listing 4.3). För att hantera just detta har det två olika lägena: `'fence'` och `'fence_and_repair'` inkluderats. Genom att förse `extractionMode` med strängen `'fence'` matchas modellens utdata mot regex-mönster för att på så vis extrahera den del som innesluts av klammerparenteser ({}) och som förhoppningsvis innehåller giltig JSON. Skulle innehållet i parenteserna däremot inkludera syntaxfel, så som missade kommatecken eller citattecken, räcker det dock inte med att använda `'fence'` eftersom att detta läget inte på något vis kontrollerar huruvida objektet i sig följer JSON-syntaxen. Detta hanteras genom att förse `extraionMode` med strängen `'fence_and_repair'`, som återanvänder samma logik som i `'fence'` men som därefter använder sig av det externa biblioteken `jsonrepair` [36] för försöka att åtgärda eventuella fel i strukturen. Man kan även förse `extractionMode` med strängen `'none'` vilket leder till att modellens genererade utdata direkt konverteras till ett objekt utan att använda varken regex-mönster eller externa bibliotek.

Det övriga funktionerna som erbjuds via `PostProcessor` appliceras först efter att strängen som modellen returnerar har konverterats till ett objekt. Dessa funktioner

har som gemensamt övergripande syfte att standardisera enskilda fält i objektet för att på så vis öka chansen att valideringen lyckas.

Fältet `normalization` i Listing 4.14 ovan används på ett objekts strängbaserade enumvärden för trimma (avlägsna blanksteg) och göra om dem till gemener. Tanken med detta är att därigenom eliminera eventuella valideringsfel till följd av skillnader i stora/små bokstäver eller i omgivande blanksteg mellan modellens utdata och ett schemas på förhand givna enumvärden. Som exempel på detta kan vi se att samtliga enumvärden som är angivna i Baschemat, i Tabell 4.2, är gemener. Om en output ifrån en modell exempelvis innehåller värdet `Natural_Scenery` för fältet `motifs`, skulle valideringen för detta fält i normala fall misslyckas eftersom att det inte är en exakt överensstämmelse med det faktiska enumvärdet natural_scenery, även om den semantiska betydelsen är densamma. Genom att aktivera funktionen `normalization` kan alltså sådana mindre avvikelser undvikas.

Även fältet `aliasMapping` kan aktiveras för att påverka svarsobjektets strängbaserad enumvärden. I detta fall har inspiration hämtats ifrån en metod vid namn `Fuzzy-matching` [37], som i korthet kan beskrivas som en strategi för approximerad matchning. Här används metoden i en något annorlunda och förenklad variant genom att förse det domänspecifika konfigurations objektet `domainConfig` med en slags uppslagsstruktur som sedan används för att mappa ett semantiskt korrekt modellsvar till ett syntaktiskt korrekt enumvärde. Ett exempel på detta visas i Listing 4.15 nedan. Anta exempelvis att det i modellens utdata för fältet `collection` var angivet `'fall'` istället för `'autumn'`. Båda svaren syftar på samma årstid men bara ett av dem skulle i normal fall anses korrekta i enlighet med basschemat, se Tabell 4.1 och 4.2. Genom att förse det domänspecifika objektet med en uppslagsstruktur för just `collections` kan alltså det semantiskt likvärdiga svaret `'fall'` mappas till det syntaktiskt korrekta svaret och därigenom accepteras som korrekt trots det ursprungliga syntaktiska skillnaderna.

```
const ALIAS_LOOKUP: Record<string, Record<string, string>> = {
collection: {
    'christmas': 'christmas_and_winter',
    'xmas': 'christmas_and_winter',
    'fall': 'autumn',
    'city': 'cities',
  },
...
}
```

Listing 4.15: Exempel på den struktur som används för aliasMapping. Den yttersta nyckeln (collection) motsvara ett av schemats fält, värdet utgörs av ett objekt som i sin tur består av nyckel-värde par där nycklen är ett potentiellt svar ifrån modellen och värdet det korrekta enumvärdet.

Efter att modellens output har omvandlas från sträng till objektrepresentation och genomgått diverse bearbetningssteg skicka den vidare till `Validator` för att valideras.

4.4.4 Validation: Validering av utdata

Komponenten `Validation` används för att validera utdatan, som antingen kommer direkt ifrån visionmodellen, eller som först har bearbetats av `PostProcessor`. Validering utförs i två olika steg: ett standardsteg som automatisk används om komponenten inkluderas i pipelinen, och ett valfritt steg som kan styras via det konfigurationsobjekt som skickas med i klassen konstruktor då pipelinen skapas. I Listing 4.16 nedan visas den typ som definierar konfigurationsobjektet.

```
type ValidatorConfig = {
  consistencyRules?: boolean
}
```

Listing 4.16: Konfigurationobjekt av typen ValidatorConfig skickas med då Validator-komponenten instantieras för att påverka hur valideringen ska genomföras. Genom att aktivera consistencyRules tillämpas kontextuella hjälpregler vid validering.

Komponentens standardförfarande är en strukturell validering som utförs genom att utdatan kontrolleras av Zods `safePrase`-metod mot det angivna schemat som ligger till grund för extraheringen, ett generellt exempel på detta framgår i Listing 3.4. Denna validering säkerställer att det fält, datatyper och villkor som angivits i schemat följs av modellen utdata. Eventuella valideringsfel, som tillhandahålls automatiskt via Zod, innehållande information om både felaktigt angivna fält och orsak till avvikelsen, lagras tillfälligt i komponenten för att sedan kunna inkluderas i en potentiell omförsöksprompt i Sektion 4.4.5.

Det valfria valideringssteget inkluderas genom att aktivera `consistencyRules` då komponenten instantieras. Denna validering sker på en semantisk nivå genom att dra nytta av det kontextuella hjälpregler som användare kan förse pipelinen med via det domänspecifika objektet, se en övergripande beskrivning av detta i Sektion 4.3. Utdatan ifrån modellen kan på så vis valideras emot användardefinierade reglerna för att uppmärksamma logiska fel. Varje regel består av ett villkor(`condition`) som specificerar under vilka förhållanden regeln gäller, en kontroll (`check`) som anger ett villkor som måste var uppfyllt om regeln gäller, en felkod (`flagAs`) och ett användarvänligt meddelande (`message`). Samtliga regler evalueras mot modellens utdata och det eventuella flaggor som signaler logiska överträdelser lagras tillfälligt, för att sedan skickas vidare till komponenten `RetryStrategy`, förutsatt att denna är inkludera i pipelinen. Ett enklare exempel på en sådan regel visas i Listing 4.17 nedan.

```
{
    condition: (output) => output['text_visible'] === false,
    check: (output) => output['text_content'] === null,
    flagAs: 'text_content_without_visible_text',
    message: 'text_content must be null when text_visible is false',
}
```

Listing 4.17: Exempel på en kontextuell hjälpregel som kan anges i det domänspecifika objektet för att förhindra logiska fel. Fältet condition anger under villka förhållanden regeln gäller, fältet check anger villket villkor som ska vara uppfyllt för att regeln ska vara giltig, fälten flagAs och message används för att signalera felet och variablen output motsvar modellens utdata. I detta fall gäller regeln om modellen i sin utdata angivit att textilbilden i fråga inte innehåller något synlig text. Om så är fallet, görs en kontroll av fältet text_content för att se efter om den null. Om regeln inte efterföljs signaleras detta med hjälp av flagAs och message.

För det baschema som primärt används under projektet, se Tabell 4.1 och 4.2, har följande hjälpregler tagits fram:

- Om `text_visible` är `false`, behöver `text_content` vara `null`
- Om `text_visible` är `true`, behöver `text_content` vara en `array` innehållande minst ett element
- Om värdet `'none'` är valt för fältet `motifs`, får inga andra värden förekomma under samma fält
- Om `has_border` är `false`, behöver `border_colour` vara `null`
- Om `has_border` är `true`, behöver `border_colour` var skiljt från `null`

Det som slutligen avgör om en viss utdata är giltig eller ej beror på huruvida komponenten körs i sitt standardläge eller om `consistencyRules` är aktiverat. I standardläget anses utdatan giltigt om den följer det angivna schemats struktur och regler, det vill säga att den klarar Zod-valideringen utan anmärkning. Om `consistencyRules` däremot är aktiverat, kommer även de användardefinierade reglerna att påverka huruvida en viss utdata anses giltigt eller ej. I detta läget skulle alltså en strukturellt korrekt utdata som bryter mot en eller flera av det kontextuella hjälpreglerna anses vara ogiltig.

Om modellens utdata bedöms som giltig vid valideringen är pipelinens uppgift färdig och den extraherade metadata kan direkt inkluderas i `PipelineResult`-objektet. Om utdatan däremot underkänns vid valideringen, och komponenten `RetryStrategy` är aktiverad, skickas det lagrade felmeddelandena vidare till pipelinens sista komponent `RetryStrategy` som kan använda sig av dessa då ett nytt försök till analys av samm bild utförs.

4.4.5 RetryStrategy: Logik för återförsök

Om en textilbild inte går igenom valideringen kan komponenten `RetryStrategy` användas för att göra ett nytt försök. Komponenten kan konfigureras med de två olika alternativen `maxAttempts`(sv. maximalt antal återförsök) och `level`(sv. nivå), se Listing 4.18.

```
type RetryOptions = {
    maxAttempts: number;
    level: 0 | 1 ;
}
```

Listing 4.18: Konfigurationsobjekt av typen RetryOptions skickas med då RetryStrategy instansieras för att påverka återförsök. Fältet maxAttempts sätter en övregräns för antalet omförsök och level påverkar omförsökets ambitionsnivå.

Det finns två olika återförsöks-nivåer att välja mellan. För nivå noll sker återförsöken med oförändrade inställningar, det vill säga modellen uppmanas att extrahera information ifrån samma bild med precis samma prompt som i det initiala försöket. Denna typ av återförsök är tänkt att kunna användas för att korrigera slumpmässiga genereringsbeteende snarare än bristande förståelse för den faktiska uppgiften. För nivå ett kompletteras ursprungsprompten vid återförsök med en explicit återkoppling baserat på det felmeddelanden som registrerade i komponenten `Validator`, se Sektion 4.4.4 ovan. Felmeddelandena kan avse antingen struktur(ifrån Zod) och semantik(ifrån kontextuella hjälpregler) tillsammans eller enbart struktur, beroende på hur `Validator` är konfigurerad. Denna typ av återförsök vägleder alltså modellen genom att tala om vad som blev fel under tidigare försök och på vilket sätt detta behöver korrigeras i nästa försök. Ett exempel på hur detta ser ut i praktiken finns i Listing 4.19 nedan. Det formatrelaterade felen(`Schema errors`) är tagna ifrån exemplet i Listing 3.4 och det semantikrelaterade felen(`Consistency issues`) är tagna ifrån Listing 4.17.

```
...

Your previous response contained the following errors.
Please correct them and try again:

Schema errors:
  - [ 'motifs' ] Array must contain at least 1 element(s)
  - [ 'designStyle' ] Invalid enum value. Expected 'illustrated' |
    ↪ 'realistic' | 'abstract', received 'geometric'

Consistency issues:
  - text_content must be null when text_visible is false
```

Listing 4.19: Exempel på felmeddelanden som läggas till i slutet av den ursprungliga användarprompten. Schema errors är relaterade till strukturella fel och Consistency issues till semantiska/logiska fel.

Om modellen fortsätter att genererar inkorrekt utdata efter att samtliga återförsök har genomförts, returneras den senast genererade varianten tillsammans med valideringsstatusen ogiltig. Om en giltig utdata genereras avbryts eventuella kvarvarande försök och den validerade modellresponsen returneras tillsammans med valideringsstatus giltig.

4.4.6 Pipelinens resultat

När samtliga steg i pipelinen är genomförda returneras ett objekt av typen `PipelineResult`, se Listing 4.20 nedan, innehållande den extraherade metadatan(`output`) samt kompletterande information om vilken bild som analyserats, om utdatans giltighet, eventuella felmeddelanden, tids- och tokenåtgång samt antal återförsök.

```
type PipelineResult = {
  imageId: string
  output: Record<string, unknown>

  outputParsed: boolean
  schemaValid: boolean
  consistencyValid: boolean

  validationErrors: string[]
  flags: string[]
  attempts: number
  durationMs: number
  modelMetrics?: ModelMetrics
  pipelineError?: string
}
```

Listing 4.20: Returtypen för pipelinen.

Resultatet ifrån bildanalysen lagras sedan i resultatloggar tillsammans med information om vilka av Pipelinens komponenter och funktioner som varit delaktiga i att producera just detta resultat. Dessa ligger sedan till grund för det tester som utförs i nästkommande sektion.

4.5 Utformning och genomförande av tester

Utvärderingen strukturerades upp i tre olika faser, där varje ny fas baserades på resultatet ifrån den förgående. Under fas 1 etableras en tillförlitlig referensnivå för varje modell, fas 2 fokuserade på att isolera och jämföra effekterna av pipelinens komponenter mot referensnivån, och i fas 3 utvärderades samtliga VLM:er tillsammans med den pipeline konfiguration som fastställdes under fas 2.

Datasetet som ligger till grund för testerna utgörs av 63 egen annoterade textilbilder. I varje enskilt test extraherar VLM:en information om samtliga av dessa

bilder i enlighet med det grundläggande schemat, se Tabell 4.1 och 4.2. Därefter jämfördes modellens outputs med grundsanningarna.

4.5.1 Ramverk för mätning

Varje fält i det grundläggande textilschemat utvärderas med en metrik som visas i Tabell 4.3 nedan.

Tabell 4.3: Tabell över vilka metriker som används för respektive fält i basschemat

Type	Metrik	Fält
Enum	Accuracy	product_type, collection, designStyle, layout, border_colour
Array	Jaccard	motifs, primary_colors, text_content
Boolean	F1	text_visible, has_border, pattern_repeat

Dessutom kombineras det ovan nämna metrikerna till ett samlat oviktat medelvärde vid namn mean_quality (sv. medelkvalitet), för varje test. Ytterligare ett sammansatt mått som används under utvärderingen är joint_quality (sv. sammanvägd kvalitet), som beräknas som produkten av mean_quality och schema compliance rate (sv. grad av schemaöverensstämmelse) och ger därigenom uttryck för både korrekthet och tillförlitlighet.

Fas 1–2 använde ett fixt värde på variabeln seed för att eliminera slumpvariation och möjliggöra jämförbarhet.Referensnivåerna från fas 1 utvärderades även över seeds 0, 42 och 123 för att analysera känslighet för seed. I fas 3 genomfördes testerna med samma seeds för att möjliggöra jämförelse med referensnivåerna.

4.5.2 Fas 1: Etablering av referensnivå

Under fas 1 testades alla tre modeller vid tre olika temperaturer och med samtliga tidigare identifierade metoderna för strukturera output: json_schema, json_object, tool_calling och none (schemat inkluderas i prompten utan att förlita sig på någon API parameter), se Sektion 4.1. De enda aktiva komponenterna i pipelinen under denna fas var PromptEngine och Validator, för att efterlikna ett icke-pipeline-baserat beteende.

Syftet var att för varje modell identifiera en temperatur och metod som på ett tillförlitligt sätt kunde producera strukturerad output. Denna första fas fungerade därigenom som en förutsättning för det andra faserna eftersom att en utvärdering av innehållet förutsätter strukturerad utdata. Efter att samtliga tester körts rangordnades modellerna efter schema compliance rate, joint_quality och mean_quality (se Tabell 5.2).

4.5.3 Fas 2: Komponentanalys

Komponentanalysen genomfördes i två steg. I det första steget(2a) utvärderas pipelinens komponenter tillsammans med en väl-presterande referensnivå ifrån fas 1 (Qwen3 med json_object) som lyckades generera giltig data i 97% av fallen. I det efterföljande steget (2b) undersökt hur pipelinens komponenter kan hjälpa sämre presterande referensnivåer med att hantera schemafel (genom att studera Qwen3 med tool_calling) och parsingfel (genom att studera samtliga modeller med none-format).

4.5.3.1 Fas 2a:

Varje pipelinekomponent utvärderades en i taget mot referensnivån för att isolera dess individuella påverkan på resultatet. Därefter utvärderades även hela pipelinen med samtliga komponenter och funktionalitet aktiverad. Syftet var att undersöka effekten av pipelinens komponenter på en redan väl fungerade referensnivå.

4.5.3.2 Fas 2b:

Under fas 2b undersöktes de av pipelinens komponenterna som hanterade ogiltiga extraheringsresultat, det vill säga PostProcessor och RetryStrateg. Syftet var att undersöka om pipelinebaserad felkorrigering kunde förbättra extraheringsresultaten.

4.5.4 Fas 3: Modelljämförelse

Under fas 3 jämfördes samtliga modeller, vardera utifrån sin bästa kombination av temperatur och metod för strukturerad output ifrån fas 1, tillsammans med den mest gynnsamma kombination av pipeline-komponenter som konstaterats under fas 2. Syftet med den avslutande fasen var att avgöra vilken modell som tillsammans med pipelinen skulle användas i tjänsten.

5

Resultat

I detta kapitel presenteras det resultat som åstadkoms under de tre utvärderingsfaserna i Sektion 4.5. I Tabell 5.1 nedan definieras centrala termer och metriken.

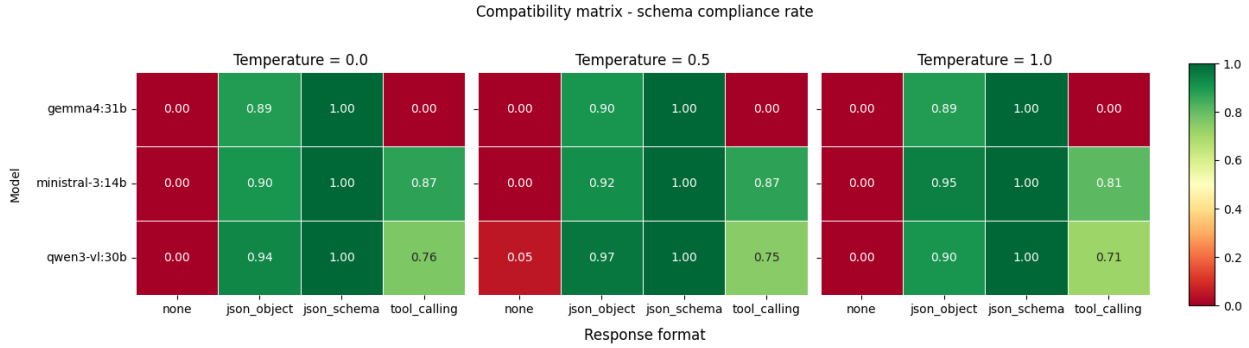
Tabell 5.1: Definitioner av termer och metriker som används under Resultatkapitlet.

Termer	**Definition**
Konfigurationer	
Models	De tre lokala VLM:erna som utvärderas: gemma4:31b, ministral-3:14b, qwen3-vl:30b.
Response format	Metod som används för att erhålla strukturerar output: none(utan att förlita sig på API-parametrar endast schema i prompt), json_object, json_schema, eller tool_calling (se Sektion 4.1.1).
Temperature	Modellens sampling temperatur; Värden som används: 0.0, 0.5, 1.0. För definition av temperatur se 3.3
Seed	Används för reproducerbar generering vid identiska förutsättningar. För vidare definition av seed se Tabell 3.3
Baseline	Referensnivå som pipelinens komponenter utvärderas mot i fas 2a,2b och 3. Bestående av modell och pipeline i dess mest grundläggande form.
Strukturerad output	Utdata som följer det angivna schemat.
Pipelinens utfall	
Pipeline error	Testfall där ett externt fel uppstod vid extrahering, exempelvis ett API-relaterade fel.
Parse failed	Testfall där modellen returnerade icke-parsbar utdata.
Schema failed	Testfall där modellen returnerade parsbar men icke schemaenlig utdata
Schema-valid	Testfall där modellen returnerade parsebar och schemaöverensstämmande utdata.
Metriker, för definition av metrikerna se 3.4	
Schema compliance rate	Andel schema-giltiga testfall utan Pipeline error.
Enum accuracy	Korrekthetsmått för enumfält
Jaccard similarity	Mått på likhet mellan två olika mängder för arrayfält.
F1	Korrekthetsmått för booleanfält
Mean output quality	Oviktat medelvärde för de tre metrikerna (enum accuracy, Jaccard, and F1) beräknat över Schema-giltiga testfall.
Joint quality	Schema compliance rate $\times$ mean output quality.
Parse rate	Andel testfall som returnerade parsbar utdata.
Δ vs baseline	Avvikelse från referensnivå definierad som konfigurations metrik minus referensnivåns värde.

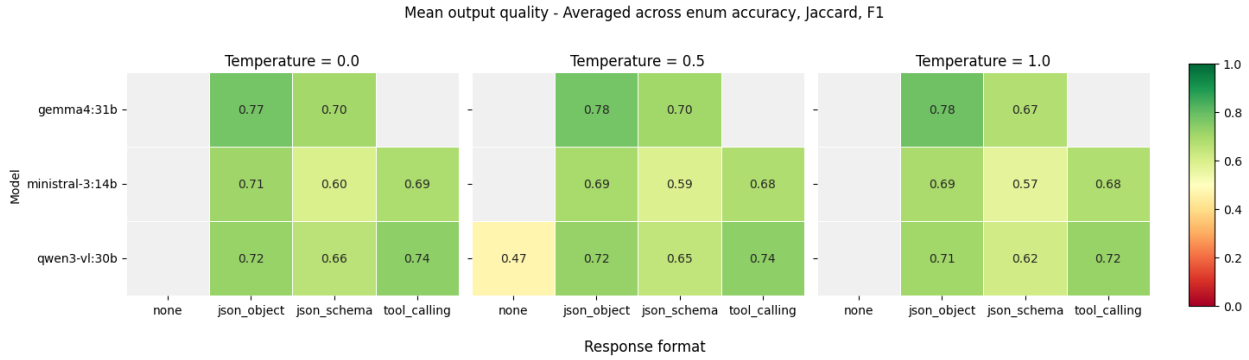
5.1 Fas 1: Etablering av referensnivå

I fas 1 utvärderades hur metod för strukturerad output och temperatur påverkar utdata i termer av kvalitet, tillförlitlighet och resursanvändning.

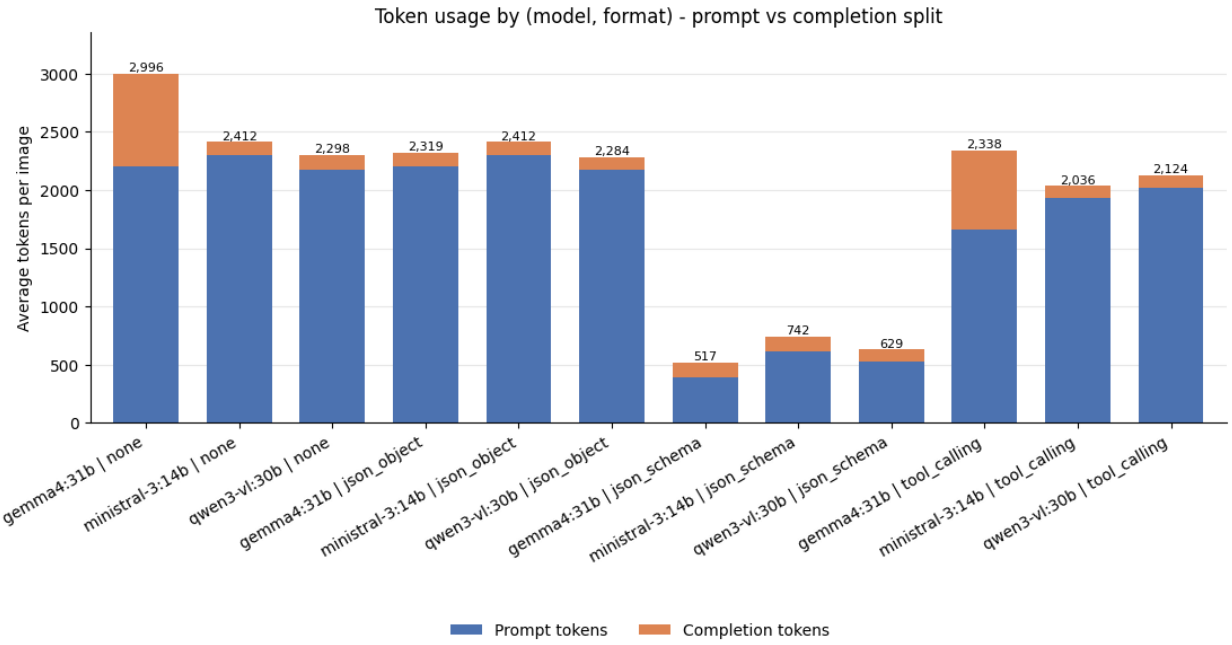
Figur 5.1: Fördelning av feltyper per modell, outputmetod och temperatur i testfall. Varje stapel visar fördelningen av utfallskategorierna Pipeline error, Parse failed, Schema failed och Schema valid för respektive test



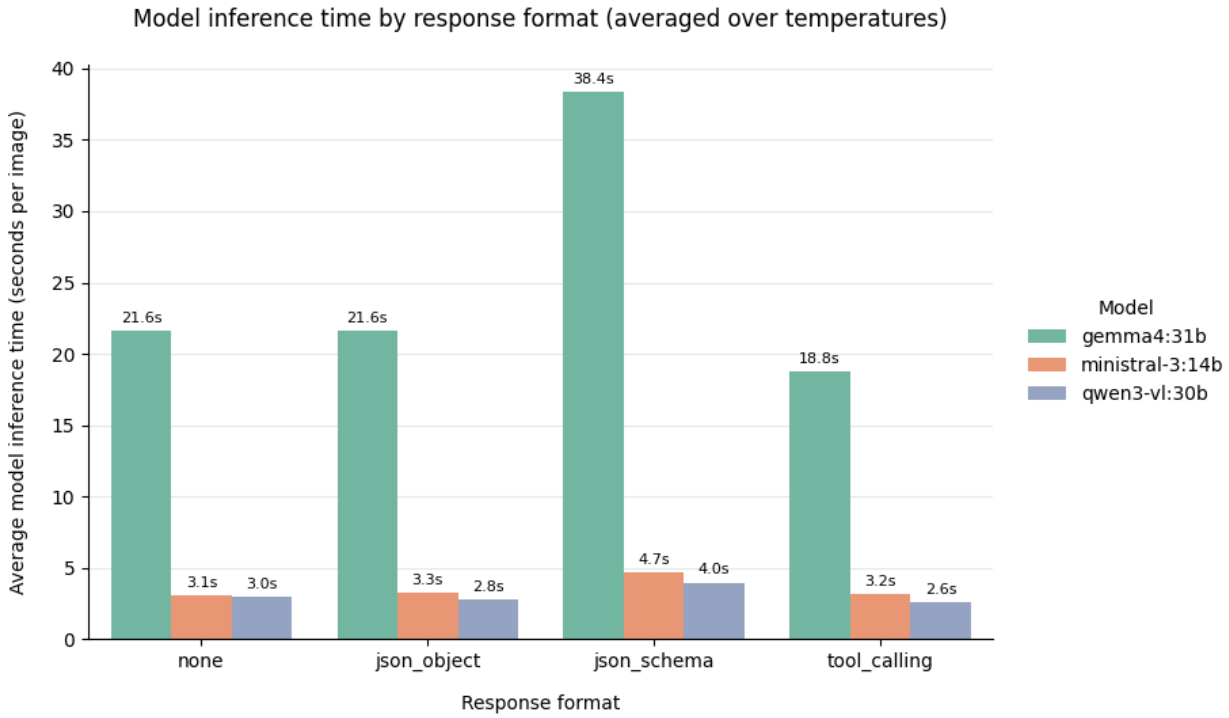
Figur 5.2: Schemaöverensstämmelse för varje modell (rad), temperatur(matris) och metod för strukturerad output (kolumn). Färgerna symboliserar grad av överensstämmelse från 0 (röd) till 1 (grön).



Figur 5.3: Genomsnittlig kvalitet på utdata för varje modell (rad), temperatur (matris) och metod för strukturerad output (kolumn). Gråa celler indikerar att det inte fanns en enda schema-giltiga utdata ifrån just den körningen. Färgerna representerar nivå av genomsnittlig kvalitet från 0 (röd) till 1 (grön).



Figur 5.4: Genomsnittlig tokenåtgång per bild för varje modell och metod för strukturell output.Varje stapel representerar medelvärdet över körningar med temperaturerna 0, 0.5 och 1. Varje stapel är även uppdelad i andelarna Prompt tokens (blå) och Completions tokens (orange).



Figur 5.5: Genomsnittlig tidsåtgång per bild för varje modell och metod för strukturell output. Varje stapel representerar medelvärdet över körningar med temperaturerna 0, 0.5 och 1.

Tabell 5.2: Rangordning av modeller: de bäst presterande konfigurationerna med schemaöverensstämmelse $\geq 0{,}95$, baserat på joint_quality

Model	Format	Temp	Compliance	Enum acc	Jaccard	F1	Mean quality	Joint quality	Tokens	Duration (ms)
gemma4:31b	json_schema	0.5	1.000	0.568	0.789	0.757	0.705	0.705	520	37,944
qwen3-vl:30b	json_object	0.5	0.968	0.639	0.752	0.781	0.724	0.701	2,285	2,704
gemma4:31b	json_schema	0.0	1.000	0.559	0.787	0.755	0.700	0.700	519	36,998
gemma4:31b	json_schema	1.0	1.000	0.540	0.756	0.726	0.674	0.674	514	40,172
qwen3-vl:30b	json_schema	0.0	1.000	0.476	0.728	0.770	0.658	0.658	628	4,285
ministral-3:14b	json_object	1.0	0.952	0.543	0.816	0.710	0.690	0.657	2,415	3,287
qwen3-vl:30b	json_schema	0.5	1.000	0.441	0.729	0.771	0.647	0.647	629	3,893
qwen3-vl:30b	json_schema	1.0	1.000	0.406	0.708	0.732	0.615	0.615	631	3,790
ministral-3:14b	json_schema	0.0	1.000	0.403	0.680	0.703	0.595	0.595	742	4,621

Samtliga konfigurationer i Tabell 5.2 ovan har en schemaöverensstämmelse $\geq 0{,}95$. Den bästa konfigurationen för varje modell är gulmarkerad och utgör referensnivån ifrån fas 1.

5.2 Fas 2: Komponentanalys

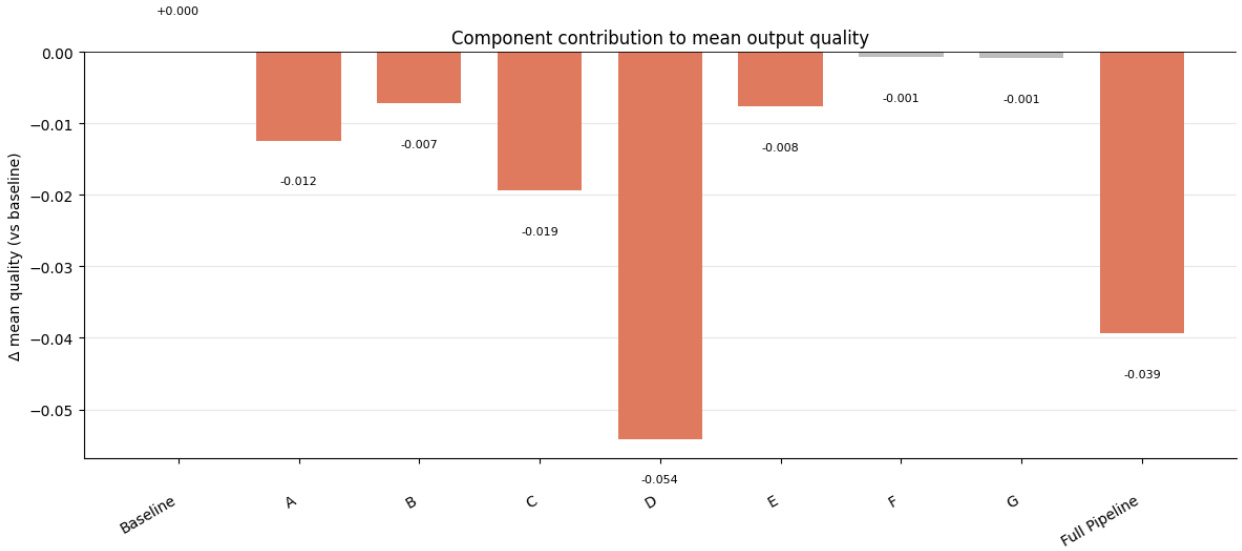
Det pipeline-konfigurationer som utvärderas under denna fas finns listade i Tabell 5.3.

Tabell 5.3: Definition av kodningen A-J som används för att representera olika konfigurationer i figurerna.

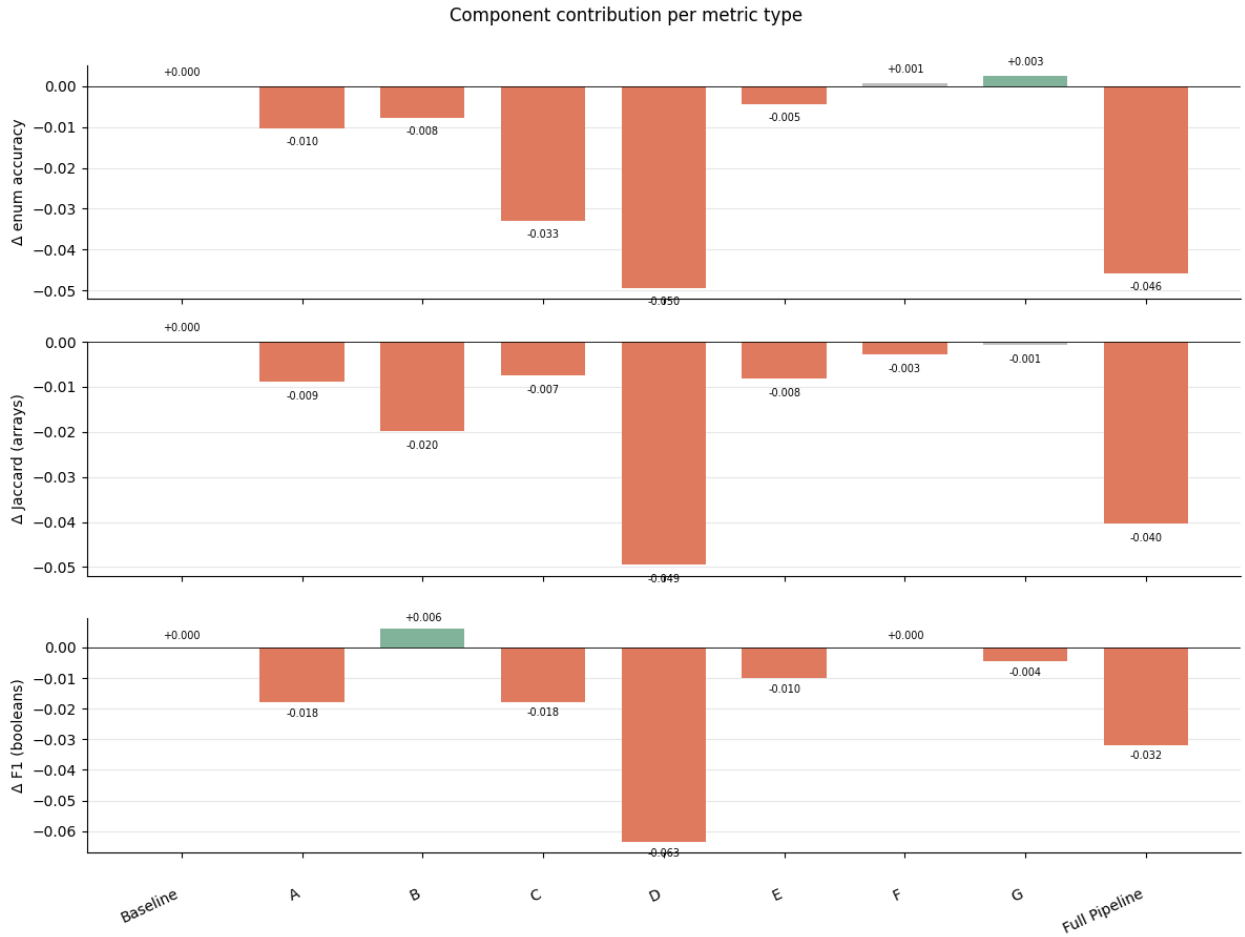
Kod	Beskrivning
Baseline	PromptEngine(schemaInjection:true)
A	PromptEngine(chainOfThough:true, schemaInjection:true)
B	PromptEngine(chainOfThough:true, fieldDescription:true)
C	PreProcessor(tiling:true)
D	PreProcessor(normalization:true)
E	Validator(consistencyRules:true) & I
F	PostProcessor(extractionMode:'fence_and_repair')
G	PostProcessor(normalization:true)
H	PostProcessor(normalization:true, extractionMode:'fence_and_repair')
I	RetryStrategy
J	H & I
Full Pipeline	B & C & D & E & H

5.2.1 2a: Pipelinens effekt på välfungerande referensnivå

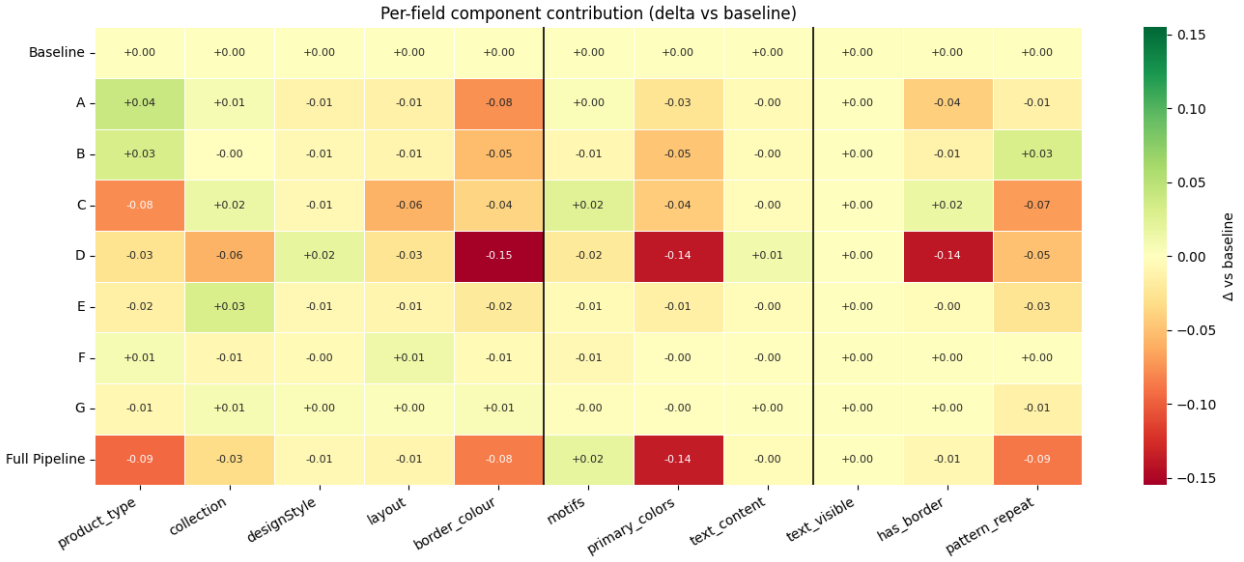
Ifrån Tabell 5.2 väljs den näst högst rankade konfigurationen: qwen3-vl:30b med json_object vid temperatur 0.5 ut som referensnivå eftersom att den har högst mean_quality och betydligt lägre genomsnittlig inferenstid än den högst rankade. Varje pipelinekomponent testas en i taget mot referensnivån och dess isolerade effekt på resultatet plottas. Samtliga beräkningar av kvalitet utfördes på schema-giltiga utfall.



Figur 5.6: Komponenternas bidrag till den genomsnittliga kvalitet på utdata. Varje stapel visar förändring i genomsnittskvalitet då en viss komponent är aktiverade jämfört med referensnivån.



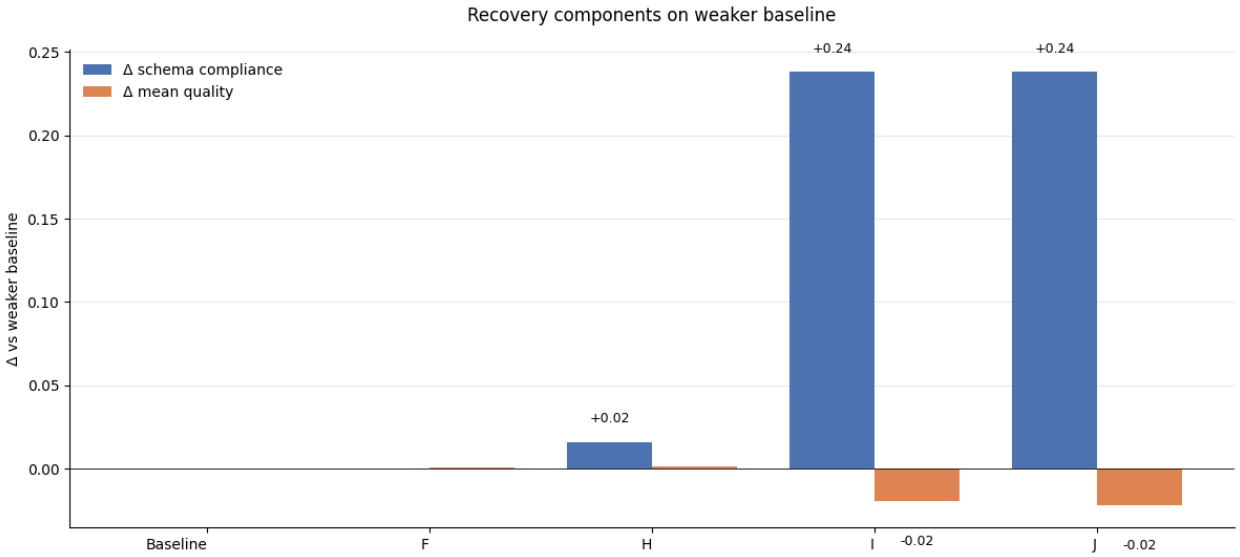
Figur 5.7: Komponenternas bidrag till den genomsnittliga kvaliteten på utdata för det olika metrikerna. Varje stapel visar förändring i genomsnittskvalitet då en viss komponent är aktiverade jämfört med referensnivån. De tre panelrna delar x-axel.



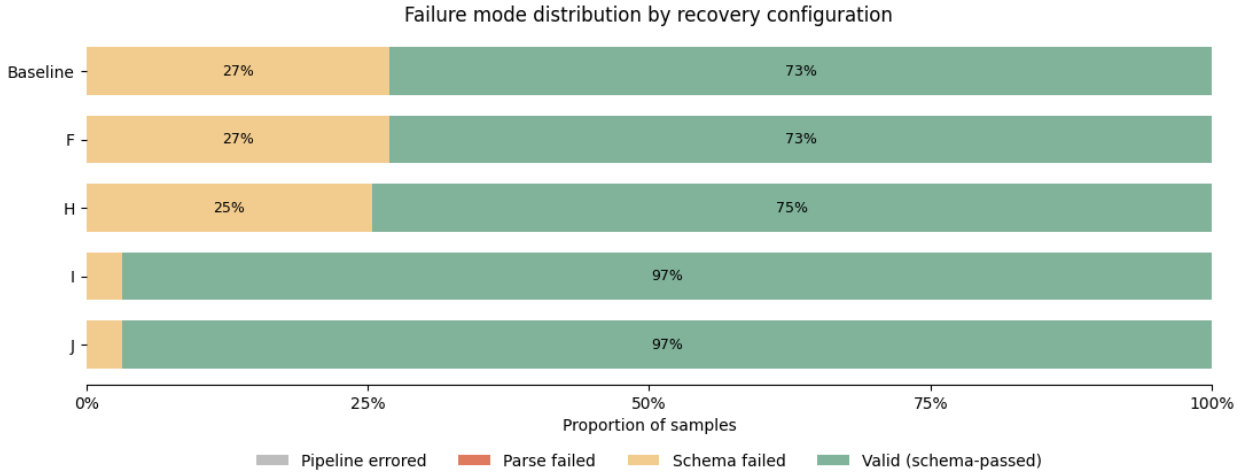
Figur 5.8: Komponenternas effekt på det individuella fälten. Varje cell visar förändringen för respektive fällts poäng i förhållande till referensnivån. Kolumnerna är grupperade efter typ(enum,array,boolean) och separeras med vertikala linjer. Färgerna anger avvikelsen från referensnivån, grön motsvara förbättring och röd försämring.

5.2.2 2b: Pipelinens effekt på en sämre referensnivå

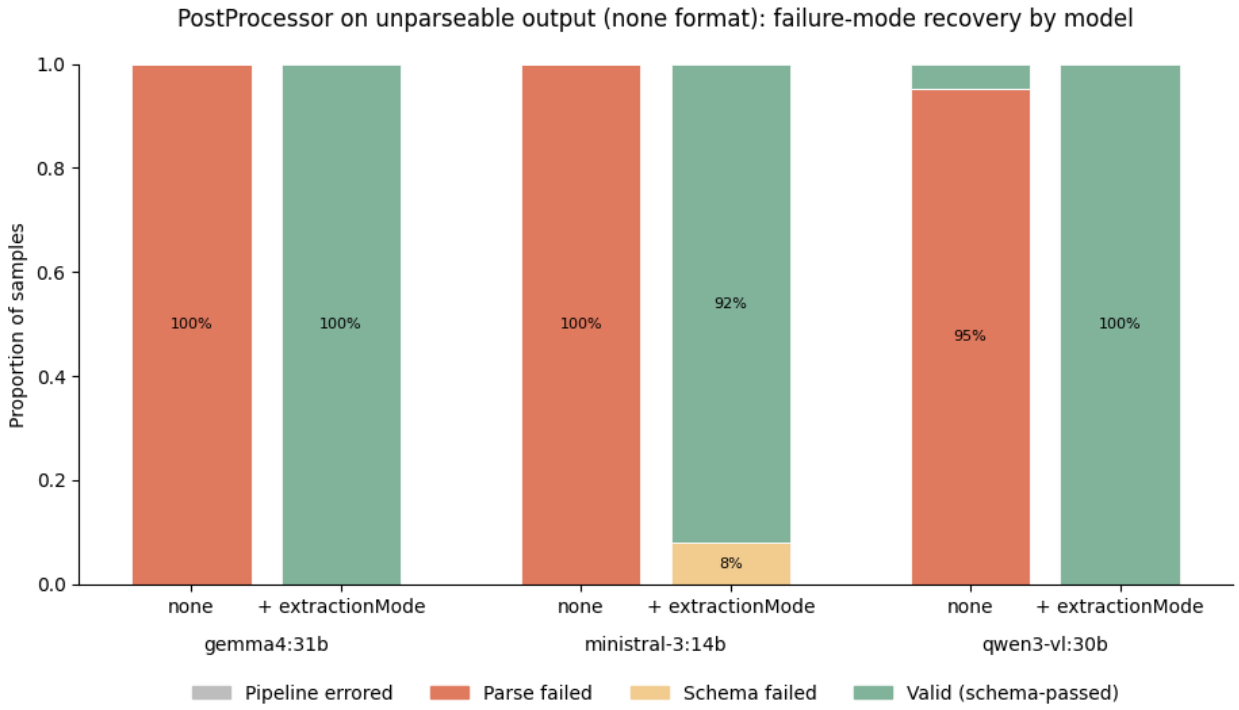
Under fas 2b testades pipelinens återhämtningskomponenter: PostProcessor och RetryStrategy, mot svagare referensnivåer ifrån fas 1. Först utvärderades komponenterna tillsammans med en referensnivå som genererade en mängd schemafel (qwen3 med temperatur 1 och tool_call-format) i Figur 5.9 och 5.10. Därefter utvärderas samtliga modeller ifrån fas 1 med none-format, det vill säga med den metod för strukturerad output som inte lyckades generera parsbar data överhuvudtaget (Figur 5.1 och 5.2) tillsammans med komponenten PostProcessors. Resultatet redovisas i Figur 5.11



Figur 5.9: Effekten av komponenterna PostProcessor och RetryStrategy på en referensnivå med medelhög prestanda (qwen3-vl:30b med temperatur 1 och tool_call-format). Staplarna visar skillnad i schemaöverensstämmelse (blå) och genomsnittlig kvalitet (orange) i relation till referensnivån.



Figur 5.10: Fördelning av feltyper per återhämtningskonfiguration för referensnivå med medelhög prestanda (qwen3-vl:30b med temperatur 1 och tool_call-format).



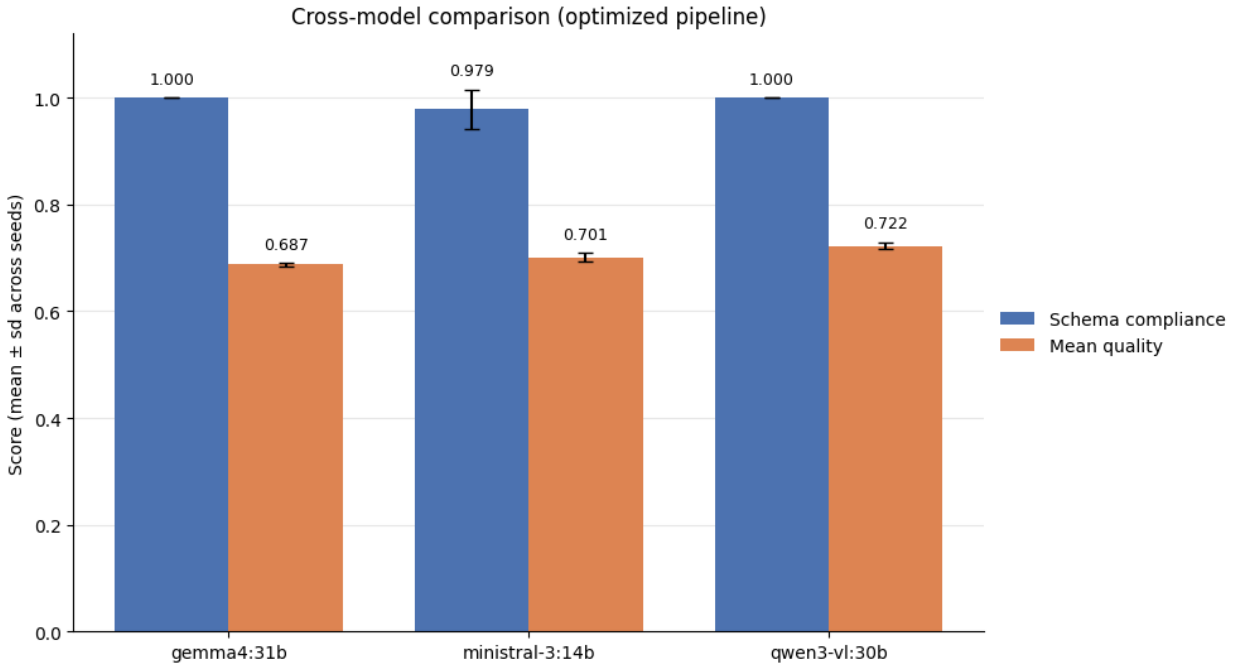
Figur 5.11: Fördelning av feltyper för samtliga modeller med(+ extracionMode) och utan (none) PostProcessorns funktion för att hantera icke-parsbar utdata ifrån modellerna. Samtliga modeller har körts med med formatet none.

5.2.3 Sammanfattning

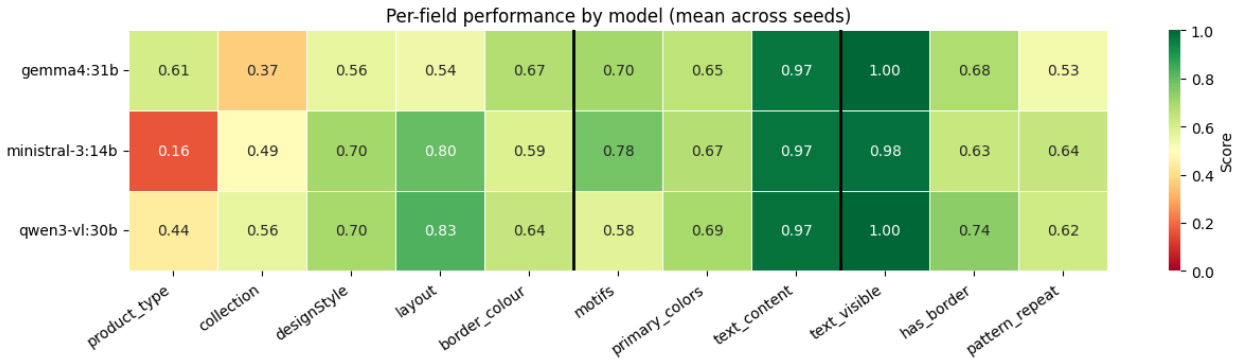
För den välfungerande referensnivån (fas 2a) observerades ingen förbättring av den genomsnittlig kvalitet då pipelinens komponenter aktiverades. Istället var effektern neutrala eller negativa. För den svagare referensnivån (2b) påverkade komponenten RetryStrategy resultatet positivt genom en ökning i schemaöverensstämmelse men utan märkbar förändring i genomsnittlig kvalitet. Komponenten PostProcessor bidrog i hög grad genom att bearbeta och extrahera den råa sträng som modellen genererade som output innan den parsades. Dessa resultat ligger till grund för pipelinekonfigurationen som används under fas 3.

5.3 Fas3: Modelljämförelse

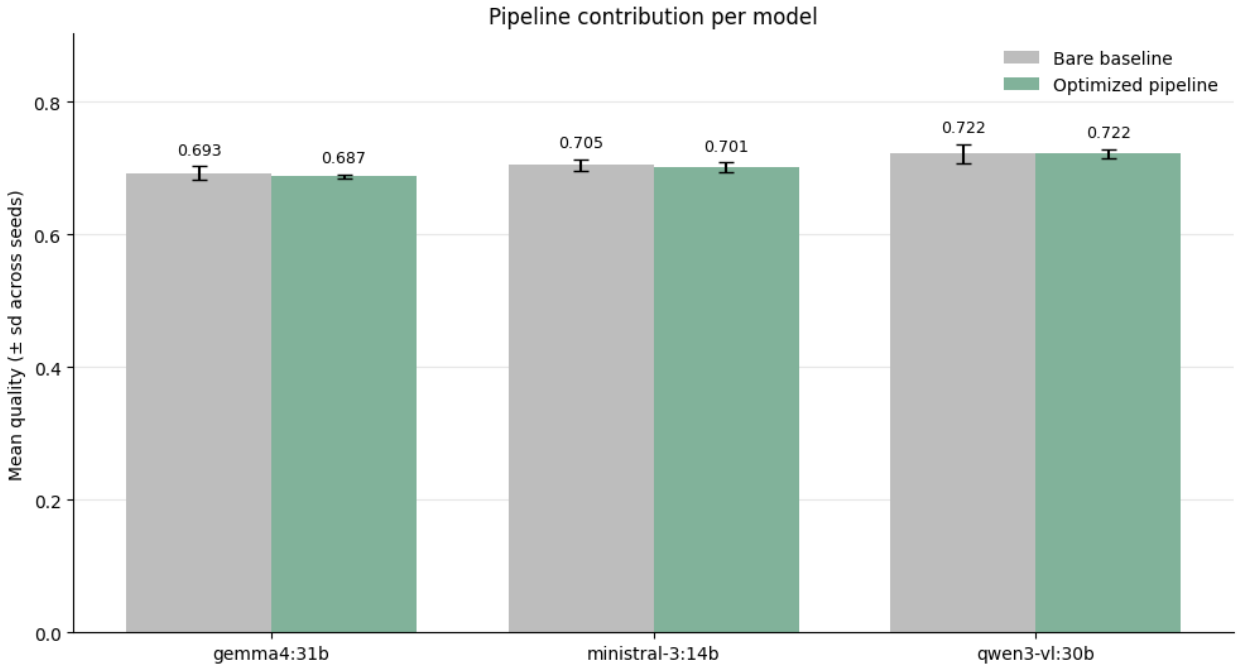
Under fas 3 utvärderades alla tre modeller med respektive bästa konfiguration (Tabell 5.2), tillsammans med den bäst presterande pipelinekonfigurationen som etablerades under fas 2: PostProcessor med full funktionalitet aktiverad, Validator utan consistencyRules och RetryStrategy. Varje modell plus pipeline kördes över tre olika seeds, med respektive modells bästa konfiguration utan pipeline som referens.



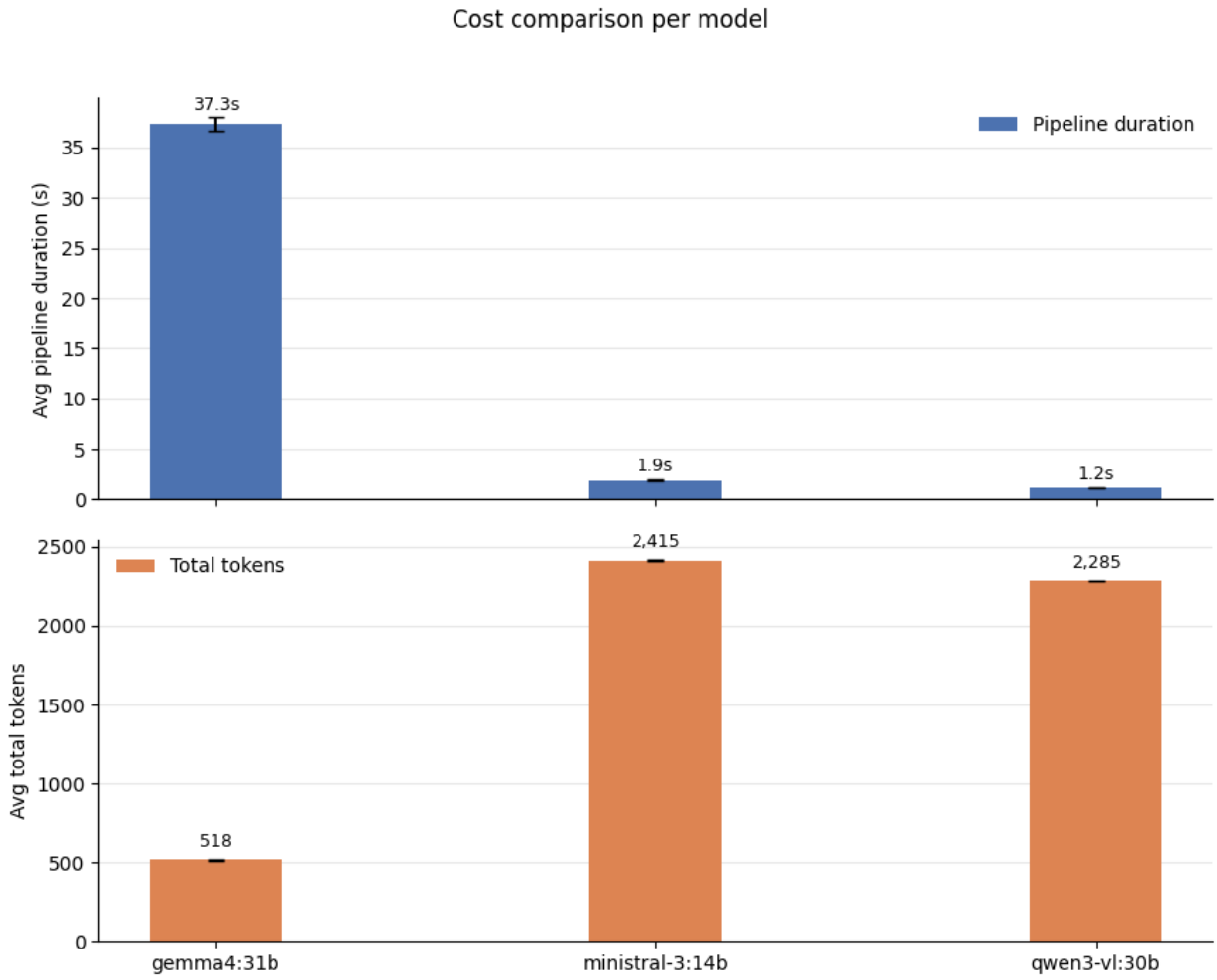
Figur 5.12: Jämförelse mellan modeller med den optimerade pipelinen. Staplarna visar medelvärden över tre seeds, felstaplar anger standardavvikelsen.



Figur 5.13: Fältvis prestanda per modell. Varje cell visar genomsnittlig fältkvalitet över tre seeds. Kolumnerna är grupperade efter typ(enum,array,boolean) och separeras med vertikala linjer. Grönt anger hög kvalitet, rött låg kvalitet.



Figur 5.14: Den optimerade pipelinens påverkan på genomsnittlig kvalitet för de tre modellerna. Bare baseline är fas 1-konfigurationen utan pipelinekomponenter, båda staplarna är medelvärden över tre seeds.



Figur 5.15: Den optimerade pipelinens kostnad för de tre modellerna. Tiden avser end-to-end pipeline per bild, tokens avser prompt och completions. Staplarna visar genomsnitt över icke-felaktiga fall och tre seeds.

Sammanfattning För nästa samtliga modeller uppkom schemaöverensstämmelsen till 1 och den genomsnittliga kvaliteten var koncentrerad till ett smalt intervall: $0.687 - 0.722$. Den optimerade pipelinen medförde inga nämnvärda förändringar relativt referensnivåerna för genomsnittlig kvalitet för samtliga modeller. Däremot observerades större skillnader i den fältvisa prestandan och för både tid och token-förbrukning.

6

Diskussion

Syftet med projektet var att ta fram en tjänst som kunde användas för att extrahera strukturerad information ifrån produktbilder med hjälp av vision språkmodeller via Ollama. Den framtagna lösningen består av en konfigurerbar modelloberoende pipeline, som utgör kärnan i själva tjänsten, och dess utformning och implementering behandlas i Sektion 4.3. Detta avsnitt inleds med en utförlig diskussion och tolkning av de resultats som framkom i resultatsektion 5, och därefter hålls en övergripande diskussion kring metodologin.

6.1 Diskussion och tolkning av resultat

För att undersöka om den framtagna pipeline-konstruktionen tillförde något värde genomfördes en utvärdering, vars resultat återfinns i Sektion 5, med syfte att avgöra vilka komponenter som bör ingå i den driftsatta tjänsten.

6.1.1 Tolkning av resultat ifrån Fas 1

Utvärderingens första fas, vars resultat redogörs för i Sektion 5.1, visade att metoden för strukturerar output, och inte temperatur, var den dominerande faktorn för att åstadkomma korrekt strukturerad utdata. I kompatibilitetsmatrisen (Figur 5.2), kan det noteras att med json_schema genereras genomgående schemaenlig output (1.0) för samtliga modeller och temperaturnivåer, medan json_object konsekvent ligger på längre nivåer, men som fortfarande ger användbar output i det flesta fall (≈0.89-0.97). Med outputmetoden none, det vill säga utan API-variabler och med schemat direkt i prompten, genererades däremot inte en enda giltig output (0.0). Metoden tool_call uppvisar tydliga skillnader mellan modellerna: för ministral uppnåddes resultat på (≈ 0.81-0.87) och för qwen3 (≈ 0.71-0.75), medan gemma4 inte genererade någon giltig output (0.0). När det kommer till temperatur, verkar den endast ha en marginell påverkan på möjligheten att åstadkomma strukturerad utdata. Qwen3 med json_object varierar endast mellan 0.90 och 0.97 över de tre temperaturnivåerna, medan json_schema förblir konstant på 1.00.

Det kan även noteras att de två felkategorierna Parse failed (icke parsbar output) och Schema failed (parsbar men ogiltig enligt schema), som i Figur 5.1 visas som

separata felkällor, klumpas ihop under samma icke-kompatibla kategori i kompatibilitetsmatrisen (Figur 5.2). För exempelvis gemma4 med outputmetod none, är problemet att output inte kan parsa, inte att den bryter mot schemat.

En intressant observation kan göras genom att jämföra kompatibilitetsmatrisen i Figur 5.2 med figuren över genomsnittlig kvalitet (Figur 5.3). Den outputmetod som konsekvent genererar strukturerad output är inte den som dominerande när det gäller genomsnittlig kvalitet. För samma modell presterar istället både json_object och tool_calling konsekvent bättre än json_schema, vilket visar att schemaefterlevnad och kvalitet inte nödvändigtvis hänger ihop. En möjlig orsak till att just json_schema presterar sämre skulle kunna vara att dess constrained decoding, se Sektion 4.1.1, också hindrar modellen från sitt bästa fria svar. Precis som för strukturerad output har temperatur endast marginell påverkan på den genomsnittliga kvaliteten.

När det gäller genomsnittlig tokenförbrukning är det återigen json_schema som presterar bäst (Figur 5.4), med cirka 517–742 tokens per bild jämfört med omkring 2000 för övriga metoder. Den genomsnittliga tidsåtgången per analysera bild i Figur 5.5 skiljer sig inte nämnvärt mellan det olika output metoderna för ministral och qwen3. Däremot sticker gemma4 ut som cirka 6 ggr långsammare än det övriga modellerna. En möjlig förklaring är att skillnaden beror på arkitekturen, gemma4 är dense-baserad medan Qwen använder MoE, vilket innebär fler aktiva parametrar per generering, se sektion 3.1. Dense-arkitektur kan dock inte ensamt förklara den långsammare inferensen, eftersom ministral också är dense-baserad men mycket snabbare. En trolig förklaring är skillnaden i storlek, där gemma4 är omkring dubbelt så stor som ministral. Detta är dock en tolkning av modellernas egenskaper snarare än ett mätresultat.

6.1.2 Tolkning av resultat ifrån Fas 2

När pipelinens olika komponenter utvärderades tillsammans med den välfungerande referensnivån, i Sektion 5.2, påverkades den genomsnittliga kvaliteten antingen inte alls eller negativt (Figur 5.6). Varken PromptEngines chain-of-thought (A) eller fieldDescription (B), Validatorns consistencyRules (E) eller PostProcessors funktionaliteter (F,G) gav någon nämnvärd påverkan på den genomsnittliga kvaliteten (under 1%). De två kraftigaste negativa effekten observerades för bildnormaliseringen (D) med ca 5% försämrad kvalitet och för kombinationen av samtliga komponenter (Full Pipeline) med ca 4% försämrad kvalitet. Att skillnaden mellan referensnivån och den kompletta pipelinen inte är summan av det individuella komponenterna tyder på att effekterna inte är rent additiva utan beror på interaktioner mellan dem. Klart är i varje fall att pipelinens komponenter inte bidra till en ökad genomsnittlig kvalitet utan snarare en försämring när den används tillsammans med en redan välfungerande referensnivå under denna utvärdering.

Det bör även noteras att exempelvis consistencyRules (E) skulle kunna påverka resultaten under andra förutsättningar, exempelvis om en annan uppsättning regler hade tillämpats eller om specifikationen varit utformad på ett annat sätt. Att komponenten inte gav något mätbart utslag i denna utvärdering innebär därför inte nödvändigtvis att den saknar värde. Samma argumentation kan även föras beträffande en del av de andra funktionerna som pipelinen erbjuder.

Figurerna 5.7 och 5.8 kan i viss mån ge en förklaring till det mindre gynnsamma effekterna som observerades ovan. Figur 5.7 visar tydligt hur bildnormaliseringen (D) leder till ett likvärdig negativt resultat för samtliga metriker: enum accuracy (-0.050), Jaccard (-0.049) och F1 (-0.063). Detta kan i Figur 5.8 därefter härledas till att påverkan är särskilt koncentrerad till färgrelaterade fält. Fältet border_colour (-0.15), primary_colours (-0.14) och has_border (-0.14). Den färgfördelning som normalisering ger upphov till för att påverka kontraster tycks alltså leda till en försämring för de färgberoende fälten. I övrigt är det svårt att dra några konkreta slutsatser eftersom att de övriga komponenterna endast uppvisar små, varierande förändringar per fält (vanligtvis inom $\pm 0.02 - 0.08$).

Figur 5.14 ifrån fas 3 bekräftar även att pipelinens effekt på outputens kvalitet i princip är försumbar för samtliga modeller och inte endast för qwen3 som utvärderades under fas 2a: gemma4 -0.006, ministral -0.004 och qwen3 0.000. Sammantaget kan det alltså konstateras att för en redan välfungerande referensnivå tillför komponenterna igen tydligt kvalitetsvinst.

Däremot visade sig somliga pipelinekomponenterna vara mer effektiva då de utvärderades tillsammans med de två mindre effektiva referensnivåer i Sektion 5.2.2. För konfigurationen med medelhög prestanda (qwen3 + tool_call + temperatur 1) visade sig komponenten RetryStrategy (I) ha en stor positiv påverkan på schemaöverensstämmelse. I Figur 5.9 kan det noteras att den ökar graden av schemaöverensstämmelse avsevärt med hela +0.24 medan skillnaden i medelkvalitet däremot förblir nästintill oförändrad. PostProcessorns bidrag (F,H) till genomsnittlig kvalitet och schemaöverensstämmelse är i princip försumbar.

Figur 5.10 ger kontext till resultaten som presenteras i Figur 5.9. Basnivåns samtliga fel uppstår på grund av att ouputen inte stämmer överens med det på förhand givna schemat: 27% Schema failed, 73% Schema valid och 0% Parse failed. PostProcessorns funktionalitet för att reparera icke-parsbar JSON (extractionMode) har alltså ingen effekt eftersom att outputen redan uppfyller kraven för parsbarhet, vilket förklara varför staplarna (F,H) i Figur 5.9 inte visar någon höjd. Komponenten RetryStrategy (I) reducerar schemafel från 27% till cirka 3%, eftersom modellen, genom att få Zods valideringsfel som återkoppling vid återförsök, kan korrigera den exakta strukturella missen.

I Figur 5.11 blir det dock tydligt att komponenten PostProcessor i hög grad bidrar genom att bearbeta och extrahera den råa sträng som modellen genererade som ut-

data innan den parsas. Med hjälp av PostProcessorns funktionalitet för att reparera json, kan i princip all data återställas och valideras mot schemat.

Sammantaget kan det alltså konstateras att pipelinens komponenter inte har något större effekt på det välpresterande referensnivåer som identifierades under fas 1. Tillsammans med det svagare referensnivåerna återfinns däremot ett värde i de komponenter som hanterar felmekanismer: hantering av schemafel via den feedback-baserade retry-mekanismen i kombination med Validator och hantering av parsefel via PostProcessorns extractionmode.

6.1.3 Tolkning av resultat ifrån Fas 3

Under fas 3 (Sektion 5.3) jämfördes samtliga modellers bäst presterande referensnivå ifrån Tabell 5.2 tillsammans med den fastställda pipelinkonfigurationen ifrån fas 2. I Figur 5.12 framgår att modellerna presterar i princip likvärdigt, samtliga uppnår ett nästan perfekt resultat för scehmaöverensstämmelse (gemma4 1.000, ministral 0.979, qwen3 1.000) och den genomsnittliga kvalitén är koncentrerad inom ett smalt intervall (0.687, 0.701, 0.722 - en spridning på 0.035). Utifrån denna figur blir det således svårt att utse en klar vinnare, även om qwen3 presterar aning bättre än de andra modellerna.

Om man däremot betraktar den fältvisa prestandan per modell i Figur 5.13, kan vissa effekter som döljs i det aggregerade resultatet komma fram till ytan. Resultaten varierar mer mellan modellerna för vissa fält, såsom product_type, collection och layout, medan de för andra fält, såsom primary_colour, text_content och text_visible, är mer jämförbara. Inga starka slutsatser kan nödvändigtvis dras ifrån figuren, men det antyder ändå att modellval bör göras utifrån vilka fält som anses viktigast. Om exempelvis layout-relaterad information är central för en viss extrahering, förefaller qwen3 vara ett bättre alternativ än gemma4. Medan det förhåller sig precis tvärt om när det exempelvis kommer till fältet motiv.

Den mest framträdande skillnaden mellan modellerna återfinns i tid och tokenåtgång (Figur 5.15). Gemma4 är oerhört mycket långsammare än det övriga modellerna med en genomsnittlig tid per bild på 37.3s jämför med ministrals 1.2s och qwen3 1.9s, en potentiell anledning till detta har redan diskuterats ovan. För tokenåtgången är beteendet däremot inverterat: för gemma ca. 500 tokens och för de övriga ca 2300 tokens per bild. Både ministral och qwen3 använder sig av ouputmetoden json_object medan gemma4 använder json_schema. För de två förstnämnda behöver det schema som extraheringen ska följa inkluderas i användarprompten (se Sektion 4.1.1), medan det för json_schema tillhandahålls via API-fältet response_format. Skillnaden skulle alltså kunna förklaras utifrån detta. Att inkludera schemat direkt i prompten fungerar som en instruktion för modellen att följa, men den kan fortfarande generera fritt, vilken eventuellt skulle kunna resultera i ytterligare resonemang eller korrigeringar ifrån modellens sida som ökar användningen av tokens. När schemat tillhandahålls genom response_format i API:et verkar constrained decoding (se Sektion 4.1.1) hindra ytterligare generering utanför formatet.

6.1.4 Slutsatser från utvärderingen

Det huvudsakliga syftet med utvärderingen var att fastställa i vilken utsträckning pipelinens komponenter bidrog till förbättrad informationsutvinning och därmed identifiera vilka komponenter som bör ingå i den slutgiltiga tjänsten. Resultaten indikerar att komponenterna som helhet inte förbättrar den genomsnittliga extraktionskvaliteten när de används tillsammans med en välpresterande vision-språkmodell. Däremot påvisades ett värde i de tre återhämtningskomponenterna: RetryStrategy för hantering av schemafel, PostProcessor med extractionMode för hantering av tolkningsfel samt Validator.

Baserat på utvärderingsresultaten rekommenderas därför en tjänst som kombinerar dessa återhämtningskomponenter med Qwen3-modellen och json_object för generering av strukturerad utdata.

6.2 Metoddiskussion

Utöver den diskussion som förts om de faktiska resultaten, finns tre metod-relaterade aspekter som kan vara intressanta att lyfta, eftersom dessa påverkar hur resultaten bör läsas och i vilka sammanhang det är relevanta.

För det första, samtliga resultat som tagits fram i Sektion 5 bygger på en jämförelse mellan modellerna output och egen-annoterade grundsanningar (med undantag för fälten product_type och collection som kommer ifrån textilföretagets redan befintliga produktdata). Det framtagna siffrorna beskriver alltså en överensstämmelsen mellan modellerna utdata och min egen subjektiva tolkning för respektive textilbild, utan någon objektiv referens. Skulle en annan person gå igenom precis samma dataset och annotera bilderna utifrån de fördefinierade listorna av enumvärden, är det troligt att resultatet skulle se annorlunda ut i enskilda fall. Däremot skulle inte detta påverka pipelinens grundläggande beteende och troligtvis inte heller det övergripande slutsatserna som kunnat konstateras vid jämförelserna mellan komponenter och modeller i resultatsektionen 5. Men det innebär ändå en viktig metodologisk begränsning.

Den andra aspekten rör framtagningen av det schema som ligger till grund för extraheringen av information. Det grundläggande schemat som togs fram i Sektion 4.2 är i huvudsak enum-baserat, bestående av kategoriska fält med fördefinierade värden, och detta var ett medvetet metodologiskt val. Kategorisk extrahering går enkelt att mäta. Att avgöra om en modells utdata överensstämmer med en annat värde ifrån en fördefinierad lista är enkelt att verifiera, medan det är betydligt svårare när utdata i stället består av fria beskrivningar av samma innehåll. Fri-text fälten introducerar större variationsrikedom i formuleringar, vilket försvårar såväl annoteringsarbetet som efterföljande mätning. Detta metodologiska val gjordes dock på

bekostnad av mer detaljerade och nyanserade extraheringar som kunde erhållits om fri-text fält inkluderats. Ursprungstanken var, precis som förklaras i Sektion 4.2, att det grundläggande schema endast skulle vara en utgångspunkt och att komplexiteten och nyanserna i vad som efterfrågades av modellerna kunde byggas vidare på allt eftersom. Detta hann tyvärr inte genomföras, men vore intressant att vidareutveckla. Men med det sagt, är en enumbaserad extrahering även ett gångbart upplägg i verkliga system så det kan ändå finnas ett praktiskt värde i det framtagna schemat.

Den tredje aspekten handlar om i vilken grad slutsatserna är generaliserbara utanför det enum-baserade schemat. Ett svar på detta kan härledas direkt ifrån diskussionen ovan. De pipeline-komponenter som faktiskt visade sig användbara och som kom att utgöra den slutgiltiga pipelinen, det vill säga PostProcessor, RetryStrategy och Validator, fungerar som globala regler som appliceras på utdatans struktur, oberoende av om schemat innehåller enums eller fritext. Pipelinen som sådan kan användas för samtliga typer av informationsextrahering, förutsatt att ett lämpligt schema definieras för den information som ska extraheras. Den modulära designen möjliggör dessutom att nya komponenter enkelt kan tillföras eller att befintliga komponenter kan modifieras vid behov.

6.3 Etiska överväganden

För att träna AI-modeller krävs det i regel en ofantlig mängd data som i allra högsta grad påverkar modellens förmåga att producera relevanta svar. Under träningen kalibreras modellens parametrar utifrån innehållet i träningsdatan, vilket leder till att de mönster och egenskaper som finns i datan också kommer att återspeglas i modellens svar. Om de dataset som modellen tränas på innehåller inkorrekt information, diskriminerande eller partiska inslag kommer detta således även att påverka modellens output.

De modeller som används i projektet är, som tidigare nämnts, förtränade och körs via Ollama utan ytterligare finjustering. Detta innebär att eventuella bias i den data som modellerna har tränats på kvarstår oförändrade och därför bör beaktas vid tolkningen av extraktionsresultaten.

6.4 Kvarstående arbete

Även om vision-pipelinens huvudsakliga funktionalitet har implementerats, återstår vissa aspekter att behandla utanför detta arbetes omfattning:

- **Mikrotjänstbaserad arkitektur** - Tjänsten har ännu inte explicit utformats som en mikrotjänst, även om den redan följer en del av de principer som kännetecknar en mikrotjänstarkitektur.
- **RabbitMQ integrering** - Tjänsten saknar för närvarande stöd för meddelandehantering via RabbitMQ.

- **Docker-containerisering** - Tjänsten saknar för närvarande Docker-stöd.
- **Utvidga specifikation** - Det grundläggande enum-baserad specifikationen avsågs utvidgas för mer nyanserad och detaljerad extrahering, men detta hann inte genomföras.

7

Slutsats

Inom ramen för detta examensarbete har en tjänst för strukturerad informationsutvinning från produktbilder med hjälp av vision-språkmodeller utvecklats. I detta projekt har tjänsten applicerats på textilprodukter, men den är designad för att vara generell och kan användas för olika typer av bildbaserad informationsutvinning. Extraktionen baseras på en användardefinierad specifikation som styr vilken information som extraheras från bilderna. Syftet är att berika sökunderlaget i Pivotise, ett system för sortimentshantering, och därigenom förbättra användarupplevelsen.

För att identifiera den mest lämpliga systemkonfigurationen utvärderades tre aktuella vision-språkmodeller, Gemma4, Ministral-3 och Qwen3-VL tillsammans med den framtagna tjänsten. Utvärderingen visade att tjänstens återhämtningskomponenter ökade stabiliteten och säkerställde en högre grad av överensstämmelse med den definierade specifikationen. Däremot kunde ingen förbättring av informationsextraheringens korrekthet konstateras. Resultaten talar för att Qwen3 med JSON Mode bör användas tillsammans med tjänstens återhämtningskomponenter.

Projektet begränsas av att utvärderingen genomförts på ett mindre dataset av textilprodukter och med grundsanningar framtagna av en enskild annoterare. De komponenter som visade positiva effekter är i huvudsak specifikationsoberoende, men deras tillämpbarhet utanför textildomänen har inte verifierats.

Litteraturförteckning

[1] O. Kaduri, S. Bagon, and T. Dekel, “What’s in the Image? A Deep-Dive into the Vision of Vision Language Models,” arXiv:2411.17491, 2024. [Online]. Tillgänglig: https://arxiv.org/abs/2411.17491. (Hämtad: 2026-05-10)

[2] Z. Li, X. Wu, H. Du, F. Liu, H. Nghiem, and G. Shi, “A Survey of State of the Art Large Vision Language Models: Alignment, Benchmark, Evaluations and Challenges,” in Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition Workshops (CVPRW), 2025, pp. 1578–1597. doi: 10.1109/CVPRW67362.2025.00147. [Online]. Tillgänglig: https://doi.org/10.1109/CVPRW67362.2025.00147. (Hämtad: 2026-05-10)

[3] S. Raschka, “Mixture-of-Experts (MoE),” Machine Learning FAQ, 2024. [Online]. Tillgänglig: https://sebastianraschka.com/faq/docs/mixture-of-experts.html. (Hämtad: 2026-05-21)

[4] Qwen Team, "Qwen3-VL-30B-A3B-Instruct,"Hugging Face, [Online]. Tillgänglig: `https://huggingface.co/Qwen/Qwen3-VL-30B-A3B-Instruct`. (Hämtad: 2026-04-03).

[5] Google, Gemma 4 31B IT,"Hugging Face, [Online]. Tillgänglig: `https://huggingface.co/google/gemma-4-31B-it`. (Hämtad: 2026-04-03).

[6] Mistral AI, Ministral-3-14B-Instruct-2512,"Hugging Face, [Online]. Tillgänglig: `https://huggingface.co/mistralai/Ministral-3-14B-Instruct-2512`. (Hämtad: 2026-04-03).

[7] OpenAI, “OpenAI open-weight models (gpt-oss),” OpenAI Help Center, 2025. [Online]. Tillgänglig: https://help.openai.com/en/articles/11870455-openai-open-weight-models-gpt-oss. (Hämtad: 2026-05-10)

[8] Ollama, Öllama,"[Online]. Tillgänglig: `https://ollama.com/`. (Hämtad: 2026-04-03).

[9] F. S. Marcondes, A. Gala, R. Magalhães, F. P. de Britto, D. Durães och P. Novais, *Natural Language Analytics with Generative Large-Language Models*. 1. uppl., Cham, Schweiz: Springer, 2025.

[10] Ollama, Öllama Library,"[Online]. Tillgänglig: `https://ollama.com/library`. (Hämtad: 2026-04-13).

[11] IBM, Large Language Models,"IBM Think, [Online]. Tillgänglig: https://www.ibm.com/think/topics/large-language-models. (Hämtad: 2026-04-13).

[12] IBM, "Instruction Tuning,"IBM Think, [Online]. Tillgänglig: https://www.ibm.com/think/topics/instruction-tuning. (Hämtad: 2026-04-14).

[13] OpenAI, öpenai (npm package): Official JavaScript/TypeScript library for the OpenAI API,"[Online]. Tillgänglig: https://www.npmjs.com/package/openai. (Hämtad: 2026-04-10).

[14] Ollama, "API Introduction," [Online]. Tillgänglig: https://docs.ollama.com/api/introduction. (Hämtad: 2026-04-08).

[15] OpenAI, ÄPI Reference Overview,"[Online]. Tillgänglig: https://developers.openai.com/api/reference/overview. (Hämtad: 2026-04-10).

[16] Ekelund, Produktkatalog / bilddataset från sortiment,Tillgänglig: https://ekelunds.com/sv, år 2026.

[17] OpenAI, "Chat Completions API Reference (Create Chat Completion),"[Online]. Tillgänglig: https://developers.openai.com/api/reference/resources/chat/subresources/completions/methods/create. (Hämtad: 2026-04-03).

[18] OpenAI, Function Calling in the OpenAI API,"[Online]. Tillgänglig: https://developers.openai.com/api/docs/guides/function-calling. (Hämtad: 2026-04-11).

[19] OpenJS Foundation, "Node.js," i Node.js Documentation, USA, 2026. [Online]. Tillgänglig: https://nodejs.org/en. Hämtad: 2026-05-16.

[20] OpenJS Foundation / Microsoft, "TypeScript," i TypeScript Documentation, USA, 2026. [Online]. Tillgänglig: https://www.typescriptlang.org/. Hämtad: 2026-05-16.

[21] OpenJS Foundation, "Express," i Express.js Documentation, USA, 2026. [Online]. Tillgänglig: https://expressjs.com/. Hämtad: 2026-05-16.

[22] Meta Platforms, "React," i React Documentation, USA, 2026. [Online]. Tillgänglig: https://react.dev/. Hämtad: 2026-05-16.

[23] Vite Team, "Vite," i Vite Documentation, 2026. [Online]. Tillgänglig: https://vite.dev/. Hämtad: 2026-05-16.

[24] Colin McDonnell, "Zod – TypeScript-first schema validation library," Zod Documentation. [Online]. Tillgänglig: https://zod.dev/.(Hämtad: 2026-04-21)

[25] "zod-to-json-schema," npm package. [Online]. Tillgänglig: https://www.npmjs.com/package/zod-to-json-schema. (Hämtad: 2026-04-21)

[26] "sharp," High performance Node.js image processing. [Online]. Tillgänglig: https://sharp.pixelplumbing.com/. (Hämtad: 2026-04-29)

[27] “libvips/libvips,” GitHub repository. [Online]. Tillgänglig: https://github.com/libvips/libvips. (Hämtad: 2026-04-29)

[28] G. Naidu, T. Zuva och E. M. Sibanda, ”A Review of Evaluation Metrics in Machine Learning Algorithms,” i *Artificial Intelligence Application in Networks and Systems*, CSOC 2023, Cham, Schweiz, 2023, ss. 19–31. [Online]. Tillgänglig: `https://doi.org/10.1007/978-3-031-35314-7_2`, Hämtad: 2026-05-16.

[29] OpenAI, Structured Outputs,Developers Docs. [Online]. Tillgängli: https://developers.openai.com/api/docs/guides/structured-outputs. (Hämtad: 2026-04-21)

[30] M. Schall and G. de Melo, “The Hidden Cost of Structure: How Constrained Decoding Affects Language Model Performance,” in *Proc. 15th Int. Conf. Recent Advances in Natural Language Processing (RANLP)*, Varna, Bulgaria, Sep. 2025, pp. 1074–1084. [Online]. Tillgänglig: https://aclanthology.org/2025.ranlp-1.124/

[31] OpenAI, "Introducing Structured Outputs in the API,ÖpenAI Blog, 2024. [Online]. Tillgänglig: https://openai.com/index/introducing-structured-outputs-in-the-api/. (Hämtad: 2026-04-21)

[32] Y. Abd Alrahman, “Good Design,” presentation slides for DAT055 Object Oriented Applications, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, unpublished.

[33] E. Gamma, R. Helm, R. Johnson och J. Vlissides, "Creational Patterns,"i *Design Patterns: Elements of Reusable Object-Oriented Software*, 1:a uppl., Reading, MA, USA: Addison-Wesley, 1994, kap. 3, ss. 97-107.

[34] A. J. de Margerie, A. Roger, and I. Rish, “Image Tiling for High-Resolution Reasoning: Balancing Local Detail with Global Context,” arXiv:2512.11167, 2025. [Online]. Tillgänglig: https://arxiv.org/abs/2512.11167. (Hämtad: 2026-04-29)

[35] J. Almeida, “Prompt Engineering: A Comparative Study of Prompting Techniques in AI Language Models,” in *Proc. 2025 IEEE Integrated STEM Education Conference (ISEC)*, 2025, pp. 1–4, doi: 10.1109/ISEC64801.2025.11147384.

[36] “jsonrepair,” npm package. [Online]. Tillgänglig: https://www.npmjs.com/package/jsonrepair. (Hämtad: 2026-05-02)

[37] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, “Robust and efficient fuzzy match for online data cleaning,” in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD ’03)*, San Diego, CA, USA, 2003, pp. 313–324.

33

INSTITUTIONEN FÖR något ämne
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS