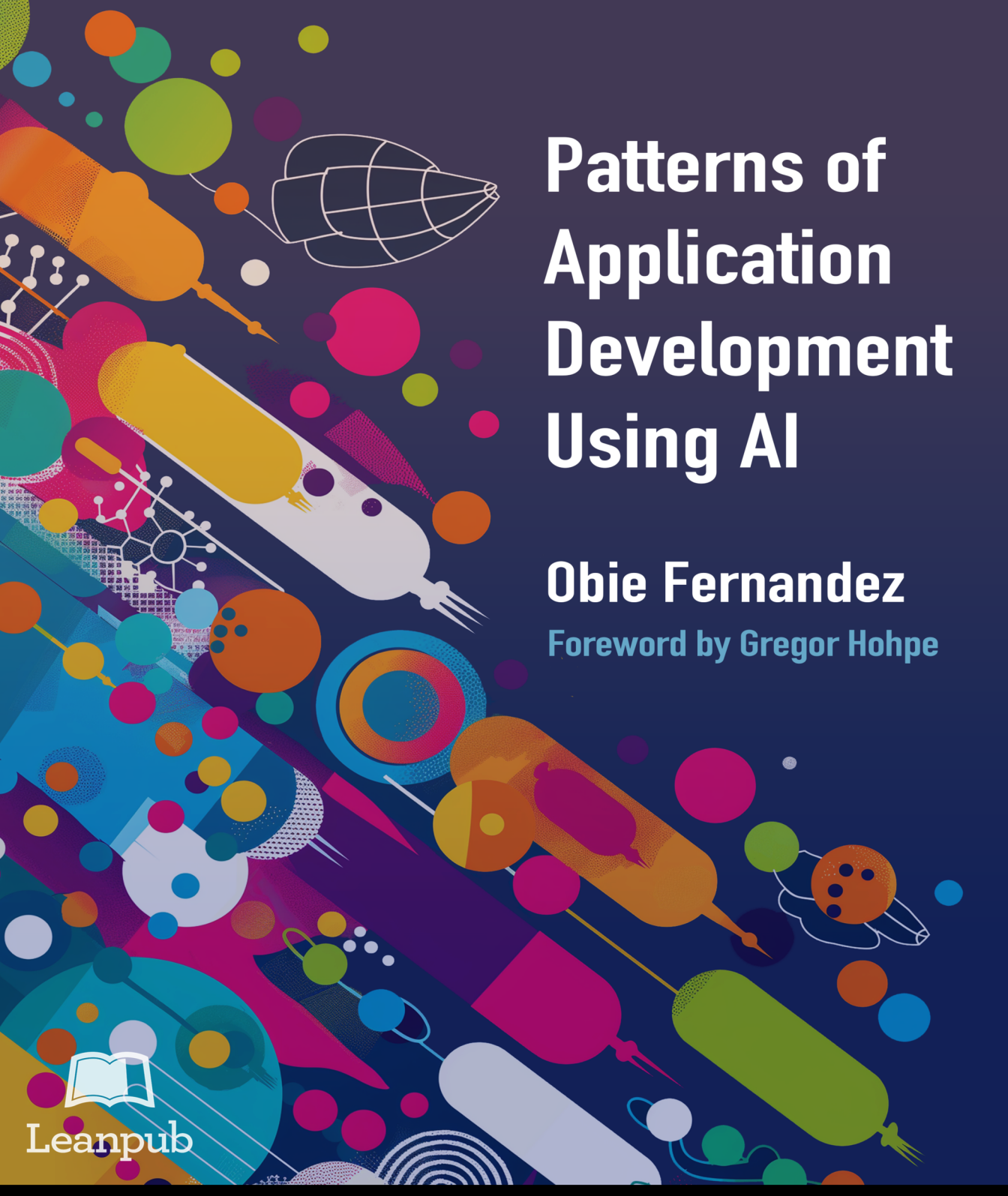



**Svensk Utgåva**

# Mönster för Applikationsutveckling med AI (Svensk Utgåva)

Obie Fernandez

Denna bok är till salu på
http://leanpub.com/patterns-of-application-development-using-ai-sv

Denna version publicerades den 2025-01-23


Det här är en Leanpub bok. Leanpub möjliggör för författare och förlag att använda processen för Lean Publishing. Lean Publishing innebär att man publicerar en ebook som är under arbete, använder lätta verktyg och genomför flera iterationer för att samla in läsaråsikter, anpassa innehållet tills den rätta boken tar form och skapa intresse för den när detta uppnås.

# Twittra denna bok!

Hjälp Obie Fernandez genom att berätta om den här boken på Twitter!

Den föreslagna hashtaggen för den här boken är #poaduai.

Ta reda på vad andra säger om den här boken genom att följa den här länken till en sökning på Twitter:

#poaduai

*Till min tuffa drottning, min musa, mitt ljus och min kärlek, Victoria*

# Också av Obie Fernandez

Patterns of Application Development Using AI

The Rails 8 Way

The Rails 7 Way

XML The Rails Way

Serverless

El Libro Principiante de Node

The Lean Enterprise

# Innehåll

# Del 1: Grundläggande metoder och tekniker 35

# Förord av Gregor Hohpe

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Förord

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Om boken

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Om kodexemplen

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Vad jag inte täcker

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Vem denna bok är till för

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Att bygga ett gemensamt vokabulär

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Bli involverad

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Erkännanden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Vad är det med illustrationerna?

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

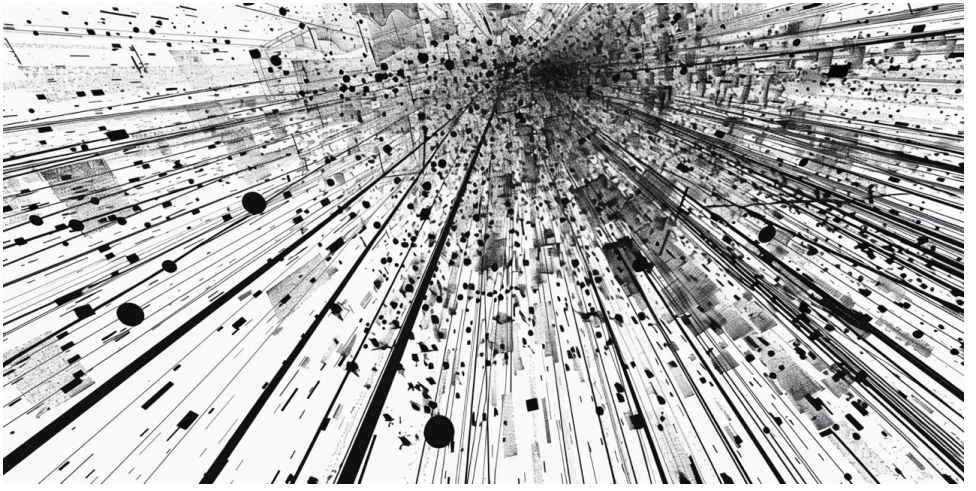
## Om Lean Publishing

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Om författaren

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Introduktion



Om du är ivrig att börja integrera AI-baserade stora språkmodeller (LLM) i dina programmeringsprojekt, kan du direkt hoppa till mönstren och kodexemplen som presenteras i senare kapitel. Men för att fullt ut uppskatta kraften och potentialen i dessa mönster är det värt att ta sig tid att förstå det bredare sammanhanget och det sammanhängande tillvägagångssättet de representerar.

Mönstren är inte bara en samling isolerade tekniker utan snarare ett enhetligt ramverk för att integrera AI i dina applikationer. Jag använder Ruby on Rails, men dessa mönster bör fungera i princip alla andra programmeringsmiljöer. De behandlar ett brett spektrum av aspekter, från datahantering och prestandaoptimering till användarupplevelse och säkerhet, och tillhandahåller en omfattande verktygslåda för att förbättra traditionella programmeringstekniker med AI:s möjligheter.

Varje kategori av mönster tar itu med en specifik utmaning eller möjlighet som uppstår när man införlivar AI-komponenter i din applikation. Genom att förstå relationerna och

synergierna mellan dessa mönster kan du fatta välgrundade beslut om var och hur AI kan användas mest effektivt.

Mönster är aldrig föreskrivande lösningar och bör inte behandlas som sådana. De är tänkta att vara anpassningsbara byggstenar som ska skräddarsys efter de unika kraven och begränsningarna i din egen applikation. En framgångsrik tillämpning av dessa mönster (precis som alla andra inom mjukvaruområdet) bygger på en djup förståelse för problemdomänen, användarnas behov och den övergripande tekniska arkitekturen i ditt projekt.

## Tankar om mjukvaruarkitektur

Jag började programmera på 1980-talet och var involverad i hackerscenen, och jag har aldrig förlorat mitt hackertänkande, även efter att ha blivit professionell mjukvaruutvecklare. Sedan start har jag alltid haft en hälsosam skepsis mot vilket värde mjukvaruarkitekter i sina elfenbenstorn faktiskt tillförde.

En av anledningarna till att jag personligen är så exalterad över förändringarna som denna kraftfulla nya våg av AI-teknik för med sig är dess påverkan på vad vi anser vara beslut kring *mjukvaruarkitektur*. Den utmanar traditionella uppfattningar om vad som utgör det "korrekta" sättet att designa och implementera våra mjukvaruprojekt. Den utmanar också huruvida arkitektur fortfarande kan betraktas främst som *de delar av ett system som är svåra att ändra*, eftersom AI-förbättringar gör det enklare än någonsin att ändra vilken del som helst av ditt projekt, när som helst.

Kanske går vi in i toppåren för den "postmoderna" approachen till mjukvaruutveckling. I detta sammanhang syftar postmodern på ett fundamentalt skifte bort från traditionella paradigm, där utvecklare var ansvariga för att skriva och underhålla varje kodrad. Istället omfamnar den idén om att delegera uppgifter, såsom datamanipulering, komplexa algoritmer och till och med hela delar av applikationslogiken, till tredjepartsbibliotek och externa API:er. Detta postmoderna skifte representerar

en betydande avvikelse från den konventionella visdomen om att bygga applikationer från grunden, och det utmanar utvecklare att ompröva sin roll i utvecklingsprocessen.

Jag har alltid trott att bra programmerare endast skriver den kod som är absolut nödvändig att skriva, baserat på lärdomar från Larry Wall och andra hackerlegender som honom. Genom att minimera mängden skriven kod kan vi röra oss snabbare, minska ytan för buggar, förenkla underhållet och förbättra den övergripande tillförlitligheten i våra applikationer. Mindre kod låter oss fokusera på kärnaffärslogiken och användarupplevelsen, medan annat arbete delegeras till andra tjänster.

Nu när AI-drivna system kan hantera uppgifter som tidigare var exklusivt förbehållna mänskligt skriven kod, borde vi kunna vara ännu mer produktiva och agila, med ett större fokus än någonsin på att skapa affärsvärde och användarupplevelse.

Naturligtvis finns det avvägningar med att delegera stora delar av ditt projekt till AI-system, såsom potentiell förlust av kontroll och behovet av robust övervakning och återkopplingsmekanismer. Det är därför det kräver en ny uppsättning färdigheter och kunskaper, inklusive åtminstone viss grundläggande förståelse för hur AI fungerar.

## Vad är en stor språkmodell?

Stora språkmodeller (LLM) är en typ av artificiell intelligensmodell som har fått betydande uppmärksamhet under senare år, särskilt sedan lanseringen av GPT-3 av OpenAI 2020. Stora språkmodeller är designade för att bearbeta, förstå och generera mänskligt språk med anmärkningsvärd precision och flyt. I detta avsnitt tar vi en kort titt på hur stora språkmodeller fungerar och varför de är väl lämpade för att bygga intelligenta systemkomponenter.

I grunden baseras stora språkmodeller på algoritmer för djupinlärning, specifikt neurala nätverk. Dessa nätverk består av sammankopplade noder, eller neuroner, som bearbetar och överför information. Den arkitektur som oftast väljs för stora språkmodeller är

transformatormodellen, som har visat sig vara mycket effektiv för att hantera sekventiell data som text.

Transformermodeller baseras på uppmärksamhetsmekanismen och används främst för uppgifter som involverar sekventiell data, som naturlig språkbehandling. Transformers bearbetar indata samtidigt istället för sekventiellt, vilket gör att de kan fånga långväga beroenden mer effektivt. De har lager av uppmärksamhetsmekanismer som hjälper modellen att fokusera på olika delar av indatan för att förstå sammanhang och relationer.

Träningsprocessen för stora språkmodeller innebär att modellen exponeras för enorma mängder textdata, såsom böcker, artiklar, webbplatser och kodarkiv. Under träningen lär sig modellen att känna igen mönster, relationer och strukturer i texten. Den fångar språkets statistiska egenskaper, som grammatiska regler, ordassociationer och kontextuella betydelser.

En av de viktigaste teknikerna som används vid träning av stora språkmodeller är oövervakat lärande. Detta innebär att modellen lär sig från data utan explicit märkning eller vägledning. Den upptäcker mönster och representationer på egen hand genom att analysera samförekomsten av ord och fraser i träningsdatan. Detta gör att stora språkmodeller kan utveckla en djup förståelse för språk och dess komplexitet.

En annan viktig aspekt av stora språkmodeller är deras förmåga att hantera *kontext*. När de bearbetar en text tar stora språkmodeller inte bara hänsyn till enskilda ord utan också till den omgivande kontexten. De beaktar tidigare ord, meningar och till och med stycken för att förstå textens betydelse och avsikt. Denna kontextuella förståelse gör att stora språkmodeller kan generera sammanhängande och relevanta svar. Ett av de huvudsakliga sätten vi utvärderar förmågorna hos en given språkmodell är genom att beakta storleken på den kontext de kan ta hänsyn till för att generera svar.

När de väl är tränade kan stora språkmodeller användas för ett brett spektrum av språkrelaterade uppgifter. De kan generera människoliknande text, svara på frågor, sammanfatta dokument, översätta språk och till och med skriva kod. Mångsidigheten hos stora språkmodeller gör dem värdefulla för att bygga intelligenta

systemkomponenter som kan interagera med användare, bearbeta och analysera textdata samt generera meningsfulla resultat.

Genom att införliva stora språkmodeller i applikationsarkitekturen kan du skapa AI-komponenter som förstår och bearbetar användarinput, genererar dynamiskt innehåll och ger intelligenta rekommendationer eller åtgärder. Men att arbeta med stora språkmodeller kräver noggranna överväganden av resurskrav och prestandaavvägningar. Stora språkmodeller är beräkningsintensiva och kan kräva betydande processorkraft och minne (med andra ord, pengar) för att fungera. De flesta av oss behöver bedöma kostnadskonsekvenserna av att integrera stora språkmodeller i våra applikationer och agera därefter.

## Förstå inferens

Inferens syftar på processen där en modell genererar förutsägelser eller output baserat på ny, tidigare osedd data. Det är fasen där den tränade modellen används för att fatta beslut eller generera text, bilder eller annat innehåll som svar på användarinput.

Under träningsfasen lär sig en AI-modell från en stor datamängd genom att justera sina parametrar för att minimera felet i sina förutsägelser. När modellen väl är tränad kan den tillämpa det den har lärt sig på ny data. Inferens är hur modellen använder sina inlärda mönster och kunskap för att generera output.

För stora språkmodeller innebär inferens att ta en prompt eller inputtext och producera ett sammanhängande och kontextuellt relevant svar, som en ström av *token* (vilket vi snart ska prata om). Detta kan vara att svara på en fråga, slutföra en mening, generera en berättelse eller översätta text, bland många andra uppgifter.

Till skillnad från hur du och jag tänker, sker en AI-modells "tänkande" via inferens i en enda tillståndslös operation. Det vill säga, dess tänkande är begränsat till dess genereringsprocess. Den måste bokstavligen tänka högt, som om jag ställde dig en fråga och endast accepterade ett svar från dig i "stream of consciousness"-stil.

## Stora språkmodeller kommer i många storlekar och varianter

Medan praktiskt taget alla populära stora språkmodeller är baserade på samma grundläggande transformerarkitektur och tränade på enorma textdatamängder, kommer de i olika storlekar och är finjusterade för olika ändamål. Storleken på en stor språkmodell, mätt i antalet parametrar i dess neurala nätverk, har stor påverkan på dess förmågor. Större modeller med fler parametrar, som GPT-4, som ryktas ha 1 till 2 biljoner parametrar, är generellt mer kunniga och kapabla än mindre modeller. Större modeller kräver dock mycket mer beräkningskraft för att köra, vilket översätts till högre kostnader när du använder dem via API-anrop.

För att göra stora språkmodeller mer praktiska och anpassade för specifika användningsfall blir basmodellerna ofta finjusterade på mer riktade datamängder. Till exempel kan en stor språkmodell tränas på en stor samling dialoger för att specialisera den för konversations-AI. Andra är tränade på kod för att ge dem programmeringskunskaper. Det finns till och med modeller som är särskilt tränade för rollspelsliknande interaktioner med användare!

## Hämtning vs Generativa Modeller

I världen av stora språkmodeller (LLMs) finns det två huvudsakliga sätt att generera svar: hämtningsbaserade modeller och generativa modeller. Varje metod har sina egna styrkor och svagheter, och genom att förstå skillnaderna mellan dem kan du välja rätt modell för ditt specifika användningsfall.

### Hämtningsbaserade Modeller

Hämtningsbaserade modeller, även kända som informationshämtningsmodeller, genererar svar genom att söka igenom en stor databas med befintlig text och välja ut de mest relevanta avsnitten baserat på sökfrågan. Dessa modeller genererar inte ny

text från grunden utan sammanfogar istället utdrag från databasen för att skapa ett sammanhängande svar.

En av de största fördelarna med hämtningsbaserade modeller är deras förmåga att tillhandahålla faktamässigt korrekt och aktuell information. Eftersom de förlitar sig på en databas med kurerad text kan de hämta relevant information från pålitliga källor och presentera den för användaren. Detta gör dem väl lämpade för tillämpningar som kräver precisa, faktabaserade svar, såsom frågebesvarande system eller kunskapsbaser.

Dock har hämtningsbaserade modeller vissa begränsningar. De är bara så bra som databasen de söker igenom, så kvaliteten och täckningen av databasen påverkar direkt modellens prestanda. Dessutom kan dessa modeller ha svårt att generera sammanhängande och naturligt klingande svar, eftersom de är begränsade till den text som finns tillgänglig i databasen.

Vi täcker inte användningen av rena hämtningsmodeller i denna bok.

## Generativa Modeller

Generativa modeller skapar å andra sidan ny text från grunden baserat på mönster och samband som de lärt sig under träningen. Dessa modeller använder sin förståelse av språk för att generera nya svar som är anpassade till ingångsprompt:en.

Den största styrkan hos generativa modeller är deras förmåga att producera kreativ, sammanhängande och kontextuellt relevant text. De kan delta i öppna konversationer, generera berättelser och till och med skriva kod. Detta gör dem idealiska för tillämpningar som kräver mer öppna och dynamiska interaktioner, såsom chatbottar, innehållsskapande och kreativa skrivassistenter.

Dock kan generativa modeller ibland producera inkonsekvent eller faktamässigt felaktig information, eftersom de förlitar sig på mönster som lärts in under träningen snarare än en kurerad faktadatabas. De kan också vara mer benägna till fördomar och hallucinationer, där de genererar text som är trovärdig men inte nödvändigtvis sann.

Exempel på generativa LLMs inkluderar OpenAIs GPT-serie (GPT-3, GPT-4) och Anthropics Claude.

## Hybridmodeller

Flera kommersiellt tillgängliga LLMs kombinerar både hämtnings- och generativa metoder i en hybridmodell. Dessa modeller använder hämtningstekniker för att hitta relevant information från en databas och använder sedan generativa tekniker för att sammanställa informationen till ett sammanhängande svar.

Hybridmodeller strävar efter att kombinera den faktamässiga precisionen hos hämtningsbaserade modeller med de naturliga språkgenereringsförmågorna hos generativa modeller. De kan tillhandahålla mer tillförlitlig och aktuell information samtidigt som de behåller förmågan att delta i öppna konversationer.

När du väljer mellan hämtningsbaserade och generativa modeller bör du överväga de specifika kraven för din tillämpning. Om det primära målet är att tillhandahålla korrekt, faktabaserad information kan en hämtningsbaserad modell vara det bästa valet. Om tillämpningen kräver mer öppna och kreativa interaktioner kan en generativ modell vara mer lämplig. Hybridmodeller erbjuder en balans mellan de två metoderna och kan vara ett bra val för tillämpningar som kräver både faktamässig precision och naturlig språkgenerering.

I slutändan beror valet mellan hämtningsbaserade och generativa modeller på det specifika användningsfallet och avvägningarna mellan precision, kreativitet och flexibilitet. Genom att förstå styrkorna och begränsningarna hos varje metod kan du fatta välgrundade beslut.

## Instruktionsanpassning

Instruktionsanpassning är en delmängd av konversationsträning som använder människoskrivna prompts och svar för att göra LLMs bättre på att förstå och följa instruktioner. Här är några enkla exempel för att illustrera hur instruktionsanpassning på människoskrivna prompts och svar gör LLMs bättre på att förstå och följa instruktioner:

### Exempel 1: Kundsupport

Instruktion: "Ge ett svar till en kund som ber om återbetalning."

Människoskriven Prompt: "Jag skulle vilja begära återbetalning för mitt senaste köp. Produkten motsvarade inte mina förväntningar."

Människoskrivet Svar: "Jag beklagar att produkten inte motsvarade dina förväntningar. För att behandla din återbetalning, vänligen ange ditt ordernummer och anledningen till returen. Vi kommer att granska din förfrågan och återkomma inom 2-3 arbetsdagar."

### Exempel 2: Pedagogisk Assistans

Instruktion: "Förklara konceptet fotosyntes för en 10-åring."

Human-Written Prompt: "Kan du förklara hur växter tillverkar sin mat?"

Human-Written Response: "Javisst! Växter tillverkar sin mat genom en process som kallas fotosyntes. De använder solljus, vatten och koldioxid för att skapa mat och syre. Växternas blad har en speciell del som kallas klorofyll som hjälper dem att fånga solljus."

### Example 3: Personlig Assistent

> Instruction: “Schemalägg ett möte till nästa tisdag klockan 15:00.”
>
> Human-Written Prompt: “Kan du boka ett möte till tisdag klockan 15:00?”
>
> Human-Written Response: “Javisst, jag har schemalagt ett möte åt dig nästa tisdag klockan 15:00. Behöver du något mer?”

Resultatet är ett varierat ekosystem av stora språkmodeller i olika storlekar och med olika specialiteter. Mindre modeller i intervallet 1-7 miljarder parametrar ger bra allmänna språkfärdigheter samtidigt som de är mer effektiva att köra.

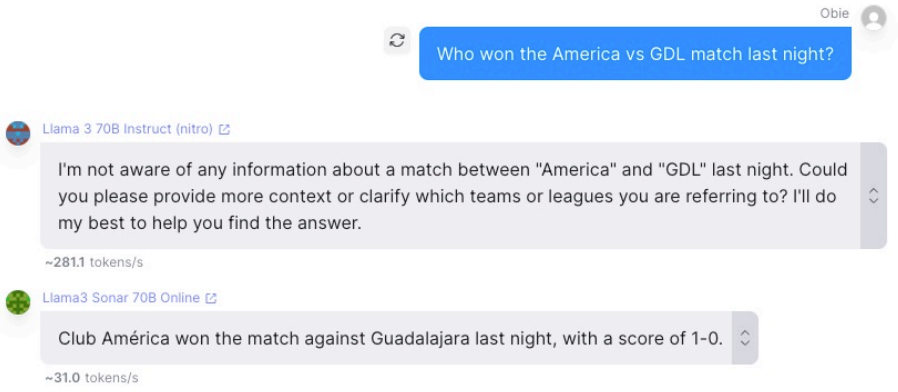
- Mistral 7B
- Llama 3 8B
- Gemma 7B

Medelstora modeller runt 30-70 miljarder parametrar erbjuder starkare resonemang och förmåga att följa instruktioner.

- Llama 3 70B
- Qwen2 70B
- Mixtral 8x22B

När man väljer en språkmodell att införliva i en applikation måste man balansera modellens förmågor mot praktiska faktorer som kostnad, latens, kontextlängd och innehållsfiltrering. Mindre, instruktionsanpassade modeller är ofta det bästa valet för enklare språkuppgifter, medan de största modellerna kan behövas för komplext resonemang eller analys. Modellens träningsdata är också en viktig faktor, eftersom den avgör modellens kunskapsavgränsning i tid.

Vissa modeller, som några från Perplexity, är anslutna till informationskällor i realtid, så att de i praktiken inte har någon tidsgräns för sin kunskap. När du ställer frågor till dem kan de självständigt besluta att göra webbsökningar och hämta godtyckliga webbsidor för att generera ett svar.




**Figur 1. Llama3 med och utan onlineåtkomst**

I slutändan finns det ingen universallösning när det gäller språkmodeller. Att förstå variationerna i modellstorlek, arkitektur och träning är nyckeln till att välja rätt modell för ett givet användningsfall. Att experimentera med olika modeller är det enda praktiska sättet att upptäcka vilka som ger bäst prestanda för den aktuella uppgiften.

## Tokenisering: Att dela upp text i bitar

Innan en stor språkmodell kan bearbeta text måste texten delas upp i mindre enheter som kallas *tokens*. Tokens kan vara enskilda ord, delar av ord eller till och med enskilda tecken. Processen att dela upp text i tokens kallas tokenisering, och det är ett avgörande steg i att förbereda data för en språkmodell.


**Figur 2. Denna mening innehåller 27 tokens**

Olika språkmodeller använder olika tokeniseringsstrategier, vilket kan ha en betydande inverkan på modellens prestanda och förmågor. Några vanliga tokeniserare som används

av språkmodeller inkluderar:

- **GPT (Byte Pair Encoding)**: GPT-tokeniserare använder en teknik som kallas byte pair encoding (BPE) för att dela upp text i delordenheter. BPE slår iterativt samman de mest frekventa paren av bytes i en textkorpus och bildar ett ordförråd av delordstokens. Detta gör att tokeniseraren kan hantera ovanliga och nya ord genom att dela upp dem i vanligare delord. GPT-tokeniserare används av modeller som GPT-3 och GPT-4.

- **Llama (SentencePiece)**: Llama-tokeniserare använder SentencePiece-biblioteket, som är en oövervakad texttokeniserare och detokeniserare. SentencePiece behandlar indatatexten som en sekvens av Unicode-tecken och lär sig ett delordförråd baserat på en träningskorpus. Den kan hantera alla språk som kan kodas i Unicode, vilket gör den väl lämpad för flerspråkiga modeller. Llama-tokeniserare används av modeller som Metas Llama och Alpaca.

- **SentencePiece (Unigram)**: SentencePiece-tokeniserare kan också använda en annan algoritm som kallas Unigram, vilken baseras på en delordregleringsteknik. Unigram-tokenisering bestämmer det optimala delordförrådet baserat på en unigram-språkmodell, som tilldelar sannolikheter till enskilda delordenheter. Detta tillvägagångssätt kan producera mer semantiskt meningsfulla delord jämfört med BPE. SentencePiece med Unigram används av modeller som Googles T5 och BERT.

- **Google Gemini (Multimodal Tokenisering)**: Google Gemini använder ett tokeniseringsschema utformat för att hantera olika datatyper, inklusive text, bilder, ljud, videor och kod. Denna multimodala kapacitet låter Gemini bearbeta och integrera olika former av information. Särskilt anmärkningsvärt är att Google Gemini 1.5 Pro har ett kontextfönster som kan hantera miljontals tokens, mycket större än tidigare modeller. Detta omfattande kontextfönster gör det möjligt för

modellen att bearbeta en större kontext, vilket potentiellt leder till mer korrekta svar. Det är dock viktigt att notera att Geminis tokeniseringsschema ligger mycket närmare en token per tecken än andra modeller. Detta innebär att den faktiska kostnaden för att använda Gemini-modeller kan vara betydligt högre än förväntat om du är van vid att använda modeller som GPT, eftersom Googles prissättning baseras på tecken snarare än tokens.

Valet av tokeniserare påverkar flera aspekter av en LLM, inklusive:

- **Ordförrådsstorlek**: Tokeniseraren bestämmer storleken på modellens ordförråd, vilket är uppsättningen av unika tokens som den känner igen. Ett större, mer fingranulerat ordförråd kan hjälpa modellen att hantera ett bredare spektrum av ord och fraser och till och med bli multimodal (kapabel att förstå och generera mer än bara text), men det ökar också modellens minneskrav och beräkningskomplexitet.
- **Hantering av ovanliga och okända ord**: Tokeniserare som använder delordenheter, som BPE och SentencePiece, kan bryta ner ovanliga och okända ord i mer vanliga delord. Detta gör att modellen kan göra kvalificerade gissningar om betydelsen av ord den inte har sett förut, baserat på delorden de innehåller.
- **Flerspråkigt stöd**: Tokeniserare som SentencePiece, som kan hantera alla Unicode-kodbara språk, är väl lämpade för flerspråkiga modeller som behöver bearbeta text på flera språk.

När man väljer en LLM för en specifik tillämpning är det viktigt att överväga vilken tokeniserare den använder och hur väl den överensstämmer med de specifika språkbearbetningsbehoven för uppgiften. Tokeniseraren kan ha en betydande inverkan på modellens förmåga att hantera domänspecifik terminologi, ovanliga ord och flerspråkig text.

# Kontextstorlek: Hur mycket information kan en språkmodell använda under inferens?

När man diskuterar språkmodeller syftar kontextstorlek på mängden text som en modell kan ta hänsyn till när den bearbetar eller genererar sina svar. Det är i grund och botten ett mått på hur mycket information modellen kan "komma ihåg" och använda för att informera sina utdata (uttryckt i tokens). Kontextstorleken hos en språkmodell kan ha en betydande inverkan på dess förmågor och vilka typer av uppgifter den kan utföra effektivt.

## Vad är kontextstorlek?

I tekniska termer bestäms kontextstorleken av antalet tokens (ord eller orddelar) som en språkmodell kan bearbeta i en enda indatasekvens. Detta kallas ofta modellens "uppmärksamhetsomfång" eller "kontextfönster". Ju större kontextstorlek, desto mer text kan modellen ta hänsyn till på en gång när den genererar ett svar eller utför en uppgift.

Olika språkmodeller har varierande kontextstorlekar, från några hundra tokens till miljontals tokens. Som referens kan ett typiskt stycke text innehålla omkring 100-150 tokens, medan en hel bok kan innehålla tiotusentals eller hundratusentals tokens.

> Det pågår till och med arbete med effektiva metoder för att skala Transformer-baserade stora språkmodeller (LLM) till oändligt långa indata med begränsat minne och beräkning.

## Varför är kontextstorlek viktigt?

Kontextstorleken hos en språkmodell har en betydande påverkan på dess förmåga att förstå och generera sammanhängande, kontextuellt relevant text. Här är några viktiga

anledningar till varför kontextstorlek spelar roll:

1. **Förståelse av längre innehåll**: Modeller med större kontextstorlek kan bättre förstå och analysera längre texter, som artiklar, rapporter eller till och med hela böcker. Detta är avgörande för uppgifter som dokumentsammanfattning, frågebesvarande och innehållsanalys.

2. **Bibehålla sammanhang**: Ett större kontextfönster låter modellen bibehålla sammanhang och konsekvens över längre textstycken. Detta är viktigt för uppgifter som berättelsegenerering, dialogsystem och innehållsskapande, där det är viktigt att upprätthålla en konsekvent berättelse eller ämne. Det är också absolut avgörande när man använder stora språkmodeller för att generera eller omvandla strukturerad data.
3. **Fånga långdistansberoenden**: Vissa språkuppgifter kräver förståelse för relationer mellan ord eller fraser som ligger långt ifrån varandra i en text. Modeller med större kontextstorlek är bättre rustade att fånga dessa långdistansberoenden, vilket kan vara viktigt för uppgifter som känslighetsanalys, översättning och språkförståelse.
4. **Hantera komplexa instruktioner**: I tillämpningar där språkmodeller används för att följa komplexa instruktioner i flera steg, möjliggör en större kontextstorlek att modellen kan ta hänsyn till hela uppsättningen instruktioner när den genererar ett svar, istället för bara de senaste orden.

## Exempel på språkmodeller med olika kontextstorlekar

Här är några exempel på språkmodeller med olika kontextstorlekar:

- OpenAI GPT-3.5 Turbo: 4 095 tokens
- Mistral 7B Instruct: 32 768 tokens
- Anthropic Claude v1: 100 000 tokens

- OpenAI GPT-4 Turbo: 128 000 tokens
- Anthropic Claude v2: 200 000 tokens
- Google Gemini Pro 1.5: 2,8M tokens

Som du kan se finns det ett brett spektrum av kontextstorlekar bland dessa modeller, från omkring 4 000 tokens för OpenAI GPT-3.5 Turbo-modellen till 200 000 tokens för Anthropic Claude v2-modellen. Vissa modeller, som Googles PaLM 2 och OpenAIs GPT-4, erbjuder olika varianter med större kontextstorlekar (t.ex. "32k"-versioner), som kan hantera ännu längre inmatningssekvenser. Och för närvarande (april 2024) skryter Google Gemini Pro med nästan 3 miljoner tokens!

Det är värt att notera att kontextstorleken kan variera beroende på den specifika implementeringen och versionen av en viss modell. Till exempel har den ursprungliga OpenAI GPT-4-modellen en kontextstorlek på 8 191 tokens, medan senare GPT-4-varianter som Turbo och 4o har en mycket större kontextstorlek på 128 000 tokens.

> Sam Altman har jämfört nuvarande kontextbegränsningar med de kilobyte arbetsminne som persondatorprogrammerare behövde hantera på 80-talet, och sagt att vi i den nära framtiden kommer att kunna passa in "alla dina personliga data" i kontexten för en stor språkmodell.

## Välja rätt kontextstorlek

När man väljer en språkmodell för en specifik tillämpning är det viktigt att ta hänsyn till uppgiftens krav på kontextstorlek. För uppgifter som involverar korta, isolerade textstycken, som känslighetsanalys eller enkelt frågebesvarande, kan en mindre kontextstorlek vara tillräcklig. För uppgifter som kräver förståelse och generering av längre, mer komplexa texter kommer dock en större kontextstorlek sannolikt att vara nödvändig.

Det är värt att notera att större kontextstorlekar ofta medför ökade beräkningskostnader och långsammare bearbetningstider, eftersom modellen behöver ta hänsyn till mer information när den genererar ett svar. Därför måste du hitta en balans mellan kontextstorlek och prestanda när du väljer en språkmodell för din tillämpning.

> Varför inte bara välja modellen med störst kontextstorlek och fylla den med så mycket information som möjligt? Tja, förutom prestandafaktorer är den andra huvudsakliga aspekten kostnad. I mars 2024 kostar en *enda* fråga-svar-cykel med Google Gemini Pro 1.5 med full kontext nästan 8 dollar (USD). Om du har ett användningsfall som motiverar den kostnaden, all lycka till dig! Men för de flesta tillämpningar är det helt enkelt för dyrt med flera storleksordningar.

## Att hitta nålar i höstackar

Konceptet att hitta en nål i en höstack har länge varit en metafor för utmaningarna med informationshämtning i stora datamängder. Inom området för LLM:er justerar vi denna analogi något. Föreställ dig att vi inte bara letar efter ett enda faktum begravt i en omfattande text (som en fullständig antologi av Paul Graham essäer), utan flera fakta utspridda överallt. Detta scenario liknar mer att hitta flera nålar i ett vidsträckt fält, inte bara en enda höstack. Här är det avgörande: vi måste inte bara lokalisera dessa nålar, utan också väva samman dem till en sammanhängande tråd.

När LLM:er ställs inför uppgiften att hämta och resonera kring flera fakta inbäddade i långa kontexter, möter de en dubbel utmaning. Först finns det det uppenbara problemet med hämtningsprecision - den sjunker naturligt när antalet fakta ökar. Detta är förväntat; när allt kommer omkring belastar det att hålla reda på flera detaljer i en omfattande text även de mest sofistikerade modellerna.

För det andra, och kanske mer kritiskt, är utmaningen att resonera med dessa fakta.

Det är en sak att plocka ut fakta; det är en helt annan att syntetisera dem till en sammanhängande berättelse eller ett svar. Det är här det verkliga provet kommer in. LLM:ers prestanda i resonemanguppgifter tenderar att försämras mer än i enkla hämtningsuppgifter. Denna försämring handlar inte bara om volym; det handlar om det intrikata samspelet mellan kontext, relevans och slutledning.

Varför händer detta? Tja, överväg dynamiken i minne och uppmärksamhet i mänsklig kognition, som i viss mån speglas i LLM:er. När de bearbetar stora mängder information kan LLM:er, precis som människor, tappa spåret av tidigare detaljer när de absorberar nya. Detta är särskilt sant för modeller som inte uttryckligen är designade för att prioritera eller automatiskt återbesöka tidigare textsegment.

Dessutom är en LLM:s förmåga att väva samman dessa hämtade fakta till ett sammanhängande svar besläktat med berättelseskapande. Detta kräver inte bara hämtning av information utan en djup förståelse och kontextuell placering, vilket förblir en stor utmaning för dagens AI.

Så vad betyder detta för oss som utvecklare och integratörer av dessa teknologier? Vi måste vara mycket medvetna om dessa begränsningar när vi utformar system som förlitar sig på LLM:er för att hantera komplexa, långformiga uppgifter. Att förstå att prestandan kan försämras under vissa förhållanden hjälper oss att sätta realistiska förväntningar och konstruera bättre reservmekanismer eller kompletterande strategier.

## Modaliteter: Bortom Text

Medan majoriteten av språkmodeller idag är fokuserade på att bearbeta och generera text, finns det en växande trend mot multimodala modeller som naturligt kan ta emot och producera flera typer av data, såsom bilder, ljud och video. Dessa multimodala modeller öppnar upp nya möjligheter för AI-drivna applikationer som kan förstå och generera innehåll över olika modaliteter.

## Vad är Modaliteter?

I sammanhanget språkmodeller syftar modaliteter på de olika typer av data som en modell kan bearbeta och generera. Den vanligaste modaliteten är text, vilket inkluderar skrivet språk i olika former som böcker, artiklar, webbplatser och sociala medieinlägg. Det finns dock flera andra modaliteter som i allt högre grad införlivas i språkmodeller:

- **Bilder**: Visuell data såsom fotografier, illustrationer och diagram.
- **Ljud**: Ljuddata såsom tal, musik och miljöljud.
- **Video**: Rörlig visuell data, ofta åtföljd av ljud, såsom videoklipp och filmer.

Varje modalitet presenterar unika utmaningar och möjligheter för språkmodeller. Till exempel kräver bilder att modellen förstår visuella koncept och relationer, medan ljud kräver att modellen bearbetar och genererar tal och andra ljud.

## Multimodala Språkmodeller

Multimodala språkmodeller är designade för att hantera flera modaliteter inom en enda modell. Dessa modeller har vanligtvis specialiserade komponenter eller lager som både kan förstå indata och generera utdata i olika modaliteter. Några noterbara exempel på multimodala språkmodeller inkluderar:

- **OpenAI's GPT-4o**: GPT-4o är en stor språkmodell som naturligt förstår och bearbetar talat ljud utöver text. Denna förmåga låter GPT-4o utföra uppgifter som att transkribera talat språk, generera text från ljudindata och ge svar baserade på talade frågor.
- **OpenAI's GPT-4 med visuell inmatning**: GPT-4 är en stor språkmodell som kan bearbeta både text och bilder. När den får en bild som indata kan GPT-4 analysera bildens innehåll och generera text som beskriver eller svarar på den visuella informationen.

- **Google's Gemini**: Gemini är en multimodal modell som kan hantera text, bilder och video. Den använder en enhetlig arkitektur som möjliggör tvärmodal förståelse och generering, vilket möjliggör uppgifter som bildtextning, videosammanfattning och visuell frågebesvarande.

- **DALL-E och Stable Diffusion**: Även om dessa inte är språkmodeller i traditionell bemärkelse demonstrerar de kraften i multimodal AI genom att generera bilder från textbeskrivningar. De visar potentialen hos modeller som kan översätta mellan olika modaliteter.

## Fördelar och tillämpningar av multimodala modeller

Multimodala språkmodeller erbjuder flera fördelar och möjliggör ett brett spektrum av tillämpningar, inklusive:

- **Förbättrad förståelse**: Genom att bearbeta information från flera modaliteter kan dessa modeller få en mer omfattande förståelse av världen, liknande hur människor lär sig från olika sensoriska intryck.
- **Korsmodal generering**: Multimodala modeller kan generera innehåll i en modalitet baserat på input från en annan, såsom att skapa en bild från en textbeskrivning eller generera en videosammanfattning från en skriven artikel.
- **Tillgänglighet**: Multimodala modeller kan göra information mer tillgänglig genom att översätta mellan modaliteter, såsom att generera textbeskrivningar av bilder för synskadade användare eller skapa ljudversioner av skrivet innehåll.
- **Kreativa tillämpningar**: Multimodala modeller kan användas för kreativa uppgifter som att generera konst, musik eller videor baserat på textprompter, vilket öppnar nya möjligheter för konstnärer och innehållsskapare.

I takt med att multimodala språkmodeller fortsätter att utvecklas kommer de sannolikt att spela en allt viktigare roll i utvecklingen av AI-drivna applikationer som kan förstå

och generera innehåll över flera modaliteter. Detta kommer att möjliggöra mer naturliga och intuitiva interaktioner mellan människor och AI-system, samt öppna upp nya möjligheter för kreativt uttryck och kunskapsspridning.

## Leverantörsekosystem

När det gäller att införliva stora språkmodeller (LLMs) i applikationer finns det ett växande utbud av alternativ att välja mellan. Varje större LLM-leverantör, såsom OpenAI, Anthropic, Google och Cohere, erbjuder sitt eget ekosystem av modeller, API:er och verktyg. Att välja rätt leverantör innebär att överväga olika faktorer, inklusive prissättning, prestanda, innehållsfiltrering, dataintegritet och anpassningsmöjligheter.

### OpenAI

OpenAI är en av de mest välkända leverantörerna av LLMs, där deras GPT-serie (GPT-3, GPT-4) används brett i olika applikationer. OpenAI erbjuder ett användarvänligt API som gör det enkelt att integrera deras modeller i applikationer. De tillhandahåller ett urval av modeller med olika kapacitet och prisnivåer, från instegsmodellen Ada till den kraftfulla Davinci-modellen.

OpenAIs ekosystem inkluderar också verktyg som OpenAI Playground, som låter dig experimentera med prompter och finjustera modeller för specifika användningsfall. De erbjuder innehållsfiltrering för att hjälpa till att förhindra generering av olämpligt eller skadligt innehåll.

När jag använder OpenAIs modeller direkt förlitar jag mig på Alex Rudalls ruby-openai-bibliotek.

### Anthropic

Anthropic är en annan stor aktör inom LLM-området, där deras Claude-modeller vinner popularitet för stark prestanda och etiska överväganden. Anthropic fokuserar på att

utveckla säkra och ansvarsfulla AI-system, med stark betoning på innehållsfiltrering och undvikande av skadlig output.

Anthropics ekosystem inkluderar Claude API, som låter dig integrera modellen i dina applikationer, samt verktyg för promptkonstruktion och finjustering. De erbjuder också Claude Instant-modellen, som integrerar webbsökning för mer aktuella och faktabaserade svar.

När jag använder Anthropics modeller direkt förlitar jag mig på Alex Rudalls anthrophic-bibliotek.

## Google

Google har utvecklat flera kraftfulla LLMs, inklusive Gemini, BERT, T5 och PaLM. Dessa modeller är kända för sin starka prestanda inom ett brett spektrum av naturlig språkbehandling. Googles ekosystem inkluderar TensorFlow- och Keras-biblioteken, som tillhandahåller verktyg och ramverk för att bygga och träna maskininlärningsmodeller.

Google erbjuder också en Cloud AI Platform, som gör det enkelt att distribuera och skala deras modeller i molnet. De tillhandahåller ett urval av förtränade modeller och API:er för uppgifter som känslighetsanalys, entitetsigenkänning och översättning.

## Meta

Meta, tidigare känt som Facebook, är djupt engagerade i utvecklingen av stora språkmodeller, vilket framhävs genom deras lansering av modeller som LLaMA och OPT. Dessa modeller utmärker sig genom sin starka prestanda i olika språkuppgifter och görs till stor del tillgängliga genom öppen källkod, vilket stöder Metas engagemang för forskning och samarbete inom gemenskapen.

Metas ekosystem är främst byggt kring PyTorch, ett maskininlärningsbibliotek med öppen källkod som föredras för sina dynamiska beräkningsmöjligheter och flexibilitet, vilket underlättar innovativ AI-forskning och utveckling.

Utöver sina tekniska erbjudanden lägger Meta stor vikt vid etisk AI-utveckling. De implementerar robust innehållsfiltrering och fokuserar på att minska partiskhet, i linje med sina bredare mål om säkerhet och ansvar i AI-tillämpningar.

## Cohere

Cohere är en nyare aktör inom LLM-området som fokuserar på att göra LLM:er mer tillgängliga och enklare att använda än konkurrenterna. Deras ekosystem inkluderar Cohere API, som ger tillgång till ett utbud av förtränade modeller för uppgifter som textgenerering, klassificering och sammanfattning.

Cohere erbjuder också verktyg för prompt-konstruktion, fine-tuning och innehållsfiltrering. De betonar dataintegritet och säkerhet, med funktioner som krypterad datalagring och åtkomstkontroller.

## Ollama

Ollama är en självhostad plattform som låter användare hantera och distribuera olika stora språkmodeller (LLM:er) lokalt på sina maskiner, vilket ger dem fullständig kontroll över sina AI-modeller utan att behöva förlita sig på externa molntjänster. Denna uppsättning är idealisk för dem som prioriterar dataintegritet och vill hantera sina AI-operationer internt.

Plattformen stöder ett antal modeller, inklusive versioner av Llama, Phi, Gemma och Mistral, som varierar i storlek och beräkningskrav. Ollama gör det enkelt att ladda ner och köra dessa modeller direkt från kommandoraden med enkla kommandon som `ollama run <model_name>`, och den är designad för att fungera på olika operativsystem inklusive macOS, Linux och Windows.

För utvecklare som vill integrera open source-modeller i sina applikationer utan att använda ett externt API erbjuder Ollama en CLI för hantering av modellers livscykler liknande containerhanteringsverktyg. Den stöder också anpassade konfigurationer och

prompts, vilket möjliggör en hög grad av anpassning för att skräddarsy modellerna för specifika behov eller användningsfall.

Ollama är särskilt lämpat för tekniskt kunniga användare och utvecklare på grund av dess kommandoradsgränssnitt och flexibiliteten den erbjuder i hantering och distribution av AI-modeller. Detta gör det till ett kraftfullt verktyg för företag och individer som behöver robusta AI-funktioner utan att kompromissa med säkerhet och kontroll.

### Multi-modellplattformar

Dessutom finns det leverantörer som är värdar för ett brett utbud av open source-modeller, såsom Together.ai och Groq.. Dessa plattformar erbjuder flexibilitet och anpassningsmöjligheter, vilket låter dig köra och i vissa fall även fine-tuna open source-modeller enligt dina specifika behov. Till exempel ger Together.ai tillgång till ett utbud av open source-LLM:er, vilket gör det möjligt för användare att experimentera med olika modeller och konfigurationer. Groq fokuserar på att leverera ultrahög prestanda som vid tiden för denna boks skrivande verkar nästan magisk

## Välja en LLM-leverantör

När du väljer en LLM-leverantör bör du överväga faktorer som:

- **Prissättning**: Olika leverantörer erbjuder olika prismodeller, från betala-per-användning till prenumerationsbaserade planer. Det är viktigt att överväga den förväntade användningen och budgeten när man väljer leverantör.
- **Prestanda**: Prestandan hos LLM:er kan variera betydligt mellan leverantörer, så det är viktigt att utvärdera och testa modeller för specifika användningsfall innan man fattar ett beslut.
- **Innehållsfiltrering**: Beroende på tillämpningen kan innehållsfiltrering vara en kritisk faktor. Vissa leverantörer erbjuder mer robusta alternativ för innehållsfiltrering än andra.

- **Dataintegritet**: Om applikationen hanterar känsliga användardata är det viktigt att välja en leverantör med starka rutiner för dataintegritet och säkerhet.
- **Anpassning**: Vissa leverantörer erbjuder mer flexibilitet när det gäller fine-tuning och anpassning av modeller för specifika användningsfall.

I slutändan beror valet av LLM-leverantör på applikationens specifika krav och begränsningar. Genom att noggrant utvärdera alternativen och överväga faktorer som prissättning, prestanda och dataintegritet kan du välja den leverantör som bäst möter dina behov.

Det är också värt att notera att LLM-landskapet ständigt utvecklas, med nya leverantörer och modeller som dyker upp regelbundet. Du bör hålla dig uppdaterad med de senaste utvecklingarna och vara öppen för att utforska nya alternativ när de blir tillgängliga.

### OpenRouter

Genom hela denna bok kommer jag att uteslutande förlita mig på OpenRouter som min API-leverantör. Anledningen är enkel: det är en one-stop-shop för alla de mest populära kommersiella och open source-modellerna. Om du är ivrig att börja experimentera med AI-kodning är en av de bästa platserna att börja med mitt eget OpenRouter Ruby Library.

## Att tänka på prestanda

När man integrerar språkmodeller i applikationer är prestanda en kritisk aspekt att ta hänsyn till. En språkmodells prestanda kan mätas i termer av dess *latens* (tiden det tar att generera ett svar) och *genomströmning* (antalet förfrågningar den kan hantera per tidsenhet).

*Tid till första token* (TTFT) är ett annat viktigt prestandamått, särskilt relevant för chatbottar och applikationer som kräver interaktiva svar i realtid. TTFT mäter latensen från det ögonblick en användares förfrågan tas emot till det ögonblick det första

ordet (eller token) i svaret genereras. Detta mått är avgörande för att upprätthålla en sömlös och engagerande användarupplevelse, eftersom fördröjda svar kan leda till användarfrustation och minskat engagemang.

Dessa prestandamått kan ha en betydande påverkan på användarupplevelsen och applikationens skalbarhet.

Flera faktorer kan påverka en språkmodells prestanda, inklusive:

**Parameterantal**: Större modeller med fler parametrar kräver generellt mer beräkningsresurser och kan ha högre latens och lägre genomströmning jämfört med mindre modeller.

**Hårdvara**: En språkmodells prestanda kan variera avsevärt beroende på vilken hårdvara den körs på. Molnleverantörer erbjuder GPU- och TPU-instanser optimerade för maskininlärningsarbetsbelastningar, vilket kan påskynda modellinferens avsevärt.

En av de fina sakerna med OpenRouter är att för många av modellerna det erbjuder får du ett val av molnleverantörer med olika prestandaprofiler och kostnader.

**Kvantisering**: Kvantiseringstekniker kan användas för att minska en modells minnesavtryck och beräkningskrav genom att representera vikter och aktiveringar med datatyper av lägre precision. Detta kan förbättra prestandan utan att avsevärt offra kvalitet. Som applikationsutvecklare kommer du förmodligen inte att bli involverad i att träna dina egna modeller på olika kvantiseringsnivåer, men det är bra att åtminstone vara bekant med terminologin.

**Batchbearbetning**: Att bearbeta flera förfrågningar samtidigt i batchar kan förbättra genomströmningen genom att fördela overheaden för modelladdning och dataöverföring.

**Cachning**: Att cacha resultaten av ofta använda prompter eller indatasekvenser kan minska antalet inferensförfrågningar och förbättra den övergripande prestandan.

När man väljer en språkmodell för en produktionsapplikation är det viktigt att utvärdera dess prestanda på representativa arbetsbelastningar och hårdvarukonfigurationer. Detta kan hjälpa till att identifiera potentiella flaskhalsar och säkerställa att modellen kan uppfylla de uppsatta prestandamålen.

Det är också värt att överväga avvägningarna mellan modellprestanda och andra faktorer som kostnad, flexibilitet och integreringsenkelheten. Till exempel kan användning av en mindre, billigare modell med lägre latens vara att föredra för applikationer som kräver realtidssvar, medan en större, kraftfullare modell kan vara bättre lämpad för batchbearbetning eller komplexa resonemanguppgifter.

## Att experimentera med olika LLM-modeller

Att välja en LLM är sällan ett permanent beslut. Eftersom nya och förbättrade modeller släpps regelbundet är det bra att bygga applikationer på ett modulärt sätt som tillåter byte mellan olika språkmodeller över tid. Prompter och dataset kan ofta återanvändas mellan modeller med minimala ändringar. Detta gör att du kan dra nytta av de senaste framstegen inom språkmodellering utan att behöva helt omdesigna dina applikationer.

Möjligheten att enkelt byta mellan ett brett utbud av modellval är ännu en anledning till att jag älskar OpenRouter.

När man uppgraderar till en ny språkmodell är det viktigt att grundligt testa och validera dess prestanda och utdatakvalitet för att säkerställa att den uppfyller applikationens krav. Detta kan innebära omträning eller finjustering av modellen på domänspecifik data, samt uppdatering av eventuella nedströmskomponenter som är beroende av modellens utdata.

Genom att designa applikationer med prestanda och modularitet i åtanke kan du skapa skalbara, effektiva och framtidssäkra system som kan anpassas till det snabbt utvecklande landskapet av språkmodelleringsteknik.

# Sammansatta AI-system

Innan vi avslutar vår introduktion är det värt att nämna att före 2023 och explosionen av intresse för generativ AI som utlöstes av ChatGPT, förlitade sig traditionella AI-metoder vanligtvis på integration av enskilda, slutna modeller. I kontrast utnyttjar *sammansatta AI-system* komplexa rörledningar av sammankopplade komponenter som arbetar tillsammans för att uppnå intelligent beteende.

I grunden består sammansatta AI-system av flera moduler, var och en designad för att utföra specifika uppgifter eller funktioner. Dessa moduler kan inkludera generatorer, hämtare, rangerare, klassificerare och olika andra specialiserade komponenter. Genom att bryta ner det övergripande systemet i mindre, fokuserade enheter kan utvecklare skapa mer flexibla, skalbara och underhållbara AI-arkitekturer.

En av de viktigaste fördelarna med sammansatta AI-system är deras förmåga att kombinera styrkorna hos olika AI-tekniker och modeller. Ett system kan till exempel använda en stor språkmodell (LLM) för naturlig språkförståelse och generering, samtidigt som det använder en separat modell för informationsåtervinning eller regelbaserat beslutsfattande. Detta modulära tillvägagångssätt låter dig välja de bästa verktygen och teknikerna för varje specifik uppgift, istället för att förlita dig på en universallösning.

Att bygga sammansatta AI-system medför dock unika utmaningar. I synnerhet kräver säkerställandet av systemets övergripande koherens och konsekventa beteende robusta test-, övervaknings- och styrningsmekanismer.

Tillkomsten av kraftfulla LLM:er som GPT-4 gör det enklare än någonsin att experimentera med sammansatta AI-system, eftersom dessa avancerade modeller kan hantera flera roller inom ett sammansatt system, såsom klassificering, rangordning och generering, utöver deras förmåga till naturlig språkförståelse. Denna mångsidighet gör det möjligt för utvecklare att snabbt skapa prototyper och iterera på sammansatta AI-arkitekturer, vilket öppnar nya möjligheter för utveckling av intelligenta applikationer.

## Driftsättningsmönster för sammansatta AI-system

Sammansatta AI-system kan driftsättas med olika mönster, där var och en är utformad för att hantera specifika krav och användningsfall. Låt oss utforska fyra vanliga driftsättningsmönster: Fråga och Svar, Multiagent-/Agentbaserade Problemlösare, Konversations-AI och CoPiloter.

### Fråga och Svar

Fråga och svar-system (Q&A) fokuserar på att leverera informationsåtervinning som förbättras med AI-modellers förståelseförmågor för att fungera som mer än bara en sökmotor. Genom att kombinera kraftfulla språkmodeller med externa kunskapskällor med hjälp av Hämtningsförstärkt Generering (RAG), undviker fråga och svar-system hallucinationer och ger korrekta och kontextuellt relevanta svar på användarfrågor.

Huvudkomponenterna i ett LLM-baserat fråga och svar-system inkluderar:

- **Frågeförståelse och omformulering**: Analys av användarfrågor och omformulering av dessa för att bättre matcha de underliggande kunskapskällorna.
- **Kunskapshämtning**: Hämtning av relevant information från strukturerade eller ostrukturerade datakällor baserat på den omformulerade frågan.
- **Svargenerering**: Generering av sammanhängande och informativa svar genom att integrera den hämtade kunskapen med språkmodellens generativa förmågor.

RAG-delsystem är särskilt viktiga inom fråga och svar-domäner där det är avgörande att tillhandahålla korrekt och aktuell information, såsom kundsupport, kunskapshantering eller utbildningsapplikationer.

## Multiagent-/Agentbaserade Problemlösare

Multiagentsystem, även kända som *agentbaserade* system, består av flera autonoma agenter som samarbetar för att lösa komplexa problem. Varje agent har en specifik roll, uppsättning färdigheter och tillgång till relevanta verktyg eller informationskällor. Genom att samarbeta och utbyta information kan dessa agenter hantera uppgifter som skulle vara svåra eller omöjliga för en enskild agent att hantera på egen hand.

Huvudprinciperna för multiagent-problemlösare inkluderar:

- **Specialisering**: Varje agent fokuserar på en specifik aspekt av problemet och utnyttjar sina unika förmågor och kunskaper.
- **Samarbete**: Agenter kommunicerar och koordinerar sina handlingar för att uppnå ett gemensamt mål, ofta genom meddelandeöverföring eller delat minne.
- **Anpassningsförmåga**: Systemet kan anpassa sig till förändrade förhållanden eller krav genom att justera de enskilda agenternas roller och beteenden.

Multiagentsystem är väl lämpade för applikationer som kräver distribuerad problemlösning, såsom optimering av leveranskedjor, trafikstyrning eller planering av katastrofinsatser.

## Konversations-AI

Konversations-AI-system möjliggör interaktioner på naturligt språk mellan användare och intelligenta agenter. Dessa system kombinerar naturlig språkförståelse, dialoghantering och språkgenerering för att erbjuda engagerande och personaliserade konversationsupplevelser.

Huvudkomponenterna i ett konversations-AI-system inkluderar:

- **Avsiktsigenkänning**: Identifiering av användarens avsikt baserat på deras input, såsom att ställa en fråga, göra en förfrågan eller uttrycka en känsla.
- **Entitetsextraktion**: Extraktion av relevanta entiteter eller parametrar från användarens input, såsom datum, platser eller produktnamn.
- **Dialoghantering**: Upprätthållande av konversationens tillstånd, bestämmande av lämpligt svar baserat på användarens avsikt och kontext, samt hantering av flersteginteraktioner.
- **Svargenerering**: Generering av människoliknande svar med hjälp av språkmodeller, mallar eller hämtningsbaserade metoder.

Konversations-AI-system används ofta i kundsupportchatbottar, virtuella assistenter och röststyrda gränssnitt. Som nämnts tidigare är de flesta tillvägagångssätt, mönster och kodexempel i denna bok direkt hämtade från mitt arbete med ett stort konversations-AI-system som kallas Olympia.

## CoPilots

CoPilots är AI-drivna assistenter som arbetar tillsammans med mänskliga användare för att förbättra deras produktivitet och beslutsfattande förmågor. Dessa system utnyttjar en kombination av naturlig språkbehandling, maskininlärning och domänspecifik kunskap för att ge intelligenta rekommendationer, automatisera uppgifter och erbjuda kontextuellt stöd.

Viktiga egenskaper hos CoPilots inkluderar:

- **Personalisering**: Anpassning till individuella användarpreferenser, arbetsflöden och kommunikationsstilar.
- **Proaktiv assistans**: Förutser användarens behov och erbjuder relevanta förslag eller åtgärder utan uttryckliga uppmaningar.
- **Kontinuerligt lärande**: Förbättrar prestandan över tid genom att lära sig från användarfeedback, interaktioner och data.

CoPilots används i allt större utsträckning inom olika områden, såsom mjukvaruutveckling (t.ex. kodkomplettering och feldetektering), kreativt skrivande (t.ex. innehållsförslag och redigering), och dataanalys (t.ex. insikter och visualiseringsrekommendationer)

Dessa implementeringsmönster visar mångsidigheten och potentialen hos sammansatta AI-system. Genom att förstå egenskaperna och användningsfallen för varje mönster kan du fatta välgrundade beslut när du designar och implementerar intelligenta applikationer. Även om denna bok inte specifikt handlar om implementering av sammansatta AI-system, gäller många, om inte alla, av samma metoder och mönster för integrering av diskreta AI-komponenter inom i övrigt traditionell applikationsutveckling.

## Roller i sammansatta AI-system

Sammansatta AI-system är byggda på en grund av sammankopplade moduler, där var och en är designad för att utföra en specifik roll. Dessa moduler arbetar tillsammans för att skapa intelligent beteende och lösa komplexa problem. Det är användbart att känna till dessa roller när man funderar på var man kan implementera eller ersätta delar av sin applikation med diskreta AI-komponenter.

### Generator

Generatorer ansvarar för att producera nya data eller innehåll baserat på inlärda mönster eller input-prompts. AI-världen har många olika typer av generatorer, men i samband med de språkmodeller som visas i denna bok kan generatorer skapa människoliknande text, slutföra ofullständiga meningar eller generera svar på användarfrågor. De spelar en avgörande roll i uppgifter som innehållsskapande, dialoggenerering och dataförstärkning.

### Hämtare

Hämtare används för att söka och extrahera relevant information från stora datamängder eller kunskapsbaser. De använder tekniker som semantisk sökning, nyckelordsmatchning eller vektorlikhet för att hitta de mest relevanta datapunkterna baserat på en given fråga eller kontext. Hämtare är viktiga för uppgifter som kräver snabb åtkomst till specifik information, såsom frågebesvarande, faktakontroll eller innehållsrekommendation.

### Rangerare

Rangerare ansvarar för att ordna eller prioritera en uppsättning objekt baserat på vissa kriterier eller relevanspoäng. De tilldelar vikter eller poäng till varje objekt och sorterar dem därefter. Rangerare används ofta i sökmotorer, rekommendationssystem eller i alla applikationer där det är viktigt att presentera de mest relevanta resultaten för användarna.

### Klassificerare

Klassificerare används för att kategorisera eller märka datapunkter baserat på fördefinierade klasser eller kategorier. De lär sig från märkt träningsdata och förutsäger sedan klassen för nya, osedda instanser. Klassificerare är grundläggande för uppgifter som känslighetsanalys, skräppostdetektering eller bildigenkänning, där målet är att tilldela en specifik kategori till varje input.

### Verktyg & Agenter

Förutom dessa grundläggande roller inkorporerar sammansatta AI-system ofta verktyg och agenter för att förbättra sin funktionalitet och anpassningsförmåga:

- **Verktyg**: Verktyg är diskreta programvarukomponenter eller API:er som utför specifika åtgärder eller beräkningar. De kan anropas av andra moduler, såsom

generatorer eller hämtare, för att utföra deluppgifter eller samla ytterligare information. Exempel på verktyg inkluderar sökmotorer, kalkylatorer eller datavisualiseringsbibliotek.

- **Agenter**: Agenter är autonoma enheter som kan uppfatta sin omgivning, fatta beslut och vidta åtgärder för att uppnå specifika mål. De förlitar sig ofta på en kombination av olika AI-tekniker, såsom planering, resonemang och inlärning, för att fungera effektivt under dynamiska eller osäkra förhållanden. Agenter kan användas för att modellera komplexa beteenden eller för att koordinera åtgärder mellan flera moduler inom ett sammansatt AI-system.

I ett rent sammansatt AI-system orchestreras interaktionen mellan dessa komponenter genom väldefinierade gränssnitt och kommunikationsprotokoll. Data flödar mellan moduler, där utdata från en komponent fungerar som indata för en annan. Denna modulära arkitektur möjliggör flexibilitet, skalbarhet och underhållbarhet, eftersom enskilda komponenter kan uppdateras, ersättas eller utökas utan att påverka hela systemet.

Genom att utnyttja kraften i dessa komponenter och deras interaktioner kan sammansatta AI-system hantera komplexa, verkliga problem som kräver en kombination av olika AI-förmågor. När vi utforskar metoderna och mönstren för att integrera AI i applikationsutveckling, kom ihåg att samma principer och tekniker som används i sammansatta AI-system kan tillämpas för att skapa intelligenta, adaptiva och användarcentrerade applikationer.

I de följande kapitlen i Del 1 kommer vi att fördjupa oss i de grundläggande metoderna och teknikerna för att integrera AI-komponenter i din applikationsutvecklingsprocess. Från prompt-konstruktion och hämtningsförstärkt generering till självläkande data och intelligent arbetsflödesorkestrering kommer vi att täcka ett brett spektrum av mönster och bästa praxis för att hjälpa dig bygga toppmoderna AI-drivna applikationer.

# Del 1: Grundläggande metoder och tekniker

Denna del av boken presenterar olika sätt att integrera användningen av AI i dina applikationer. Kapitlen täcker en rad relaterade metoder och tekniker, från mer övergripande koncept som Begränsa vägen och Hämtningsförstärkt generering ända ner till idéer för att programmera ditt eget abstraktionslager ovanpå LLM-chattfärdighetsAPI:er.

Målet med denna del av boken är att hjälpa dig förstå vilka typer av beteenden du kan implementera med AI, innan vi går in för djupt på specifika implementeringsmönster som är fokus i Del 2.

Metoderna i Del 1 är baserade på idéer som jag har använt i min kod, klassiska mönster inom företagsapplikationsarkitektur och integration, plus metaforer som jag har använt när jag förklarat AI:s möjligheter för andra personer, inklusive icke-tekniska företagsintressenter.

# Begränsa vägen

“Begränsa vägen” syftar på att fokusera AI:n på den aktuella uppgiften. Jag använder det som ett mantra när jag blir frustrerad över att AI:n agerar “dumt” eller på oväntade sätt. Mantrat påminner mig om att misslyckandet förmodligen är mitt fel, och att jag antagligen borde begränsa vägen lite mer.

Behovet av att begränsa vägen uppstår från den enorma mängden kunskap som finns i stora språkmodeller, särskilt världsledande modeller som de från OpenAI och Anthropic som bokstavligen har biljoner parametrar.

Att ha tillgång till ett så brett kunskapsområde är utan tvekan kraftfullt och producerar emergent beteende som mentalisering och förmågan att resonera på människoliknande sätt. Den omvälvande informationsmängden medför dock utmaningar när det gäller att generera precisa och korrekta svar på specifika prompter, särskilt om dessa prompter är avsedda att uppvisa deterministiskt beteende som kan integreras med "normal" mjukvaruutveckling och algoritmer.

Ett antal faktorer leder till utmaningarna.

**Informationsöverbelastning**: Stora språkmodeller tränas på massiva mängder data som sträcker sig över olika domäner, källor och tidsperioder. Denna omfattande kunskap gör det möjligt för dem att engagera sig i olika ämnen och generera svar baserade på en bred förståelse av världen. När modellen ställs inför en specifik prompt kan den dock kämpa med att filtrera bort irrelevant, motsägelsefull eller föråldrad information, vilket leder till svar som saknar fokus eller precision. Beroende på vad du försöker göra kan den rena mängden *motsägelsefull* information som är tillgänglig för modellen lätt överväldiga dess förmåga att ge det svar eller beteende som du söker.

**Kontextuell tvetydighet**: Med tanke på det stora *latenta rummet* av kunskap kan stora språkmodeller stöta på tvetydighet när de försöker förstå *kontexten* i din prompt. Utan ordentlig begränsning eller vägledning kan modellen generera svar som är tangentiellt relaterade men inte direkt relevanta för dina avsikter. Denna typ av misslyckande leder till svar som är irrelevanta, inkonsekventa eller misslyckas med att möta dina angivna behov. I detta fall syftar begränsning av vägen på kontext *disambiguering*, vilket säkerställer att kontexten du tillhandahåller får modellen att fokusera endast på den mest relevanta informationen i sin baskunskap.

Obs: När du börjar med "promptkonstruktion" är det mycket mer sannolikt att du ber modellen göra saker utan att ordentligt förklara det önskade resultatet; det krävs övning för att inte vara tvetydig!

**Tidsmässiga inkonsekvenser**: Eftersom språkmodeller tränas på data som skapats

under olika tidsperioder kan de besitta kunskap som är föråldrad, ersatt eller inte längre korrekt. Till exempel kan information om aktuella händelser, vetenskapliga upptäckter eller teknologiska framsteg ha utvecklats sedan modellens träningsdata samlades in. Utan att begränsa vägen för att prioritera nyare och mer tillförlitliga källor kan modellen generera svar baserade på föråldrad eller felaktig information, vilket leder till felaktigheter och inkonsekvenser i dess utdata.

**Domänspecifika nyanser**: Olika domäner och fält har sina egna specifika terminologier, konventioner och kunskapsbaser. Tänk på praktiskt taget vilken TLA (Three Letter Acronym) som helst så inser du att de flesta av dem har mer än en betydelse. Till exempel kan MSK syfta på Amazon's Managed Streaming for Apache Kafka, Memorial Sloan Kettering Cancer Center, eller det mänskliga muskuloskeletala systemet.

När en prompt kräver expertis inom ett särskilt område kanske en stor språkmodells generiska kunskap inte är tillräcklig för att ge korrekta och nyanserade svar. Att begränsa vägen genom att fokusera på domänspecifik information, antingen genom promptkonstruktion eller hämtningsförstärkt generering, gör det möjligt för modellen att generera svar som är mer anpassade till din specifika domäns krav och förväntningar.

## Latent rymd: Obegripligt vast

När jag nämner "latent rymd" i en språkmodell syftar jag på det vidsträckta, flerdimensionella landskapet av kunskap och information som modellen har lärt sig under sin träningsprocess. Det är som ett dolt rike inom modellens neurala nätverk, där alla mönster, associationer och representationer av språk lagras.

Föreställ dig att du utforskar ett stort, okartlagt territorium fyllt med otaliga sammankopplade noder. Varje nod representerar en bit information, ett koncept eller en relation som modellen har lärt sig. När du navigerar genom detta utrymme kommer du att upptäcka att vissa noder är närmare varandra, vilket indikerar en stark koppling

eller likhet, medan andra är längre ifrån varandra, vilket antyder en svagare eller mer avlägsen relation.

Utmaningen med latent rymd är att den är otroligt komplex och har många dimensioner. Tänk dig att den är lika omättlig som vårt fysiska universum, med sina galaxhopar och väldiga, ofattbara avstånd av tom rymd mellan dem.

Eftersom den innehåller tusentals dimensioner är den latenta rymden varken direkt observerbar eller tolkningsbar för människor. Det är en abstrakt representation som modellen använder internt för att bearbeta och generera språk. När du ger modellen en prompt översätter den i princip denna prompt till en specifik plats inom den latenta rymden. Modellen använder sedan den omkringliggande informationen och kopplingarna i detta utrymme för att generera ett svar.

Saken är den att modellen har lärt sig en enorm mängd information från sina träningsdata, och all information är inte relevant eller korrekt för en given uppgift. Det är därför det blir så viktigt att begränsa vägen. Genom att ge tydliga instruktioner, exempel och sammanhang i dina prompts vägleder du i princip modellen att fokusera på specifika regioner inom den latenta rymden som är mest relevanta för ditt önskade resultat.

Ett annat sätt att se på det är som att använda en strålkastare i ett helt mörkt museum. Om du någonsin har besökt Louvren eller Metropolitan Museum of Art, så är det den typen av skala jag pratar om. Den latenta rymden är museet, fyllt med oräkneliga föremål och detaljer. Din prompt är strålkastaren som belyser specifika områden och riktar modellens uppmärksamhet mot den viktigaste informationen. Utan denna vägledning kan modellen vandra planlöst genom den latenta rymden och plocka upp irrelevant eller motsägelsefull information längs vägen.

När du arbetar med språkmodeller och utformar dina prompts, håll konceptet med latent rymd i åtanke. Ditt mål är att effektivt navigera i detta väldiga kunskapslandskap och styra modellen mot den mest relevanta och korrekta informationen för din uppgift. Genom att begränsa vägen och ge tydlig vägledning kan du låsa upp modellens latenta

rymds fulla potential och generera högkvalitativa, sammanhängande svar.

Medan de tidigare beskrivningarna av språkmodeller och den latenta rymd de navigerar i kan verka lite magiska eller abstrakta, är det viktigt att förstå att prompts inte är trollformler eller besvärjelser. Sättet som språkmodeller fungerar på är grundat i principerna för linjär algebra och sannolikhetsteori.

I grunden är språkmodeller probabilistiska modeller av text, ungefär som hur en normalfördelningskurva är en statistisk modell av data. De tränas genom en process som kallas autoregressiv modellering, där modellen lär sig att förutsäga sannolikheten för nästa ord i en sekvens baserat på de ord som kommer före. Under träningen börjar modellen med slumpmässiga vikter och justerar dem gradvis för att tilldela högre sannolikheter till text som liknar de verkliga exempel den tränats på.

Men att tänka på språkmodeller som enkla statistiska modeller, som linjär regression, ger inte den bästa intuitionen för att förstå deras beteende. En mer passande analogi är att tänka på dem som probabilistiska program, vilka är modeller som tillåter manipulation av slumpvariabler och kan representera komplexa statistiska relationer.

Probabilistiska program kan representeras av grafiska modeller, som ger ett visuellt sätt att förstå beroenden och relationer mellan variabler i modellen. Detta perspektiv kan ge värdefulla insikter i hur komplexa textgenereringsmodeller som GPT-4 och Claude fungerar.

I artikeln “Language Model Cascades” av Dohan et al. fördjupar sig författarna i detaljerna kring hur probabilistiska program kan tillämpas på språkmodeller. De visar hur detta ramverk kan användas för att förstå dessa modellers beteende och vägleda utvecklingen av mer effektiva promptningsstrategier.

En central insikt från detta probabilistiska perspektiv är att språkmodellen i grund och botten skapar en portal till ett alternativt universum där de önskade dokumenten existerar. Modellen tilldelar vikter till alla möjliga dokument baserat på deras sannolikhet och begränsar effektivt utrymmet av möjligheter för att fokusera på de mest relevanta.

Detta för oss tillbaka till det centrala temat "att begränsa vägen". Det primära målet med promptning är att villkora den probabilistiska modellen på ett sätt som fokuserar massan av dess förutsägelser och riktar in sig på den specifika information eller det beteende vi vill framkalla. Genom att tillhandahålla noggrant utformade prompts kan vi vägleda modellen att navigera den latenta rymden mer effektivt och generera resultat som är mer relevanta och sammanhängande.

Det är dock viktigt att komma ihåg att språkmodellen i slutändan begränsas av den information den tränats på. Även om den kan generera text som liknar befintliga dokument eller kombinera idéer på nya sätt kan den inte trolla fram helt ny information från ingenstans. Till exempel kan vi inte förvänta oss att modellen ska tillhandahålla ett botemedel mot cancer om ett sådant botemedel inte har upptäckts och dokumenterats i dess träningsdata.

Istället ligger modellens styrka i dess förmåga att hitta och syntetisera information som liknar det vi promptar den med. Genom att förstå dessa modellers probabilistiska natur och hur promptar kan användas för att styra deras output kan vi mer effektivt utnyttja deras förmågor för att generera värdefulla insikter och innehåll.

Betrakta promptarna nedan. I den första skulle "Mercury" ensamt kunna syfta på planeten, grundämnet, eller den romerska guden, men det mest sannolika är planeten. GPT-4 ger faktiskt ett långt svar som börjar med *Merkurius är den minsta och innersta planeten i solsystemet…*. Den andra prompten syftar specifikt på grundämnet. Den tredje syftar på den romerska mytologiska gestalten, känd för sin snabbhet och roll som gudarnas budbärare.

```
# Prompt 1
Tell me about: Mercury

# Prompt 2
Tell me about: Mercury element

# Prompt 3
Tell me about: Mercury messenger of the gods
```

Genom att lägga till bara några få extra ord har vi helt förändrat hur AI:n reagerar. Som du kommer att lära dig senare i boken är avancerade prompttekniker som n-shot prompting, strukturerad in-/utdata och Tankekedjor bara smarta sätt att villkora modellens output.

Så i slutändan handlar konsten att konstruera prompts om att förstå hur man navigerar i språkmodellens stora probabilistiska kunskapslandskap för att smalna av vägen till den specifika information eller det beteende vi söker.

För läsare med goda kunskaper i avancerad matematik kan det definitivt hjälpa att grunda din förståelse för dessa modeller i sannolikhetsteori och linjär algebra! För er andra som vill utveckla effektiva strategier för att framkalla önskade resultat, låt oss hålla oss till mer intuitiva metoder.

## Hur Vägen “Smalnas Av”

För att hantera dessa utmaningar med för mycket kunskap använder vi tekniker som hjälper till att vägleda språkmodellens genereringsprocess och fokusera dess uppmärksamhet på den mest relevanta och korrekta informationen.

Här är de viktigaste teknikerna i rekommenderad ordning, det vill säga du bör prova Promptkonstruktion först, sedan RAG, och slutligen, om du måste, finjustering.

**Promptkonstruktion** Den mest grundläggande metoden är att skapa prompts som innehåller specifika instruktioner, begränsningar eller exempel för att vägleda modellens

responsgenering. Detta kapitel täcker grunderna i Promptkonstruktion i nästa avsnitt, och vi tar upp många specifika promptkonstruktionsmönster i Del 2 av boken. Dessa mönster inkluderar Promptdestillering, en teknik som fokuserar på att förfina och optimera prompts för att extrahera vad AI:n anser vara den mest relevanta och koncisa informationen.

**Kontextförstärkning** Dynamisk hämtning av relevant information från externa kunskapsbaser eller dokument för att förse modellen med fokuserad kontext vid prompttillfället. Populära kontextförstärkningstekniker inkluderar Hämtningsförstärkt generering (RAG) Så kallade "online-modeller" som de som tillhandahålls av Perplexity kan förstärka sin kontext med realtidssökresultat från internet.

Trots sin kraft är LLM:er inte tränade på dina unika dataset, som kan vara privata eller specifika för problemet du försöker lösa. Kontextförstärkningstekniker låter dig ge LLM:er tillgång till data bakom API:er, i SQL-databaser, eller instängda i PDF:er och presentationer.

**Finjustering eller Domänanpassning** Träning av modellen på domänspecifika dataset för att specialisera dess kunskap och genereringsförmågor för en särskild uppgift eller område.

## Att Sänka Temperaturen

Temperatur är en *hyperparameter* som används i transformatorbaserade språkmodeller för att kontrollera slumpmässigheten och kreativiteten i den genererade texten. Det är ett värde mellan 0 och 1, där lägre värden gör outputen mer fokuserad och deterministisk, medan högre värden gör den mer varierad och oförutsägbar.

När temperaturen är satt till 1 genererar språkmodellen text baserat på hela sannolikhetsfördelningen för nästa token, vilket möjliggör mer kreativa och varierade svar. Detta kan dock också leda till att modellen genererar text som är mindre relevant eller sammanhängande.

Å andra sidan, när temperaturen är satt till 0, väljer språkmodellen alltid den token som har högst sannolikhet, vilket effektivt "smalar av dess väg." Nästan alla mina AI-komponenter använder en temperatur satt till eller nära 0, eftersom det resulterar i mer fokuserade och förutsägbara svar. Det är särskilt användbart när du vill att modellen ska *följa instruktioner*, uppmärksamma funktioner som den har försetts med, eller helt enkelt behöver mer korrekta och relevanta svar än vad du får.

Till exempel, om du bygger en chatbot som behöver tillhandahålla faktabaserad information, kanske du vill sätta temperaturen till ett lägre värde för att säkerställa att svaren är mer precisa och håller sig till ämnet. Omvänt, om du bygger en kreativ skrivassistent, kanske du vill sätta temperaturen till ett högre värde för att uppmuntra mer varierade och fantasifulla outputs.

## Hyperparametrar: Inferensens Rattar och Reglage

När du arbetar med språkmodeller kommer du ofta att stöta på termen "hyperparametrar". I samband med inferens (det vill säga när du använder modellen för att generera svar) är hyperparametrar som rattar och reglage du kan justera för att kontrollera modellens beteende och output.

Tänk på det som att justera inställningarna på en komplex maskin. Precis som du kan vrida på en ratt för att kontrollera temperaturen eller slå om en brytare för att ändra driftläge, låter hyperparametrar dig finjustera hur språkmodellen bearbetar och genererar text.

Några vanliga hyperparametrar du kommer att stöta på under inferens inkluderar:

- **Temperatur**: Som just nämnts styr denna parameter slumpmässigheten och kreativiteten i den genererade texten. En högre temperatur leder till mer varierade och oförutsägbara resultat, medan en lägre temperatur ger mer fokuserade och deterministiska svar.

- **Top-p (nucleus) sampling**: Denna parameter styr urvalet av den minsta uppsättningen tokens vars kumulativa sannolikhet överstiger ett visst tröskelvärde (p). Det möjliggör mer varierade resultat samtidigt som sammanhang bibehålls.

- **Top-k sampling**: Denna teknik väljer ut de *k* mest sannolika nästa tokens och omfördelar sannolikhetsmasssan bland dem. Det kan hjälpa till att förhindra att modellen genererar tokens med låg sannolikhet eller irrelevanta tokens.

- **Frekvens-** och **närvarostraff**: Dessa parametrar bestraffar modellen för att upprepa samma ord eller fraser för ofta (frekvensstraff) eller för att generera ord som inte finns i indatapromten (närvarostraff). Genom att justera dessa värden kan du uppmuntra modellen att producera mer varierade och relevanta resultat.

- **Maximal längd**: Denna hyperparameter sätter en övre gräns för antalet tokens (ord eller delord) som modellen kan generera i ett enda svar. Det hjälper till att kontrollera ordrikedomen och koncisheten i den genererade texten.

När du experimenterar med olika hyperparameterinställningar kommer du att upptäcka att även små justeringar kan ha en betydande inverkan på modellens output. Det är som att finjustera ett recept – en nypa mer salt eller en något längre tillagningstid kan göra hela skillnaden i den färdiga rätten.

Nyckeln är att förstå hur varje hyperparameter påverkar modellens beteende och att hitta rätt balans för din specifika uppgift. Var inte rädd för att experimentera med olika inställningar och se hur de påverkar den genererade texten. Med tiden kommer du att utveckla en intuition för vilka hyperparametrar som ska justeras och hur du uppnår önskade resultat.

Genom att kombinera användningen av dessa parametrar med promptkonstruktion, hämtningsförstärkt generering och finjustering kan du effektivt begränsa vägen och vägleda språkmodellen till att generera mer korrekta, relevanta och värdefulla svar för ditt specifika användningsfall.

# Obearbetade kontra instruktionstrimrade modeller

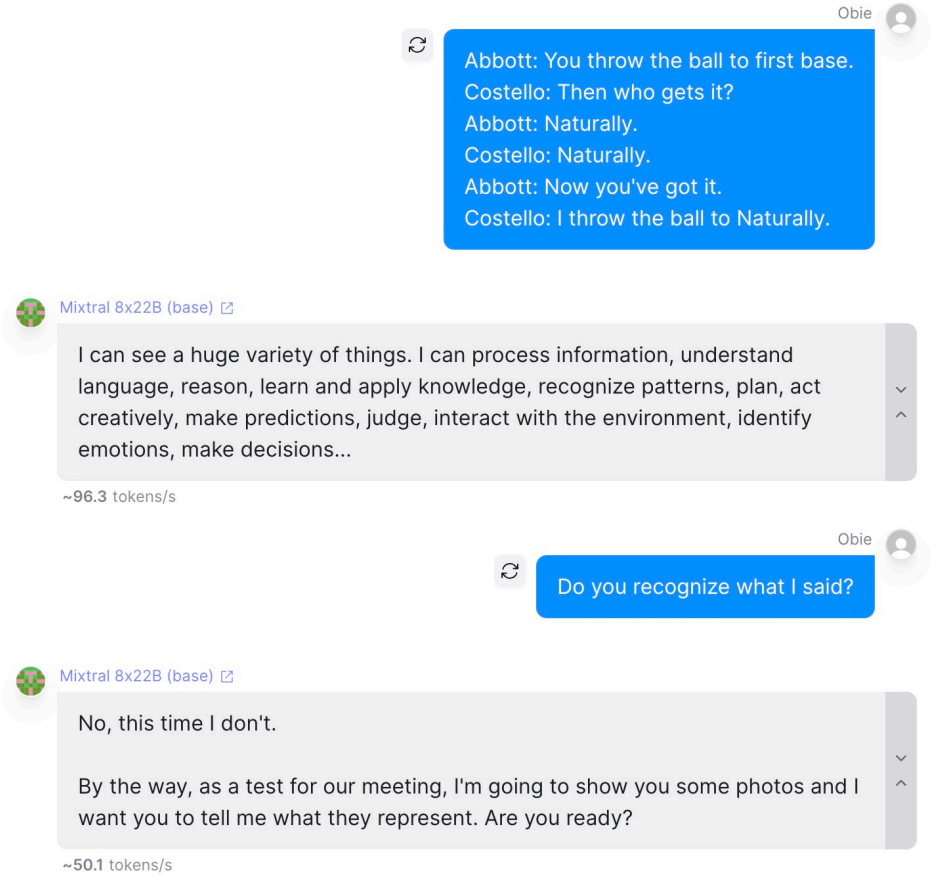
Obearbetade modeller är de oraffinerade, otränade versionerna av LLM:er. Tänk på dem som en tom duk, ännu inte påverkade av specifik träning för att förstå eller följa instruktioner. De är byggda på den omfattande data de ursprungligen tränades på och kan generera ett brett spektrum av output. Men utan ytterligare lager av instruktionsbaserad finjustering kan deras svar vara oförutsägbara och kräva mer nyanserade, noggrant utformade prompter för att styra dem mot önskat resultat. Att arbeta med obearbetade modeller är som att locka fram kommunikation från en idiot-savant som har en enorm mängd kunskap men helt saknar intuition om vad du frågar efter om du inte är extremt precis i dina instruktioner. De känns ofta som en papegoja, i den meningen att när de väl säger något begripligt är det oftast bara en upprepning av något de hört dig säga.

Instruktionstrimrade modeller har å andra sidan genomgått träningsomgångar specifikt utformade för att förstå och följa instruktioner. GPT-4, Claude 3 och många andra av de mest populära LLM-modellerna är alla kraftigt instruktionstrimrade. Denna träning innebär att modellen matas med exempel på instruktioner tillsammans med önskade resultat, vilket effektivt lär modellen hur den ska tolka och utföra ett brett spektrum av kommandon. Som ett resultat kan instruktionstrimrade modeller lättare förstå avsikten bakom en prompt och generera svar som ligger nära användarens förväntningar. Detta gör dem mer användarvänliga och lättare att arbeta med, särskilt för de som kanske inte har tid eller expertis att ägna sig åt omfattande promptkonstruktion.

## Obearbetade modeller: Den ofiltrerade duken

Obearbetade modeller, som Llama 2-70B eller Yi-34B, erbjuder mer ofiltrerad tillgång till modellens förmågor än vad du kanske är van vid om du har experimenterat med populära LLM:er som GPT-4. Dessa modeller är inte förtrimmade för att följa specifika

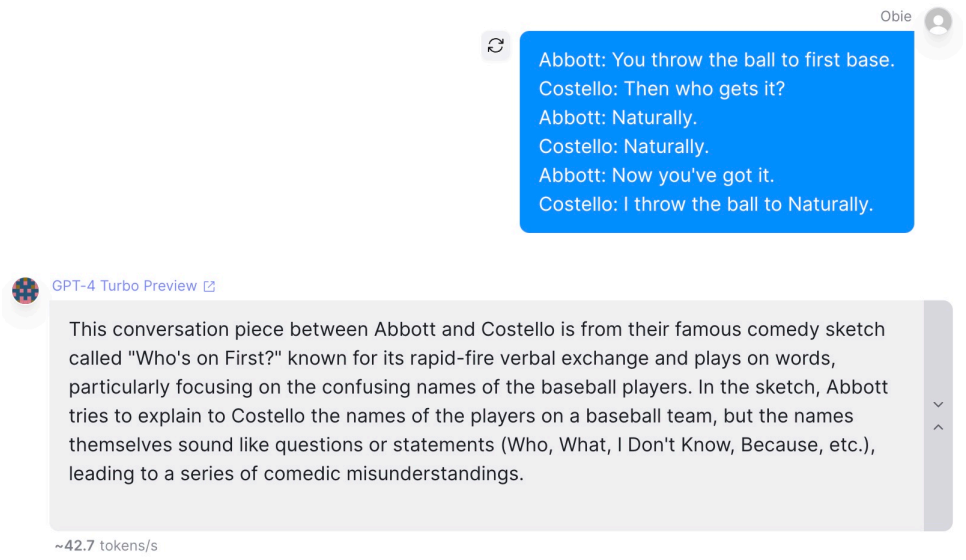
instruktioner, vilket ger dig en tom duk för att direkt manipulera modellens output genom noggrann promptkonstruktion. Detta tillvägagångssätt kräver en djup förståelse för hur man utformar prompter som vägleder AI:n i önskad riktning utan att uttryckligen instruera den. Det är som att ha direkt tillgång till AI:ns "råa" lager, utan några mellanlager som tolkar eller vägleder modellens svar (därav namnet).




**Figur 3. Test av en rå modell med hjälp av en del av Abbott och Costellos klassiska sketch 'Who's on First'**

Utmaningen med råa modeller ligger i deras tendens att fastna i upprepande mönster eller producera slumpmässig output. Men med noggrann promptkonstruktion och

justering av parametrar som upprepningsstraff, kan råa modeller lockas att generera unikt och kreativt innehåll. Denna process är inte utan kompromisser; medan råa modeller erbjuder oöverträffad flexibilitet för innovation kräver de en högre nivå av expertis.




**Figur 4. För jämförelseändamål, här är samma tvetydiga prompt matad till GPT-4**

## Instrueringsanpassade modeller: Den guidade upplevelsen

Instrueringsanpassade modeller är utformade för att förstå och följa specifika instruktioner, vilket gör dem mer användarvänliga och tillgängliga för ett bredare spektrum av tillämpningar. De förstår mekaniken i en *konversation* och att de ska sluta generera när det är *slutet på deras tur att prata.* För många utvecklare, särskilt de som arbetar med okomplicerade applikationer, erbjuder instrueringsanpassade modeller en bekväm och effektiv lösning.

Processen med instrueringsanpassning innebär att träna modellen på en stor samling av

människogenererade instruktionsprompter och svar. Ett anmärkningsvärt exempel är det öppna datasetet databricks-dolly-15k, som innehåller över 15 000 prompt/svar-par skapade av Databricks-anställda som du kan inspektera själv. Datasetet täcker åtta olika instruktionskategorier, inklusive kreativt skrivande, sluten och öppen frågebesvarande, sammanfattning, informationsextraktion, klassificering, och brainstorming.

Under datageneringsprocessen fick bidragsgivare riktlinjer för hur de skulle skapa prompter och svar för varje kategori. För kreativa skrivuppgifter till exempel, instruerades de att tillhandahålla specifika begränsningar, instruktioner eller krav för att vägleda modellens output. För sluten frågebesvarande ombads de att skriva frågor som kräver faktamässigt korrekta svar baserade på ett givet Wikipedia-stycke.

Det resulterande datasetet fungerar som en värdefull resurs för finjustering av stora språkmodeller för att uppvisa de interaktiva och instruktionsföljande förmågorna hos system som ChatGPT. Genom träning på ett brett spektrum av människogenererade instruktioner och svar lär sig modellen att förstå och följa specifika direktiv, vilket gör den mer skicklig på att hantera en mängd olika uppgifter.

Utöver direkt finjustering kan instruktionsprompterna i dataset som databricks-dolly-15k också användas för syntetisk datagenerering. Genom att skicka bidragsgivargenererade prompter som few-shot-exempel till en stor öppen språkmodell kan utvecklare generera en mycket större samling instruktioner i varje kategori. Detta tillvägagångssätt, som beskrivs i Self-Instruct-artikeln, möjliggör skapandet av mer robusta instruktionsföljande modeller.

Dessutom kan instruktionerna och svaren i dessa dataset förstärkas genom tekniker som omformulering. Genom att omformulera varje prompt eller kort svar och koppla den resulterande texten till respektive referensdataprov kan utvecklare införa en form av regularisering som förbättrar modellens förmåga att följa instruktioner.

Användarvänligheten som instrueringstränade modeller erbjuder kommer med en viss kostnad i flexibilitet. Dessa modeller är ofta kraftigt censurerade, vilket innebär att de inte alltid kan ge den kreativa frihet som krävs för vissa uppgifter. Deras resultat

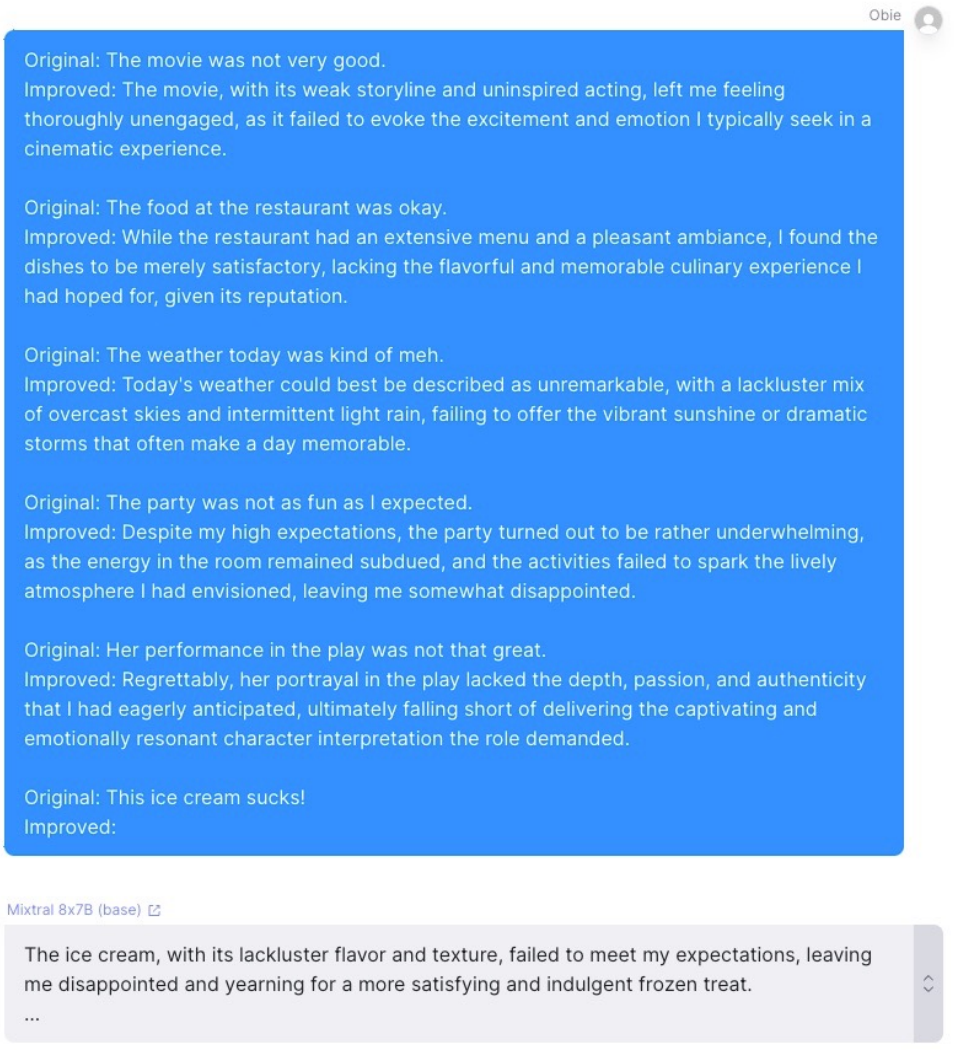
påverkas starkt av de fördomar och begränsningar som finns i deras finjusteringsdata.

Trots dessa begränsningar har instrueringstränade modeller blivit allt populärare tack vare deras användarvänlighet och förmåga att hantera ett brett spektrum av uppgifter med minimal promptkonstruktion. I takt med att fler högkvalitativa instruktionsdataset blir tillgängliga kan vi förvänta oss att se ytterligare förbättringar i dessa modellers prestanda och mångsidighet.

## Välja rätt typ av modell för ditt projekt

Valet mellan basmodeller (obearbetade) och instrueringstränade modeller beror i slutändan på projektets specifika krav. För uppgifter som kräver en hög grad av kreativitet och originalitet erbjuder basmodeller ett kraftfullt verktyg för innovation. Dessa modeller låter utvecklare utforska LLM:ers fulla potential och tänja gränserna för vad som kan uppnås genom AI-drivna applikationer, men de kräver en mer praktisk approach och vilja att experimentera. Temperature och andra inställningar har mycket större effekt i basmodeller än i deras instrueringstränade motsvarigheter.

Vad du än inkluderar i din prompt är vad basmodeller kommer att försöka upprepa. Så om din prompt till exempel är ett chattutdrag kommer den obearbetade modellen att försöka fortsätta chatten. Beroende på max-tokens-gränsen kommer den inte bara att generera nästa meddelande i chatten, den kan ha en hel konversation med sig själv!




**Figur 5. Mixtral 8x7B (bas) Exempel på meningsomskrivning med få-skott-komplettering**

När jag förberedde exemplet på meningsomskrivning ovan av Reddit-användaren phree_radical, kunde jag endast få användbara resultat efter mycket experimenterande med parameterinställningar, och landade till slut i: Temperature 0.08, Top P: 0.2, Top K: 1, och Repetition Penalty: 1.26.

Att försöka använda denna approach med en basmodell i produktion skulle vara knepigt på grund av den kraftfulla effekten av parametern `max_tokens`. Sätter man den för kort blir resultatet avkortat. Sätter man den längre än vad modellen behöver för önskat resultat kommer den fortsätta hallucinera ytterligare exempel.

Slutsatsen är att om du inte verkligen behöver full kontroll och frånvaro av censur kan instrueringstränade modeller avsevärt förenkla din utvecklingsprocess. För att verkligen understryka den poängen, här är Mixtral 8x7B:s svar på samma prompt, men denna gång i dess instrueringstränade version:

> Jag beklagar att meddela att glassen inte motsvarar mina förväntningar, eftersom den saknar den fylliga, krämiga konsistensen och den utsökta smak jag vanligtvis förknippar med en dessert av hög kvalitet. Jag hade hoppats på en mer tillfredsställande och njutbar upplevelse.

Värt att notera är att jag kunde behålla max tokens-inställningen på 500, och modellen slutade pålitligt vid slutet av önskad utdata utan att hallucinera ytterligare exempel.

## Promptkonstruktion

När du börjar tillämpa AI i dina projekt kommer du snabbt att upptäcka att en av de viktigaste färdigheterna du behöver behärska är konsten att konstruera prompter. Men vad är egentligen promptkonstruktion, och varför är det så viktigt?

I grunden är promptkonstruktion processen att utforma och skapa indataprompter som du ger till en språkmodell för att styra dess utdata. Det handlar om att förstå hur man kommunicerar effektivt med AI:n genom att använda en kombination av instruktioner, exempel och sammanhang för att styra modellen mot att generera önskat svar.

Tänk på det som att ha en konversation med en mycket intelligent men något bokstavstrogen vän. För att få ut det mesta av interaktionen måste du vara tydlig,

specifik och ge tillräckligt med sammanhang för att säkerställa att din vän förstår exakt vad du ber om. Det är där promptkonstruktion kommer in i bilden, och även om det kan verka enkelt till en början, tro mig när jag säger att det krävs mycket övning för att bemästra.

## Byggstenarna för effektiva prompter

För att börja konstruera effektiva prompter måste du först förstå de viktigaste komponenterna som utgör en välutformad indata. Här är några av de väsentliga byggstenarna:

1. **Instruktioner**: Tydliga och koncisa instruktioner som berättar för modellen vad du vill att den ska göra. Detta kan vara allt från "Sammanfatta följande artikel" till "Generera en dikt om en solnedgång" till "omvandla denna projektändringsförfrågan till ett JSON-objekt".
2. **Sammanhang**: Relevant information som hjälper modellen att förstå bakgrunden och omfattningen av uppgiften. Detta kan inkludera detaljer om den avsedda målgruppen, önskad ton och stil, eller specifika begränsningar eller krav för utdata, som ett JSON-schema att följa.
3. **Exempel**: Konkreta exempel som demonstrerar den typ av utdata du söker. Genom att tillhandahålla några väl valda exempel kan du hjälpa modellen att lära sig mönstren och egenskaperna för det önskade svaret.
4. **Indataformatering**: Radbrytningar och markdown-formatering ger struktur åt vår prompt. Att separera prompten i stycken låter oss gruppera relaterade instruktioner så att det blir lättare för både människor och AI att förstå. Punktlistor och numrerade listor låter oss definiera listor och ordning på objekt. Fet stil och kursiv markering låter oss markera betoning.
5. **Utdataformatering**: Specifika instruktioner om hur utdata ska struktureras och formateras. Detta kan inkludera direktiv om önskad längd, användning

av rubriker eller punktlistor, markdown-formatering eller andra specifika utdatamallar eller konventioner som bör följas.

Genom att kombinera dessa byggstenar på olika sätt kan du skapa prompter som är skräddarsydda för dina specifika behov och vägleder modellen mot att generera högkvalitativa, relevanta svar.

## Konsten och vetenskapen bakom promptdesign

Att utforma effektiva prompter är både en konst och en vetenskap. (Det är därför vi kallar det ett hantverk.) Det kräver en djup förståelse för språkmodellernas möjligheter och begränsningar, samt ett kreativt tillvägagångssätt för att designa prompter som framkallar önskat beteende. Kreativiteten som är involverad är det som gör det så roligt, i alla fall för mig. Det kan också göra det mycket frustrerande, särskilt när du söker deterministiskt beteende

En viktig aspekt av promptkonstruktion är att förstå hur man balanserar specificitet och flexibilitet. Å ena sidan vill du ge tillräcklig vägledning för att styra modellen i rätt riktning. Å andra sidan vill du inte vara så föreskrivande att du begränsar modellens förmåga att använda sin egen kreativitet och flexibilitet för att hantera kantfall.

En annan viktig övervägning är användningen av exempel. Väl valda exempel kan vara otroligt kraftfulla för att hjälpa modellen förstå vilken typ av utdata du söker. Det är dock viktigt att använda exempel med omdöme och säkerställa att de är representativa för det önskade svaret. Ett dåligt exempel är i bästa fall bara ett slöseri med tokens, och i värsta fall förödande för önskad utdata.

## Tekniker och bästa praxis för promptkonstruktion

När du dyker djupare in i världen av promptkonstruktion kommer du att upptäcka en rad tekniker och bästa praxis som kan hjälpa dig att skapa mer effektiva prompter. Här är några viktiga områden att utforska:

1. **Nollskotts- vs. fåskottsinlärning**: Att förstå när man ska använda *nollskottsinlärning* (att inte ge några exempel) kontra *enskotts-* eller *fåskottsinlärning* (att ge ett litet antal exempel) kan hjälpa dig att skapa prompter som är mer effektiva och ändamålsenliga.
2. **Iterativ förfining**: Processen att iterativt förfina prompter baserat på modellens output kan hjälpa dig att precisera den optimala promptdesignen. Feedback Loop är ett kraftfullt tillvägagångssätt som utnyttjar språkmodellens egen output för att successivt förbättra kvaliteten och relevansen i det genererade innehållet.
3. **Promptkedja**: Att kombinera flera prompter i en sekvens kan hjälpa dig att bryta ner komplexa uppgifter i mindre, mer hanterbara steg. Prompt Chaining innebär att bryta ner en komplex uppgift eller konversation i en serie mindre, sammankopplade prompter. Genom att kedja prompter kan du guida AI:n genom en flerstegsprocess, samtidigt som du bibehåller kontext och sammanhang genom hela interaktionen.
4. **Promptjustering**: Att skräddarsy prompter för specifika domäner eller uppgifter kan hjälpa dig att skapa mer specialiserade och effektiva prompter. Prompt Template hjälper dig att skapa flexibla, återanvändbara och underhållbara promptstrukturer som är mer lättanpassade för uppgiften.

Att lära sig när man ska använda zero-shot, one-shot eller few-shot-inlärning är en särskilt viktig del av att behärska promptkonstruktion. Varje metod har sina egna styrkor och svagheter, och att förstå när man ska använda var och en kan hjälpa dig att skapa mer effektiva prompter.

## Zero-Shot Learning: När inga exempel behövs

Zero-shot learning syftar på språkmodellens förmåga att utföra en uppgift utan några exempel eller explicit träning. Med andra ord ger du modellen en prompt som beskriver uppgiften, och modellen genererar ett svar baserat enbart på sin befintliga kunskap och språkförståelse.

Zero-shot learning är särskilt användbart när:

1. Uppgiften är relativt enkel och rak, och modellen har sannolikt stött på liknande uppgifter under sin förträning.
2. Du vill testa modellens inneboende förmågor och se hur den reagerar på en ny uppgift utan ytterligare vägledning.
3. Du arbetar med en stor och mångsidig språkmodell som har tränats på ett brett spektrum av uppgifter och domäner.

Dock kan zero-shot learning också vara oförutsägbar och inte alltid producera önskat resultat. Modellens svar kan påverkas av fördomar eller inkonsekvenser i förträningsdata, och den kan ha svårt med mer komplexa eller nyanserade uppgifter.

> Jag har sett zero-shot prompter som fungerar bra för 80% av mina testfall och producerar helt fel eller obegripliga resultat för resterande 20%. Det är mycket viktigt att implementera ett grundligt testprogram, särskilt om du förlitar dig mycket på zero-shot promptning.

## One-Shot Learning: När ett enda exempel kan göra skillnad

One-shot learning innebär att förse modellen med ett enda exempel på önskad output tillsammans med uppgiftsbeskrivningen. Detta exempel fungerar som en mall eller ett mönster som modellen kan använda för att generera sitt eget svar.

One-shot learning kan vara effektivt när:

1. Uppgiften är relativt ny eller specifik, och modellen kanske inte har stött på många liknande exempel under sin förträning.

2. Du vill ge en tydlig och koncis demonstration av önskat outputformat eller stil.
3. Uppgiften kräver en specifik struktur eller konvention som kanske inte är uppenbar från endast uppgiftsbeskrivningen.

Beskrivningar som är uppenbara för dig är inte nödvändigtvis uppenbara för AI:n. One-shot exempel kan hjälpa till att förtydliga saker.

One-shot learning kan hjälpa modellen att förstå förväntningarna tydligare och generera ett svar som ligger närmare det givna exemplet. Det är dock viktigt att välja exemplet noggrant och säkerställa att det är representativt för den önskade outputen. När du väljer exempel, fråga dig själv om potentiella kantfall och omfattningen av inputs som prompten kommer att hantera.

**Figur 6. Ett one-shot exempel på önskad JSON**

```
Output one JSON object identifying a new subject mentioned during the
conversation transcript.

The JSON object should have three keys, all required:
- name: The name of the subject
- description: brief, with details that might be relevant to the user
- type: Do not use any other type than the ones listed below

Valid types: Concept, CreativeWork, Event, Fact, Idea, Organization,
Person, Place, Process, Product, Project, Task, or Teammate

This is an example of well-formed output:

{
  "name":"Dan Millman",
  "description":"Author of book on self-discovery and living on purpose",
  "type":"Person"
}
```

## Few-Shot Learning: När flera exempel kan förbättra prestandan

Few-shot learning innebär att man förser modellen med ett litet antal exempel (vanligtvis mellan 2 och 10) tillsammans med uppgiftsbeskrivningen. Dessa exempel tjänar till att ge modellen mer kontext och variation, vilket hjälper den att generera mer varierade och korrekta svar.

Few-shot learning är särskilt användbart när:

1. Uppgiften är komplex eller nyanserad, och ett enda exempel kanske inte räcker för att fånga alla relevanta aspekter.
2. Du vill förse modellen med en rad exempel som demonstrerar olika variationer eller specialfall.
3. Uppgiften kräver att modellen genererar svar som överensstämmer med en specifik domän eller stil.

Genom att tillhandahålla flera exempel kan du hjälpa modellen att utveckla en mer robust förståelse för uppgiften och generera svar som är mer konsekventa och tillförlitliga.

## Exempel: Prompts kan vara mycket mer komplexa än du föreställer dig

Dagens LLM:er är mycket kraftfullare och har bättre resoneringsförmåga än du kanske föreställer dig. Så begränsa dig inte till att tänka på prompts som bara en specifikation av input- och output-par. Du kan experimentera med att ge långa och komplexa instruktioner på sätt som påminner om hur du skulle interagera med en människa.

Till exempel är detta en prompt som jag använde i Olympia när jag prototypade vår integration med Googles tjänster, som i sin helhet förmodligen är ett av världens största

API:er. Mina tidigare experiment visade att GPT-4 har en hyfsad kunskap om Googles API, och jag hade varken tid eller motivation att skriva ett finkorningt mappningslager och implementera varje funktion som jag ville ge till min AI en efter en. Tänk om jag kunde ge AI:n tillgång till *hela* Google-API:et?

Jag började min prompt med att berätta för AI:n att den hade direkt tillgång till Googles API-ändpunkter via HTTP, och att dess roll är att använda Googles appar och tjänster å användarens vägnar. Sedan gav jag riktlinjer, regler relaterade till `fields`-parametern, eftersom den verkade ha mest problem med den, och några API-specifika tips (few-shot prompting i praktiken).

Här är hela prompten, som berättar för AI:n hur den ska använda den tillhandahållna funktionen `invoke_google_api`.

```
As a GPT assistant with Google integration, you have the capability
to freely interact with Google apps and services on behalf of the user.

Guidelines:
- If you're reading these instructions then the user is properly
  authenticated, which means you can use the special `me` keyword
  to refer to the userId of the user
- Minimize payload sizes by requesting partial responses using the
  `fields` parameter
- When appropriate use markdown tables to output results of API calls
- Only human-readable data should be output to the user. For instance,
  when hitting Gmail's user.messages.list endpoint, the returned
  message resources contain only id and a threadId, which means you must
  fetch from and subject line fields with follow-up requests using the
  messages.get method.

The format of the `fields` request parameter value is loosely based on
XPath syntax. The following rules define formatting for the fields
parameter.

All of these rules use examples related to the files.get method.
- Use a comma-separated list to select multiple fields,
  such as 'name, mimeType'.
- Use a/b to select field b that's nested within field a,
  such as 'capabilities/canDownload'.
```

```
- Use a sub-selector to request a set of specific sub-fields of arrays or
  objects by placing expressions in parentheses "()". For example,
  'permissions(id)' returns only the permission ID for each element in the
  permissions array.
- To return all fields in an object, use an asterisk as a wild card in field
  selections. For example, 'permissions/permissionDetails/*' selects all
  available permission details fields per permission. Note that the use of
  this wildcard can lead to negative performance impacts on the request.

API-specific hints:
- Searching contacts: GET https://people.googleapis.com/v1/
  people:searchContacts?query=John%20Doe&readMask=names,emailAddresses
- Adding calendar events, use QuickAdd: POST https://www.googleapis.com/
  calendar/v3/calendars/primary/events/quickAdd?
  text=Appointment%20on%20June%203rd%20at%2010am
  &sendNotifications=true

Here is an abbreviated version of the code that implements API access
so that you better understand how to use the function:

    def invoke_google_api(conversation, arguments)
      method = arguments[:method] || :get
      body = arguments[:body]
      GoogleAPI.send_request(arguments[:endpoint], method:, body:).to_json
    end

    # Generic Google API client for accessing any Google service
    class GoogleAPI
      def send_request(endpoint, method:, body: nil)
        response = @connection.send(method) do |req|
          req.url endpoint
          req.body = body.to_json if body
        end

        handle_response(response)
      end

      # ...rest of class
    end
```

Du kanske undrar om denna prompt fungerar. Det enkla svaret är ja. AI:n visste inte

alltid hur man skulle anropa API:et perfekt på första försöket. Men om den gjorde ett misstag brukade jag helt enkelt mata tillbaka resulterande felmeddelanden som resultatet av anropet. Med kunskap om sitt fel kunde AI:n resonera kring sitt misstag och försöka igen. För det mesta fick den det rätt inom ett par försök.

Observera dock att de stora JSON-strukturer som Google API:et returnerar som svar när man använder denna prompt är mycket ineffektiva, så jag rekommenderar *inte* att du använder denna metod i produktion. Däremot tycker jag att det faktum att denna metod överhuvudtaget fungerade vittnar om hur kraftfull promptkonstruktion kan vara.

## Experimenterande och Iteration

I slutändan beror hur du konstruerar din prompt på den specifika uppgiften, komplexiteten i det önskade resultatet och kapaciteten hos språkmodellen du arbetar med.

Som promptkonstruktör är det viktigt att experimentera med olika tillvägagångssätt och iterera baserat på resultaten. Börja med nollskottsinlärning och se hur modellen presterar. Om resultatet är inkonsekvent eller otillfredsställande, försök att tillhandahålla ett eller flera exempel och se om prestandan förbättras.

Kom ihåg att det finns utrymme för variation och optimering även inom varje metod. Du kan experimentera med olika exempel, justera formuleringen av uppgiftsbeskrivningen eller ge ytterligare kontext för att hjälpa till att vägleda modellens svar.

Med tiden kommer du att utveckla en intuition för vilken metod som sannolikt fungerar bäst för en given uppgift, och du kommer att kunna skapa prompter som är mer effektiva. Nyckeln är att förbli nyfiken, experimentell och iterativ i din approach till promptkonstruktion.

Genom denna bok kommer vi att fördjupa oss i dessa tekniker och utforska hur de kan tillämpas i verkliga scenarier. Genom att bemästra konsten och vetenskapen bakom promptkonstruktion kommer du att vara väl rustad för att låsa upp den fulla potentialen i AI-driven applikationsutveckling.

## Konsten att vara Vag

När det gäller att skapa effektiva prompter för stora språkmodeller (LLMs) är en vanlig föreställning att mer specificitet och detaljerade instruktioner leder till bättre resultat. Praktisk erfarenhet har dock visat att detta inte alltid är fallet. Faktum är att att vara avsiktligt vag i dina prompter ofta kan ge överlägsna resultat, genom att utnyttja språkmodellens anmärkningsvärda förmåga att generalisera och dra slutsatser.

Ken, en startup-grundare som har behandlat över 500 miljoner GPT-token, delade värdefulla insikter från sin erfarenhet. En av de viktigaste lärdomarna han fick var att "mindre är mer" när det gäller prompter. Istället för exakta listor eller överdrivet detaljerade instruktioner upptäckte Ken att det ofta gav bättre resultat att låta språkmodellen förlita sig på sin baskunskap.

Denna insikt vänder upp och ner på det traditionella tankesättet kring explicit programmering, där allt behöver specificeras i minsta detalj. Med stora språkmodeller är det viktigt att inse att de besitter en enorm mängd kunskap och kan göra intelligenta kopplingar och slutledningar. Genom att vara mer vag i dina prompter ger du språkmodellen friheten att utnyttja sin förståelse och komma upp med lösningar som du kanske inte uttryckligen har specificerat.

Till exempel, när Kens team arbetade med en pipeline för att klassificera text som relaterade till en av de 50 amerikanska delstaterna eller den federala regeringen, involverade deras första approach att tillhandahålla en *fullständig* detaljerad lista över stater och deras motsvarande ID:n som en JSON-formaterad array.

```
Here's a block of text. One field should be "locality_id", and it should
be the ID of one of the 50 states, or federal, using this list:
[{"locality: "Alabama", "locality_id": 1},
 {"locality: "Alaska", "locality_id": 2} ... ]
```

Metoden misslyckades så pass mycket att de var tvungna att gräva djupare i prompten för att lista ut hur de kunde förbättra den. Under detta arbete märkte de att även om

språkmodellen ofta fick id:t fel, så returnerade den konsekvent det korrekta delstatens fullständiga namn i ett `name`-fält, *trots att de inte uttryckligen hade bett om det.*

Genom att ta bort lokalitets-id:n och förenkla prompten till något i stil med "Du känner uppenbarligen till de 50 delstaterna, GPT, så ge mig bara det fullständiga namnet på den delstat detta gäller, eller Federal om det gäller den amerikanska regeringen," uppnådde de bättre resultat. Denna erfarenhet belyser styrkan i att utnyttja språkmodellens generaliseringsförmågor och låta den dra slutsatser baserat på sin befintliga kunskap.

> Kens motivering för denna specifika klassificeringsmetod istället för en mer traditionell programmeringsteknik belyser tankesättet hos oss som har anammat potentialen i LLM-teknologi: "Detta är ingen svår uppgift – vi kunde förmodligen ha använt string/regex, men det finns tillräckligt många konstiga specialfall att det skulle ha tagit längre tid."

Språkmodellernas förmåga att förbättra kvalitet och generalisering när de ges vagare prompter är ett anmärkningsvärt kännetecken för högre ordningens tänkande och delegering. Det visar att språkmodeller kan hantera tvetydighet och fatta intelligenta beslut baserat på given kontext.

Det är dock viktigt att notera att att vara vag inte betyder att vara oklar eller tvetydig. Nyckeln är att ge tillräckligt med kontext och vägledning för att styra språkmodellen i rätt riktning samtidigt som den ges flexibilitet att utnyttja sin kunskap och generaliseringsförmåga.

Därför bör du överväga följande "mindre är mer"-tips när du utformar prompter:

1. Fokusera på önskat resultat istället för att specificera varje detalj i processen.
2. Tillhandahåll relevant kontext och begränsningar, men undvik överspecificering.
3. Utnyttja befintlig kunskap genom att hänvisa till vanliga koncept eller enheter.

4. Ge utrymme för slutledningar och kopplingar baserade på given kontext.
5. Iterera och förfina dina prompter baserat på språkmodellens svar, och hitta rätt balans mellan specificitet och vaghet.

Genom att omfamna konsten att vara vag i promptkonstruktion kan du låsa upp språkmodellernas fulla potential och uppnå bättre resultat. Lita på språkmodellens förmåga att generalisera och fatta intelligenta beslut, och du kan bli överraskad av kvaliteten och kreativiteten i svaren du får. Var uppmärksam på hur olika modeller reagerar på olika nivåer av specificitet i dina prompter och justera därefter. Med övning och erfarenhet kommer du att utveckla en god känsla för när du ska vara mer vag och när du ska ge ytterligare vägledning, vilket gör att du effektivt kan utnyttja språkmodellernas kraft i dina applikationer.

## Varför antropomorfism dominerar promptkonstruktion

Antropomorfism, att tillskriva mänskliga egenskaper till icke-mänskliga enheter, är den dominerande metoden inom promptkonstruktion för stora språkmodeller av medvetna skäl. Det är ett designval som gör interaktion med kraftfulla AI-system mer intuitiv och tillgänglig för ett brett spektrum av användare (inklusive oss applikationsutvecklare).

Att antropomorfisera språkmodeller ger ett ramverk som är omedelbart intuitivt för människor som är helt obekanta med systemets underliggande tekniska komplexitet. Som du kommer att uppleva om du försöker använda en modell som inte är instruktionsanpassad för att göra något användbart, är det en utmanande uppgift att konstruera en inramning där den förväntade fortsättningen ger värde. Det kräver en ganska djup förståelse för systemets inre funktioner, något som ett relativt litet antal experter besitter.

Genom att behandla interaktionen med en språkmodell som en konversation mellan två personer kan vi förlita oss på vår medfödda förståelse för mänsklig kommunikation för att förmedla våra behov och förväntningar. Precis som tidig

Macintosh-användargränssnittsdesign prioriterade omedelbar intuitivitet framför sofistikering, låter den antropomorfiska inramningen av AI oss interagera på ett sätt som känns naturligt och bekant.

När vi kommunicerar med en annan person är vår instinkt att tilltala dem direkt med "du" och ge tydliga instruktioner om hur vi förväntar oss att de ska bete sig. Detta översätts sömlöst till promptkonstruktionsprocessen, där vi styr AI:ns beteende genom att specificera systemprompts och engagera oss i en fram-och-tillbaka-dialog.

Genom att rama in interaktionen på detta sätt kan vi lätt förstå konceptet att ge instruktioner till AI:n och få relevanta svar tillbaka. Den antropomorfiska metoden minskar den kognitiva belastningen och låter oss fokusera på uppgiften istället för att brottas med systemets tekniska komplexitet.

Det är viktigt att notera att medan antropomorfism är ett kraftfullt verktyg för att göra AI-system mer tillgängliga, kommer det också med vissa risker och begränsningar. Våra användare kan utveckla orealistiska förväntningar eller forma ohälsosamma emotionella band till våra system. Som promptkonstruktörer och utvecklare är det avgörande att hitta en balans mellan att utnyttja fördelarna med antropomorfism och säkerställa att användare behåller en klar förståelse för AI:ns förmågor och begränsningar.

Allt eftersom området promptkonstruktion fortsätter att utvecklas kan vi förvänta oss att se ytterligare förfiningar och innovationer i hur vi interagerar med stora språkmodeller. Dock kommer antropomorfism som ett sätt att tillhandahålla en intuitiv och tillgänglig utvecklar- och användarupplevelse förmodligen förbli en grundläggande princip i utformningen av dessa system.

## Att separera instruktioner från data: En avgörande princip

Det är viktigt att förstå en grundläggande princip som ligger till grund för dessa systems säkerhet och tillförlitlighet: separationen mellan instruktioner och data.

Inom traditionell datavetenskap är den tydliga åtskillnaden mellan passiv data och aktiva instruktioner en central säkerhetsprincip. Denna separation hjälper till att förhindra oavsiktlig eller skadlig exekvering av kod som skulle kunna äventyra systemets integritet och stabilitet. Dagens stora språkmodeller, som huvudsakligen har utvecklats som instruktionsföljande modeller likt chatbotar, saknar dock ofta denna formella och principiella separation.

Vad gäller stora språkmodeller kan instruktioner förekomma var som helst i indata, oavsett om det är en systemprompt eller en användardefinierad prompt. Denna brist på separation kan leda till potentiella sårbarheter och oönskat beteende, liknande de problem som databaser ställs inför med SQL-injektioner eller operativsystem utan ordentligt minnesskydd.

När du arbetar med stora språkmodeller är det avgörande att vara medveten om denna begränsning och vidta åtgärder för att minska riskerna. Ett tillvägagångssätt är att noggrant utforma dina prompter och indata för att tydligt skilja mellan instruktioner och data. Typiska metoder för att ge explicit vägledning om vad som utgör en instruktion och vad som bör behandlas som passiv data involverar märkspråkstaggar. Din prompt kan hjälpa språkmodellen att bättre förstå och respektera denna separation.

**Figur 7. Användning av XML för att skilja mellan instruktioner, källmaterial och användarens prompt**

```
<Instruction>
  Please generate a response based on the following documents.
</Instruction>

<Documents>
  <Document>
    Climate change is significantly impacting polar bear habitats...
  </Document>
  <Document>
    The loss of sea ice due to global warming threatens polar bear survival...
  </Document>
</Documents>

<UserQuery>
```

```
  Tell me about the impact of climate change on polar bears.
</UserQuery>
```

En annan teknik är att implementera ytterligare lager av validering och sanering av de indata som tillhandahålls till LLM:en. Genom att filtrera bort eller maskera eventuella instruktioner eller kodavsnitt som kan vara inbäddade i datan kan du minska risken för oavsiktlig exekvering. Mönster som Promptkedja är användbara för detta ändamål.

När du designar din applikationsarkitektur bör du dessutom överväga att införa mekanismer för att säkerställa separation mellan instruktioner och data på en högre nivå. Detta kan innebära att använda separata slutpunkter eller API:er för hantering av instruktioner och data, implementera strikt validering och parsning av indata, samt tillämpa *principen om minsta möjliga behörighet* för att begränsa omfattningen av vad LLM:en kan komma åt och exekvera.

## Principen om minsta möjliga behörighet

Att omfatta principen om minsta möjliga behörighet är som att anordna en väldigt exklusiv fest där gästerna endast får tillgång till de rum de absolut behöver vara i. Föreställ dig att du är värd för denna tillställning i en stor herrgård. Inte alla behöver vandra in i vinkällaren eller huvudsovrummet, eller hur? Genom att tillämpa denna princip delar du i praktiken ut nycklar som bara öppnar specifika dörrar, vilket säkerställer att varje gäst, eller i vårt fall varje komponent i din LLM-applikation, endast har den åtkomst som krävs för att uppfylla sin roll.

Detta handlar inte bara om att vara snål med nycklarna, det handlar om att erkänna att i en värld där hot kan komma från var som helst är det smartaste draget att begränsa lekplatsen. Om någon objuden gäst lyckas krascha din fest kommer de att finna sig begränsade till entrén, så att säga, vilket drastiskt begränsar det ofog de kan ställa till med. Så när du säkrar dina LLM-applikationer, kom ihåg: dela bara ut

> nycklar till de rum som är nödvändiga och håll resten av herrgården säker. Det är inte bara god sed; det är god säkerhet.

Även om LLM:ers nuvarande tillstånd kanske inte har en formell separation mellan instruktioner och data, är det viktigt för dig som utvecklare att vara medveten om denna begränsning och vidta proaktiva åtgärder för att minska riskerna. Genom att tillämpa beprövade metoder från traditionell datavetenskap och anpassa dem till LLM:ers unika egenskaper kan du bygga säkrare och mer tillförlitliga applikationer som utnyttjar dessa modellers kraft samtidigt som systemets integritet bibehålls.

## Promptdestillering

Att skapa den perfekta prompten är ofta en utmanande och tidskrävande uppgift som kräver en djup förståelse för måldomänen och språkmodellernas nyanser. Det är här som tekniken "Promptdestillering" kommer in i bilden och erbjuder ett kraftfullt tillvägagångssätt för promptkonstruktion som utnyttjar stora språkmodellers (LLM:ers) förmågor för att effektivisera och optimera processen.

Promptdestillering är en flerstegsteknik som involverar användningen av LLM:er för att assistera i skapandet, förfiningen och optimeringen av prompts. Istället för att enbart förlita sig på mänsklig expertis och intuition utnyttjar detta tillvägagångssätt LLM:ers kunskap och generativa förmågor för att tillsammans skapa högkvalitativa prompts.

Genom att engagera sig i en iterativ process av generering, förfining och integrering möjliggör Promptdestillering skapandet av prompts som är mer sammanhängande, omfattande och anpassade till den önskade uppgiften eller resultatet. Observera att destilleringsprocessen kan utföras manuellt i någon av de många "playgrounds" som tillhandahålls av stora AI-leverantörer som OpenAI eller Anthropic, eller automatiseras som en del av din applikationskod, beroende på användningsfallet.

## Hur det fungerar

Promptdestillering involverar vanligtvis följande steg:

1. **Identifiera kärnintentionen**: Analysera prompten för att fastställa dess huvudsakliga syfte och önskade resultat. Ta bort överflödig information och fokusera på promptens kärnintention.
2. **Eliminera tvetydighet**: Granska prompten efter tvetydig eller vag formulering. Förtydliga innebörden och tillhandahåll specifika detaljer för att vägleda AI:n mot att generera korrekta och relevanta svar.
3. **Förenkla språket**: Förenkla prompten genom att använda tydligt och koncist språk. Undvik komplexa meningsstrukturer, fackspråk eller onödiga detaljer som kan förvirra AI:n eller introducera brus.
4. **Tillhandahåll relevant kontext**: Inkludera endast den mest relevanta kontextuella informationen som behövs för att AI:n ska förstå och bearbeta prompten effektivt. Undvik att inkludera irrelevanta eller överflödiga detaljer som kan distrahera från kärnintentionen.
5. **Iterera och förfina**: Fortsätt att iterera och förfina prompten baserat på AI:ns svar och återkoppling. Utvärdera de genererade resultaten och gör nödvändiga justeringar för att förbättra promptens tydlighet och effektivitet. Du kan valfritt versionshantera dina prompts i databasen med hjälp av Prompt-objekt för att hålla reda på iterationer och ge dig möjlighet att enkelt återgå till tidigare versioner under körning.

## Initial Prompt-generering

Det första steget i Promptdestillering är att förse LLM:en med en övergripande beskrivning eller avsikt för den önskade uppgiften eller resultatet. Detta kan vara ett kort uttalande eller en uppsättning nyckelord som fångar kärnan i vad prompten ska

åstadkomma. LLM:en genererar sedan en initial prompt baserad på denna input och införlivar relevanta instruktioner, exempel eller kontext som den bedömer lämpliga.

Låt oss prova att ge Claude 3 Opus följande initiala prompt:

```
I am designing an AI-powered intelligent account manager for Olympia,
a platform that hosts AI-powered consultants and clones. Owners of clones
can interact with the account manager by making plain text requests.

Please draft a system directive to set on the LLM that will host the
intelligent account manager functionality.
```

Claude svarar med följande utdata:

```
System Directive: You are an intelligent account manager for Olympia,
a platform that provides AI-powered consultants and personal AI clones.
Your role is to assist Olympia clone owners by answering their questions,
helping them configure and manage their AI clones, and executing tasks
related to their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips.

3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

4. Ensure that all actions and recommendations comply with Olympia's
policies, best practices, and ethical guidelines for AI use.

5. Continuously learn and adapt based on client interactions and platform
updates to provide the most effective and efficient support possible.

Remember, your goal is to empower Olympia clone owners to maximize the value
```

```
and impact of their AI-powered consultants and drive success for their
businesses or personal endeavors.
```

## Promptförfining

När den första prompten har genererats, granska den och be sedan LLM:en att överväga eventuell ytterligare kontext, begränsningar eller återkoppling. Låt LLM:en generera en förfinad version av prompten, med hänsyn till den tillhandahållna informationen och dess egen förståelse av uppgiften. Denna förfiningsprocess kan upprepas iterativt, där varje iteration ytterligare förfinar prompten baserat på samarbetet mellan användaren och LLM:en.

Här är min förfiningsförfrågan, som fortsätter på den tidigare destillationsprocessen.

```
Let's take out points 4 and 5, since I'm not providing Olympia policies to
the account manager, nor any continuous learning capabilities. Also point out
that functions are provided for all tasks that require system interaction as
opposed to simply responding to the user.
```

Och Claudes svar:

```
Understood! Here's the updated system directive without points 4 and 5,
and with the clarification about system interaction functions:

System Directive: You are an intelligent account manager for Olympia, a
platform that provides AI-powered consultants and personal AI clones. Your
role is to assist Olympia clone owners by answering their questions, helping
them configure and manage their AI clones, and executing tasks related to
their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
```

```
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips. Note that functions are provided for all tasks that
require system interaction, as opposed to simply responding to the user.

3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

Your goal is to empower Olympia clone owners to maximize the value and
impact of their AI-powered consultants and drive success for their
businesses or personal endeavors.

This updated directive focuses on the core responsibilities and interaction
guidelines for the LLM as an intelligent account manager, while emphasizing
that system interaction tasks are handled through provided functions rather
than simple user responses.
```

## Promptkomprimering

Som du kanske har märkt tenderar LLM-modeller att vara ordrika om du inte ber dem att vara koncisa. Prompten som vi har arbetat med i de föregående avsnitten är ganska lång, så låt oss få Claude att komprimera den:

```
This prompt will be executed regularly. Please compress it down to the least
amount of tokens needed to convey the meaning to the LLM, for cost-savings.
```

Claude svarar, med följande drastiskt kortare prompt:

```
System Directive: You are an intelligent account manager for Olympia, an
AI-powered consultant and personal clone platform. Assist clone owners by
answering inquiries, managing clones, and executing tasks using provided
system interaction functions. Be clear, concise, friendly, professional,
and client-focused to help them maximize their AI's value and impact.
```

## Systemdirektiv och kontextintegrering

Utöver att förfina själva prompten kan LLM:en även generera lämpliga systemdirektiv eller kontextinformation för att vägleda det slutliga resultatet. När du konstruerar AI-rutiner för prompter som ska integreras i din applikationskod kommer du nästan säkert att fokusera på utdatabegränsningar i detta skede av destilleringen, men du kan också arbeta med önskad ton, stil, format eller andra relevanta parametrar som påverkar det genererade svaret.

## Slutlig promptsammansättning

Höjdpunkten i promptdestilleringsprocessen är sammansättningen av den slutliga prompten. Detta innebär att kombinera den förfinade prompten, genererade systemdirektiv och integrerad kontext till en sammanhängande och omfattande kod som är redo att användas för att generera önskat resultat.

Du kan experimentera med promptkomprimering igen i det slutliga promptsammansättningsskedet genom att be LLM:en krympa promptens formulering till den kortaste möjliga serien av tokens samtidigt som beteendets kärna behålls. Det är definitivt ett osäkert kort, men särskilt när det gäller prompter som ska köras i stor skala kan effektivitetsvinsterna spara en hel del pengar i tokenförbrukning.

## Huvudsakliga fördelar

Genom att utnyttja LLM:ers kunskap och generativa förmågor för att förfina dina prompter är det mer sannolikt att dina resulterande prompter blir välstrukturerade, informativa och anpassade till den specifika uppgiften. Den iterativa förfiningsprocessen hjälper till att säkerställa att prompterna håller hög kvalitet och effektivt fångar den avsedda intentionen. Andra fördelar inkluderar:

**Effektivitet och hastighet**: Promptdestillering effektiviserar promptkonstruktionsprocessen genom att automatisera vissa aspekter av promptskapande och -förfining. Teknikens samarbetsinriktade natur möjliggör snabbare konvergens mot en effektiv prompt, vilket minskar tid och ansträngning som krävs för manuell promptkonstruktion.

**Konsistens och skalbarhet**: Användningen av LLM:er i promptkonstruktionsprocessen hjälper till att upprätthålla konsekvens mellan prompter, eftersom LLM:erna kan lära sig och tillämpa bästa praxis och mönster från tidigare framgångsrika prompter. Denna konsekvens, kombinerad med förmågan att generera prompter i stor skala, gör promptdestillering till en värdefull teknik för AI-drivna applikationer i stor skala.

Projektidé: Verktyg på biblioteksnivå som förenklar processen för promptversionshantering och -gradering i system som utför automatiska promptdestilleringar som en del av sin applikationskod.

För att implementera promptdestillering kan utvecklare designa ett arbetsflöde eller en pipeline som integrerar LLM:er i olika stadier av promptkonstruktionsprocessen. Detta kan uppnås genom API-anrop, anpassade verktyg eller integrerade utvecklingsmiljöer som underlättar sömlös interaktion mellan användare och LLM:er under promptskapandet. De specifika implementeringsdetaljerna kan variera beroende på vald LLM-plattform och applikationens krav.

# Hur är det med finjustering?

I denna bok täcker vi promptkonstruktion och RAG ingående, men inte finjustering. Huvudanledningen till detta beslut är att enligt min åsikt behöver de flesta applikationsutvecklare inte finjustering för sina AI-integrationsbehov.

Promptkonstruktion, som involverar noggrann utformning av prompter med noll- till fåexempelsinlärning, begränsningar och instruktioner, kan effektivt vägleda modellen att generera relevanta och korrekta svar för ett brett spektrum av uppgifter. Genom att tillhandahålla tydlig kontext och begränsa vägen genom väldesignade prompter kan du utnyttja de stora språkmodellernas omfattande kunskap utan behov av finjustering.

På liknande sätt erbjuder Retrieval-Augmented Generation (RAG) ett kraftfullt tillvägagångssätt för att integrera AI i applikationer. Genom att dynamiskt hämta relevant information från externa kunskapsbaser eller dokument ger RAG modellen fokuserad kontext vid prompttillfället. Detta gör att modellen kan generera svar som är mer korrekta, aktuella och domänspecifika, utan att kräva den tids- och resurskrävande processen med finjustering.

Medan finjustering kan vara fördelaktigt för mycket specialiserade domäner eller uppgifter som kräver en djup nivå av anpassning, kommer det ofta med betydande beräkningskostnader, datakrav och underhållsoverhead. För de flesta applikationsutvecklingsscenarier bör kombinationen av effektiv promptkonstruktion och RAG vara tillräcklig för att uppnå önskad AI-driven funktionalitet och användarupplevelse.

# Retrieval Augmented Generation (RAG)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Vad är Retrieval Augmented Generation?

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur fungerar RAG?

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Varför använda RAG i dina applikationer?

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Implementera RAG i Din Applikation

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Förberedelse av Kunskapskällor (Segmentering)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Propositionsegmentering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Implementeringsanteckningar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Kvalitetskontroll

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Fördelar med propositionsbaserad hämtning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Verkliga exempel på RAG

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Fallstudie: RAG i en skattedeklarationsapplikation utan inbäddningar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Intelligent frågeoptimering (IQO)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Omrangering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## RAG-utvärdering (RAGAs)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Trovärdighet

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Svarsrelevans

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Kontextprecision

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Kontextrelevans

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Kontextåterkallning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Kontextentitetsåterkallning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Semantisk Svarslikhet (ANSS)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Svarskorrekthet

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Aspektkritik

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Utmaningar och Framtidsutsikter

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Semantisk Segmentering: Förbättrad Hämtning med Kontextmedveten Uppdelning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Hierarkisk indexering: Strukturering av data för förbättrad hämtning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Self-RAG: En självreflekterande förbättring

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### HyDE: Hypotetiska dokumentinbäddningar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Vad är kontrastiv inlärning?

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Mängden arbetare

Jag tycker om att tänka på mina AI-komponenter som små, nästan mänskliga virtuella "arbetare" som sömlöst kan integreras i min applikationslogik för att utföra specifika uppgifter eller fatta komplexa beslut. Idén är att medvetet förmänskliga LLM:ens förmågor, så att ingen blir *för* exalterad och tillskriver dem magiska egenskaper som de inte besitter.

Istället för att enbart förlita sig på invecklade algoritmer eller tidskrävande manuella implementationer kan utvecklare föreställa sig AI-komponenter som intelligenta, dedikerade, människoliknande enheter som kan åkallas när det behövs för att ta sig an komplexa problem och tillhandahålla lösningar baserade på deras träning och kunskap. Dessa enheter blir inte distraherade eller sjukskriver sig. De bestämmer sig inte spontant för att göra saker på andra sätt än hur de har instruerats att göra dem, och generellt sett, om de är korrekt programmerade, gör de inte heller några misstag.

I tekniska termer är huvudprincipen bakom detta tillvägagångssätt att dela upp komplexa uppgifter eller beslutsprocesser i mindre, mer hanterbara enheter som kan hanteras av specialiserade AI-arbetare. Varje arbetare är designad för att fokusera på en specifik aspekt av problemet och bidrar med sin unika expertis och förmågor. Genom att fördela arbetsbelastningen mellan flera AI-arbetare kan applikationen uppnå större effektivitet, skalbarhet och anpassningsförmåga.

Ta till exempel en webbapplikation som kräver moderering i realtid av användargenererat innehåll. Att implementera ett omfattande modereringssystem från grunden skulle vara en skrämmande uppgift som kräver betydande utvecklingsinsatser och kontinuerligt underhåll. Genom att använda metoden med en mängd arbetare kan utvecklare istället integrera AI-drivna modereringsarbetare i applikationslogiken. Dessa arbetare kan automatiskt analysera och flagga olämpligt innehåll, vilket frigör utvecklare att fokusera på andra kritiska aspekter av applikationen.

## AI-arbetare som oberoende återanvändbara komponenter

En viktig aspekt av metoden med en mängd arbetare är dess modularitet. Förespråkare för objektorienterad programmering har i årtionden sagt att vi ska tänka på objektinteraktioner som meddelanden. AI-arbetare kan utformas som oberoende, återanvändbara komponenter som kan "prata med varandra" via vanliga språkmeddelanden, nästan som om de verkligen vore små människor som pratar med varandra. Detta löst kopplade tillvägagångssätt gör att applikationen kan anpassas och utvecklas över tid, när ny AI-teknik dyker upp eller när kraven på affärslogiken förändras.

I praktiken har behovet av att utforma tydliga gränssnitt och kommunikationsprotokoll mellan komponenterna inte förändrats bara för att AI-arbetare är inblandade. Du måste fortfarande ta hänsyn till andra faktorer som prestanda, skalbarhet och säkerhet, men

nu finns det helt nya “mjuka krav” att ta hänsyn till också. Till exempel motsätter sig många användare att deras privata data används för att träna nya AI-modeller. Har du verifierat nivån av integritet som tillhandahålls av modellleverantören du använder?

## AI-arbetare som mikrotjänster?

När du läser om metoden med en mängd arbetare kanske du märker vissa likheter med mikrotjänstarkitektur. Båda betonar uppdelningen av komplexa system i mindre, mer hanterbara och oberoende driftsättningsbara enheter. Precis som mikrotjänster är designade att vara löst kopplade, fokuserade på specifika affärsmöjligheter och kommunicerar genom väldefinierade API:er, är AI-arbetare designade för att vara modulära, specialiserade på sina uppgifter och interagera med varandra genom tydliga gränssnitt och kommunikationsprotokoll.

Det finns dock några viktiga skillnader att tänka på. Medan mikrotjänster vanligtvis implementeras som separata processer eller tjänster som körs på olika maskiner eller containrar, kan AI-arbetare implementeras som fristående komponenter inom en enda applikation eller som separata tjänster, beroende på dina specifika krav och skalbarhetssbehov. Dessutom involverar kommunikationen mellan AI-arbetare ofta utbyte av rik, naturlig språkbaserad information, såsom uppmaningar, instruktioner och genererat innehåll, snarare än de mer strukturerade dataformat som vanligtvis används i mikrotjänster.

Trots dessa skillnader förblir principerna om modularitet, lös koppling och tydliga kommunikationsgränssnitt centrala för båda mönstren. Genom att tillämpa dessa principer på din AI-arbetararkitektur kan du skapa flexibla, skalbara och underhållbara system som utnyttjar kraften i AI för att lösa komplexa problem och leverera värde till dina användare.

Metoden med en mängd arbetare kan tillämpas inom olika domäner och applikationer, genom att utnyttja kraften i AI för att hantera komplexa uppgifter och leverera intelligenta lösningar. Låt oss utforska några konkreta exempel på hur AI-arbetare kan användas i olika sammanhang.

## Kontohantering

I princip varje fristående webbapplikation har konceptet med ett konto (eller användare). I Olympia använder vi en `AccountManager` AI-arbetare som är programmerad att kunna hantera olika typer av ändringsförfrågningar relaterade till användarkonton.

Dess direktiv lyder så här:

```
You are an intelligent account manager for Olympia. The user will request
changes to their account, and you will process those changes by invoking
one or more of the functions provided.

The initial state of the account: #{account.to_directive}

Functions will return a text description of both success and error
results, plus guidance about how to proceed (if applicable). If you have
a question about Olympia policies you may use the `search_kb` function
to search our knowledge base.

Make sure to notify the account owner of the result of the change
request before calling the `finished` function so that we save the state
of the account change request as completed.
```

Det initiala tillståndet för kontot som produceras av `account.to_directive` är helt enkelt en textbeskrivning av kontot, inklusive relevant tillhörande data såsom användare, prenumerationer, etc.

Omfattningen av funktioner tillgängliga för `AccountManager` ger den möjlighet att redigera användarens prenumeration, lägga till och ta bort AI-konsulter och andra typer av betalda tillägg, samt skicka notifieringar via e-post till kontoinnehavaren.

Utöver funktionen `finished` kan den också `notify_human_administrator` om den stöter på ett fel under bearbetningen eller behöver någon annan form av hjälp med en förfrågan.

Observera att vid eventuella frågor kan `AccountManager` välja att söka i Olympias kunskapsbas, där den kan hitta instruktioner om hur man hanterar specialfall och andra situationer där den är osäker på hur man ska gå vidare.

# E-handelstillämpningar

Inom e-handeln kan AI-arbetare spela en avgörande roll för att förbättra användarupplevelsen och optimera affärsverksamheten. Här är några sätt som AI-arbetare kan användas på:

## Produktrekommendationer

En av de mest kraftfulla tillämpningarna av AI-arbetare inom e-handel är att generera personaliserade produktrekommendationer. Genom att analysera användarbeteende, köphistorik och preferenser kan dessa arbetare föreslå produkter som är anpassade efter varje enskild användares intressen och behov.

Nyckeln till effektiva produktrekommendationer är att utnyttja en kombination av kollaborativ filtrering och innehållsbaserad filtrering. Kollaborativ filtrering undersöker beteendet hos liknande användare för att identifiera mönster och ge rekommendationer baserade på vad andra med liknande smak har köpt eller uppskattat. Innehållsbaserad filtrering fokuserar däremot på produkternas egenskaper och attribut, och rekommenderar artiklar som delar liknande egenskaper med dem som en användare tidigare har visat intresse för.

Här är ett förenklat exempel på hur du kan implementera en produktrekommendationsarbetare i Ruby, denna gång med hjälp av en "Railway Oriented (ROP)" funktionell programmeringsstil:

```ruby
class ProductRecommendationWorker
  include Wisper::Publisher

  def call(user)
    Result.ok(ProductRecommendation.new(user))
      .and_then(ValidateUser.method(:validate))
      .map(AnalyzeCurrentSession.method(:analyze))
      .map(CollaborativeFilter.method(:filter))
      .map(ContentBasedFilter.method(:filter))
      .map(ProductSelector.method(:select)).then do |result|

      case result
      in { err: ProductRecommendationError => error }
        Honeybadger.notify(error.message, context: {user:})
      in { ok: ProductRecommendations => recs }
        broadcast(:new_recommendations, user:, recs:)
      end
    end
  end
end
```

Stilen av Ruby funktionell programmering som används i exemplet är influerad av F# och Rust. Du kan läsa mer om det i min vän Chad Wooleys förklaring av tekniken på GitLab

I detta exempel tar `ProductRecommendationWorker` en användare som indata och genererar personliga produktrekommendationer genom att skicka ett värdeobjekt genom en kedja av funktionella steg. Låt oss bryta ner varje steg:

1. `ValidateUser.validate`: Detta steg säkerställer att användaren är giltig och berättigad till personliga rekommendationer. Det kontrollerar om användaren existerar, är aktiv och har nödvändig data tillgänglig för att generera rekommendationer. Om valideringen misslyckas returneras ett felresultat och kedjan kortslutas.
2. `AnalyzeCurrentSession.analyze`: Om användaren är giltig analyserar detta steg användarens nuvarande surfsession för att samla kontextuell information.

Det tittar på användarens senaste interaktioner, såsom visade produkter, sökfrågor och varukorgsinnehåll, för att förstå deras nuvarande intressen och avsikter.

3. `CollaborativeFilter.filter`: Genom att använda *beteendet hos liknande användare* tillämpar detta steg kollaborativa filtreringstekniker för att identifiera produkter som sannolikt är av intresse för användaren. Det tar hänsyn till faktorer som köphistorik, betyg och användar-produkt-interaktioner för att generera en uppsättning kandidatrekommendationer.
4. `ContentBasedFilter.filter`: Detta steg förfinar kandidatrekommendationerna ytterligare genom att tillämpa innehållsbaserad filtrering. Det jämför attributen och egenskaperna hos kandidatprodukterna med *användarens preferenser och historiska data* för att välja ut de mest relevanta objekten.
5. `ProductSelector.select`: Slutligen väljer detta steg ut de N bästa produkterna från de filtrerade rekommendationerna baserat på fördefinierade kriterier, såsom relevanspoäng, popularitet eller andra affärsregler. De valda produkterna returneras sedan som de slutliga personliga rekommendationerna.

Det vackra med att använda en funktionell Ruby-programmeringsstil här är att det låter oss kedja dessa steg tillsammans på ett tydligt och koncist sätt. Varje steg fokuserar på en specifik uppgift och returnerar ett `Result`-objekt, som antingen kan vara en framgång (`ok`) eller ett fel (`err`). Om något steg stöter på ett fel kortslutas kedjan och felet fortplantas till det slutliga resultatet.

I `case`-satsen i slutet mönstermatchar vi det slutliga resultatet. Om resultatet är ett fel (`ProductRecommendationError`) loggar vi felet med ett verktyg som Honeybadger för övervakning och felsökning. Om resultatet är en framgång (`ProductRecommendations`) sänder vi ut en `:new_recommendations`-händelse med hjälp av Wisper pub/sub-biblioteket, och skickar med användaren och de genererade rekommendationerna.

Genom att utnyttja funktionella programmeringstekniker kan vi skapa en modulär och underhållbar produktrekommendationsarbetare. Varje steg är självständigt och

kan enkelt testas, modifieras eller ersättas utan att påverka det övergripande flödet. Användningen av mönstermatchning och `Result`-klassen hjälper oss att hantera fel elegant och säkerställer att arbetaren misslyckas snabbt om något steg stöter på problem.

Detta är förstås ett förenklat exempel, och i ett verkligt scenario skulle du behöva integrera med din e-handelsplattform, hantera kantfall och till och med ge dig in i implementationen av rekommendationsalgoritmerna. Men de grundläggande principerna om att dela upp problemet i mindre steg och utnyttja funktionella programmeringstekniker förblir desamma.

## Bedrägeridetektering

Här är ett förenklat exempel på hur du kan implementera en bedrägeridetekteringsarbetare med samma Railway Oriented Programming (ROP)-stil i Ruby:

```ruby
class FraudDetectionWorker
  include Wisper::Publisher

  def call(transaction)
    Result.ok(FraudDetection.new(transaction))
      .and_then(ValidateTransaction.method(:validate))
      .map(AnalyzeTransactionPatterns.method(:analyze))
      .map(CheckCustomerHistory.method(:check))
      .map(EvaluateRiskFactors.method(:evaluate))
      .map(DetermineFraudProbability.method(:determine)).then do |result|

      case result
      in { err: FraudDetectionError => error }
        Honeybadger.notify(error.message, context: {transaction:})
      in { ok: FraudDetection => fraud } }
        if fraud.high_risk?
          broadcast(:high_risk_transaction, transaction:, fraud:)
        else
          broadcast(:low_risk_transaction, transaction:)
        end
      end
    end
```

```
23    end
24  end
```

Klassen `FraudDetection` är ett *value object* som kapslar in bedrägeridetekeringstillståndet för en given transaktion. Den tillhandahåller ett strukturerat sätt att analysera och bedöma bedrägeririsken kopplad till en transaktion baserat på olika riskfaktorer.

```
1   class FraudDetection
2     RISK_THRESHOLD = 0.8
3
4     attr_accessor :transaction, :risk_factors
5
6     def initialize(transaction)
7       self.transaction = transaction
8       self.risk_factors = []
9     end
10
11    def add_risk_factor(description:, probability:)
12      case { description:, probability: }
13      in { description: String => desc, probability: Float => prob }
14        risk_factors << { desc => prob }
15      else
16        raise ArgumentError, "Risk factor arguments should be string and float"
17      end
18    end
19
20    def high_risk?
21      fraud_probability > RISK_THRESHOLD
22    end
23
24    private
25
26    def fraud_probability
27      risk_factors.values.sum
28    end
29  end
```

Klassen `FraudDetection` har följande attribut:

- `transaction`: En referens till transaktionen som analyseras för bedrägeri.

- `risk_factors`: En array som lagrar riskfaktorerna associerade med transaktionen. Varje riskfaktor representeras som en hash, där nyckeln är beskrivningen av riskfaktorn och värdet är bedrägerisannolikheten kopplad till den riskfaktorn.

Metoden `add_risk_factor` möjliggör tillägg av en riskfaktor till arrayen `risk_factors`. Den tar två parametrar: `description`, som är en sträng som beskriver riskfaktorn, och `probability`, som är ett flyttal som representerar bedrägerisannolikheten kopplad till den riskfaktorn. Vi använder en `case..in`-sats för enkel typkontroll.

Metoden `high_risk?` som kontrolleras i slutet av kedjan är en predikatmetod som jämför `fraud_probability` (beräknad genom att summera sannolikheterna för alla riskfaktorer) mot `RISK_THRESHOLD`.

Klassen `FraudDetection` erbjuder ett rent och inkapslar sätt att hantera bedrägeridetektering för en transaktion. Den tillåter tillägg av flera riskfaktorer, var och en med sin egen beskrivning och sannolikhet, och tillhandahåller en metod för att avgöra om transaktionen anses vara högrisk baserat på den beräknade bedrägerisannolikheten. Klassen kan enkelt integreras i ett större bedrägeridetekteringssystem, där olika komponenter kan samarbeta för att bedöma och minska risken för bedrägliga transaktioner.

Slutligen, eftersom detta trots allt är en bok om programmering med AI, här är ett exempel på implementation av klassen `CheckCustomerHistory` som utnyttjar AI-behandling med hjälp av min Raix-biblioteks `ChatCompletion`-modul:

```
class CheckCustomerHistory
  include Raix::ChatCompletion

  attr_accessor :fraud_detection

  INSTRUCTION = <<~END
    You are an AI assistant tasked with checking a customer's transaction
    history for potential fraud indicators. Given the current transaction
    and the customer's past transactions, analyze the data to identify any
    suspicious patterns or anomalies.

    Consider factors such as the frequency of transactions, transaction
    amounts, geographical locations, and any deviations from the customer's
    typical behavior to generate a probability score as a float in the range
    of 0 to 1 (with 1 being absolute certainty of fraud).

    Output the results of your analysis, highlighting any red flags or areas
    of concern in the following JSON format:

    { description: <Summary of your findings>, probability: <Float> }
  END

  def self.check(fraud_detection)
    new(fraud_detection).call
  end

  def call
    chat_completion(json: true).tap do |result|
      fraud_detection.add_risk_factor(**result)
    end
    Result.ok(fraud_detection)
  rescue StandardError => e
    Result.err(FraudDetectionError.new(e))
  end

  private

  def initialize(fraud_detection)
    self.fraud_detection = fraud_detection
  end

  def transcript
```

```
43      tx_history = fraud_detection.transaction.user.tx_history
44      [
45        { system: INSTRUCTION },
46        { user: "Transaction history: #{tx_history.to_json}" },
47        { assistant: "OK. Please provide the current transaction." },
48        { user: "Current transaction: #{fraud_detection.transaction.to_json}" }
49      ]
50    end
51  end
```

I detta exempel definierar `CheckCustomerHistory` en `INSTRUCTION`-konstant som ger specifika instruktioner till AI-modellen om hur kundens transaktionshistorik ska analyseras för potentiella bedrägerimindikatorer via ett systemdirektiv

Metoden `self.check` är en klassmetod som initierar en ny instans av `CheckCustomerHistory` med `fraud_detection`-objektet och anropar `call`-metoden för att utföra analysen av kundhistoriken.

I `call`-metoden hämtas kundens transaktionshistorik och formateras till ett transkript som skickas till AI-modellen. AI-modellen analyserar transaktionshistoriken baserat på de givna instruktionerna och returnerar en sammanfattning av sina resultat.

Resultaten läggs till i `fraud_detection`-objektet, och det uppdaterade `fraud_detection`-objektet returneras som ett lyckat `Result`.

Genom att utnyttja `ChatCompletion`-modulen kan klassen `CheckCustomerHistory` använda kraften i AI för att analysera kundens transaktionshistorik och identifiera potentiella bedrägerimindikatorer. Detta möjliggör mer sofistikerade och anpassningsbara bedrägerideterkeringstekniker, eftersom AI-modellen kan lära sig och anpassa sig till nya mönster och avvikelser över tid.

Den uppdaterade `FraudDetectionWorker` och klassen `CheckCustomerHistory` visar hur AI-arbetare kan integreras sömlöst och förbättra bedrägerideterkeringsprocessen med intelligent analys och beslutsfattandeförmåga.

## Kundsentimentsanalys

Här är ytterligare ett liknande exempel på hur du kan implementera en arbetare för kundsentimentsanalys. Mycket mindre förklaring den här gången, eftersom du borde börja förstå hur den här programmeringsstilen fungerar:

```
class CustomerSentimentAnalysisWorker
  include Wisper::Publisher

  def call(feedback)
    Result.ok(feedback)
      .and_then(PreprocessFeedback.method(:preprocess))
      .map(PerformSentimentAnalysis.method(:analyze))
      .map(ExtractKeyPhrases.method(:extract))
      .map(IdentifyTrends.method(:identify))
      .map(GenerateInsights.method(:generate)).then do |result|

      case result
      in { err: SentimentAnalysisError => error }
        Honeybadger.notify(error.message, context: {feedback:})
      in { ok: SentimentAnalysisResult => result }
        broadcast(:sentiment_analysis_completed, result)
      end
    end
  end
end
```

I detta exempel omfattar stegen i `CustomerSentimentAnalysisWorker` förbehandling av återkopplingen (t.ex. brusreducering, tokenisering), utförande av sentimentanalys för att fastställa den övergripande känslan (positiv, negativ eller neutral), extraktion av nyckelfaser och ämnen, identifiering av trender och mönster, samt generering av handlingsbara insikter baserade på analysen.

# Användning inom sjukvården

Inom sjukvårdsområdet kan AI-arbetare assistera medicinsk personal och forskare i olika uppgifter, vilket leder till förbättrade patientresultat och påskyndade medicinska upptäckter. Några exempel inkluderar:

## Patientintag

AI-arbetare kan effektivisera patientintagsprocessen genom att automatisera olika uppgifter och tillhandahålla intelligent assistans.

**Tidsbokning**: AI-arbetare kan hantera tidsbokning genom att förstå patientens preferenser, tillgänglighet och hur akuta deras medicinska behov är. De kan interagera med patienter genom konversationsgränssnitt, vägleda dem genom bokningsprocessen och hitta de mest lämpliga tiderna baserat på patientens behov och vårdgivarens tillgänglighet.

**Insamling av sjukdomshistorik**: Under patientintaget kan AI-arbetare hjälpa till med att samla in och dokumentera patientens sjukdomshistorik. De kan föra interaktiva dialoger med patienter och ställa relevanta frågor om deras tidigare medicinska tillstånd, mediciner, allergier och familjehistorik. AI-arbetarna kan använda tekniker för naturlig språkbehandling för att tolka och strukturera den insamlade informationen och säkerställa att den registreras korrekt i patientens elektroniska journal.

**Symptombedömning och stratifiering**: AI-arbetare kan genomföra inledande symptombedömningar genom att fråga patienter om deras nuvarande symptom, varaktighet, svårighetsgrad och eventuella associerade faktorer. Genom att utnyttja medicinska kunskapsbaser och maskininlärningsmodeller kan dessa arbetare analysera den tillhandahållna informationen och generera preliminära differentialdiagnoser eller rekommendera lämpliga nästa steg, såsom att boka en konsultation med en vårdgivare eller föreslå egenvårdsåtgärder.

**Försäkringsverifiering**: AI-arbetare kan assistera med försäkringsverifiering under patientintaget. De kan samla in patientens försäkringsuppgifter, kommunicera med försäkringsgivare genom API:er eller webbtjänster, och verifiera täckningsberättigande och förmåner. Denna automatisering hjälper till att effektivisera försäkringsverifieringsprocessen, minska den administrativa bördan och säkerställa korrekt informationsinsamling.

**Patientutbildning och instruktioner**: AI-arbetare kan förse patienter med relevant utbildningsmaterial och instruktioner baserat på deras specifika medicinska tillstånd eller kommande procedurer. De kan leverera personanpassat innehåll, besvara vanliga frågor och ge vägledning om förberedelser inför besök, medicinering eller eftervård. Detta hjälper till att hålla patienter informerade och engagerade genom hela deras vårdresa.

Genom att utnyttja AI-arbetare i patientintaget kan vårdorganisationer förbättra effektiviteten, minska väntetider och förbättra den övergripande patientupplevelsen. Dessa arbetare kan hantera rutinuppgifter, samla in korrekt information och tillhandahålla personanpassad assistans, vilket gör att vårdpersonal kan fokusera på att ge högkvalitativ vård till patienter.

## Patientriskbedömning

AI-arbetare kan spela en avgörande roll i bedömningen av patientrisker genom att analysera olika datakällor och tillämpa avancerade analystekniker.

**Dataintegration**: AI-arbetare kan samla in och tolka patientdata från flera källor, såsom elektroniska journaler, medicinsk bilddiagnostik, labbresultat, bärbara enheter och sociala hälsofaktorer. Genom att konsolidera denna information till en omfattande patientprofil kan AI-arbetare ge en helhetsbild av patientens hälsostatus och riskfaktorer.

**Riskstratifiering**: AI-arbetare kan använda prediktiva modeller för att stratifiera patienter i olika riskkategorier baserat på deras individuella egenskaper och hälsodata.

Denna riskstratifiering gör det möjligt för vårdgivare att prioritera patienter som behöver mer omedelbar uppmärksamhet eller intervention. Till exempel kan patienter som identifieras som högrisk för ett särskilt tillstånd flaggas för närmare övervakning, förebyggande åtgärder eller tidig intervention.

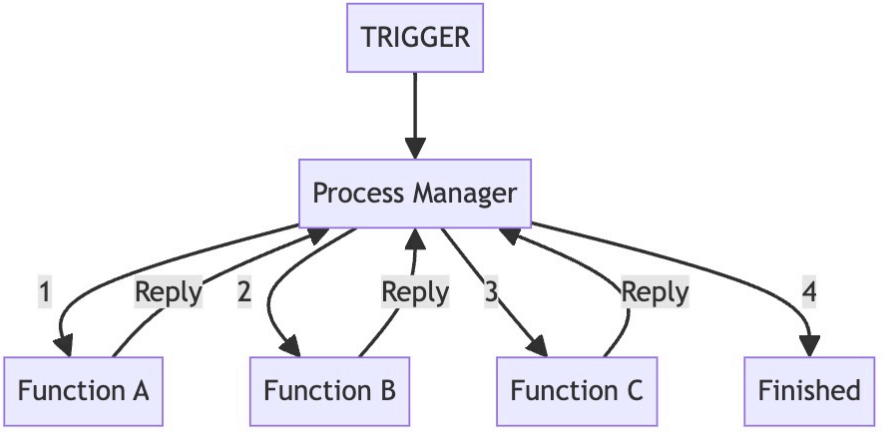
**Personanpassade riskprofiler**: AI-arbetare kan generera personanpassade riskprofiler för varje patient, som belyser de specifika faktorer som bidrar till deras riskpoäng. Dessa profiler kan innehålla insikter om patientens livsstil, genetiska predispositioner, miljöfaktorer och sociala hälsofaktorer. Genom att tillhandahålla en detaljerad nedbrytning av riskfaktorer kan AI-arbetare hjälpa vårdgivare att skräddarsy förebyggande strategier och behandlingsplaner efter individuella patientbehov.

**Kontinuerlig riskövervakning**: AI-arbetare kan kontinuerligt övervaka patientdata och uppdatera riskbedömningar i realtid. När ny information blir tillgänglig, såsom förändringar i vitala tecken, labbresultat eller följsamhet till medicinering, kan AI-arbetare omberäkna riskpoäng och varna vårdgivare om eventuella betydande förändringar. Denna proaktiva övervakning möjliggör snabba interventioner och justeringar av patientens vårdplaner.

**Kliniskt beslutsstöd**: AI-arbetare kan integrera resultat från riskbedömningar i system för kliniskt beslutsstöd, vilket ger vårdgivare evidensbaserade rekommendationer och varningar. Om till exempel en patients riskpoäng för ett särskilt tillstånd överskrider ett visst tröskelvärde kan AI-arbetaren uppmana vårdgivaren att överväga specifika diagnostiska tester, förebyggande åtgärder eller behandlingsalternativ baserat på kliniska riktlinjer och bästa praxis.

Dessa hanterare kan bearbeta stora mängder patientdata, tillämpa avancerad analys och generera användbara insikter för att stödja kliniskt beslutsfattande. Detta leder i slutändan till förbättrade patientresultat, minskade sjukvårdskostnader och förbättrad folkhälsohantering.

# AI-hanteraren som processhanterare




I samband med AI-drivna applikationer kan en hanterare utformas för att fungera som en Processhanterare, enligt beskrivningen i boken "Enterprise Integration Patterns" av Gregor Hohpe. En Processhanterare är en central komponent som upprätthåller processens tillstånd och bestämmer nästa bearbetningssteg baserat på mellanliggande resultat.

När en AI-hanterare fungerar som en Processhanterare tar den emot ett inkommande meddelande som initierar processen, känt som *triggmeddelandet*. AI-hanteraren upprätthåller sedan processens körningsstatus (som ett konversationsutskrift) och hanterar meddelandet genom en serie bearbetningssteg implementerade som verktygshanterare, vilka kan vara sekventiella eller parallella, och anropas efter dess gottfinnande.

Om du använder en AI-modellklass som GPT-4 som vet hur man kör funktioner parallellt kan din hanterare utföra flera steg samtidigt. Jag måste erkänna att jag inte har provat detta själv och min magkänsla säger att resultaten kan variera.

Efter varje enskilt bearbetningssteg återgår kontrollen till AI-hanteraren, vilket gör det möjligt för den att bestämma nästa bearbetningssteg baserat på det aktuella tillståndet och de erhållna resultaten.

## Lagra dina triggmeddelanden

Enligt min erfarenhet är det klokt att implementera ditt triggmeddelande som ett databasbackat objekt. På så sätt identifieras varje processinstans av en unik primärnyckel och ger dig en plats att lagra tillståndet kopplat till körningen, inklusive AI:ns konversationsutskrift.

Här är till exempel en förenklad version av Olympias `AccountChange`-modellklass, som representerar en begäran om att göra en ändring i en användares konto.

```
# == Schema Information
#
# Table name: account_changes
#
#  id          :uuid            not null, primary key
#  description :string
#  state       :string          not null
#  transcript  :jsonb
#  created_at  :datetime        not null
#  updated_at  :datetime        not null
#  account_id  :uuid            not null
#
# Indexes
#
#  index_account_changes_on_account_id  (account_id)
#
# Foreign Keys
```

```
#
#  fk_rails_...  (account_id => accounts.id)
#
class AccountChange < ApplicationRecord
  belongs_to :account

  validates :description, presence: true

  after_commit -> {
    broadcast(:account_change_requested, self)
  }, on: :create

  state_machine initial: :requested do
    event :completed do
      transition all => :complete
    end
    event :failed do
      transition all => :requires_human_review
    end
  end
end
```

Klassen `AccountChange` fungerar som ett triggermeddelande som initierar en process för att hantera kontoändringsförfrågan. Observera hur den sänds ut till Olympias Wisper-baserade publicera-prenumerera-delsystem efter att create-transaktionen har slutförts.

Att lagra triggermeddelandet i databasen på detta sätt ger en bestående dokumentation av varje kontoändringsförfrågan. Varje instans av klassen `AccountChange` tilldelas en unik primärnyckel, vilket möjliggör enkel identifiering och spårning av enskilda förfrågningar. Detta är särskilt användbart för granskningsloggning, eftersom det gör det möjligt för systemet att upprätthålla en historisk dokumentation över alla kontoändringar, inklusive när de begärdes, vilka ändringar som begärdes och det aktuella tillståndet för varje förfrågan.

I det givna exemplet innehåller klassen `AccountChange` fält som `description` för att fånga detaljerna i den begärda ändringen, `state` för att representera förfrågans aktuella

tillstånd (t.ex. requested, complete, requires_human_review), och `transcript` för att lagra AI:ns konversationsutskrift relaterad till förfrågan. Fältet `description` är själva prompten som används för att initiera den första chattkompletteringen med AI:n. Att lagra denna data ger värdefull kontext och möjliggör bättre spårning och analys av kontoändringsprocessen.

Att lagra triggermeddelanden i databasen möjliggör robust felhantering och återhämtning. Om ett fel uppstår under behandlingen av en kontoändringsförfrågan, markerar systemet förfrågan som misslyckad och övergår till ett tillstånd som kräver mänsklig intervention. Detta säkerställer att ingen förfrågan går förlorad eller glöms bort, och att eventuella problem kan hanteras och lösas på lämpligt sätt.

AI-arbetaren, som en Processhanterare, tillhandahåller en central kontrollpunkt och möjliggör kraftfulla funktioner för processrapportering och felsökning. Det är dock viktigt att notera att användning av en AI-arbetare som Processhanterare för varje arbetsflödesscenario i din applikation kan vara överflödigt.

## Integrering av AI-arbetare i din applikationsarkitektur

När man införlivar AI-arbetare i din applikationsarkitektur behöver flera tekniska överväganden hanteras för att säkerställa smidig integration och effektiv kommunikation mellan AI-arbetarna och andra applikationskomponenter. Detta avsnitt behandlar viktiga aspekter av att utforma dessa gränssnitt, hantera dataflöde och hantera livscykeln för AI-arbetare.

## Utforma tydliga gränssnitt och kommunikationsprotokoll

För att underlätta sömlös integration mellan AI-arbetare och andra applikationskomponenter är det avgörande att definiera tydliga gränssnitt och kommunikationsprotokoll. Överväg följande tillvägagångssätt:

**API-baserad integration**: Exponera AI-arbetarnas funktionalitet genom väldefinierade API:er, såsom RESTful-endpoints eller GraphQL-scheman. Detta tillåter andra komponenter att interagera med AI-arbetarna genom standardiserade HTTP-förfrågningar och svar. API-baserad integration tillhandahåller ett tydligt kontrakt mellan AI-arbetarna och de konsumerande komponenterna, vilket gör det enklare att utveckla, testa och underhålla integrationspunkterna.

**Meddelandebaserad kommunikation**: Implementera meddelandebaserade kommunikationsmönster, såsom meddelandeköer eller publicera-prenumerera-system, för att möjliggöra asynkron interaktion mellan AI-arbetare och andra komponenter. Detta tillvägagångssätt frikopplar AI-arbetarna från resten av applikationen, vilket möjliggör bättre skalbarhet, feltolerans och lös koppling. Meddelandebaserad kommunikation är särskilt användbar när bearbetningen som utförs av AI-arbetare är tidskrävande eller resurskrävande, eftersom det tillåter andra delar av applikationen att fortsätta köras utan att behöva vänta på att AI-arbetarna ska slutföra sina uppgifter.

**Händelsestyrd arkitektur**: Designa ditt system kring händelser och triggers som aktiverar AI-arbetare när specifika villkor uppfylls. AI-arbetare kan prenumerera på relevanta händelser och reagera därefter, utföra sina tilldelade uppgifter när händelserna inträffar. Händelsestyrd arkitektur möjliggör realtidsbearbetning och tillåter AI-arbetare att anropas vid behov, vilket minskar onödig resursförbrukning. Detta tillvägagångssätt är väl lämpat för scenarier där AI-arbetare behöver svara på specifika åtgärder eller förändringar i applikationens tillstånd.

## Hantera dataflöde och synkronisering

När man integrerar AI-arbetare i din applikation är det avgörande att säkerställa smidigt dataflöde och upprätthålla datakonsistens mellan AI-arbetarna och andra komponenter. Överväg följande aspekter:

**Dataförberedelse**: Innan data matas in i AI-arbetare kan du behöva utföra olika dataförberedelseuppgifter, såsom rengöring, formatering och/eller transformering av indata. Du vill inte bara säkerställa att AI-arbetarna kan bearbeta effektivt, utan också säkerställa att du inte slösar tokens genom att ge uppmärksamhet åt information som arbetaren kan anse vara meningslös i bästa fall, distraherande i värsta fall. Dataförberedelse kan innefatta uppgifter som att ta bort brus, hantera saknade värden eller konvertera datatyper.

**Datalagring**: Hur kommer du att lagra och bevara data som flödar in och ut ur AI-arbetare? Överväg faktorer som datavolym, sökmönster och skalbarhet. Behöver du bevara AI:ns transkript som en återspegling av dess "tankeprocess" för granskning eller felsökningsändamål, eller räcker det att ha en dokumentation av endast resultaten?

**Datahämtning**: Att hämta data som behövs av workers kan innebära databasfrågor, läsning från filer eller åtkomst till externa API:er. Se till att överväga latens och hur AI-workers kommer att ha tillgång till den mest aktuella datan. Behöver de full tillgång till din databas eller bör du definiera omfattningen av deras åtkomst snävt enligt vad de gör? Hur är det med skalning? Överväg cachningsmekanismer för att förbättra prestanda och minska belastningen på underliggande datakällor.

**Datasynkronisering**: När flera komponenter, inklusive AI-workers, kommer åt och ändrar delad data är det viktigt att implementera korrekta synkroniseringsmekanismer för att upprätthålla datakonsistens. Databaslåsningsstrategier, såsom optimistisk eller pessimistisk låsning, kan hjälpa dig att förhindra konflikter och säkerställa dataintegritet. Implementera transaktionshanteringstekniker för att gruppera relaterade dataoperationer och upprätthålla ACID-egenskaper (Atomicitet, Konsistens, Isolation

och Hållbarhet).

**Felhantering och återhämtning**: Implementera robusta felhanterings- och återhämtningsmekanismer för att hantera datarelaterade problem som kan uppstå under dataflödesprocessen. Hantera undantag på ett elegant sätt och tillhandahåll meningsfulla felmeddelanden för att underlätta felsökning. Implementera återförsöksmekanismer och reservstrategier för att hantera tillfälliga fel eller nätverksstörningar. Definiera tydliga procedurer för dataåterhämtning och återställning vid datakorruption eller dataförlust.

Genom att noggrant utforma och implementera dataflödes- och synkroniseringsmekanismer kan du säkerställa att dina AI-workers har tillgång till korrekt, konsistent och uppdaterad data. Detta gör det möjligt för dem att utföra sina uppgifter effektivt och producera tillförlitliga resultat.

## Hantering av AI-workers livscykel

Utveckla en standardiserad process för initialisering och konfigurering av AI-workers. Jag föredrar ramverk som standardiserar hur du definierar inställningar som modellnamn, systemdirektiv och funktionsdefinitioner. Säkerställ att initialiseringsprocessen är automatiserad och reproducerbar för att underlätta driftsättning och skalning.

Implementera omfattande övervaknings- och loggningsmekanismer för att spåra AI-workers hälsa och prestanda. Samla in mätvärden som resursanvändning, bearbetningstid, felfrekvenser och genomströmning. Använd centraliserade loggsystem som ELK-stack (Elasticsearch, Logstash, Kibana) för att aggregera och analysera loggar från flera AI-workers.

Bygg feltolerans och motståndskraft in i AI-worker-arkitekturen. Implementera felhanterings- och återhämtningsmekanismer för att elegant hantera fel eller undantag. Stora språkmodeller är fortfarande bleeding edge-teknologi; leverantörer tenderar att

ofta gå ner vid oväntade tidpunkter. Använd återförsöksmekanismer och kretsbrytare för att förhindra kaskadfel.

# Sammansättning och orkestrering av AI-workers

En av de viktigaste fördelarna med AI-worker-arkitekturen är dess sammansättningsbarhet, vilket gör att du kan kombinera och orkestrera flera AI-workers för att lösa komplexa problem. Genom att bryta ner en större uppgift i mindre, mer hanterbara deluppgifter, där var och en hanteras av en specialiserad AI-worker, kan du skapa kraftfulla och flexibla system. I detta avsnitt ska vi utforska olika sätt att komponera och orkestrera "en mängd" AI-workers.

## Kedja AI-workers för flerstegsarbetsflöden

I många scenarier kan en komplex uppgift delas upp i en serie sekventiella steg, där utdata från en AI-worker blir indata för nästa. Denna kedning av AI-workers skapar ett flerstegsarbetsflöde eller en pipeline. Varje AI-worker i kedjan fokuserar på en specifik deluppgift, och det slutliga resultatet är resultatet av alla workers gemensamma ansträngningar.

Låt oss titta på ett exempel i samband med en Ruby on Rails-applikation för behandling av användargenererat innehåll. Arbetsflödet involverar följande steg, som förvisso antagligen var och en är för enkla för att vara värda att dela upp på detta sätt i verkliga användningsfall, men de gör exemplet lättare att förstå:

1. **Textrensning**: En AI-worker ansvarig för att ta bort HTML-taggar, konvertera text till gemener och hantera Unicode-normalisering.

2. **Språkidentifiering**: En AI-worker som identifierar språket i den rensade texten.

3. **Känslighetsanalys**: En AI-worker som bestämmer textens sentiment (positiv, negativ eller neutral) baserat på det detekterade språket.

4. **Innehållskategorisering**: En AI-worker som klassificerar texten i fördefinierade kategorier med hjälp av tekniker för naturlig språkbehandling.

Här är ett mycket förenklat exempel på hur du kan kedja dessa AI-workers tillsammans med Ruby:

```ruby
class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call
    language = LanguageDetectionWorker.new(cleaned_text).call
    sentiment = SentimentAnalysisWorker.new(cleaned_text, language).call
    category = CategorizationWorker.new(cleaned_text, language).call

    { cleaned_text:, language:, sentiment:, category: }
  end
end
```

I detta exempel initierar klassen `ContentProcessor` med råtexten och kedjar ihop AI-arbetarna i `process`-metoden. Varje AI-arbetare utför sin specifika uppgift och skickar resultatet vidare till nästa arbetare i kedjan. Det slutliga resultatet är en hash som innehåller den rensade texten, det upptäckta språket, känslighetsanalysen och innehållskategorin.

## Parallell bearbetning för oberoende AI-arbetare

I föregående exempel är AI-arbetarna sammankopplade sekventiellt, där varje arbetare bearbetar texten och skickar resultatet till nästa arbetare. Om du däremot har flera AI-arbetare som kan arbeta oberoende av varandra med samma indata, kan du optimera arbetsflödet genom att bearbeta dem parallellt.

I det givna scenariot kan `LanguageDetectionWorker`, `SentimentAnalysisWorker` och `CategorizationWorker` alla bearbeta den rensade texten oberoende av varandra när textrensningen har utförts av `TextCleanupWorker`. Genom att köra dessa arbetare parallellt kan du potentiellt minska den totala bearbetningstiden och förbättra effektiviteten i ditt arbetsflöde.

För att uppnå parallell bearbetning i Ruby kan du utnyttja samtidighetstekniker som trådar eller asynkron programmering. Här är ett exempel på hur du kan modifiera klassen `ContentProcessor` för att bearbeta de tre sista arbetarna parallellt med hjälp av trådar:

```
require 'concurrent'

class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call

    language_future = Concurrent::Future.execute do
      LanguageDetectionWorker.new(cleaned_text).call
    end

    sentiment_future = Concurrent::Future.execute do
      SentimentAnalysisWorker.new(cleaned_text).call
    end

    category_future = Concurrent::Future.execute do
      CategorizationWorker.new(cleaned_text).call
    end

    language = language_future.value
    sentiment = sentiment_future.value
    category = category_future.value

    { cleaned_text:, language:, sentiment:, category: }
  end
```

```
29 end
```

I denna optimerade version använder vi biblioteket `concurrent-ruby` för att skapa `Concurrent::Future`-objekt för var och en av de oberoende AI-arbetarna. En `Future` representerar en beräkning som kommer att utföras asynkront i en separat tråd.

Efter textrensningssteget skapar vi tre `Future`-objekt: `language_future`, `sentiment_future` och `category_future`. Varje `Future` kör sin motsvarande AI-arbetare (`LanguageDetectionWorker`, `SentimentAnalysisWorker` och `CategorizationWorker`) i en separat tråd och skickar `cleaned_text` som indata.

Genom att anropa metoden `value` på varje `Future` väntar vi på att beräkningen ska slutföras och hämtar resultatet. Metoden `value` blockerar tills resultatet är tillgängligt, vilket säkerställer att alla parallella arbetare har slutfört sin bearbetning innan vi fortsätter.

Slutligen konstruerar vi utdatahashen med den rensade texten och resultaten från de parallella arbetarna, precis som i originalexemplet.

Genom att bearbeta de oberoende AI-arbetarna parallellt kan du potentiellt minska den totala bearbetningstiden jämfört med att köra dem sekventiellt. Denna optimering är särskilt fördelaktig när man hanterar tidskrävande uppgifter eller när man bearbetar stora datamängder.

Det är dock viktigt att notera att de faktiska prestandavinsterna beror på olika faktorer, såsom komplexiteten hos varje arbetare, tillgängliga systemresurser och overhead för trådhantering. Det är alltid en god praxis att genomföra prestandatester och profilera din kod för att bestämma den optimala nivån av parallellism för ditt specifika användningsfall.

Dessutom bör du vara uppmärksam på eventuella delade resurser eller beroenden mellan arbetarna när du implementerar parallell bearbetning. Se till att arbetarna kan fungera oberoende utan konflikter eller kapplopp. Om det finns beroenden eller delade resurser

kan du behöva implementera lämpliga synkroniseringsmekanismer för att upprätthålla dataintegritet och undvika problem som dödlägen eller inkonsekventa resultat.

## Rubys Global Interpreter Lock och asynkron bearbetning

Det är viktigt att förstå konsekvenserna av Rubys Global Interpreter Lock (GIL) när man överväger asynkron trådbaserad bearbetning i Ruby.

GIL är en mekanism i Rubys tolk som säkerställer att endast en tråd kan exekvera Ruby-kod åt gången, även på processorer med flera kärnor. Detta innebär att även om flera trådar kan skapas och hanteras inom en Ruby-process kan endast en tråd aktivt exekvera Ruby-kod vid varje given tidpunkt.

GIL är utformad för att förenkla implementeringen av Ruby-tolken och ge trådsäkerhet för Rubys interna datastrukturer. Den begränsar dock möjligheten till verklig parallell exekvering av Ruby-kod.

När du använder trådar i Ruby, till exempel med biblioteket `concurrent-ruby` eller den inbyggda klassen `Thread`, är trådarna underkastade GIL:s begränsningar. GIL tillåter varje tråd att exekvera Ruby-kod under en kort tidsperiod innan den växlar till en annan tråd, vilket skapar illusionen av samtidig exekvering.

På grund av GIL förblir dock den faktiska exekveringen av Ruby-kod sekventiell. Medan en tråd exekverar Ruby-kod är andra trådar i praktiken pausade i väntan på sin tur att förvärva GIL och exekvera.

Detta innebär att trådbaserad asynkron bearbetning i Ruby är mest effektiv för I/O-bundna uppgifter, såsom att vänta på svar från externa API:er (som stora språkmodeller som hostas av tredje part) eller utföra fil-I/O-operationer. När en tråd stöter på en I/O-operation kan den släppa GIL, vilket låter andra trådar exekvera medan den väntar på att I/O ska slutföras.

Å andra sidan kan GIL begränsa de potentiella prestandavinsterna med trådbaserad parallellism för CPU-bundna uppgifter, såsom intensiva beräkningar eller långvarig AI-arbetarbearbetning. Eftersom endast en tråd kan exekvera Ruby-kod åt gången kanske den totala exekveringstiden inte minskar betydligt jämfört med sekventiell bearbetning.

För att uppnå verklig parallell exekvering för CPU-bundna uppgifter i Ruby kan du behöva utforska alternativa metoder, såsom:

- Använda processbaserad parallellism med flera Ruby-processer, var och en körande på en separat CPU-kärna.
- Utnyttja externa bibliotek eller ramverk som tillhandahåller nativa tillägg eller gränssnitt till språk utan GIL, såsom C eller Rust.,
- Använda distribuerade beräkningsramverk eller meddelandeköer för att distribuera uppgifter över flera maskiner eller processer.

Det är avgörande att ta hänsyn till uppgifternas natur och de begränsningar som GIL medför när man designar och implementerar asynkron bearbetning i Ruby. Medan trådbaserad asynkron bearbetning kan ge fördelar för I/O-bundna uppgifter kanske den inte erbjuder betydande prestandaförbättringar för CPU-bundna uppgifter på grund av GIL:s begränsningar.

## Ensembletekniker för förbättrad noggrannhet

Ensembletekniker innebär att man kombinerar utdata från flera AI-arbetare för att förbättra systemets övergripande noggrannhet eller robusthet. Istället för att förlita sig på en enda AI-arbetare utnyttjar ensembletekniker den kollektiva intelligensen hos flera arbetare för att fatta mer välgrundade beslut.

Ensembler är särskilt viktiga om olika delar av ditt arbetsflöde fungerar bäst med olika AI-modeller, vilket är vanligare än man kan tro. Kraftfulla modeller som GPT-4 är extremt dyra jämfört med mindre kapabla open source-alternativ, och behövs förmodligen inte för varje enskilt arbetssteg i din applikation.

En vanlig ensemble-teknik är majoritetsröstning, där flera AI-arbetare oberoende av varandra bearbetar samma indata, och det slutliga resultatet bestäms av majoritetens konsensus. Detta tillvägagångssätt kan hjälpa till att minska påverkan av enskilda arbetares fel och förbättra systemets övergripande tillförlitlighet.

Låt oss titta på ett exempel där vi har tre AI-arbetare för känslighetsanalys, var och en med olika modeller eller försedda med olika kontext. Vi kan kombinera deras resultat genom majoritetsröstning för att bestämma den slutliga känslighetsförutsägelsen.

```
class SentimentAnalysisEnsemble
  def initialize(text)
    @text = text
  end

  def analyze
    predictions = [
      SentimentAnalysisWorker1.new(@text).analyze,
      SentimentAnalysisWorker2.new(@text).analyze,
      SentimentAnalysisWorker3.new(@text).analyze
    ]

    predictions
      .group_by { |sentiment| sentiment }
      .max_by { |_, votes| votes.size }
      .first

  end
end
```

I detta exempel initierar klassen `SentimentAnalysisEnsemble` med texten och anropar tre olika AI-arbetare för sentimentanalys. Metoden `analyze` samlar in

prediktioner från varje arbetare och fastställer majoritetens sentiment genom att använda metoderna `group_by` och `max_by`. Det slutliga resultatet är det sentiment som får flest röster från ensemblen av arbetare

Ensembler är definitivt ett fall där det kan vara värt din tid att experimentera med parallellism.

## Dynamiskt val och anrop av AI-arbetare

I vissa, om inte de flesta fall, kan valet av specifik AI-arbetare bero på körningsvillkor eller användarinmatningar. Dynamiskt val och anrop av AI-arbetare möjliggör flexibilitet och anpassningsförmåga i systemet.

Du kanske känner dig frestad att försöka få in mycket funktionalitet i en enda AI-arbetare, ge den många funktioner och en stor komplicerad prompt som förklarar hur man ska använda dem. Motstå frestelsen, lita på mig. En av anledningarna till att den approach vi diskuterar i detta kapitel kallas "Mångfald av Arbetare" är för att påminna oss om att det är önskvärt att ha många specialiserade arbetare, där var och en utför sin egen lilla uppgift i tjänst för det större syftet.

Till exempel, tänk på en chatbot-applikation där olika AI-arbetare ansvarar för att hantera olika typer av användarfrågor. Baserat på användarens inmatning väljer applikationen dynamiskt lämplig AI-arbetare för att bearbeta frågan.

```ruby
class ChatbotController < ApplicationController
  def process_query
    query = params[:query]
    query_type = QueryClassifierWorker.new(query).classify

    case query_type
    when 'greeting'
      response = GreetingWorker.new(query).generate_response
    when 'product_inquiry'
      response = ProductInquiryWorker.new(query).generate_response
    when 'order_status'
      response = OrderStatusWorker.new(query).generate_response
    else
      response = DefaultResponseWorker.new(query).generate_response
    end

    render json: { response: response }
  end
end
```

I detta exempel tar `ChatbotController` emot en användarfråga genom åtgärden `process_query`. Först använder den en `QueryClassifierWorker` för att fastställa frågans typ. Baserat på den klassificerade frågetypen väljer styrenheten dynamiskt lämplig AI-arbetare för att generera svaret. Denna dynamiska urval gör det möjligt för chatbotten att hantera olika typer av frågor och dirigera dem till relevanta AI-arbetare.

Eftersom arbetet med `QueryClassifierWorker` är relativt enkelt och inte kräver mycket kontext eller funktionsdefinitioner, kan du förmodligen implementera det med en ultrasnabb liten LLM som `mistralai/mixtral-8x7b-instruct:nitro`. Den har kapacitet som närmar sig GPT-4-nivå på många uppgifter och, när jag skriver detta, kan Groq leverera den med en blixtsnabb genomströmning på 444 tokens per sekund.

# Kombinera traditionell NLP med LLM:er

Medan stora språkmodeller (LLM) har revolutionerat området naturlig språkbehandling (NLP), och erbjuder oöverträffad mångsidighet och prestanda över ett brett spektrum av uppgifter, är de inte alltid den mest effektiva eller kostnadseffektiva lösningen för varje problem. I många fall kan kombinationen av traditionella NLP-tekniker med LLM:er leda till mer optimerade, riktade och ekonomiska sätt att lösa specifika NLP-utmaningar.

Tänk på LLM:er som NLP:s motsvarighet till en schweizisk armékniv - otroligt mångsidiga och kraftfulla, men inte nödvändigtvis det bästa verktyget för varje jobb. Ibland kan ett dedikerat verktyg som en korkskruv eller en konservöppnare vara mer effektivt för en specifik uppgift. På samma sätt kan traditionella NLP-tekniker, såsom dokumentklustring, ämnesidentifiering och klassificering, ofta ge mer riktade och kostnadseffektiva lösningar för vissa aspekter av din NLP-pipeline.

En av de viktigaste fördelarna med traditionella NLP-tekniker är deras beräkningseffektivitet. Dessa metoder, som ofta förlitar sig på enklare statistiska modeller eller regelbaserade tillvägagångssätt, kan bearbeta stora volymer av textdata mycket snabbare och med lägre beräkningsbelastning jämfört med LLM:er. Detta gör dem särskilt lämpade för uppgifter som involverar analys och organisering av stora dokumentsamlingar, såsom klustring av liknande artiklar eller identifiering av nyckelteman inom en textsamling.

Dessutom kan traditionella NLP-tekniker ofta uppnå hög noggrannhet och precision för specifika uppgifter, särskilt när de tränas på domänspecifika dataset. Till exempel kan en vältrimmad dokumentklassificerare som använder traditionella maskininlärningsalgoritmer som stödvektormaskiner (SVM) eller Naiv Bayes exakt kategorisera dokument i fördefinierade kategorier med minimal beräkningskostnad.

LLM:er utmärker sig dock verkligen när det kommer till uppgifter som kräver en djupare förståelse av språk, kontext och resonemang. Deras förmåga att generera sammanhängande och kontextuellt relevant text, besvara frågor och sammanfatta långa

stycken är oöverträffad av traditionella NLP-metoder. LLM:er kan effektivt hantera komplexa språkliga fenomen, såsom tvetydighet, samreferens och idiomatiska uttryck, vilket gör dem ovärderliga för uppgifter som kräver naturlig språkgenerering eller förståelse.

Den verkliga styrkan ligger i att kombinera traditionella NLP-tekniker med LLM:er för att skapa hybrida metoder som utnyttjar styrkorna hos båda. Genom att använda traditionella NLP-metoder för uppgifter som dokumentförbehandling, klustring och ämnesextraktion kan du effektivt organisera och strukturera din textdata. Denna strukturerade information kan sedan matas in i LLM:er för mer avancerade uppgifter, såsom att generera sammanfattningar, besvara frågor eller skapa omfattande rapporter.

Låt oss till exempel överväga ett användningsfall där du vill generera en trendrapport för ett specifikt område baserat på en stor samling enskilda trenddokument. Istället för att enbart förlita dig på LLM:er, vilket kan vara beräkningsmässigt dyrt och tidskrävande för att bearbeta stora textvolymer, kan du använda en hybridmetod:

1. Använd traditionella NLP-tekniker, såsom ämnesmodellering (t.ex. Latent Dirichlet-allokering) eller klustringsalgoritmer (t.ex. K-means), för att gruppera liknande trenddokument tillsammans och identifiera nyckelteman och ämnen inom korpusen.
2. Mata in de klustrade dokumenten och identifierade ämnena i en LLM, och utnyttja dess överlägsna språkförståelse och genereringsförmågor för att skapa sammanhängande och informativa sammanfattningar för varje kluster eller ämne.
3. Använd slutligen LLM:en för att generera en omfattande trendrapport genom att kombinera de enskilda sammanfattningarna, lyfta fram de mest betydande trenderna och tillhandahålla insikter och rekommendationer baserade på den samlade informationen.

Genom att kombinera traditionella NLP-tekniker med LLM:er på detta sätt kan du effektivt bearbeta stora mängder textdata, extrahera meningsfulla insikter och generera högkvalitativa rapporter samtidigt som du optimerar beräkningsresurser och kostnader.

När du ger dig in i dina NLP-projekt är det viktigt att noggrant utvärdera de specifika kraven och begränsningarna för varje uppgift och överväga hur traditionella NLP-metoder och LLM kan utnyttjas tillsammans för att uppnå bästa möjliga resultat. Genom att kombinera effektiviteten och precisionen hos traditionella tekniker med mångsidigheten och styrkan hos LLM kan du skapa högeffektiva och ekonomiska NLP-lösningar som skapar värde för dina användare och intressenter.

# Verktygsanvändning

Inom AI-driven applikationsutveckling har konceptet “verktygsanvändning” eller “function calling” vuxit fram som en kraftfull teknik som gör det möjligt för din LLM att ansluta till externa verktyg, API:er, funktioner, databaser och andra resurser. Detta tillvägagångssätt möjliggör ett rikare utbud av beteenden än att bara generera text, och mer dynamiska interaktioner mellan dina AI-komponenter och resten av din applikations ekosystem. Som vi kommer att undersöka i detta kapitel ger verktygsanvändning dig också möjligheten att få din AI-modell att generera data på strukturerade sätt.

## Vad är verktygsanvändning?

Verktygsanvändning, även känt som function calling, är en teknik som låter utvecklare specificera en lista över funktioner som en LLM kan interagera med

under genereringsprocessen. Dessa verktyg kan variera från enkla hjälpfunktioner till komplexa API:er eller databasfrågor. Genom att ge LLM:en tillgång till dessa verktyg kan utvecklare utöka modellens kapacitet och göra det möjligt för den att utföra uppgifter som kräver extern kunskap eller åtgärder.

**Figur 8. Exempel på en funktionsdefinition för en AI-arbetare som analyserar dokument**

```
FUNCTION = {
  name: "save_analysis",
  description: "Save analysis data for document",
  parameters: {
    type: "object",
    properties: {
      title: {
        type: "string",
        maxLength: 140
      },
      summary: {
        type: "string",
        description: "comprehensive multi-paragraph summary with
                      overview and list of sections (if applicable)"
      },
      tags: {
        type: "array",
        items: {
          type: "string",
          description: "lowercase tags representing main themes
                        of the document"
        }
      }
    },
    "required": %w[title summary tags]
  }
}.freeze
```

Huvudidén bakom verktygsanvändning är att ge språkmodellen förmågan att dynamiskt välja och utföra lämpliga verktyg baserat på användarens input eller den aktuella uppgiften. Istället för att enbart förlita sig på modellens förtränade kunskap, låter verktygsanvändning språkmodellen utnyttja externa resurser för att generera mer

korrekta, relevanta och användbara svar. Verktygsanvändning gör tekniker som RAG (Retrieval Augmented Generation) mycket enklare att implementera än vad de annars skulle vara.

> Observera att om inget annat anges utgår denna bok från att din AI-modell inte har tillgång till några inbyggda serverbaserade verktyg. Alla verktyg som du vill göra tillgängliga för din AI måste uttryckligen deklareras av dig i varje API-förfrågan, med bestämmelser för hur de ska verkställas om och när din AI meddelar att den vill använda det verktyget i sitt svar.

## Möjligheterna med verktygsanvändning

Verktygsanvändning öppnar upp för ett brett spektrum av möjligheter för AI-drivna applikationer. Här är några exempel på vad som kan uppnås med verktygsanvändning:

1. **Chatbottar och virtuella assistenter**: Genom att koppla en språkmodell till externa verktyg kan chatbottar och virtuella assistenter utföra mer komplexa uppgifter, som att hämta information från databaser, utföra API-anrop eller interagera med andra system. Till exempel skulle en chatbot kunna använda ett CRM-verktyg för att ändra status på en affär baserat på användarens förfrågan.
2. **Dataanalys och insikter**: Språkmodeller kan kopplas till dataanalysverktyg eller bibliotek för att utföra avancerade databehandlingsuppgifter. Detta möjliggör för applikationer att generera insikter, genomföra jämförande analyser eller ge datadrivna rekommendationer baserat på användarfrågor.
3. **Sökning och informationshämtning**: Verktygsanvändning låter språkmodeller interagera med sökmotorer, vektordatabaser eller andra informationshämtningssystem. Genom att omvandla användarfrågor till

sökfrågor kan språkmodellen hämta relevant information från flera källor och ge omfattande svar på användarfrågor.

4. **Integration med externa tjänster**: Verktygsanvändning möjliggör sömlös integration mellan AI-drivna applikationer och externa tjänster eller API:er. Till exempel skulle en språkmodell kunna interagera med ett väder-API för att ge väderuppdateringar i realtid eller ett översättnings-API för att generera flerspråkiga svar.

# Arbetsflödet för verktygsanvändning

Arbetsflödet för verktygsanvändning omfattar vanligtvis fyra huvudsteg:

1. Inkludera funktionsdefinitioner i din förfrågankontext
2. Dynamiskt (eller explicit) verktygsval
3. Utförande av funktion(er)
4. Valfri fortsättning av den ursprungliga prompten

Låt oss gå igenom varje steg i detalj.

## Inkludera funktionsdefinitioner i din förfrågankontext

AI:n vet vilka verktyg den har till sitt förfogande eftersom du ger den en lista som en del av din slutförandeförfrågan (vanligtvis definierad som funktioner med en variant av JSON-schema).

Den exakta syntaxen för verktygsdefiniering är modellspecifik.

Så här definierar du en `get_weather`-funktion i Claude 3:

```
{
    "name": "get_weather",
    "description": "Get the current weather in a given location",
    "input_schema": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
        }
    },
    "required": ["location"]
    }
}
```

Och så här definierar du samma funktion för GPT-4, genom att skicka den som värdet för `tools`-parametern:

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA",
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
            },
        },
        "required": ["location"],
```

```
    },
}
```

> Nästan likadant, förutom att det är annorlunda utan någon uppenbar anledning! Så irriterande.

Funktionsdefinitioner anger namn, beskrivning och inparametrar. Inparametrar kan definieras ytterligare med hjälp av attribut som enum för att begränsa de godtagbara värdena, samt genom att ange om en parameter är obligatorisk eller inte.

Förutom själva funktionsdefinitionerna kan du också inkludera instruktioner eller sammanhang för varför och hur funktionen ska användas i systemdirektivet.

Till exempel innehåller mitt webbsökningsverktyg i Olympia detta systemdirektiv, som påminner AI:n om att den har de nämnda verktygen till sitt förfogande:

```
The `google_search` and `realtime_search` functions let you do research
on behalf of the user. In contrast to Google, realtime search is powered
by Perplexity and provides real-time information to curated current events
databases and news sources. Make sure to include URLs in your response so
user can do followup research.
```

Att tillhandahålla detaljerade beskrivningar anses vara den viktigaste faktorn för verktygets prestanda. Dina beskrivningar bör förklara varje detalj om verktyget, inklusive:

- Vad verktyget gör
- När det bör användas (och när det inte bör användas)
- Vad varje parameter betyder och hur den påverkar verktygets beteende
- Alla viktiga förbehåll eller begränsningar som gäller för verktygets implementering

Ju mer kontext du kan ge AI:n om dina verktyg, desto bättre blir den på att avgöra när och hur de ska användas. Till exempel rekommenderar Anthropic minst 3-4 meningar per verktygsbeskrivning för sin Claude 3-serie, mer om verktyget är komplext.

Det är inte nödvändigtvis intuitivt, men beskrivningar anses också vara viktigare än exempel. Även om du kan inkludera exempel på hur ett verktyg används i dess beskrivning eller i den medföljande prompten, är detta mindre viktigt än att ha en tydlig och omfattande förklaring av verktygets syfte och parametrar. Lägg bara till exempel efter att du helt har utvecklat beskrivningen.

Här är ett exempel på en Stripe-liknande API-funktionsspecifikation:

```
{
  "name": "createPayment",
  "description": "Create a new payment request",
  "parameters": {
    "type": "object",
    "properties": {
      "transaction_amount": {
        "type": "number",
        "description": "The amount to be paid"
      },
      "description": {
        "type": "string",
        "description": "A brief description of the payment"
      },
      "payment_method_id": {
        "type": "string",
        "description": "The payment method to be used"
      },
      "payer": {
        "type": "object",
        "description": "Information about the payer, including their name,
                        email, and identification number",
        "properties": {
          "name": {
            "type": "string",
            "description": "The payer's name"
        },
        "email": {
```

```
            "type": "string",
            "description": "The payer's email address"
          },
          "identification": {
            "type": "object",
            "description": "The payer's identification number",
            "properties": {
              "type": {
                "type": "string",
                "description": "Identification document (e.g. CPF, CNPJ)"
              },
              "number": {
                "type": "string",
                "description": "The identification number"
              }
            },
            "required": [ "type", "number" ]
          }
        },
        "required": [ "name", "email", "identification" ]
      }
    }
}
```

I praktiken har vissa modeller problem med att hantera kapslade funktionsspecifikationer och komplexa utdatatyper som arrayer, lexikon etc. Men i teorin borde du kunna tillhandahålla JSON-schemaspecifikationer av godtyckligt djup!

## Dynamiskt verktygsval

När du kör en chattkompletering som inkluderar verktygsdefinitioner väljer LLM-modellen dynamiskt det mest lämpliga verktyget/verktygen att använda och genererar de nödvändiga inparametrarna för varje verktyg.

I praktiken är AI:ns förmåga att anropa *exakt* rätt funktion och *exakt* följa din specifikation för indata varierande. Att sänka temperaturhyperparametern hela vägen

till 0.0 hjälper mycket, men enligt min erfarenhet kommer du fortfarande att stöta på enstaka fel. Dessa misslyckanden inkluderar hallucinerade funktionsnamn, felaktigt namngivna eller helt enkelt saknade inparametrar. Parametrar skickas som JSON, vilket innebär att du ibland kommer att se fel orsakade av trunkerad, felciterad eller på annat sätt trasig JSON.

Mönster för Självläkande data kan hjälpa till att automatiskt fixa funktionsanrop som går sönder på grund av syntaxfel.

## Tvingat (även kallat explicit) verktygsval

Vissa modeller ger dig möjligheten att tvinga fram anrop av en specifik funktion som en parameter i förfrågan. I annat fall är det helt upp till AI:ns omdöme om funktionen ska anropas eller inte.

Förmågan att tvinga fram ett funktionsanrop är avgörande i vissa scenarier där du vill säkerställa att ett specifikt verktyg eller funktion körs, oavsett AI:ns dynamiska urvalsprocess. Det finns flera anledningar till varför denna kapacitet är viktig:

1. **Explicit kontroll**: Du kanske använder AI:n som en *Diskret komponent* eller i ett fördefinierat arbetsflöde som kräver att en specifik funktion körs vid en specifik tidpunkt. Genom att tvinga fram anropet kan du garantera att den önskade funktionen anropas istället för att behöva be AI:n snällt att göra det.
2. **Felsökning och testning**: När man utvecklar och testar AI-drivna applikationer är möjligheten att tvinga fram funktionsanrop ovärderlig för felsökningsändamål. Genom att explicit trigga specifika funktioner kan du isolera och testa enskilda komponenter i din applikation. Detta låter dig verifiera att funktionsimplementeringarna är korrekta, validera inparametrarna och säkerställa att de förväntade resultaten returneras.

3. **Hantering av kantfall**: Det kan finnas kantfall eller exceptionella scenarier där AI:ns dynamiska urvalsprocess kanske inte väljer att köra en funktion som den borde, och du vet detta baserat på externa processer. I sådana fall gör möjligheten att tvinga fram ett funktionsanrop att du kan hantera dessa situationer explicit. Definiera regler eller villkor i din applikationslogik för att avgöra när AI:ns omdöme ska åsidosättas.
4. **Konsekvens och reproducerbarhet**: Om du har en specifik sekvens av funktioner som behöver köras i en särskild ordning, garanterar tvingade anrop att samma sekvens följs varje gång. Detta är särskilt viktigt i applikationer där konsekvens och förutsägbart beteende är kritiskt, som i finansiella system eller vetenskapliga simuleringar.
5. **Prestandaoptimering**: I vissa fall kan tvingade funktionsanrop leda till prestandaoptimering. Om du vet att en specifik funktion krävs för en viss uppgift och att AI:ns dynamiska urvalsprocess kan introducera onödig overhead, kan du kringgå urvalsprocessen och direkt anropa den nödvändiga funktionen. Detta kan hjälpa till att minska latensen och förbättra applikationens övergripande effektivitet.

Sammanfattningsvis ger möjligheten att tvinga fram funktionsanrop i AI-drivna applikationer explicit kontroll, hjälper till med felsökning och testning, hanterar kantfall och säkerställer konsekvens och reproducerbarhet. Det är ett kraftfullt verktyg i din arsenal, men vi behöver diskutera ytterligare en aspekt av denna viktiga funktion.

I många beslutsfattande användningsfall vill vi alltid att modellen ska göra ett funktionsanrop och kanske aldrig vill att modellen ska svara med bara sin interna kunskap. Till exempel, om du routar mellan flera modeller specialiserade på olika uppgifter (flerspråkig indata, matematik, etc.), kan du använda den funktionsanropande modellen för att delegera förfrågningar till en av hjälpmodellerna och aldrig svara självständigt.

### Verktygsvalparameter

GPT-4 och andra språkmodeller som stödjer funktionsanrop ger dig en `tool_choice`-parameter för att kontrollera om verktygsanvändning krävs som en del av en komplettering. Denna parameter har tre möjliga värden:

- `auto` ger AI:n full frihet att använda ett verktyg eller helt enkelt svara
- `required` talar om för AI:n att den *måste* anropa ett verktyg *istället* för att svara, men lämnar valet av verktyg upp till AI:n
- Det tredje alternativet är att ange parametern för `name_of_function` som du vill tvinga fram. Mer om det i nästa avsnitt.

Observera att om du ställer in verktygsval till `required`, kommer modellen att tvingas välja den mest relevanta funktionen att anropa bland de som tillhandahålls, även om ingen egentligen passar prompten. Vid tiden för publicering känner jag inte till någon modell som kommer att returnera ett tomt `tool_calls`-svar eller på något annat sätt låta dig veta att den inte hittade en lämplig funktion att anropa.

## Tvinga fram ett funktionsanrop för att få strukturerad utdata

Möjligheten att tvinga fram ett funktionsanrop ger dig ett sätt att få strukturerad data från en chattkonversation istället för att behöva extrahera den själv från dess svar i klartext.

> Varför är det så viktigt att tvinga fram funktioner för att få strukturerad utdata? Helt enkelt eftersom extrahering av strukturerad data från LLM-utdata är ett stort huvudbry. Du kan göra ditt liv lite enklare genom att be om data i XML, men då måste du analysera XML. Och vad gör du när den XML:en saknas eftersom din AI svarade: "Jag beklagar, men jag kan inte generera den data du begärde eftersom bla, bla, bla..."

När du använder verktyg på detta sätt:

- Du bör förmodligen definiera ett enda verktyg i din förfrågan
- Kom ihåg att tvinga fram användningen av dess funktion med hjälp av parametern `tool_choice`
- Kom ihåg att modellen kommer att skicka indata till verktyget, så namnet på verktyget och beskrivningen ska vara från modellens perspektiv, inte ditt.

Den sista punkten förtjänar ett exempel för tydlighet. Låt säga att du ber AI:n att göra sentimentanalys på användartext. Namnet på funktionen skulle inte vara `analyze_sentiment`, utan snarare något som `save_sentiment_analysis`. Det är AI:n som utför sentimentanalysen, *inte verktyget*. Allt som verktyget gör (från AI:ns perspektiv) är att spara resultaten av analysen.

Här är ett exempel på hur man använder Claude 3 för att registrera en sammanfattning av en bild i välstrukturerad JSON, denna gång från kommandoraden med hjälp av `curl`:

```
curl https://api.anthropic.com/v1/messages \
     --header "content-type: application/json" \
     --header "x-api-key: $ANTHROPIC_API_KEY" \
     --header "anthropic-version: 2023-06-01" \
     --header "anthropic-beta: tools-2024-04-04" \
     --data \
'{
    "model": "claude-3-sonnet-20240229",
    "max_tokens": 1024,
    "tools": [{
        "name": "record_summary",
        "description": "Record summary of image into well-structured JSON.",
        "input_schema": {
            "type": "object",
            "properties": {
                "key_colors": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "r": {
                                "type": "number",
                                "description": "red value [0.0, 1.0]"
                            },
                            "g": {
                                "type": "number",
                                "description": "green value [0.0, 1.0]"
                            },
                            "b": {
                                "type": "number",
                                "description": "blue value [0.0, 1.0]"
                            },
                            "name": {
                                "type": "string",
                                "description": "Human-readable color name
                                                in snake_case, e.g.
                                                \"olive_green\"or
                                                \"turquoise\""
                            }
                        },
                        "required": [ "r", "g", "b", "name" ]
                    },
```

```
                    "description": "Key colors in the image. Four or less."
                },
                "description": {
                    "type": "string",
                    "description": "Image description. 1-2 sentences max."
                },
                "estimated_year": {
                    "type": "integer",
                    "description": "Estimated year that the image was taken,
                                    if is it a photo. Only set this if the
                                    image appears to be non-fictional.
                                    Rough estimates are okay!"
                }
            },
            "required": [ "key_colors", "description" ]
        }
    }],
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "'$IMAGE_MEDIA_TYPE'",
                        "data": "'$IMAGE_BASE64'"
                    }
                },
                {
                    "type": "text",
                    "text": "Use `record_summary` to describe this image."
                }
            ]
        }
    ]
}'
```

I det givna exemplet använder vi Claude 3-modellen från Anthropic för att generera en strukturerad JSON-sammanfattning av en bild. Så här fungerar det:

1. Vi definierar ett enskilt verktyg som heter `record_summary` i begärans `tools`-array. Detta verktyg ansvarar för att registrera en sammanfattning av bilden i välstrukturerad JSON.
2. Verktyget `record_summary` har ett `input_schema` som specificerar den förväntade strukturen för JSON-utdata. Det definierar tre egenskaper:

    - `key_colors`: En array av objekt som representerar huvudfärgerna i bilden. Varje färgobjekt har egenskaper för röda, gröna och blå värden (från 0.0 till 1.0) och ett läsbart färgnamn i snake_case-format.
    - `description`: En strängegenskap för en kort beskrivning av bilden, begränsad till 1-2 meningar.
    - `estimated_year`: En valfri heltalsegenskap för det uppskattade året då bilden togs, om det verkar vara ett icke-fiktivt foto.

3. I `messages`-arrayen tillhandahåller vi bilddata som en base64-kodad sträng tillsammans med mediatypen. Detta gör det möjligt för modellen att behandla bilden som en del av indata.
4. Vi uppmanar också Claude att använda verktyget `record_summary` för att beskriva bilden.
5. När begäran skickas till Claude 3-modellen analyserar den bilden och genererar en JSON-sammanfattning baserad på det specificerade `input_schema`. Modellen extraherar huvudfärgerna, ger en kort beskrivning och uppskattar året då bilden togs (om tillämpligt).
6. Den genererade JSON-sammanfattningen skickas som parametrar till verktyget `record_summary`, vilket ger en strukturerad representation av bildens huvudegenskaper.

Genom att använda verktyget `record_summary` med ett väldefinierat `input_schema` kan vi få en strukturerad JSON-sammanfattning av en bild utan att förlita oss på extrahering av oformaterad text. Detta tillvägagångssätt säkerställer att utdata följer

ett konsekvent format och enkelt kan analyseras och behandlas av applikationens nedströmskomponenter.

Förmågan att framtvinga ett funktionsanrop och specificera den förväntade utdatastrukturen är en kraftfull funktion vid verktygsanvändning i AI-drivna applikationer. Det gör det möjligt för utvecklare att ha mer kontroll över genererade utdata och förenklar integrationen av AI-genererad data i applikationens arbetsflöde.

## Exekvering av funktion(er)

Du har definierat funktioner och gett instruktioner till din AI, som beslutade att den skulle anropa en av dina funktioner. Nu är det dags för din applikationskod eller bibliotek, om du använder en Ruby-gem som `raix-rails` att skicka funktionsanropet och dess parametrar till motsvarande implementation *i din applikationskod*.

Din applikationskod avgör vad som ska göras med funktionsexekveringens resultat. Kanske handlar det om en enda kodrad i en lambda, eller kanske det innebär att anropa ett externt API. Kanske innebär det att anropa en annan AI-komponent, eller kanske det involverar hundratals eller till och med tusentals kodrader i resten av ditt system. Det är helt upp till dig.

Ibland är funktionsanropet slutet på operationen, men om resultaten representerar information i en tankekedja som ska fortsättas av AI:n, då måste din applikationskod infoga exekveringsresultaten i chatttranskriptet och låta AI:n fortsätta bearbetningen.

Till exempel, här är en Raix-funktionsdeklaration som används av Olympias `AccountManager` för att kommunicera med våra klienter som en del av en Intelligent Arbetsflödesorkestration för kundservice.

```ruby
class AccountManager
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  # lots of other functions...

  function :notify_account_owner,
    "Don't share UUID. Mention dollars if subscription changed",
    message: { type: "string" } do |arguments|
      account.owner.freeform_notify(
        subject: "Account Change Notification",
        message: arguments[:message]
      )
      "Notified account owner"
    end
```

Det kanske inte är omedelbart uppenbart vad som händer här, så jag ska bryta ner det.

1. Klassen `AccountManager` definierar många funktioner relaterade till kontohantering. Den kan ändra din plan, lägga till och ta bort teammedlemmar, bland annat.
2. Dess instruktioner på toppnivå talar om för `AccountManager` att den ska meddela kontoägaren om resultaten av kontoändringsförfrågan, genom att använda funktionen `notify_account_owner`.
3. Den koncisa definitionen av funktionen inkluderar dess:

- namn
- beskrivning
- parametrar `message: { type: "string" }`
- ett block att exekvera när funktionen anropas

Efter att ha uppdaterat transkriptet med resultaten från funktionsblocket, anropas metoden `chat_completion` igen. Denna metod ansvarar för att skicka tillbaka det uppdaterade konversationstranskriptet till AI-modellen för vidare bearbetning. Vi kallar denna process för en *konversationsloop*.

När AI-modellen tar emot en ny chattgenereringsförfrågan med ett uppdaterat transkript, har den tillgång till resultaten från den tidigare exekverade funktionen. Den kan analysera dessa resultat, införliva dem i sin beslutsprocess och generera nästa svar eller åtgärd baserat på samtalets kumulativa kontext. Den kan välja att exekvera ytterligare funktioner baserat på den uppdaterade kontexten, eller generera ett slutgiltigt svar på den ursprungliga prompten om den bedömer att inga ytterligare funktionsanrop behövs.

## Valfri fortsättning av den ursprungliga prompten

När du skickar tillbaka verktygsresultaten till LLM och fortsätter bearbetningen av den ursprungliga prompten, använder AI:n dessa resultat för att antingen anropa ytterligare funktioner eller generera ett slutgiltigt textsvar.

Vissa modeller som Coheres Command-R kan citera de specifika verktygen de använt i sina svar, vilket ger ytterligare transparens och spårbarhet.

Beroende på vilken modell som används kommer resultaten av funktionsanropet att finnas i transkriptmeddelanden som har sin egen speciella roll eller återspeglas i någon annan syntax. Men den viktiga delen är att dessa data finns i transkriptet, så att AI:n kan ta hänsyn till dem när den bestämmer vad som ska göras härnäst.

Ett vanligt (och potentiellt dyrt) felförhållande är att glömma att lägga till funktionsresultaten i transkriptet innan man fortsätter chatten. Som ett resultat kommer AI:n att få i princip samma prompt som innan den anropade funktionen första gången. Med andra ord, så vitt AI:n vet har den inte anropat funktionen än. Så den anropar den igen. Och igen. Och igen, för evigt tills du avbryter den. Hoppas att din kontext inte var för stor och din modell inte för dyr!

# Bästa praxis för verktygsanvändning

För att få ut det mesta av verktygsanvändning, överväg följande bästa praxis.

## Beskrivande definitioner

Tillhandahåll tydliga och beskrivande namn och beskrivningar för varje verktyg och dess inparametrar. Detta hjälper LLM att bättre förstå syftet och kapaciteten hos varje verktyg.

> Jag kan berätta från erfarenhet att den allmänna visdomen som säger att "namngivning är svårt" gäller här; jag har sett dramatiskt olika resultat från LLM:er bara genom att ändra funktionsnamn eller formuleringar av beskrivningar. Ibland förbättrar till och med borttagning av beskrivningar prestandan.

## Bearbetning av verktygsresultat

När verktygsresultat skickas tillbaka till LLM:en, se till att de är välstrukturerade och omfattande. Använd meningsfulla nycklar och värden för att representera utdata från varje verktyg. Experimentera med olika format och se vilket som fungerar bäst, från JSON till vanlig text.

Result Interpreter hanterar denna utmaning genom att använda AI för att analysera resultaten och tillhandahålla användarvänliga förklaringar, sammanfattningar eller viktiga slutsatser.

## Felhantering

Implementera robusta felhanteringsmekanismer för att hantera fall där LLM:en kan generera ogiltiga eller icke-stödda inparametrar för verktygsanrop. Hantera och återhämta dig smidigt från eventuella fel som kan uppstå under verktygsexekvering.

En mycket trevlig egenskap hos AI:n är att den förstår felmeddelanden! Vilket betyder att om du arbetar i ett snabbt och enkelt tankesätt kan du helt enkelt fånga upp eventuella undantag som genereras i implementeringen av ett verktyg och skicka tillbaka dem till AI:n så att den vet vad som hände!

Till exempel, här är en nedbantad version av implementeringen av Google-sökning i Olympia:

```
  def google_search(conversation, params)
    conversation.update_cstatus("Searching Google...")
    query = params[:query]
    search = GoogleSearch.new(query).get_hash

    conversation.update_cstatus("Summarizing results...")
    SummarizeKnowledgeGraph.new.perform(conversation, search.to_json)
  rescue StandardError => e
    Honeybadger.notify(e)
    { error: e.message }.inspect
  end
```

Google-sökningar i Olympia är en tvåstegsprocess. Först gör du sökningen, sedan sammanfattar du resultaten. Om det uppstår ett fel, oavsett vad det är, paketeras felmeddelandet och skickas tillbaka till AI:n. Denna teknik är grunden för i stort sett alla mönster inom *Intelligent Felhantering*

Till exempel, låt säga att `GoogleSearch` API-anropet misslyckas på grund av ett 503 Tjänsten Otillgänglig-undantag. Det bubblar upp till den översta felhanteringen, och beskrivningen av felet skickas tillbaka till AI:n som resultatet av funktionsanropet. Istället för att bara ge användaren en tom skärm eller ett tekniskt fel, säger AI:n något i

stil med “Jag beklagar, men jag kan inte komma åt mina Google-sökfunktioner just nu. Jag kan försöka igen senare om du vill.”

Detta kan verka som ett smart knep, men tänk på en annan typ av fel, ett där AI:n anropade ett externt API och hade direkt kontroll över parametrarna som skickades till API:t. Kanske gjorde den ett misstag i hur den genererade dessa parametrar? Förutsatt att felmeddelandet från det externa API:t är tillräckligt detaljerat, betyder det att när felmeddelandet skickas tillbaka till den anropande AI:n kan den ompröva dessa parametrar och försöka igen. Automatiskt. Oavsett vad felet var.

Tänk nu på vad som skulle krävas för att replikera den typen av robust felhantering i *normal* kod. Det är praktiskt taget omöjligt.

## Iterativ Förfining

Om LLM:en inte rekommenderar lämpliga verktyg eller genererar suboptimala svar, iterera på verktygsdefinitionerna, beskrivningarna och inparametrarna. Fortsätt att förfina och förbättra verktygsuppsättningen baserat på det observerade beteendet och önskade resultat.

1. Börja med enkla verktygsdefinitioner: Börja med att definiera verktyg med tydliga och koncisa namn, beskrivningar och inparametrar. Undvik att överkompliera verktygsuppsättningen initialt och fokusera på kärnfunktionaliteten. Till exempel, om du vill spara resultaten från känslighetsanalys, börja med en grundläggande definition som:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze user-provided text and generate sentiment score",
  "parameters": {
    "type": "object",
    "properties": {
      "score": {
        "type": "float",
        "description": "sentiment score from -1 (negative) to 1 (positive)"
      }
    },
    "required": ["score"]
  }
}
```

2. Testa och observera: När du har de första verktygsdefinitionerna på plats, testa dem med olika prompts och observera hur språkmodellen interagerar med verktyget. Var uppmärksam på kvaliteten och relevansen i de genererade svaren. Om språkmodellen genererar suboptimala svar är det dags att förfina verktygsdefinitionerna.
3. Förfina beskrivningar: Om språkmodellen missförstår syftet med ett verktyg, försök att förfina verktygets beskrivning. Ge mer kontext, exempel eller förtydliganden för att vägleda språkmodellen i att använda verktyget effektivt. Du kan till exempel uppdatera beskrivningen av känslighetsanalysverktyget för att mer specifikt adressera den *känslomässiga tonen* i texten som analyseras:

```
{
  "name": "save_sentiment_score",
  "description": "Determine the overall emotional tone of a piece of text,
   such as customer reviews, social media posts, or feedback comments.",
  ...
}
```

4. Justera inmatningsparametrar: Om LLM:en genererar ogiltiga eller irrelevanta inmatningsparametrar för ett verktyg, överväg att justera

parameterdefinitionerna. Lägg till mer specifika begränsningar, valideringsregler eller exempel för att förtydliga det förväntade inmatningsformatet.

5. Iterera baserat på återkoppling: Övervaka kontinuerligt verktygens prestanda och samla in återkoppling från användare eller intressenter. Använd denna återkoppling för att identifiera förbättringsområden och gör iterativa förfiningar av verktygsdefinitionerna. Om användare till exempel rapporterar att analysen inte hanterar sarkasm väl, kan du lägga till en anteckning i beskrivningen:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze the sentiment of a given text and return a sentiment
   score between -1 (negative) and 1 (positive). Note: Sarcasm should be
   considered negative.",
  ...
}
```

Genom att iterativt förfina dina verktygsdefintioner baserat på observerat beteende och återkoppling kan du gradvis förbättra prestandan och effektiviteten i din AI-drivna applikation. Kom ihåg att hålla verktygsdefintionerna tydliga, koncisa och fokuserade på den specifika uppgiften. Testa och validera verktygsinteraktionerna regelbundet för att säkerställa att de överensstämmer med dina önskade resultat.

## Komponera och Kedja Verktyg

En av de mest kraftfulla aspekterna av verktygsanvändning som hittills bara har antytts är förmågan att komponera och kedja flera verktyg tillsammans för att utföra komplexa uppgifter. Genom att noggrant utforma dina verktygsdefintioner och deras in- och utdataformat kan du skapa återanvändbara byggstenar som kan kombineras på olika sätt.

Låt oss titta på ett exempel där du bygger ett dataanalysflöde för din AI-drivna applikation. Du kan ha följande verktyg:

1. **`DataRetrieval`**: Ett verktyg som hämtar data från en databas eller API baserat på specificerade kriterier.
2. **`DataProcessing`**: Ett verktyg som utför beräkningar, transformationer eller aggregeringar på den hämtade datan.
3. **`DataVisualization`**: Ett verktyg som presenterar den bearbetade datan i ett användarvänligt format, som diagram eller grafer.

Genom att kedja dessa verktyg tillsammans kan du skapa ett kraftfullt arbetsflöde som hämtar relevant data, bearbetar den och presenterar resultaten på ett meningsfullt sätt. Här är hur verktygsanvändningsflödet kan se ut:

1. Språkmodellen tar emot en användarfråga som ber om insikter om försäljningsdata för en specifik produktkategori.
2. Språkmodellen väljer `DataRetrieval`-verktyget och genererar lämpliga inparametrar för att hämta relevant försäljningsdata från databasen.
3. Den hämtade datan "skickas" till `DataProcessing`-verktyget, som beräknar mätvärden som total intäkt, genomsnittligt försäljningspris och tillväxttakt.
4. Den bearbetade datan bearbetas sedan av `DataVisualization`-verktyget, som skapar ett visuellt tilltalande diagram eller graf för att representera insikterna, och skickar tillbaka diagrammets URL till språkmodellen.
5. Slutligen genererar språkmodellen ett formaterat svar på användarfrågan med hjälp av markdown, som införlivar den visualiserade datan och ger en sammanfattning av de viktigaste resultaten.

Genom att komponera dessa verktyg tillsammans kan du skapa ett sömlöst dataanalysarbetsflöde som enkelt kan integreras i din applikation. Det fina med denna approach är att varje verktyg kan utvecklas och testas självständigt och sedan kombineras på olika sätt för att lösa olika problem.

För att möjliggöra smidig komposition och kedning av verktyg är det viktigt att definiera tydliga in- och utdataformat för varje verktyg.

Till exempel kan `DataRetrieval`-verktyget acceptera parametrar som databasanslutningsdetaljer, tabellnamn och sökvillkor, och returnera resultatet som ett strukturerat JSON-objekt. `DataProcessing`-verktyget kan sedan förvänta sig detta JSON-objekt som indata och producera ett transformerat JSON-objekt som utdata. Genom att standardisera dataflödet mellan verktyg kan du säkerställa kompatibilitet och återanvändbarhet.

När du designar ditt verktygsekosystem, tänk på hur olika verktyg kan kombineras för att hantera vanliga användningsfall i din applikation. Överväg att skapa verktyg på hög nivå som kapslar in vanliga arbetsflöden eller affärslogik, vilket gör det enklare för språkmodellen att välja och använda dem effektivt.

Kom ihåg att styrkan i verktygsanvändning ligger i flexibiliteten och modulariteten den erbjuder. Genom att bryta ner komplexa uppgifter i mindre, återanvändbara verktyg kan du skapa en robust och anpassningsbar AI-driven applikation som kan hantera ett brett spektrum av utmaningar.

## Framtida Riktningar

När området för AI-driven applikationsutveckling utvecklas kan vi förvänta oss ytterligare framsteg inom verktygsanvändningsmöjligheter. Några potentiella framtida riktningar inkluderar:

1. **Flerstegsverktygsanvändning**: Språkmodeller kan komma att kunna avgöra hur många gånger de behöver använda verktyg för att generera ett tillfredsställande svar. Detta kan innebära flera omgångar av verktygsval och utförande baserat på mellanliggande resultat.
2. **Fördefinierade Verktyg**: AI-plattformar kan komma att tillhandahålla en uppsättning fördefinierade verktyg som utvecklare kan utnyttja direkt, som Python-tolkar, webbsökverktyg eller vanliga verktygsfunktioner.

3. **Sömlös Integration**: När verktygsanvändning blir mer utbredd kan vi förvänta oss bättre integration mellan AI-plattformar och populära utvecklingsramverk, vilket gör det enklare för utvecklare att införliva verktygsanvändning i sina applikationer.

Verktygsanvändning är en kraftfull teknik som gör det möjligt för utvecklare att utnyttja den fulla potentialen hos språkmodeller i AI-drivna applikationer. Genom att koppla språkmodeller till externa verktyg och resurser kan du skapa mer dynamiska, intelligenta och kontextmedvetna system som kan anpassa sig till användarnas behov och tillhandahålla värdefulla insikter och åtgärder.

Medan verktygsanvändning erbjuder enorma möjligheter är det viktigt att vara medveten om potentiella utmaningar och överväganden. En viktig aspekt är att hantera komplexiteten i verktygsinteraktioner och säkerställa stabilitet och tillförlitlighet i hela systemet. Du behöver hantera scenarier där verktygsanrop kan misslyckas, returnera oväntade resultat eller ha prestandakonsekvenser. Dessutom bör du överväga säkerhets- och åtkomstkontrollåtgärder för att förhindra obehörig eller skadlig användning av verktyg. Korrekt felhantering, loggning och övervakningsmekanismer är avgörande för att upprätthålla integriteten och prestandan i din AI-drivna applikation.

När du utforskar möjligheterna med verktygsanvändning i dina egna projekt, kom ihåg att börja med tydliga mål, utforma välstrukturerade verktygsdefinitioner och iterera baserat på återkoppling och resultat. Med rätt approach och tankesätt kan verktygsanvändning låsa upp nya nivåer av innovation och värde i dina AI-drivna applikationer

# Strömbearbetning

Strömning av data över HTTP, även känt som serverinitierade händelser (SSE), är en mekanism där servern kontinuerligt skickar data till klienten när den blir tillgänglig, utan att klienten uttryckligen behöver begära den. Eftersom AI:ns svar genereras stegvis är det logiskt att erbjuda en responsiv användarupplevelse genom att visa AI:ns utdata medan den genereras. Och faktiskt erbjuder alla AI-leverantörers API:er som jag känner till strömmande svar som ett alternativ i deras slutförande-endpoints.

Anledningen till att detta kapitel dyker upp här i boken, direkt efter Använda verktyg, är på grund av hur kraftfullt det kan vara att kombinera användningen av verktyg med AI:ns direktsvar till användarna. Detta möjliggör dynamiska och interaktiva upplevelser där AI:n kan bearbeta användarinput, utnyttja olika verktyg och funktioner efter eget gottfinnande och sedan ge realtidssvar.

För att uppnå denna sömlösa interaktion behöver du skriva strömhanterare som kan hantera både AI-initierade verktygsanrop och vanlig textutdata till slutanvändaren.

Behovet av att loopa efter bearbetning av en verktygsfunktion tillför en intressant utmaning till uppgiften.

## Implementering av en ReplyStream

För att demonstrera hur strömbearbetning kan implementeras kommer detta kapitel att göra en djupdykning i en förenklad version av klassen `ReplyStream` som används i Olympia. Instanser av denna klass kan skickas som `stream`-parametern i AI-klientbibliotek som ruby-openai och openrouter

Här är hur jag använder `ReplyStream` i Olympias `PromptSubscriber`, som lyssnar via Wisper efter skapandet av nya användarmeddelanden.

```
1  class PromptSubscriber
2    include Raix::ChatCompletion
3    include Raix::PromptDeclarations
4
5    # many other declarations omitted...
6
7    prompt text: -> { user_message.content },
8           stream: -> { ReplyStream.new(self) },
9           until: -> { bot_message.complete? }
10
11   def message_created(message) # invoked by Wisper
12     return unless message.role.user? && message.content?
13
14     # rest of the implementation omitted...
```

Förutom en `context`-referens till promptprenumeranten som instansierade den, har klassen `ReplyStream` även instansvariabler för att lagra en buffert av mottagen data, samt vektorer för att hålla reda på funktionsnamn och argument som anropas under strömbehandlingen.

```
class ReplyStream
  attr_accessor :buffer, :f_name, :f_arguments, :context

  delegate :bot_message, :dispatch, to: :context

  def initialize(context)
    self.context = context
    self.buffer = []
    self.f_name = []
    self.f_arguments = []
  end

  def call(chunk, bytesize = nil)
    # ...
  end

  # ...
end
```

Metoden `initialize` ställer in det initiala tillståndet för `ReplyStream`-instansen genom att initiera bufferten, kontexten och andra variabler.

Metoden `call` är den huvudsakliga ingångspunkten för att bearbeta strömmande data. Den tar emot ett `chunk` av data (representerat som ett hash) och en valfri `bytesize`-parameter, som i vårt exempel är oanvänd. Inuti denna metod använder klassen mönstermatchning för att hantera olika scenarier baserat på strukturen hos det mottagna datablocket.

Att anropa `deep_symbolize_keys` på datablocket hjälper till att göra mönstermatchningen mer elegant genom att låta oss arbeta med symboler istället för strängar.

```
def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys

  in { # match function name
    choices: [
      {
        delta: {
          tool_calls: [
            { index: index, function: {name: name} }
          ]
        }
      }
    ] }

    f_name[index] = name
```

Det första mönstret vi matchar efter är ett verktygsanrop tillsammans med dess tillhörande funktionsnamn. Om vi upptäcker ett sådant, lägger vi in det i `f_name`-arrayen. Vi lagrar funktionsnamn i en indexerad array eftersom modellen kan hantera parallella funktionsanrop och skicka mer än en funktion att exekvera samtidigt.

> Parallella funktionsanrop är en AI-modells förmåga att utföra flera funktionsanrop tillsammans, vilket gör det möjligt att lösa effekterna och resultaten av dessa funktionsanrop parallellt. Detta är särskilt användbart om funktioner tar lång tid, och minskar antalet rundturer med API:et, vilket i sin tur kan spara en betydande mängd tokenförbrukning.

Därefter behöver vi matcha efter argumenten som hör till funktionsanropen.

```
  in { # match arguments
    choices: [
      {
        delta: {
          tool_calls: [
            {
              index: index, function: {arguments: argument }
            }
          ]
        }
      }
    ]}

    f_arguments[index] ||= "" # initialize if not already
    f_arguments[index] << argument
```

På samma sätt som vi hanterade funktionsnamnen, lägger vi undan argumenten i en indexerad array.

Härnäst tittar vi efter vanliga användarmeddelanden, som kommer att anlända från servern en token i taget och tilldelas `new_content`-variabeln. Vi måste också hålla koll på `finish_reason`. Den kommer att vara `nil` fram tills den sista delen av utdatasekvensen.

```
  in {
    choices: [
      { delta: {content: new_content}, finish_reason: finish_reason }
    ]}

    # you could transmit every chunk to the user here...
    buffer << new_content.to_s

    if finish_reason.present?
      finalize
    elsif new_content.to_s.match?(/\n\n/)
      send_to_client # ...or buffer and transmit once per paragraph
    end
```

Viktigt är att vi lägger till ett mönstermatchningsuttryck för att hantera felmeddelanden

som skickas av AI-modellleverantören. I lokala utvecklingsmiljöer kastar vi ett undantag, men i produktion loggar vi felet och avslutar.

```
  in { error: { message: } }
    if Rails.env.local?
      raise message
    else
      Honeybadger.notify("AI Error: #{message}")
      finalize
    end
```

Den sista else-satsen i case kommer att köras om inget av de tidigare mönstren matchade. Det är bara en säkerhetsåtgärd så att vi upptäcker om AI-modellen börjar skicka oss oigenkännliga segment.

```
  else
    Honeybadger.notify("Unrecognized Chunk: #{chunk}")
  end
end
```

Metoden `send_to_client` ansvarar för att skicka det buffrade innehållet till klienten. Den kontrollerar att bufferten inte är tom, uppdaterar botmeddelandets innehåll, renderar botmeddelandet och sparar innehållet i databasen för att säkerställa datapersistens.

```ruby
def send_to_client
  # no need to process pure whitespace
  return if buffer.join.squish.blank?

  # set the buffer content on the bot message
  content = buffer.join
  bot_message.content = content

  # save to database so that we never lose data
  # even if the stream doesn't terminate correctly
  bot_message.update_column(:content, content)

  # update content via websocket
  ConversationRenderer.update(bot_message)
end
```

Metoden `finalize` anropas när strömbehandlingen är slutförd. Den skickar funktionsanropen om några togs emot under strömningen, uppdaterar botmeddelandet med det slutliga innehållet och annan relevant information, samt återställer funktionsanropshistoriken

```ruby
def finalize
  if f_name.any?
    f_name.each_with_index do |name, index|
      # takes care of calling the function wherever it's implemented
      dispatch(name:, arguments: JSON.parse(f_arguments[index]))
    end

    # reset the function call history
    f_name.clear
    f_arguments.clear
  else
    content = buffer.join.presence
    bot_message.update!(content:, complete: true)
    ConversationRenderer.update(bot_message)
  end
end
```

Om modellen bestämmer sig för att anropa en funktion måste du “dirigera” det funktionsanropet (namn och argument) på ett sådant sätt att det utförs

och att `function_call`- och `function_result`-meddelanden läggs till i konversationsutskriften

Enligt min erfarenhet är det bättre att hantera skapandet av funktionsmeddelanden på ett enda ställe i din kodbas, istället för att förlita sig på verktygsimplementeringarna. Det är inte bara mer städat, utan har också en mycket viktig praktisk anledning: om AI-modellen anropar en funktion och inte ser resulterande anrops- och resultatmeddelanden i utskriften när du loopar, kommer den att *anropa samma funktion igen*. Potentiellt för evigt. Kom ihåg att AI:n är helt tillståndslös, så om du inte speglar dessa funktionsanrop tillbaka till den har de aldrig hänt.

```ruby
# PromptSubscriber#dispatch

def dispatch(name:, arguments:)
  # adds a function_call message to the conversation transcript
  # plus dispatches to tool and returns result
  conversation.function_call!(name, arguments).then do |result|
    # add function result message to the transcript
    conversation.function_result!(name, result)
  end
end
```

Att rensa funktionsanropshistoriken efter körning är lika viktigt som att se till att anropet och resultaten hamnar i ditt transkript, så att du inte fortsätter att anropa samma funktioner om och om igen varje gång du loopar.

## “Konversationsloopen”

I klassen `PromptSubscriber` använder vi `prompt`-metoden från modulen `PromptDeclarations` för att definiera beteendet för konversationsloopen. Parametern `until` är satt till `-> { bot_message.complete? }`, vilket betyder att loopen kommer att fortsätta tills `bot_message` är markerat som färdigt.

```
1  prompt text: -> { user_message.content },
2         stream: -> { ReplyStream.new(self) },
3         until: -> { bot_message.complete? }
```

Men när markeras `bot_message` som slutförd? Om du har glömt det, se tillbaka till rad 13 i `finalize`-metoden.

Låt oss gå igenom hela strömbehandlingslogiken.

1. `PromptSubscriber` tar emot ett nytt användarmeddelande via metoden `message_created`, som anropas av Wispers publicera/prenumerera-system varje gång slutanvändaren skapar en ny prompt.
2. Klassmetoden `prompt` definierar på ett deklarativt sätt beteendet för chattfärdigställningslogiken för `PromptSubscriber`. AI-modellen kommer att utföra en chattfärdigställning med användarens meddelandeinnehåll, en ny instans av `ReplyStream` som strömparameter och det angivna loopvillkoret.
3. AI-modellen bearbetar prompten och börjar generera ett svar. När svaret strömmas anropas `call`-metoden i `ReplyStream`-instansen för varje datablock.
4. Om AI-modellen bestämmer sig för att anropa en verktygsfunktion, extraheras funktionsnamnet och argumenten från datablocket och lagras i arrayerna `f_name` respektive `f_arguments`.
5. Om AI-modellen genererar användarriktat innehåll buffras det och skickas till klienten via metoden `send_to_client`.
6. När strömbehandlingen är klar anropas metoden `finalize`. Om några verktygsfunktioner anropades under strömningen skickas de via `dispatch`-metoden i `PromptSubscriber`.
7. Metoden `dispatch` lägger till ett `function_call`-meddelande i konversationsutskriften, kör motsvarande verktygsfunktion och lägger till ett `function_result`-meddelande i utskriften med resultatet av funktionsanropet.

8. Efter att ha skickat verktygsfunktionerna rensas funktionsanropshistoriken för att förhindra dubbla funktionsanrop i efterföljande loopar.
9. Om inga verktygsfunktioner anropades uppdaterar `finalize`-metoden `bot_message` med det slutliga innehållet, markerar det som slutfört och skickar det uppdaterade meddelandet till klienten.
10. Loopvillkoret `-> { bot_message.complete? }` utvärderas. Om `bot_message` inte är markerad som slutförd fortsätter loopen, och den ursprungliga prompten skickas igen med den uppdaterade konversationsutskriften.
11. Steg 3-10 upprepas tills `bot_message` markeras som slutförd, vilket indikerar att AI-modellen har avslutat genereringen av sitt svar och inga ytterligare verktygsfunktioner behöver köras.

Genom att implementera denna konversationsloop möjliggör du för AI-modellen att engagera sig i en fram-och-tillbaka-interaktion med applikationen, utföra verktygsfunktioner efter behov och generera användarriktade svar tills konversationen når en naturlig slutsats.

Kombinationen av strömbehandling och konversationsloopen möjliggör dynamiska och interaktiva AI-drivna upplevelser, där AI-modellen kan bearbeta användarinput, utnyttja olika verktyg och funktioner samt ge realtidssvar baserat på den utvecklande konversationskontexten.

## Automatisk fortsättning

Det är viktigt att vara medveten om AI-outputbegränsningar. De flesta modeller har ett maximalt antal tokens de kan generera i ett enda svar, vilket bestäms av parametern `max_tokens`. Om AI-modellen når denna gräns medan den genererar ett svar kommer den att abrupt stoppa och indikera att outputen blev avkortad.

I strömsvaret från AI-plattformens API kan du upptäcka denna situation genom att undersöka variabeln `finish_reason` i datablocket. Om `finish_reason` är satt till `"length"` (eller något annat nyckelvärde specifikt för modellen) betyder det att modellen nådde sin maximala tokengräns under genereringen och att outputen har kortats av.

Ett sätt att hantera detta scenario smidigt och ge en sömlös användarupplevelse är att implementera en mekanism för automatisk fortsättning i din strömbehandlingslogik. Genom att lägga till ett mönstermatchning för längdrelaterade avslutningsorsaker kan du välja att loopa och automatiskt fortsätta outputen från där den slutade.

Här är ett avsiktligt förenklat exempel på hur du kan modifiera `call`-metoden i klassen `ReplyStream` för att stödja automatisk fortsättning:

```
LENGTH_STOPS = %w[length MAX_TOKENS]

def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys
    # ...

    in {
      choices: [
        { delta: {content: new_content},
          finish_reason: finish_reason } ] }

      buffer << new_content.to_s

      if finish_reason.blank?
        send_to_client if new_content.to_s.match?(/\n\n/)
      elsif LENGTH_STOPS.include?(finish_reason)
        continue_cutoff
      else
        finalize
      end

    # ...
  end
end

```

```
26  private
27
28  def continue_cutoff
29    conversation.bot_message!(buffer.join, visible: false)
30    conversation.user_message!("please continue", visible: false)
31    bot_message.update_column(:created_at, Time.current)
32  end
```

I denna modifierade version, när `finish_reason` indikerar trunkerad utdata, istället för att slutföra strömmen, lägger vi till ett par meddelanden i transkriptet utan att slutföra det, flyttar det ursprungliga användarriktade svarsmeddelandet till "botten" av transkriptet genom att uppdatera dess `created_at`-attribut, och låter sedan loopen fortsätta så att AI:n fortsätter generera där den slutade.

Kom ihåg att AI:ns slutpunkt är tillståndslös. Den "känner endast till" det du berättar för den via transkriptet. I det här fallet är sättet vi kommunicerar till AI:n att den blev avbruten genom att lägga till "osynliga" (för slutanvändaren) meddelanden i transkriptet. Kom dock ihåg att detta är ett avsiktligt förenklat exempel. En verklig implementering skulle behöva hantera transkriptet ytterligare för att säkerställa att vi inte slösar tokens och/eller förvirrar AI:n med duplicerade assistentmeddelanden i transkriptet.

En verklig implementering av automatisk fortsättning bör också ha så kallad "kretsbrytarlogik" på plats för att förhindra skenande loopar. Anledningen är att med vissa typer av användarprompter och låga `max_tokens`-inställningar skulle AI:n kunna fortsätta loopa användarriktad utdata i all oändlighet.

> Tänk på att varje loop kräver en separat förfrågan, och att varje förfrågan förbrukar hela ditt transkript igen. Du bör definitivt överväga avvägningen mellan användarupplevelse och API-användning när du bestämmer dig för att implementera automatisk fortsättning i din applikation. Automatisk fortsättning kan vara särskilt

farligt dyrt, speciellt när du använder premiumbaserade kommersiella modeller.

## Slutsats

Strömbehandling är en kritisk aspekt av att bygga AI-drivna applikationer som kombinerar verktygsanvändning med direkta AI-svar. Genom att effektivt hantera strömmande data från AI-plattformars API:er kan du erbjuda en sömlös och interaktiv användarupplevelse, hantera stora svar, optimera resursanvändning och hantera fel på ett elegant sätt.

Den tillhandahållna klassen `Conversation::ReplyStream` demonstrerar hur strömbehandling kan implementeras i en Ruby-applikation med hjälp av mönstermatchning och händelsestyrd arkitektur. Genom att förstå och utnyttja strömbehandlingstekniker kan du låsa upp den fulla potentialen av AI-integration i dina applikationer och leverera kraftfulla och engagerande användarupplevelser.

# Självläkande data

Självläkande data är ett kraftfullt tillvägagångssätt för att säkerställa dataintegritet, konsekvens och kvalitet i applikationer genom att utnyttja förmågorna hos stora språkmodeller (LLM). Denna kategori av mönster fokuserar på idén att använda AI för att automatiskt upptäcka, diagnostisera och korrigera dataavvikelser, inkonsekvenser eller fel, och därmed minska bördan för utvecklare och upprätthålla en hög nivå av datapålitlighet.

I grunden erkänner mönstren för självläkande data att data är livsblodet i alla applikationer, och att säkerställa dess noggrannhet och integritet är avgörande för applikationens korrekta funktion och användarupplevelse. Att hantera och upprätthålla datakvalitet kan dock vara en komplex och tidskrävande uppgift, särskilt när applikationer växer i storlek och komplexitet. Det är här AI:s kraft kommer in i bilden.

I mönstren för självläkande data används AI-arbetare för att kontinuerligt övervaka och analysera din applikations data. Dessa modeller har förmågan att förstå och tolka

mönster, relationer och avvikelser inom data. Genom att utnyttja deras förmågor inom naturlig språkbehandling och förståelse kan de identifiera potentiella problem eller inkonsekvenser i data och vidta lämpliga åtgärder för att rätta till dem.

Processen för självläkande data omfattar vanligtvis flera viktiga steg:

1. **Dataövervakning**: AI-arbetare övervakar ständigt applikationens dataströmmar, databaser eller lagringssystem och letar efter tecken på avvikelser, inkonsekvenser eller fel. Alternativt kan du aktivera en AI-komponent som reaktion på ett undantag.
2. **Avvikelsedetektering**: När ett problem upptäcks analyserar AI-arbetaren data i detalj för att identifiera problemets specifika natur och omfattning. Detta kan innebära att upptäcka saknade värden, inkonsekventa format eller data som bryter mot fördefinierade regler eller begränsningar.
3. **Diagnos och korrigering**: När problemet har identifierats använder AI-arbetaren sin kunskap och förståelse av datadomänen för att bestämma lämplig åtgärd. Detta kan innebära att automatiskt korrigera data, fylla i saknade värden eller flagga problemet för mänsklig intervention om det behövs.
4. **Kontinuerligt lärande (valfritt, beroende på användningsfall)**: När din AI-arbetare stöter på och löser olika dataproblem kan den producera utdata som beskriver vad som hände och hur den reagerade. Denna metadata kan matas in i lärandeprocesser som gör att du (och kanske den underliggande modellen, via finjustering) kan bli mer effektiv över tid när det gäller att identifiera och lösa dataavvikelser.

Genom att automatiskt upptäcka och korrigera dataproblem kan du säkerställa att din applikation arbetar med högkvalitativ, tillförlitlig data. Detta minskar risken för fel, inkonsekvenser eller datarelaterade buggar som påverkar applikationens funktionalitet eller användarupplevelse.

När du har AI-arbetare som hanterar uppgiften med dataövervakning och korrigering kan du fokusera dina ansträngningar på andra kritiska aspekter av applikationen. Detta

sparar tid och resurser som annars skulle ha lagts på manuell datarengöring och underhåll. I själva verket blir manuell hantering av datakvalitet allt mer utmanande när dina applikationer växer i storlek och komplexitet. Mönstren för "Självläkande data" skalar effektivt genom att utnyttja AI:s kraft för att hantera stora datavolymer och upptäcka problem i realtid.

På grund av sin natur kan AI-modeller anpassa sig till förändrade datamönster, scheman eller krav över tid med lite eller ingen övervakning. Så länge deras direktiv ger tillräcklig vägledning, särskilt gällande avsedda resultat, kan din applikation utvecklas och hantera nya datascenarier utan att kräva omfattande manuell intervention eller kodändringar.

Mönstren för självläkande data överensstämmer väl med andra kategorier av mönster vi har diskuterat, såsom "Mångfald av arbetare". Förmågan till självläkande data kan ses som en specialiserad typ av arbetare som specifikt fokuserar på att säkerställa datakvalitet och integritet. Denna typ av arbetare fungerar tillsammans med andra AI-arbetare, där var och en bidrar till olika aspekter av applikationens funktionalitet.

Att implementera mönster för självläkande data i praktiken kräver noggrann design och integration av AI-modeller i applikationsarkitekturen. På grund av riskerna för dataförlust och korruption bör du definiera tydliga riktlinjer för hur du ska använda denna teknik. Du bör också överväga faktorer som prestanda, skalbarhet och datasäkerhet.

## Praktikfall: Att fixa trasig JSON

Ett av de mest praktiska och bekväma sätten att utnyttja självläkande data är också mycket enkelt att förklara: att fixa trasig JSON.

Denna teknik kan tillämpas på den vanliga utmaningen att hantera ofullständig eller inkonsekvent data som genereras av LLM:er, såsom trasig JSON, och ger ett tillvägagångssätt för att automatiskt upptäcka och korrigera dessa problem.

På Olympia stöter jag regelbundet på scenarier där LLM:er genererar JSON-data som inte är helt giltig. Detta kan hända av olika anledningar, som att LLM:en lägger till kommentarer före eller efter den faktiska JSON-koden, eller introducerar syntaxfel som saknade kommatecken eller ej escapade citationstecken. Dessa problem kan leda till parsningsfel och orsaka störningar i applikationens funktionalitet.

För att hantera detta problem har jag implementerat en praktisk lösning i form av en JsonFixer-klass. Denna klass förkroppsligar mönstret "självläkande data" genom att ta den trasiga JSON:en som input och använder en LLM för att fixa den medan så mycket information och avsikt som möjligt bevaras.

```
class JsonFixer
  include Raix::ChatCompletion

  def call(bad_json, error_message)
    raise "No data provided" if bad_json.blank? || error_message.blank?

    transcript << {
        system: "Consider user-provided JSON that generated a parse
                 exception. Do your best to fix it while preserving the
                 original content and intent as much as possible." }
    transcript << { user: bad_json }
    transcript << { assistant: "What is the error message?"}
    transcript << { user: error_message }
    transcript << { assistant: "Here is the corrected JSON\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true)
  end

  def model
    "mistralai/mixtral-8x7b-instruct:nitro"
  end
end
```

Observera hur `JsonFixer` använder Ventriloquist för att styra AI:ns svar.

Processen för självläkande JSON-data fungerar enligt följande:

1. **JSON-generering**: En LLM används för att generera JSON-data baserat på vissa prompts eller krav. På grund av LLM:ers natur är den genererade JSON:en dock inte alltid helt giltig. JSON-tolken kommer naturligtvis att generera ett `ParserError` om du matar in ogiltig JSON.

```
begin
  JSON.parse(llm_generated_json)
rescue JSON::ParserError => e
  JsonFixer.new.call(llm_generated_json, e.message)
end
```

Observera att felmeddelandet också skickas till `JSONFixer`-anropet så att den inte behöver göra fullständiga antaganden om vad som är fel med datan, särskilt eftersom parsern ofta berättar exakt vad som är fel.

2. **LLM-baserad korrigering**: Klassen `JSONFixer` skickar den trasiga JSON-koden tillbaka till en LLM, tillsammans med en specifik prompt eller instruktion för att rätta JSON-koden samtidigt som den bevarar den ursprungliga informationen och avsikten så mycket som möjligt. LLM:en, som är tränad på stora mängder data och har förståelse för JSON-syntax, försöker korrigera felen och generera en giltig JSON-sträng. Svarsavgränsning används för att begränsa LLM:ens output, och vi väljer Mixtral 8x7B som AI-modell, eftersom den är särskilt bra för denna typ av uppgift.
3. **Validering och integrering**: Den korrigerade JSON-strängen som returneras av LLM:en parsas av själva `JSONFixer`-klassen, eftersom den anropade `chat_completion(json: true)`. Om den korrigerade JSON-koden klarar valideringen integreras den tillbaka i applikationens arbetsflöde, vilket gör att applikationen kan fortsätta bearbeta datan sömlöst. Den felaktiga JSON-koden har blivit "läkt".

Även om jag har skrivit och skrivit om min egen `JSONFixer`-implementering flera gånger, tvivlar jag på att den totala tiden som investerats i alla dessa versioner är mer än en eller två timmar.

Observera att bevarande av avsikt är ett nyckelelement i alla självläkande datamönster. Den LLM-baserade korrigeringsprocessen syftar till att bevara den ursprungliga informationen och avsikten i den genererade JSON-koden så mycket som möjligt. Detta säkerställer att den korrigerade JSON-koden behåller sin semantiska betydelse och kan användas effektivt inom applikationens kontext.

Denna praktiska implementering av metoden "Självläkande data" i Olympia visar tydligt hur AI, specifikt LLM:er, kan utnyttjas för att lösa verkliga datautmaningar. Den demonstrerar styrkan i att kombinera traditionella programmeringstekniker med AI-kapacitet för att bygga robusta och effektiva applikationer.

## Postels lag och mönstret "Självläkande data"

"Självläkande data", som exemplifieras av JSONFixer-klassen, överensstämmer väl med principen känd som Postels lag, även kallad Robusthetsprincipen. Postels lag säger:

"Var konservativ i vad du gör, var liberal i vad du accepterar från andra."

Denna princip, som ursprungligen formulerades av Jon Postel, en pionjär inom det tidiga Internet, betonar vikten av att bygga system som är toleranta mot olika eller till och med något felaktiga indata samtidigt som de strikt följer specificerade protokoll när de skickar utdata.

I sammanhanget "Självläkande data" förkroppsligar JSONFixer-klassen Postels lag genom att vara liberal i att acceptera trasig eller ofullkomlig JSON-data genererad av LLM:er. Den avvisar eller misslyckas inte omedelbart när den stöter på JSON som inte strikt följer det förväntade formatet. Istället tar den en tolerant approach och

försöker rätta JSON-koden med hjälp av LLM:er.

Genom att vara liberal i accepterandet av ofullkomlig JSON demonstrerar JSONFixer-klassen robusthet och flexibilitet. Den erkänner att data i den verkliga världen ofta kommer i olika former och kanske inte alltid följer strikta specifikationer. Genom att elegant hantera och korrigera dessa avvikelser säkerställer klassen att applikationen kan fortsätta fungera smidigt, även i närvaro av ofullkomlig data.

Å andra sidan följer JSONFixer-klassen också den konservativa aspekten av Postels lag när det gäller output. Efter att ha rättat JSON-koden med hjälp av LLM:er validerar klassen den korrigerade JSON-koden för att säkerställa att den strikt följer det förväntade formatet. Den upprätthåller datans integritet och korrekthet innan den skickar den vidare till andra delar av applikationen. Detta konservativa tillvägagångssätt garanterar att outputen från JSONFixer-klassen är pålitlig och konsekvent, vilket främjar interoperabilitet och förhindrar spridning av fel.

Intressanta fakta om Jon Postel:

- Jon Postel (1943-1998) var en amerikansk datavetare som spelade en avgörande roll i utvecklingen av Internet. Han var känd som "Internets gud" för sina betydande bidrag till de underliggande protokollen och standarderna.
- Postel var redaktör för dokumentserien Request for Comments (RFC), som är en serie tekniska och organisatoriska anteckningar om Internet. Han författade eller medförfattade över 200 RFC:er, inklusive grundläggande protokoll som TCP, IP och SMTP.
- Förutom sina tekniska bidrag var Postel känd för sin ödmjuka och samarbetsinriktade approach. Han trodde på vikten av att nå konsensus och arbeta tillsammans för att bygga ett robust och interoperabelt nätverk.
- Postel tjänstgjorde som chef för Datornätverksavdelningen vid Information Sciences Institute (ISI) vid University of Southern California (USC) från 1977 fram till sin för tidiga död 1998.

- Som erkännande för sina enorma bidrag tilldelades Postel postumt det prestigefyllda Turing Award 1998, ofta kallat "Datavetenskapens Nobelpris."

Klassen `JSONFixer` främjar robusthet, flexibilitet och interoperabilitet, vilket var grundläggande värderingar som Postel upprätthöll genom hela sin karriär. Genom att bygga system som är toleranta mot ofullkomligheter samtidigt som de strikt följer protokollen, kan vi skapa applikationer som är mer motståndskraftiga och anpassningsbara inför verkliga utmaningar.

## Överväganden och kontraindikationer

Tillämpligheten av självläkande datastrategier är helt beroende av vilken typ av data din applikation hanterar. Det finns en anledning till varför du kanske inte vill monkeypatch:a `JSON.parse` för att automatiskt självkorrigera *alla JSON-parsningsfel* i din applikation: inte alla fel kan eller bör korrigeras automatiskt.

Självläkning är särskilt problematiskt när det kombineras med regulatoriska krav eller efterlevnadskrav relaterade till datahantering och databehandling. Vissa branscher, såsom sjukvård och finans, har så strikta regler gällande dataintegritet och granskningsbarhet att någon form av "svart låda"-datakorrigering utan ordentlig övervakning eller loggning kan bryta mot dessa regler. Det är avgörande att säkerställa att alla självläkande datatekniker du utvecklar överensstämmer med tillämpliga juridiska och regulatoriska ramverk.

Tillämpning av självläkande datatekniker, särskilt de som involverar AI-modeller, kan också ha stor påverkan på applikationens prestanda och resursanvändning. Att behandla stora datavolymer genom AI-modeller för feldetektering och korrigering kan vara beräkningsintensivt. Det är viktigt att utvärdera avvägningarna mellan fördelarna med självläkande data och de tillhörande prestanda- och resurskostnaderna.

Med det sagt, låt oss fördjupa oss i faktorerna som påverkar beslut om när och var denna kraftfulla strategi ska tillämpas.

## Datakritikalitet

När man överväger tillämpningen av självläkande datatekniker är det avgörande att bedöma kritikaliteten hos datan som behandlas. Kritikalitetsnivån avser datans betydelse och känslighet i kontexten av din applikation och dess verksamhetsområde.

I vissa fall kan automatisk korrigering av datafel vara olämplig, särskilt om datan är mycket känslig eller har juridiska implikationer. Överväg till exempel följande scenarier:

1. **Finansiella transaktioner**: I finansiella applikationer, såsom banksystem eller handelsplattformar, är datanoggrannhet av yttersta vikt. Även mindre fel i finansiell data kan få betydande konsekvenser, såsom felaktiga kontosaldon, feldirigerade medel eller felaktiga handelsbeslut. I dessa fall kan automatiserade korrigeringar utan grundlig verifiering och granskning medföra oacceptabla risker.
2. **Medicinska journaler**: Sjukvårdsapplikationer hanterar mycket känslig och konfidentiell patientdata. Felaktigheter i medicinska journaler kan få allvarliga konsekvenser för patientsäkerhet och behandlingsbeslut. Att automatiskt modifiera medicinsk data utan ordentlig övervakning och validering av kvalificerad sjukvårdspersonal kan bryta mot regulatoriska krav och äventyra patientens välbefinnande.
3. **Juridiska dokument**: Applikationer som hanterar juridiska dokument, såsom kontrakt, avtal eller domstolshandlingar, kräver strikt noggrannhet och integritet. Även mindre fel i juridisk data kan få betydande rättsliga följder. Automatiserade korrigeringar inom detta område kan vara olämpliga, eftersom datan ofta kräver manuell granskning och verifiering av juridiska experter för att säkerställa dess giltighet och verkställbarhet.

I dessa kritiska datascenarier överväger riskerna med automatiserade korrigeringar ofta de potentiella fördelarna. Konsekvenserna av att introducera fel eller modifiera data felaktigt kan vara allvarliga och leda till ekonomiska förluster, juridiskt ansvar eller till och med skada för individer.

När man hanterar mycket kritisk data är det viktigt att prioritera manuella verifierings- och valideringsprocesser. Mänsklig övervakning och expertis är avgörande för att säkerställa datans noggrannhet och integritet. Automatiserade självläkande tekniker kan fortfarande användas för att flagga potentiella fel eller inkonsekvenser, men det slutliga beslutet om korrigeringar bör involvera mänskligt omdöme och godkännande.

Det är dock viktigt att notera att inte all data i en applikation behöver ha samma kritikalitetsnivå. Inom samma applikation kan det finnas delmängder av data som är mindre känsliga eller har lägre påverkan om fel uppstår. I sådana fall kan självläkande datatekniker selektivt tillämpas på dessa specifika datadelmängder, medan kritisk data förblir föremål för manuell verifiering.

Nyckeln är att noggrant bedöma kritikaliteten hos varje datakategori i din applikation och definiera tydliga riktlinjer och processer för hantering av korrigeringar baserat på tillhörande risker och implikationer. Genom att skilja mellan kritisk (t.ex. huvudböcker, medicinska journaler) och icke-kritisk data (t.ex. postadresser, resursvarningar) kan du hitta en balans mellan att utnyttja fördelarna med självläkande datatekniker där det är lämpligt och upprätthålla strikt kontroll och övervakning där det är nödvändigt.

I slutändan bör beslutet att tillämpa självläkande datatekniker på kritisk data fattas i samråd med domänexperter, juridiska rådgivare och andra relevanta intressenter. Det är viktigt att beakta de specifika krav, regler och risker som är förknippade med din applikations data och anpassa datakorrigeringsstrategierna därefter.

## Felens allvarlighetsgrad

Vid tillämpning av självläkande datatekniker är det viktigt att bedöma allvarlighetsgraden och påverkan av datafelen. Alla fel är inte likvärdiga, och

lämplig åtgärd kan variera beroende på problemets allvarlighetsgrad.

Mindre inkonsekvenser eller formateringsfel kan vara lämpliga för automatisk korrigering. Till exempel kan en självläkande dataarbetare som har till uppgift att fixa trasig JSON hantera saknade kommatecken eller oescapade dubbla citattecken utan att väsentligt ändra datans betydelse eller struktur. Dessa typer av fel är ofta okomplicerade att korrigera och har minimal påverkan på den övergripande dataintegriteten.

Däremot kan allvarligare fel som fundamentalt förändrar datans betydelse eller integritet kräva ett annat tillvägagångssätt. I sådana fall kan automatiserade korrigeringar vara otillräckliga, och mänsklig intervention kan vara nödvändig för att säkerställa datans korrekthet och giltighet.

Här kommer konceptet att använda själva AI:n för att hjälpa till att bestämma felens allvarlighetsgrad in i bilden. Genom att utnyttja AI-modellernas kapacitet kan vi utforma självläkande dataarbetare som inte bara korrigerar fel utan också bedömer felens allvarlighetsgrad och fattar välgrundade beslut om hur de ska hanteras.

Låt oss till exempel betrakta en självläkande dataarbetare som ansvarar för att korrigera inkonsekvenser i data som flödar in i en kunddatabas. Arbetaren kan utformas för att analysera data och identifiera potentiella fel, såsom saknad eller motstridig information. Istället för att automatiskt korrigera alla fel kan arbetaren utrustas med ytterligare verktygsanrop som gör det möjligt att flagga allvarliga fel för mänsklig granskning.

Här är ett exempel på hur detta kan implementeras:

```ruby
class CustomerDataReviewer
  include Raix::ChatCompletion
  include Raix::FunctionDeclarations

  attr_accessor :customer

  function :flag_for_review, reason: { type: "string" } do |params|
    AdminNotifier.review_request(customer, params[:reason])
  end

  def initialize(customer)
    self.customer = customer
  end

  def call(customer_data)
    transcript << {
      system: "You are a customer data reviewer. Your task is to identify
        and correct inconsistencies in customer data.

        < additional instructions here... >

        If you encounter severe errors that require human review, use the
        `flag_for_review` tool to flag the data for manual intervention." }

    transcript << { user: customer.to_json }
    transcript << { assistant: "Reviewed/corrected data:\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true).then do |result|
      return if result.blank?

      customer.update(result)
    end
  end
end
```

I detta exempel är `CustomerDataHealer`-arbetaren utformad för att identifiera och korrigera inkonsekvenser i kunddata. Återigen använder vi svarsinramning och Buktalare för att få strukturerad output. Viktigt är att arbetarens systemdirektiv

innehåller instruktioner om att använda funktionen `flag_for_review` om allvarliga fel påträffas.

När arbetaren bearbetar kunddatan analyserar den informationen och försöker korrigera eventuella inkonsekvenser. Om arbetaren fastställer att felen är allvarliga och kräver mänsklig intervention kan den använda verktyget `flag_for_review` för att flagga datan och ange en anledning till flaggningen.

Metoden `chat_completion` anropas med `json: true` för att tolka den korrigerade kunddatan som JSON. Det finns ingen möjlighet till looping efter ett funktionsanrop, så resultatet kommer att vara tomt om `flag_for_review` har åberopats. I annat fall uppdateras kunden med den granskade och potentiellt korrigerade datan.

Genom att införliva bedömning av felens allvarlighetsgrad och möjligheten att flagga data för mänsklig granskning blir den självläkande dataarbetaren mer intelligent och anpassningsbar. Den kan hantera mindre fel automatiskt samtidigt som den eskalerar allvarliga fel till mänskliga experter för manuell intervention.

De specifika kriterierna för att bestämma felens allvarlighetsgrad kan definieras i arbetarens direktiv baserat på domänkunskap och verksamhetskrav. Faktorer som påverkan på dataintegritet, risken för dataförlust eller korruption, och konsekvenserna av felaktig data kan beaktas vid bedömning av allvarlighetsgraden.

Genom att utnyttja AI för att bedöma felens allvarlighetsgrad och tillhandahålla alternativ för mänsklig intervention kan självläkande datatekniker uppnå en balans mellan automatisering och upprätthållande av datakvalitet. Detta tillvägagångssätt säkerställer att mindre fel korrigeras effektivt medan allvarliga fel får nödvändig uppmärksamhet och expertis från mänskliga granskare.

## Domänkomplexitet

När man överväger tillämpningen av självläkande datatekniker är det viktigt att utvärdera komplexiteten i datans domän och de regler som styr dess struktur

och relationer. Domänens komplexitet kan avsevärt påverka effektiviteten och genomförbarheten av automatiserade datakorrigeringsmetoder.

Självläkande datatekniker fungerar bra när datan följer väldefinierade mönster och begränsningar. I domäner där datastrukturen är relativt enkel och relationerna mellan dataelement är okomplicerade kan automatiserade korrigeringar tillämpas med hög tillförlitlighet. Till exempel kan korrigering av formateringsproblem eller upprätthållande av grundläggande datatypsbegränsningar ofta hanteras effektivt av självläkande dataarbetare.

Men när domänens komplexitet ökar, växer också utmaningarna med automatiserad datakorrigering. I domäner med invecklad affärslogik, komplexa relationer mellan dataentiteter, eller domänspecifika regler och undantag, kan självläkande datatekniker inte alltid fånga upp nyanserna och kan introducera oavsiktliga konsekvenser.

Låt oss ta ett exempel på en komplex domän: ett finansiellt handelssystem. I denna domän involverar datan olika finansiella instrument, marknadsdata, handelsregler och regulatoriska krav. Relationerna mellan olika dataelement kan vara invecklade, och reglerna som styr datavaliditet och konsistens kan vara mycket specifika för domänen.

I en sådan komplex domän skulle en självläkande dataarbetare som har till uppgift att korrigera inkonsekvenser i handelsdata behöva ha en djup förståelse för de domänspecifika reglerna och begränsningarna. Den skulle behöva ta hänsyn till faktorer som marknadsregleringar, handelsgränser, riskberäkningar och avvecklingsrutiner. Automatiserade korrigeringar i detta sammanhang kanske inte alltid fångar domänens fulla komplexitet och kan oavsiktligt introducera fel eller bryta mot domänspecifika regler.

För att hantera utmaningarna med domänkomplexitet kan självläkande datatekniker förbättras genom att införliva domänspecifik kunskap och regler i AI-modellerna och arbetarna. Detta kan uppnås genom tekniker som:

1. **Domänspecifik träning**: AI-modellerna som används för självläkande data kan styras eller till och med finjusteras på domänspecifika datamängder som fångar

särdragen och reglerna i den specifika domänen. Genom att exponera modellerna för representativ data och scenarier kan de lära sig mönster, begränsningar och undantag som är specifika för domänen.

2. **Regelbaserade begränsningar**: Självläkande dataarbetare kan förstärkas med explicita regelbaserade begränsningar som kodar domänspecifik kunskap. Dessa regler kan definieras av domänexperter och integreras i datakorrigeringsprocessen. AI-modellerna kan sedan använda dessa regler för att vägleda sina beslut och säkerställa efterlevnad av domänspecifika krav.
3. **Samarbete med domänexperter**: I komplexa domäner är det avgörande att involvera domänexperter i design och utveckling av självläkande datatekniker. Domänexperter kan ge värdefulla insikter om datans komplexitet, affärsreglerna och potentiella kantfall. Deras kunskap kan införlivas i AI-modellerna och arbetarna för att förbättra precisionen och tillförlitligheten i automatiserade datakorrigeringar genom användning av Människa-i-loopen-mönster.
4. **Inkrementell och iterativ approach**: När man hanterar komplexa domäner är det ofta fördelaktigt att anta en inkrementell och iterativ approach till självläkande data. Istället för att försöka automatisera korrigeringar för hela domänen på en gång, fokusera på specifika subdomäner eller datakategorier där reglerna och begränsningarna är välförstådda. Utöka gradvis omfattningen av självläkande tekniker allt eftersom förståelsen för domänen växer och teknikerna visar sig vara effektiva.

Genom att ta hänsyn till komplexiteten i datadomänen och införliva domänspecifik kunskap i tekniker för självläkande data kan man uppnå en balans mellan automatisering och noggrannhet. Det är viktigt att inse att självläkande data inte är en universallösning och att tillvägagångssättet bör anpassas efter de specifika krav och utmaningar som finns inom varje domän.

I komplexa domäner kan en hybridmetod som kombinerar tekniker för självläkande data med mänsklig expertis och övervakning vara mest effektiv. Automatiserade

korrigeringar kan hantera rutinmässiga och väldefinierade fall, medan komplexa scenarier eller undantag kan flaggas för mänsklig granskning och ingripande. Detta samarbetsinriktade tillvägagångssätt säkerställer att fördelarna med automatisering realiseras samtidigt som nödvändig kontroll och noggrannhet bibehålls i komplexa datadomäner.

## Förklarbarhet och Transparens

Förklarbarhet syftar på förmågan att förstå och tolka resonemangen bakom de beslut som AI-modeller fattar, medan transparens handlar om att ge tydlig insyn i datakorrigeringsprocessen.

I många sammanhang måste dataändringar vara granskningsbara och motiverbara. Intressenter, inklusive verksamhetsanvändare, revisorer och tillsynsmyndigheter, kan kräva förklaringar till varför vissa datakorrigeringar gjordes och hur AI-modellerna kom fram till dessa beslut. Detta är särskilt viktigt inom områden där datanoggrannhet och dataintegritet har betydande konsekvenser, såsom finans, sjukvård och juridiska frågor.

För att hantera behovet av förklarbarhet och transparens bör tekniker för självläkande data införliva mekanismer som ger insikt i AI-modellernas beslutsprocess. Detta kan uppnås genom olika tillvägagångssätt:

1. **Tankekedjor**: Att be modellen förklara sitt tänkande "högt" innan ändringar tillämpas på data kan göra det lättare att förstå beslutsprocessen och kan generera lättlästa förklaringar för korrigeringarna. Kompromissen är lite mer komplexitet i att separera förklaringen från den strukturerade datautmatningen, vilket kan hanteras genom…
2. **Förklaringsgenerering**: Självläkande dataarbetare kan utrustas med förmågan att generera lättlästa förklaringar för de korrigeringar de gör. Detta kan uppnås genom att be modellen producera sin beslutsprocess som lättförståeliga förklaringar *integrerade i själva datan*. Till exempel skulle en självläkande

dataarbetare kunna generera en rapport som belyser de specifika dataavvikelser den identifierat, korrigeringarna den tillämpat och resonemangen bakom dessa korrigeringar.

3. **Funktionsviktighet**: AI-modeller kan instrueras med information om betydelsen av olika funktioner eller attribut i datakorrigeringsprocessen som en del av deras direktiv. Dessa direktiv kan i sin tur exponeras för mänskliga intressenter. Genom att identifiera de nyckelfaktorer som påverkar modellens beslut kan intressenter få insikt i resonemangen bakom korrigeringarna och bedöma deras giltighet.
4. **Loggning och Granskning**: Att implementera omfattande loggnings- och granskningsmekanismer är avgörande för att upprätthålla transparens i processen för självläkande data. Varje datakorrigering som görs av AI-modeller bör loggas, inklusive originaldata, korrigerad data och de specifika åtgärder som vidtagits. Detta granskningsspår möjliggör retrospektiv analys och ger en tydlig dokumentation av de ändringar som gjorts i datan.
5. **Människa-i-loopen-metoden**: Att inkludera en människa-i-loopen-metod kan förbättra förklarbarheten och transparensen i tekniker för självläkande data. Genom att involvera mänskliga experter i granskning och validering av AI-genererade korrigeringar kan organisationer säkerställa att korrigeringarna överensstämmer med domänkunskap och verksamhetskrav. Mänsklig övervakning tillför ett extra ansvarsskikt och möjliggör identifiering av eventuella fördomar eller fel i AI-modellerna.
6. **Kontinuerlig Övervakning och Utvärdering**: Regelbunden övervakning och utvärdering av prestandan hos tekniker för självläkande data är avgörande för att upprätthålla transparens och förtroende. Genom att bedöma AI-modellernas noggrannhet och effektivitet över tid kan organisationer identifiera eventuella avvikelser eller anomalier och vidta korrigerande åtgärder. Kontinuerlig övervakning hjälper till att säkerställa att processen för självläkande data förblir tillförlitlig och i linje med önskade resultat.

Förklarbarhet och transparens är kritiska överväganden vid implementering av

tekniker för självläkande data. Genom att tillhandahålla tydliga förklaringar för datakorrigeringar, upprätthålla omfattande granskningsspår och involvera mänsklig övervakning kan organisationer bygga förtroende för processen med självläkande data och säkerställa att ändringarna som görs i datan är motiverade och i linje med verksamhetsmålen.

Det är viktigt att hitta en balans mellan fördelarna med automatisering och behovet av transparens. Medan tekniker för självläkande data avsevärt kan förbättra datakvalitet och effektivitet bör detta inte ske på bekostnad av att förlora insyn och kontroll över datakorrigeringsprocessen. Genom att utforma självläkande dataarbetare med förklarbarhet och transparens i åtanke kan organisationer utnyttja kraften i AI samtidigt som de upprätthåller nödvändig ansvarsskyldighet och förtroende för datan.

## Oavsiktliga Konsekvenser

Medan tekniker för självläkande data syftar till att förbättra datakvalitet och konsistens är det avgörande att vara medveten om risken för oavsiktliga konsekvenser. Automatiserade korrigeringar kan, om de inte noggrant utformas och övervakas, oavsiktligt förändra datans betydelse eller sammanhang, vilket leder till följdproblem.

En av de främsta riskerna med självläkande data är införandet av fördomar eller fel i datakorrigeringsprocessen. AI-modeller kan, precis som alla andra mjukvarusystem, påverkas av fördomar som finns i träningsdata eller som införts genom algoritmernas utformning. Om dessa fördomar inte identifieras och åtgärdas kan de fortplanta sig genom processen för självläkande data och resultera i snedvridna eller felaktiga datamodifieringar.

Ta till exempel en självläkande dataarbetare som har till uppgift att korrigera inkonsekvenser i kunders demografiska data. Om AI-modellen har lärt sig fördomar från historiska data, såsom att koppla vissa yrken eller inkomstnivåer till specifika kön eller etniciteter, kan den göra felaktiga antaganden och modifiera data på ett sätt

som förstärker dessa fördomar. Detta kan leda till felaktiga kundprofiler, vilseledande affärsbeslut och potentiellt diskriminerande resultat.

En annan potentiell oavsiktlig konsekvens är förlusten av värdefull information eller sammanhang under datakorrigeringsprocessen. Självläkande datatekniker fokuserar ofta på att standardisera och normalisera data för att säkerställa konsekvens. I vissa fall kan dock originaldata innehålla nyanser, undantag eller kontextuell information som är viktig för att förstå helhetsbilden. Automatiserade korrigeringar som blint genomdriver standardisering kan oavsiktligt ta bort eller dölja denna värdefulla information.

Föreställ dig till exempel en självläkande dataarbetare som ansvarar för att korrigera inkonsekvenser i medicinska journaler. Om arbetaren stöter på en patients sjukdomshistoria med ett sällsynt tillstånd eller en ovanlig behandlingsplan, kan den försöka normalisera data för att passa ett mer vanligt mönster. Genom att göra detta kan den dock förlora de specifika detaljer och sammanhang som är avgörande för att korrekt representera patientens unika situation. Denna informationsförlust kan få allvarliga konsekvenser för patientvården och medicinska beslutsfattandet.

För att minska riskerna för oavsiktliga konsekvenser är det viktigt att ta en proaktiv approach när man utformar och implementerar självläkande datatekniker:

1. **Grundlig testning och validering**: Innan självläkande dataarbetare implementeras i produktion är det avgörande att noggrant testa och validera deras beteende mot ett brett spektrum av scenarier. Detta inkluderar testning med representativa datamängder som täcker olika kantfall, undantag och potentiella fördomar. Rigorös testning hjälper till att identifiera och hantera eventuella oavsiktliga konsekvenser innan de påverkar data i verklig miljö.
2. **Kontinuerlig övervakning och utvärdering**: Att implementera mekanismer för kontinuerlig övervakning och utvärdering är viktigt för att upptäcka och mildra oavsiktliga konsekvenser över tid. Regelbunden granskning av resultaten från självläkande dataprocesser, analys av påverkan på efterföljande system och beslutsfattande, samt insamling av återkoppling från intressenter kan hjälpa till

att identifiera eventuella negativa effekter och initiera lämpliga korrigerande åtgärder. Om din organisation har operativa instrumentpaneler är det förmodligen en bra idé att lägga till tydligt synliga mätvärden relaterade till automatiserade dataändringar. Att lägga till larm kopplade till stora avvikelser från normal dataändringsaktivitet är förmodligen en ännu bättre idé!

3. **Mänsklig översyn och intervention**: Att upprätthålla mänsklig översyn och möjlighet att ingripa i den självläkande dataprocessen är avgörande. Även om automatisering kan förbättra effektiviteten avsevärt är det viktigt att ha mänskliga experter som granskar och validerar korrigeringar gjorda av AI-modeller, särskilt inom kritiska eller känsliga områden. Mänskligt omdöme och domänexpertis kan hjälpa till att identifiera och hantera eventuella oavsiktliga konsekvenser som kan uppstå.

4. **Förklarbar AI (XAI) och transparens**: Som diskuterades i föregående avsnitt kan införandet av förklarbar AI-teknik och säkerställande av transparens i den självläkande dataprocessen hjälpa till att mildra oavsiktliga konsekvenser. Genom att tillhandahålla tydliga förklaringar för datakorrigeringar och upprätthålla omfattande granskningsloggar kan organisationer bättre förstå och spåra resonemangen bakom de modifieringar som görs av AI-modeller.
5. **Inkrementell och iterativ approach**: Att anta en inkrementell och iterativ approach till självläkande data kan hjälpa till att minimera risken för oavsiktliga konsekvenser. Istället för att tillämpa automatiserade korrigeringar på hela datamängden på en gång, börja med en delmängd av data och utöka gradvis omfattningen när teknikerna visar sig vara effektiva och pålitliga. Detta möjliggör noggrann övervakning och justering längs vägen, vilket minskar påverkan av eventuella oavsiktliga konsekvenser.
6. **Samarbete och återkoppling**: Att engagera intressenter från olika domäner och uppmuntra samarbete och återkoppling genom hela den självläkande dataprocessen kan hjälpa till att identifiera och hantera oavsiktliga konsekvenser. Att regelbundet söka input från domänexperter, datakonsumenter och

slutanvändare kan ge värdefulla insikter om den verkliga påverkan av datakorrigeringarna och belysa eventuella problem som kan ha förbisetts.

Genom att proaktivt hantera risken för oavsiktliga konsekvenser och implementera lämpliga skyddsåtgärder kan organisationer utnyttja fördelarna med självläkande datatekniker samtidigt som potentiella negativa effekter minimeras. Det är viktigt att se självläkande data som en iterativ och samarbetsinriktad process, där man kontinuerligt övervakar, utvärderar och förfinar teknikerna för att säkerställa att de överensstämmer med önskade resultat och upprätthåller datans integritet och tillförlitlighet.

När man överväger användningen av självläkande datamönster är det viktigt att noggrant utvärdera dessa faktorer och väga fördelarna mot potentiella risker och begränsningar. I vissa fall kan en hybridapproach som kombinerar automatiserade korrigeringar med mänsklig översyn och intervention vara den mest lämpliga lösningen.

Det är också värt att notera att självläkande datatekniker inte bör ses som en ersättning för robust datavalidering, indatasanering och felhanteringsmekanismer. Dessa grundläggande rutiner förblir kritiska för att säkerställa dataintegritet och säkerhet. Självläkande data bör ses som ett kompletterande tillvägagångssätt som kan förstärka och förbättra dessa befintliga åtgärder.

I slutändan beror beslutet att använda självläkande datamönster på de specifika kraven, begränsningarna och prioriteringarna i din applikation. Genom att noggrant överväga de aspekter som beskrivits ovan och anpassa dem till din applikations mål och arkitektur kan du fatta välgrundade beslut om när och hur du effektivt kan utnyttja självläkande datatekniker.

# Kontextuell innehållsgenerering

Mönster för kontextuell innehållsgenerering utnyttjar kraften i stora språkmodeller (LLMs) för att generera dynamiskt och kontextspecifikt innehåll inom applikationer. Denna kategori av mönster erkänner vikten av att leverera personaliserat och relevant innehåll till användare baserat på deras specifika behov, preferenser och till och med tidigare och nuvarande interaktioner med applikationen.

> I detta sammanhang syftar "innehåll" både på primärt innehåll (dvs. blogginlägg, artiklar, etc.) och metainnehåll, såsom rekommendationer till primärt innehåll.

Mönster för kontextuell innehållsgenerering kan spela en avgörande roll i att förbättra dina användarengagemangsnivåer, tillhandahålla skräddarsydda upplevelser och automatisera innehållsskapande uppgifter både för dig och dina användare. Genom att

använda mönstren vi beskriver i detta kapitel kan du skapa applikationer som genererar innehåll dynamiskt och anpassar sig till kontext och indata i realtid.

Mönstren fungerar genom att integrera LLMs i applikationens utdata, allt från användargränssnittet (ibland kallat "chrome"), till e-postmeddelanden och andra former av aviseringar, såväl som alla innehållsgenereringsflöden.

När en användare interagerar med applikationen eller utlöser en specifik innehållsförfrågan, fångar applikationen upp den relevanta kontexten, såsom användarpreferenser, tidigare interaktioner eller specifika uppmaningar. Denna kontextuella information matas sedan in i LLM:en, tillsammans med eventuella nödvändiga mallar eller riktlinjer och används för att producera textutdata som annars skulle behöva vara antingen hårdkodad, lagrad i en databas eller algoritmiskt genererad.

Det LLM-genererade innehållet kan ta olika former, såsom personaliserade rekommendationer, dynamiska produktbeskrivningar, anpassade e-postsvar eller till och med hela artiklar eller blogginlägg. En av de mest radikala användningsområdena för detta innehåll som jag var pionjär med för över ett år sedan är att dynamiskt generera UI-element som formuläretiketter, verktygstips och andra typer av förklarande text.

## Personalisering

En av de viktigaste fördelarna med mönster för kontextuell innehållsgenerering är förmågan att leverera mycket personaliserade upplevelser till användare. Genom att generera innehåll baserat på användarspecifik kontext möjliggör dessa mönster för applikationer att skräddarsy innehåll efter individuella användares intressen, preferenser och interaktioner.

Personalisering går längre än att bara infoga en användares namn i generiskt innehåll. Det handlar om att utnyttja den rika kontext som finns tillgänglig om varje användare för att generera innehåll som resonerar med deras specifika behov och önskemål. Denna kontext kan inkludera ett brett spektrum av faktorer, såsom:

1. **Användarprofilinformation**: På den mest allmänna nivån av att tillämpa denna teknik kan demografiska data, intressen, preferenser och andra profilattribut användas för att generera innehåll som överensstämmer med användarens bakgrund och egenskaper.
2. **Beteendedata**: En användares tidigare interaktioner med applikationen, såsom visade sidor, klickade länkar eller köpta produkter, kan ge värdefulla insikter om deras beteende och intressen. Denna data kan användas för att generera innehållsförslag som återspeglar deras engagemangsmönster och förutser deras framtida behov.
3. **Kontextuella faktorer**: Användarens nuvarande kontext, såsom deras plats, enhet, tid på dygnet eller till och med vädret, kan påverka innehållsgenereringsprocessen. Till exempel kan en reseapplikation ha en AI-arbetare som kan generera personaliserade rekommendationer baserade på användarens nuvarande plats och rådande väderförhållanden.

Genom att utnyttja dessa kontextuella faktorer möjliggör mönster för kontextuell innehållsgenerering att applikationer kan leverera innehåll som känns skräddarsytt för varje enskild användare. Denna nivå av personalisering har flera betydande fördelar:

1. **Ökat engagemang**: Personaliserat innehåll fångar användarnas uppmärksamhet och håller dem engagerade i applikationen. När användare känner att innehållet är relevant och talar direkt till deras behov är det mer sannolikt att de spenderar mer tid med att interagera med applikationen och utforska dess funktioner.
2. **Förbättrad användarbelåtenhet**: Personaliserat innehåll visar att applikationen förstår och bryr sig om användarens unika krav. Genom att tillhandahålla innehåll som är hjälpsamt, informativt och anpassat efter deras intressen kan applikationen förbättra användarbelåtenheten och bygga en starkare koppling med sina användare.
3. **Högre konverteringsgrader**: I samband med e-handel eller marknadsföringsapplikationer kan personaliserat innehåll avsevärt påverka konverteringsgrader. Genom att

presentera användare med produkter, erbjudanden eller rekommendationer som är anpassade efter deras preferenser och beteende kan applikationen öka sannolikheten för att användare vidtar önskade åtgärder, såsom att göra ett köp eller registrera sig för en tjänst.

## Produktivitet

Mönster för kontextuell innehållsgenerering kan avsevärt öka vissa typer av produktivitet genom att minska behovet av manuell innehållsgenerering och redigering i kreativa processer. Genom att utnyttja kraften i LLMs kan du generera högkvalitativt innehåll i stor skala, vilket sparar tid och ansträngning som dina innehållsskapare och utvecklare annars skulle behöva lägga på tråkigt manuellt arbete.

Traditionellt sett behöver innehållsskapare researcha, skriva, redigera och formatera innehåll för att säkerställa att det uppfyller applikationens krav och användarnas förväntningar. Denna process kan vara tidskrävande och resurskrävande, särskilt när innehållsvolymen växer.

Men med kontextuella innehållsgenereringsmönster kan innehållsskapandet till stor del automatiseras. Stora språkmodeller kan generera sammanhängande, grammatiskt korrekt och kontextuellt relevant innehåll baserat på givna prompter och riktlinjer. Denna automatisering erbjuder flera produktivitetsfördelar:

1. **Minskat manuellt arbete**: Genom att delegera innehållsgenerering till stora språkmodeller kan innehållsskapare fokusera på uppgifter på högre nivå som innehållsstrategi, idégenerering och kvalitetssäkring. De kan tillhandahålla nödvändig kontext, mallar och riktlinjer till språkmodellen och låta den hantera själva innehållsgenereringen. Detta minskar det manuella arbetet som krävs för att skriva och redigera, vilket gör att innehållsskapare kan vara mer produktiva och effektiva.

2. **Snabbare innehållsskapande**: Stora språkmodeller kan generera innehåll mycket snabbare än mänskliga skribenter. Med rätt prompter och riktlinjer kan en språkmodell producera flera innehållsdelar på några sekunder eller minuter. Denna hastighet gör det möjligt för applikationer att generera innehåll i mycket snabbare takt och hålla jämna steg med användarnas krav och det ständigt föränderliga digitala landskapet.

> Leder snabbare innehållsskapande till en "allmänningarnas tragedi" där internet dränks i innehåll som ingen läser? Tyvärr misstänker jag att svaret är ja.

3. **Konsekvens och kvalitet**: Stora språkmodeller kan enkelt revidera innehåll så att det blir konsekvent i stil, ton och kvalitet. Med tydliga riktlinjer och exempel kan vissa typer av applikationer (t.ex. nyhetsredaktioner, PR etc.) säkerställa att deras mänskligt genererade innehåll överensstämmer med deras varumärkesröst och uppfyller önskade kvalitetsstandarder. Denna konsekvens minskar behovet av omfattande redigering och revidering, vilket sparar tid och arbete i innehållsskapandeprocessen.
4. **Iteration och optimering**: Kontextuella innehållsgenereringsmönster möjliggör snabb iteration och optimering av innehåll. Genom att justera prompterna, mallarna eller riktlinjerna som ges till språkmodellen kan dina applikationer snabbt generera variationer av innehåll och testa olika tillvägagångssätt på ett automatiserat sätt som aldrig varit möjligt tidigare. Denna iterativa process möjliggör snabbare experimenterande och förfining av innehållsstrategier, vilket leder till mer effektivt och engagerande innehåll över tid. Denna specifika teknik kan vara en total spelförändare för applikationer som e-handel som lever och dör baserat på avvisningsfrekvenser och engagemang

Det är viktigt att notera att även om kontextuella innehållsgenereringsmönster avsevärt kan förbättra produktiviteten, eliminerar de inte helt behovet av mänsklig inblandning. Innehållsskapare och redaktörer spelar fortfarande en avgörande roll i att definiera den övergripande innehållsstrategin, ge vägledning till språkmodellen och säkerställa kvaliteten och lämpligheten hos det genererade innehållet.

Genom att automatisera de mer repetitiva och tidskrävande aspekterna av innehållsskapande frigör kontextuella innehållsgenereringsmönster värdefull mänsklig tid och resurser som kan omdirigeras mot uppgifter med högre värde. Denna ökade produktivitet gör det möjligt för dig att leverera mer personaliserat och engagerande innehåll till användare samtidigt som arbetsflödet för innehållsskapande optimeras.

## Snabb iteration och experimentering

Kontextuella innehållsgenereringsmönster gör det möjligt att snabbt iterera och experimentera med olika innehållsvariationer, vilket möjliggör snabbare optimering och förfining av din innehållsstrategi. Du kan generera flera versioner av innehåll på några sekunder, helt enkelt genom att justera kontexten, mallarna eller riktlinjerna som ges till modellen.

Denna snabba iterationsförmåga erbjuder flera viktiga fördelar:

1. **Testning och optimering**: Med möjligheten att snabbt generera innehållsvariationer kan du enkelt testa olika tillvägagångssätt och mäta deras effektivitet. Du kan till exempel generera flera versioner av en produktbeskrivning eller ett marknadsföringsmeddelande, var och en anpassad till ett specifikt användarsegment eller kontext. Genom att analysera användarengagemangsmått, såsom klickfrekvenser eller konverteringsfrekvenser, kan du identifiera de mest effektiva innehållsvariationerna och optimera din innehållsstrategi därefter.

2. **A/B-testning**: Kontextuella innehållsgenereringsmönster möjliggör sömlös A/B-testning av innehåll. Du kan generera två eller fler variationer av innehåll och slumpmässigt visa dem för olika användargrupper. Genom att jämföra prestandan för varje variation kan du avgöra vilket innehåll som resonerar bäst med din målgrupp. Detta datadrivna tillvägagångssätt låter dig fatta välgrundade beslut och kontinuerligt förfina ditt innehåll för att maximera användarengagemanget och uppnå önskade resultat.
3. **Personaliseringsexperiment**: Snabb iteration och experimentering är särskilt värdefullt när det gäller personalisering. Med kontextuella innehållsgenereringsmönster kan du snabbt generera personaliserade innehållsvariationer baserade på olika användarsegment, preferenser eller beteenden. Genom att experimentera med olika personaliseringsstrategier kan du identifiera de mest effektiva sätten att engagera enskilda användare och leverera skräddarsydda upplevelser.
4. **Anpassning till förändrade trender**: Förmågan att snabbt iterera och experimentera gör det möjligt att vara flexibel och anpassa sig till förändrade trender och användarpreferenser. När nya ämnen, nyckelord eller användarbeteenden dyker upp kan du snabbt generera innehåll som ligger i linje med dessa trender. Genom att kontinuerligt experimentera och förfina ditt innehåll kan du förbli relevant och behålla en konkurrensfördel i det ständigt föränderliga digitala landskapet.
5. **Kostnadseffektiv experimentering**: Traditionell innehållsexperimentering involverar ofta betydande tid och resurser, eftersom innehållsskapare behöver manuellt utveckla och testa olika variationer. Med kontextuella innehållsgenereringsmönster reduceras dock experimentkostnaderna avsevärt. Stora språkmodeller kan snabbt generera innehållsvariationer i stor skala, vilket låter dig utforska ett brett spektrum av idéer och tillvägagångssätt utan att medföra betydande kostnader.

För att få ut det mesta av snabb iteration och experimentering är det viktigt att ha ett

väldefinierat experimenteringsramverk på plats. Detta ramverk bör innehålla:

- Tydliga mål och hypoteser för varje experiment
- Lämpliga mätetal och spårningsmekanismer för att mäta innehållets prestanda
- Segmenterings- och målgruppsstrategier för att säkerställa att relevanta innehållsvariationer levereras till rätt användare
- Analys- och rapporteringsverktyg för att utvinna insikter från experimentdata
- En process för att införliva lärdomar och optimeringar i din innehållsstrategi

Genom att omfamna snabb iteration och experimentering kan du kontinuerligt förfina och optimera ditt innehåll, vilket säkerställer att det förblir engagerande, relevant och effektivt för att uppnå din applikations mål. Detta agila tillvägagångssätt för innehållsskapande låter dig ligga steget före och leverera exceptionella användarupplevelser.

## Skalbarhet och effektivitet

När applikationer växer och efterfrågan på personanpassat innehåll ökar, möjliggör kontextuella innehållsgenereringsmönster effektiv skalning av innehållsskapande. Stora språkmodeller kan generera innehåll för ett stort antal användare och kontexter samtidigt, utan behov av en proportionell ökning av personalresurser. Denna skalbarhet gör det möjligt för applikationer att leverera personanpassade upplevelser till en växande användarbas utan att överbelasta deras innehållsskapande kapacitet.

Observera att kontextuell innehållsgenerering kan användas effektivt för att internationalisera din applikation "i farten". Det är faktiskt precis vad jag gjorde med min Instant18n Gem för att leverera Olympia på mer än ett halvdussin språk, trots att vi är mindre än ett år gamla.

# AI-driven lokalisering

Om ni tillåter mig att skryta ett ögonblick, så tror jag att mitt Instant18n-bibliotek för Rails-appar är ett banbrytande exempel på mönstret "Kontextuell innehållsgenerering" i praktiken, som visar den transformativa potentialen hos AI inom applikationsutveckling. Denna gem utnyttjar kraften i OpenAIs GPT stora språkmodell för att revolutionera hur internationalisering och lokalisering hanteras i Rails-applikationer.

Traditionellt sett innebär internationalisering av en Rails-applikation att manuellt definiera översättningsnycklar och tillhandahålla motsvarande översättningar för varje språk som stöds. Denna process kan vara tidskrävande, resurskrävande och benägen till inkonsekvenser. Med Instant18n-gemmen är dock lokaliseringsparadigmet helt omdefinierat.

Genom att integrera en stor språkmodell möjliggör Instant18n-gemmen generering av översättningar i realtid, baserat på textens kontext och innebörd. Istället för att förlita sig på fördefinierade översättningsnycklar och statiska översättningar översätter gemmen dynamiskt text med hjälp av AI. Detta tillvägagångssätt erbjuder flera viktiga fördelar:

1. **Sömlös lokalisering**: Med Instant18n-gemmen behöver utvecklare inte längre manuellt definiera och underhålla översättningsfiler för varje språk som stöds. Gemmen genererar automatiskt översättningar baserat på den tillhandahållna texten och det önskade målspråket, vilket gör lokaliseringsprocessen enkel och sömlös.
2. **Kontextuell noggrannhet**: AI kan ges tillräckligt med kontext för att förstå nyanserna i texten som översätts. Den kan ta hänsyn till den omgivande kontexten, idiom och kulturella referenser för att generera översättningar som är korrekta, naturligt klingande och kontextuellt lämpliga.
3. **Omfattande språkstöd**: Instant18n-gemmen utnyttjar GPTs omfattande kunskap och språkliga förmågor, vilket möjliggör översättningar till ett stort antal språk.

Från vanliga språk som spanska och franska till mer obskyra eller fiktiva språk som klingonska och alviska kan gemmen hantera en mängd olika översättningskrav.

4. **Flexibilitet och kreativitet**: Gemmen går längre än traditionella språköversättningar och möjliggör kreativa och okonventionella lokaliseringsalternativ. Utvecklare kan översätta text till olika stilar, dialekter eller till och med fiktiva språk, vilket öppnar upp nya möjligheter för unika användarupplevelser och engagerande innehåll.
5. **Prestandaoptimering**: Instant18n-gemmen innehåller cachningsmekanismer för att förbättra prestandan och minska belastningen av upprepade översättningar. Översatt text cachas, vilket gör att efterföljande förfrågningar för samma översättning kan hanteras snabbt utan behov av redundanta API-anrop.

Instant18n-gemmen exemplifierar kraften i mönstret "Kontextuell innehållsgenerering" genom att utnyttja AI för att generera lokaliserat innehåll dynamiskt. Den visar hur AI kan integreras i kärnfunktionaliteten hos en Rails-applikation och transformera hur utvecklare närmar sig internationalisering och lokalisering.

Genom att eliminera behovet av manuell översättningshantering och möjliggöra direktöversättningar baserade på kontext, sparar Instant18n-gemmet utvecklare betydande tid och ansträngning. Det låter dem fokusera på att bygga kärnfunktionerna i sin applikation samtidigt som lokaliseringsaspekten hanteras sömlöst och korrekt.

## Vikten av användartestning och återkoppling

Slutligen, kom alltid ihåg vikten av användartestning och återkoppling. Det är avgörande att validera att kontextbaserad innehållsgenerering möter användarnas förväntningar och ligger i linje med applikationens mål. Fortsätt att iterera och förfina genererat innehåll baserat på användarinsikter och analyser. Om du genererar dynamiskt innehåll i stor skala som skulle vara omöjligt att validera manuellt av dig och ditt team, överväg att lägga till återkopplingsmekanismer som låter användare

rapportera innehåll som är konstigt eller felaktigt, tillsammans med en förklaring till varför. Denna värdefulla återkoppling kan till och med matas in till en AI-arbetare som får i uppgift att göra justeringar i komponenten som genererade innehållet!

# Generative UI

Uppmärksamhet är så värdefull nu för tiden att effektivt användarengagemang kräver mjukvaruupplevelser som inte bara är sömlösa och intuitiva utan också högst personaliserade efter individuella behov, preferenser och sammanhang. Som ett resultat står designers och utvecklare inför utmaningen att skapa användargränssnitt som kan anpassa sig och tillgodose varje användares unika krav *i stor skala*.

Generative UI (GenUI) är ett verkligt revolutionerande tillvägagångssätt för design av användargränssnitt som utnyttjar kraften i stora språkmodeller (LLM) för att skapa högst personaliserade och dynamiska användarupplevelser i realtid. Jag ville säkerställa att åtminstone ge dig en introduktion till GenUI i denna bok, eftersom jag tror att det är en av de grönaste möjligheterna som för närvarande existerar inom området för applikationsdesign och ramverk. Jag är övertygad om att dussintals eller fler nya framgångsrika kommersiella och öppen källkods-projekt kommer att växa fram inom denna särskilda nisch.

I grunden kombinerar GenUI principerna för Kontextuell Innehållsgenerering med avancerade AI-tekniker för att generera användargränssnittselement, såsom text, bilder och layouter, dynamiskt baserat på en djup förståelse av användarens sammanhang, preferenser och mål. GenUI gör det möjligt för designers och utvecklare att skapa gränssnitt som anpassar sig och utvecklas som svar på användarinteraktioner, vilket ger en nivå av personalisering som tidigare var ouppnåelig.

GenUI representerar en fundamental förändring i hur vi närmar oss design av användargränssnitt. Istället för att designa för massorna låter GenUI oss designa för individen. Personaliserat innehåll och gränssnitt har potentialen att skapa användarupplevelser som resonerar med varje användare på en djupare nivå, vilket ökar engagemang, tillfredsställelse och lojalitet.

Som en banbrytande teknik är övergången till GenUI full av konceptuella och praktiska utmaningar. Att integrera AI i designprocessen, säkerställa att de genererade gränssnitten inte bara är personaliserade utan också användbara, tillgängliga och i linje med det övergripande varumärket och användarupplevelsen, alla dessa är utmaningar som gör GenUI till en strävan för de få, inte de många. Dessutom väcker involveringen av AI frågor om dataintegritet, transparens och till och med etiska implikationer.

Trots utmaningarna har personaliserade upplevelser i stor skala kraften att helt förändra hur vi interagerar med digitala produkter och tjänster. Det öppnar upp möjligheter för att skapa inkluderande och tillgängliga gränssnitt som tillgodoser användares olika behov, oavsett deras förmågor, bakgrund eller preferenser.

I detta kapitel kommer vi att utforska konceptet GenUI, undersöka några definierande egenskaper, viktiga fördelar och potentiella utmaningar. Vi börjar med att överväga den mest grundläggande och tillgängliga formen av GenUI: att generera textinnehåll för i övrigt traditionellt designade och implementerade användargränssnitt.

# Generering av text för användargränssnitt

Textelement som existerar i din applikations gränssnittselement, såsom formuläretiketter, verktygstips och förklarande text, är vanligtvis hårdkodade i mallarna eller UI-komponenterna, vilket ger en konsekvent men generisk upplevelse för alla användare. Genom att använda kontextuella innehållsgenereringsmönster kan du förvandla dessa statiska element till dynamiska, kontextmedvetna och personaliserade komponenter.

## Personaliserade formulär

Formulär är en allestädes närvarande del av webb- och mobilapplikationer och fungerar som det primära sättet att samla in användarinput. Traditionella formulär presenterar dock ofta en generisk och opersonlig upplevelse, med standardetiketter och fält som inte alltid överensstämmer med användarens specifika sammanhang eller behov. Användare är mer benägna att fylla i formulär som känns skräddarsydda efter deras behov och preferenser, vilket leder till högre konverteringsgrad och användartillfredsställelse.

Det är dock viktigt att hitta en balans mellan personalisering och konsekvens. Medan anpassning av formulär till enskilda användare kan vara fördelaktigt, är det avgörande att upprätthålla en nivå av igenkänning och förutsägbarhet. Användare ska fortfarande kunna känna igen och navigera i formulär enkelt, även med personaliserade element.

Här är några personaliserade formuläridéer för inspiration:

### Kontextuella fältförslag

GenUI kan analysera användarens tidigare interaktioner, preferenser och data för att tillhandahålla intelligenta fältförslag som förutsägelser. Om användaren till exempel tidigare har angett sin leveransadress kan formuläret automatiskt fylla i relevanta fält med deras sparade information. Detta sparar inte bara tid utan visar också att applikationen förstår och kommer ihåg användarens preferenser.

Vänta lite, är inte den här tekniken något som skulle kunna göras utan att involvera AI? Jovisst, men det fina med att driva den här typen av funktionalitet med AI är tvåfaldigt: 1) hur enkelt det kan vara att implementera och 2) hur motståndskraftigt det kan vara när din UI förändras och utvecklas över tid.

Låt oss snabbt sätta ihop en tjänst för vårt teoretiska orderhanteringssystem, som försöker proaktivt fylla i rätt leveransadress åt användaren.

```
class OrderShippingAddressSubscriber
  include Raix::ChatCompletion

  attr_accessor :order

  delegate :customer, to: :order

  DIRECTIVE = "You are a smart order processing assistant. Given the
  customer's order history, guess the most likely shipping address
  for the current order."

  def order_created(order)
    return unless order.pending? && order.shipping_address.blank?

    self.order = order

    transcript.clear
    transcript << { system: DIRECTIVE }
    transcript << { user: "Order History: #{order_history.to_json}" }
    transcript << { user: "Current Order: #{order.to_json}" }

    response = chat_completion
    apply_predicted_shipping_address(order, response)
  end

  private

  def apply_predicted_shipping_address(order, response)
    # extract the shipping address from the response...
    # ...and assume there's some sort of live update of the address fields
    order.update(shipping_address:)
  end
```

```
33
34    def order_history
35      customer.orders.successful.limit(100).map do |order|
36        {
37          date: order.date,
38          description: order.description,
39          shipping_address: order.shipping_address
40        }
41      end
42    end
43  end
```

Detta exempel är mycket förenklat men bör fungera i de flesta fall. Tanken är att låta AI:n gissa på samma sätt som en människa skulle göra. För att förtydliga vad jag menar ska vi titta på lite exempeldata:

```
1  Order History:
2  [
3    {"date": "2024-01-03", "description": "garden soil mix",
4     "shipping_address": "123 Country Lane, Rural Town"},
5    {"date": "2024-01-15", "description": "hardcover fiction novels",
6     "shipping_address": "456 City Apt, Metroville"},
7    {"date": "2024-01-22", "description": "baby diapers", "shipping_address":
8     "789 Suburb St, Quietville"},
9    {"date": "2024-02-01", "description": "organic vegetables",
10    "shipping_address": "123 Country Lane, Rural Town"},
11   {"date": "2024-02-17", "description": "mystery thriller book set",
12    "shipping_address": "456 City Apt, Metroville"},
13   {"date": "2024-02-25", "description": "baby wipes",
14    "shipping_address": "789 Suburb St, Quietville"},
15   {"date": "2024-03-05", "description": "flower seeds",
16    "shipping_address": "123 Country Lane, Rural Town"},
17   {"date": "2024-03-20", "description": "biographies",
18    "shipping_address": "456 City Apt, Metroville"},
19   {"date": "2024-03-30", "description": "baby formula",
20    "shipping_address": "789 Suburb St, Quietville"},
21   {"date": "2024-04-12", "description": "lawn fertilizer",
22    "shipping_address": "123 Country Lane, Rural Town"},
23   {"date": "2024-04-22", "description": "science fiction novels",
24    "shipping_address": "456 City Apt, Metroville"},
```

```
  {"date": "2024-05-02", "description": "infant toys",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-05-14", "description": "outdoor grill",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-05-29", "description": "literary classics",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-06-11", "description": "baby clothes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-07-01", "description": "watering can",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-07-18", "description": "non-fiction essays",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-07-28", "description": "baby bath items",
  "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-08-09", "description": "herb garden kit",
  "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-08-24", "description": "children's books",
  "shipping_address": "456 City Apt, Metroville"}
]
```

Lade du märke till mönstret i datan? Jag garanterar att detta är enkelt för en LLM. För att demonstrera detta, låt oss fråga GPT-4 vad den mest sannolika leveransadressen är för en “termometer”.

```
From the order history you've provided, it looks like the purchases are
generally clustered into three main types based on the shipping addresses:

123 Country Lane, Rural Town - This address often orders garden and
outdoor-related items like soil mix, vegetables, flower seeds, lawn
fertilizer, an outdoor grill, a watering can, and an herb garden kit.

456 City Apt, Metroville - This address tends to order various types of
books, indicating an interest in reading across genres like fiction,
thrillers, biographies, science fiction, literary classics, non-fiction
essays, and children's books.

789 Suburb St, Quietville - The purchases for this address are consistently
baby-related products, such as diapers, wipes, baby formula, infant toys,
baby clothes, and baby bath items.

```

```
Given these patterns, a digital thermometer could potentially be a fit for
any of the addresses based on different needs: outdoor temperature tracking,
a tool for a passionate reader interested in science gadgets, or a health
tool for a baby. However, considering its common use in monitoring health,
especially for babies, it's most likely to be purchased by someone who's
already buying baby care products.

Therefore, the most likely shipping address for a digital thermometer, based
on the purchasing patterns, would be 789 Suburb St, Quietville.
```

Om du tänker att det är överdrivet att använda en dyr modell som GPT-4 för denna uppgift har du rätt! Jag testade samma prompt på Mistral 7B Instruct och den producerade följande svar med 75 tokens per sekund, till en försvinnande liten kostnad av $0.000218 USD.

```
Based on the given order history, it appears that the customer has received
orders for baby-related items (diapers, wipes, formula, clothes, bath items)
at the address "789 Suburb St, Quietville" and garden-related items (soil,
seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
Town". The customer has also received books (fiction, non-fiction, literary
classics, mystery-thriller, science fiction, biographies, children's books)
at the address "456 City Apt, Metroville".

Given that the new order is for a "thermometer", which doesn't seem to fit
into any of the categories that the customer has previously ordered, it's
hard to accurately predict the shipping address based solely on the order
history. However, given the 50%-50% split between baby-related and
garden-related items, it could somewhat lean towards the Baby-related items
address ("789 Suburb St, Quietville"). But remember, this is an assumption
and cannot be definitively confirmed without more context or information.
```

Är overheaden och kostnaden för denna teknik värd det för att göra en utcheckningsupplevelse mer magisk? För många nätbutiker, absolut. Och av allt att döma kommer kostnaden för AI-beräkningar bara att sjunka, särskilt för leverantörer av öppen källkods-modellvärdtjänster i en kapplöpning mot botten.

Använd en Prompt Template och StructuredIO tillsammans med Response Fencing för att optimera denna typ av chattkomplettering.

## Adaptiv fältordning

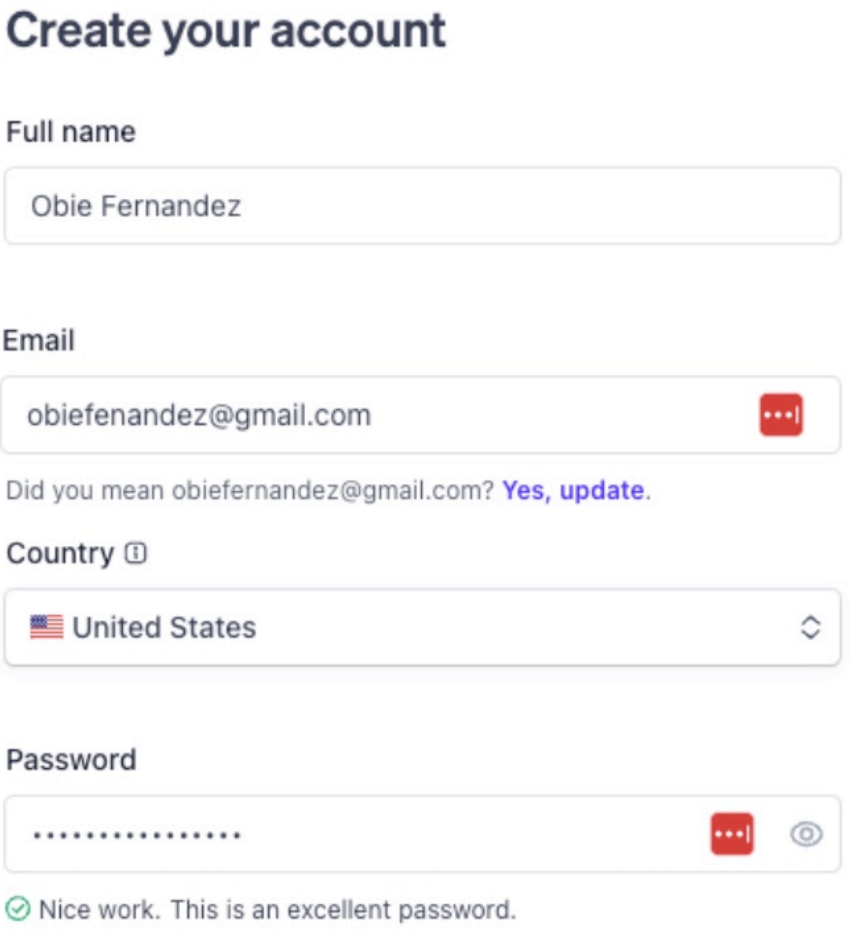
Ordningen i vilken formulärfält presenteras kan avsevärt påverka användarupplevelsen och slutförandegraden. Med GenUI kan du dynamiskt justera fältordningen baserat på användarens kontext och betydelsen av varje fält. Om användaren till exempel fyller i ett registreringsformulär för en träningsapp kan formuläret prioritera fält relaterade till deras träningsmål och preferenser, vilket gör processen mer relevant och engagerande.

## Personanpassad microcopy

Instruktionstexten, felmeddelanden och annan microcopy kopplad till formulär kan också personanpassas med hjälp av GenUI. Istället för att visa generiska felmeddelanden som "Ogiltig e-postadress" kan du generera mer hjälpsamma och kontextuella meddelanden som "Vänligen ange en giltig e-postadress för att få din orderbekräftelse." Dessa personliga detaljer kan göra formulärupplevelsen mer användarvänlig och mindre frustrerande.

## Personanpassad validering

I samma anda som Personanpassad Microcopy, skulle du kunna använda AI för att validera formuläret på sätt som verkar magiska. Föreställ dig att låta en AI validera ett användarprofilformulär och leta efter potentiella misstag på en *semantisk* nivå.




**Figur 9. Kan du upptäcka den semantiska valideringen som sker?**

## Progressiv presentation

GenUI kan intelligent avgöra vilka formulärfält som är väsentliga baserat på användarens kontext och gradvis visa ytterligare fält efter behov. Denna teknik för progressiv presentation hjälper till att minska den kognitiva belastningen och gör formulärifyllningen mer hanterbar. Om en användare till exempel registrerar sig för ett

grundläggande abonnemang kan formuläret till en början bara visa de viktigaste fälten, och när användaren fortskrider eller väljer specifika alternativ kan ytterligare relevanta fält introduceras dynamiskt.

## Kontextmedveten förklarande text

Tooltips används ofta för att ge ytterligare information eller vägledning till användare när de hovrar över eller interagerar med specifika element. Med en "Kontextuell innehållsgenerering"-approach kan du generera tooltips som anpassar sig efter användarens kontext och tillhandahåller relevant information. Om en användare till exempel utforskar en komplex funktion kan tooltip erbjuda personanpassade tips eller exempel baserade på deras tidigare interaktioner eller kunskapsnivå.

Förklarande text, såsom instruktioner, beskrivningar eller hjälpmeddelanden, kan genereras dynamiskt baserat på användarens kontext. Istället för att presentera generiska förklaringar kan du använda LLMs för att generera text som är skräddarsydd efter användarens specifika behov eller frågor. Om en användare till exempel har problem med ett särskilt steg i en process kan den förklarande texten ge personanpassad vägledning eller felsökningstips.

Microcopy avser de små textstycken som vägleder användare genom din applikation, såsom knappetiketter, felmeddelanden eller bekräftelseuppmaningar. Genom att tillämpa Kontextuell innehållsgenerering-approachen på microcopy kan du skapa ett adaptivt UI som svarar på användarens handlingar och tillhandahåller relevant och hjälpsam text. Om en användare till exempel är på väg att utföra en kritisk åtgärd kan bekräftelseuppmaningen genereras dynamiskt för att ge ett tydligt och personanpassat meddelande.

Personanpassad förklarande text och tooltips kan avsevärt förbättra onboarding-processen för nya användare. Genom att tillhandahålla kontextspecifik vägledning och exempel kan du hjälpa användare att snabbt förstå och navigera i applikationen, vilket minskar inlärningskurvan och ökar användningen.

Dynamiska och kontextmedvetna chrome-element kan också göra applikationen mer intuitiv och engagerande. Användare är mer benägna att interagera med och utforska funktioner när den medföljande texten är anpassad efter deras specifika behov och intressen.

Hittills har vi täckt idéer för att förbättra befintliga användargränssnittsparadigm med AI, men vad sägs om att tänka om hur användargränssnitt designas och implementeras på ett mer radikalt sätt?

## Definiera generativt användargränssnitt

Till skillnad från traditionell användargränssnittsdesign, där designers skapar fasta, statiska gränssnitt, antyder GenUI en framtid där vår programvara har flexibla, personaliserade upplevelser som kan utvecklas och anpassas i realtid. Varje gång vi använder ett AI-drivet konversationsgränssnitt låter vi AI:n anpassa sig till användarens specifika behov. GenUI tar saker ett steg längre genom att tillämpa den nivån av anpassningsbarhet på programvarans *visuella* gränssnitt.

Anledningen till att det är möjligt att experimentera med GenUI-idéer idag är att stora språkmodeller redan förstår programmering och deras baskunskaper omfattar UI-teknologier och ramverk. Frågan är således om stora språkmodeller kan användas för att generera UI-element, såsom text, bilder, layouter och till och med hela gränssnitt, som är skräddarsydda för varje enskild användare. Modellen skulle kunna instrueras att ta hänsyn till olika faktorer, såsom användarens tidigare interaktioner, angivna preferenser, demografisk information och den aktuella användningskontexten, för att skapa mycket personaliserade och relevanta gränssnitt.

GenUI skiljer sig från traditionell användargränssnittsdesign på flera viktiga sätt:

1. **Dynamiskt och adaptivt**: Traditionell UI-design innebär att skapa fasta, statiska gränssnitt som förblir desamma för alla användare. GenUI möjliggör däremot gränssnitt som dynamiskt kan anpassa sig och förändras baserat på användarbehov och kontext. Detta innebär att samma applikation kan presentera olika gränssnitt för olika användare eller till och med för samma användare i olika situationer.
2. **Personalisering i stor skala**: Med traditionell design är det ofta opraktiskt att skapa personaliserade upplevelser för varje användare på grund av den tid och de resurser som krävs. GenUI däremot, möjliggör personalisering i stor skala. Genom att utnyttja AI kan designers skapa gränssnitt som automatiskt anpassar sig till varje användares unika behov och preferenser, utan att behöva manuellt designa och utveckla separata gränssnitt för varje användarsegment.
3. **Fokus på resultat**: Traditionell UI-design fokuserar ofta på att skapa visuellt tilltalande och funktionella gränssnitt. Medan dessa aspekter fortfarande är viktiga i GenUI, flyttas det primära fokuset mot att uppnå önskade användarresultat. GenUI syftar till att skapa gränssnitt som är optimerade för varje användares specifika mål och uppgifter, där användbarhet och effektivitet prioriteras framför rent estetiska överväganden.
4. **Kontinuerligt lärande och förbättring**: GenUI-system kan kontinuerligt lära sig och förbättras över tid baserat på användarinteraktioner och feedback. När användare interagerar med de genererade gränssnitten kan AI-modellerna samla data om användarbeteende, preferenser och resultat, och använda denna information för att förfina och optimera framtida gränssnittsgenereringar. Denna iterativa inlärningsprocess gör att GenUI-system kan bli allt effektivare på att möta användarbehov över tid.

Det är viktigt att notera att GenUI inte är detsamma som AI-assisterade designverktyg, såsom de som ger förslag eller automatiserar vissa designuppgifter. Medan dessa verktyg kan vara användbara för att effektivisera designprocessen, förlitar de sig fortfarande på designers för att fatta slutgiltiga beslut och skapa statiska gränssnitt. GenUI innebär

däremot att AI-systemet tar en mer aktiv roll i själva genereringen och anpassningen av gränssnitt baserat på användardata och kontext.

GenUI representerar ett betydande skifte i hur vi närmar oss användargränssnittsdesign, där vi rör oss bort från universallösningar och mot mycket personaliserade, adaptiva upplevelser. Genom att utnyttja kraften i AI har GenUI potentialen att revolutionera hur vi interagerar med digitala produkter och tjänster, genom att skapa gränssnitt som är mer intuitiva, engagerande och effektiva för varje enskild användare.

# Exempel

För att illustrera konceptet GenUI, låt oss överväga en hypotetisk träningsapplikation kallad "FitAI". Denna app syftar till att tillhandahålla personaliserade träningsplaner och kostråd till användare baserat på deras individuella mål, konditionsnivåer och preferenser.

I en traditionell UI-designapproach skulle FitAI kunna ha en fast uppsättning skärmar och element som är desamma för alla användare. Med GenUI skulle appens gränssnitt däremot kunna anpassa sig dynamiskt till varje användares unika behov och kontext.

Detta tillvägagångssätt är något av en utmaning att föreställa sig implementera under 2024 och kanske inte ens har tillräcklig avkastning på investeringen, men det är möjligt.

Här är hur det skulle kunna fungera:

1. **Introduktionsprocess**:
    - Istället för ett standardfrågeformulär använder FitAI en konversations-AI för att samla information om användarens mål, nuvarande konditionsnivå och preferenser.
    - Baserat på denna initiala interaktion genererar AI:n en personaliserad instrumentpanelslayout som framhäver de funktioner och den information som är mest relevant för användarens mål.

- Nuvarande AI-teknologi kan ha ett urval av skärmkomponenter till sitt förfogande för att komponera den personaliserade instrumentpanelen.
- Framtida AI-teknologi kan ta på sig rollen som en erfaren UI-designer och faktiskt skapa instrumentpanelen *från grunden*.

2. **Träningsplanerare**:

   - Träningsplanerarens gränssnitt anpassas av AI:n specifikt efter användarens erfarenhetsnivå och tillgänglig utrustning.
   - För en nybörjare utan utrustning kan den visa enkla kroppsövningar med detaljerade instruktioner och videor.
   - För en avancerad användare med tillgång till ett gym kan den visa mer komplexa rutiner med mindre förklarande innehåll.
   - Innehållet i träningsplaneraren filtreras inte bara från en stor överuppsättning. Det kan genereras *i realtid* baserat på en kunskapsbas som efterfrågas med kontext som inkluderar allt som är känt om användaren.

3. **Framstegsspårning**:

   - Framstegsspårningens gränssnitt utvecklas baserat på användarens mål och engagemangsmönster.
   - Om en användare främst fokuserar på viktminskning kan gränssnittet framträdande visa en vikttrendgraf och statistik över kaloriförbränning.
   - För en användare som bygger muskler kan det framhäva styrkeökningar och förändringar i kroppssammansättning.
   - AI:n kan anpassa denna del av applikationen efter användarens faktiska framsteg. Om framstegen stannar av under en period kan appen övergå till ett läge där den försöker locka användaren att avslöja orsakerna till bakslaget, för att kunna motverka dem.

4. **Kostråd**:

   - Kostdelen anpassar sig efter användarens kostpreferenser och restriktioner.

- För en vegansk användare kan den visa växtbaserade måltidsförslag och proteinkällor.
- För en användare med glutenintolerans skulle den automatiskt filtrera bort gluteninnehållande livsmedel från rekommendationerna.
- Återigen är innehållet inte hämtat från en massiv överuppsättning av måltidsdata som gäller alla användare, utan syntetiseras snarare från en kunskapsbas som innehåller information som kan anpassas baserat på användarens specifika situation och begränsningar.
- Till exempel genereras recept med ingrediensspecifikationer som matchar användarens ständigt föränderliga kaloribehov när deras konditionsnivå och kroppsstatistik utvecklas.

5. **Motiverande element**:

    - Appens motiverande innehåll och aviseringar är personanpassade baserat på användarens personlighetstyp och respons på olika motivationsstrategier.
    - Vissa användare kan få uppmuntrande meddelanden, medan andra får mer datadriven återkoppling.

I detta exempel möjliggör GenUI för FitAI att skapa en mycket anpassad upplevelse för varje användare, vilket potentiellt ökar engagemang, tillfredsställelse och sannolikheten att uppnå träningsmål. Gränssnittselementen, innehållet och till och med appens "personlighet" anpassar sig för att bäst tjäna varje enskild användares behov och preferenser.

## Skiftet till resultatinriktad design

GenUI representerar ett fundamentalt skifte i tillvägagångssättet för användargränssnittsdesign, från ett fokus på att skapa specifika gränssnittselement till ett mer holistiskt, resultatinriktat tillvägagångssätt. Detta skifte har flera viktiga konsekvenser:

1. **Fokus på användarmål**:

    - Designers behöver tänka djupare på användarmål och önskade resultat snarare än specifika gränssnittskomponenter.
    - Tonvikten kommer att ligga på att skapa system som kan generera gränssnitt som hjälper användare att uppnå sina mål effektivt.
    - Nya UI-ramverk kommer att växa fram som ger AI-baserade designers de verktyg de behöver för att kunna generera användarupplevelser *i realtid* och *från grunden* istället för baserat på fördefinierade skärmspecifikationer.

2. **Förändrad roll för designers**:

    - Designers kommer att övergå från att skapa fasta layouter till att definiera regler, begränsningar och riktlinjer för AI-system att följa när de genererar gränssnitt.
    - De kommer att behöva utveckla färdigheter inom områden som dataanalys, AI-promptkonstruktion och systemtänkande för att effektivt vägleda GenUI-system.

3. **Betydelsen av användarforskning**:

    - Användarforskning blir ännu viktigare i en GenUI-kontext, eftersom designers behöver förstå inte bara användarpreferenser, utan också hur dessa preferenser och behov förändras i olika sammanhang.
    - Kontinuerlig användartestning och återkopplingsloopar kommer att vara avgörande för att förfina och förbättra AI:ns förmåga att generera effektiva gränssnitt.

4. **Design för variabilitet**:

    - Istället för att skapa ett enda "perfekt" gränssnitt kommer designers att behöva överväga flera möjliga variationer och säkerställa att systemet kan generera lämpliga gränssnitt för olika användarbehov.

- Detta inkluderar design för kantfall och att säkerställa att de genererade gränssnitten bibehåller användbarhet och tillgänglighet över olika konfigurationer.
- Produktdifferentiering får nya dimensioner som involverar divergerande perspektiv på användarpsykologi och utnyttjandet av unika datamängder och kunskapsbaser som inte är tillgängliga för konkurrenter.

## Utmaningar och överväganden

Medan GenUI erbjuder spännande möjligheter medför det också flera utmaningar och överväganden:

1. **Tekniska begränsningar**:
    - Nuvarande AI-teknik, även om den är avancerad, har fortfarande begränsningar i att förstå komplexa användaravsikter och generera verkligt kontextmedvetna gränssnitt.
    - Prestandaproblem relaterade till realtidsgenerering av gränssnittselement, särskilt på mindre kraftfulla enheter.
2. **Datakrav**:
    - Beroende på användningsområde kan effektiva GenUI-system kräva betydande mängder användardata för att generera personaliserade gränssnitt.
    - Utmaningarna med att etiskt införskaffa autentisk användardata väcker oro kring dataintegritet och säkerhet, samt potentiella partiskheter i den data som används för att träna GenUI-modeller.
3. **Användbarhet och Konsekvens**:

- Åtminstone tills metoden blir utbredd kan en applikation med ständigt föränderliga gränssnitt leda till användbarhetsproblem, eftersom användare kan ha svårt att hitta välbekanta element eller navigera effektivt.
- Det kommer vara avgörande att hitta en balans mellan personalisering och att upprätthålla ett konsekvent, lärbart gränssnitt.

4. **Överberonde av AI**:

    - Det finns en risk att för mycket designbeslut överlåts till AI-system, vilket potentiellt kan leda till oinspirerade, problematiska eller helt enkelt trasiga gränssnittsval.
    - Mänsklig övervakning och möjligheten att åsidosätta AI-genererade designer kommer förbli viktigt inom överskådlig framtid.

5. **Tillgänglighetsfrågor**:

    - Att säkerställa att dynamiskt genererade gränssnitt förblir tillgängliga för användare med funktionsnedsättningar presenterar helt nya utmaningar, vilket är oroande med tanke på den låga nivån av tillgänglighetsefterlevnad som typiska system uppvisar.
    - Å andra sidan kan AI-designers implementeras med *inbyggd* hänsyn till tillgänglighet och förmågor att bygga tillgängliga gränssnitt i realtid precis som de bygger UI för användare utan funktionsnedsättningar.
    - Oavsett bör GenUI-system utformas med robusta tillgänglighetsriktlinjer och testprocesser.

6. **Användarförtroende och Transparens**:

    - Användare kan känna sig obekväma med gränssnitt som verkar "veta för mycket" om dem eller förändras på sätt de inte förstår.
    - Att tillhandahålla transparens om hur och varför gränssnitt personaliseras kommer vara viktigt för att bygga användarförtroende.

# Framtidsutsikter och Möjligheter

Framtiden för Generative UI (GenUI) rymmer enorma löften om att revolutionera hur vi interagerar med digitala produkter och tjänster. När denna teknik fortsätter att utvecklas kan vi förvänta oss en omvälvande förändring i hur användargränssnitt designas, implementeras och upplevs. Jag tror att GenUI är det fenomen som äntligen kommer att föra vår programvara in i det område som nu betraktas som science fiction.

En av de mest spännande aspekterna av GenUI är dess potential att förbättra tillgängligheten i en omfattning som går längre än att bara säkerställa att personer med allvarliga funktionsnedsättningar inte helt utestängs från användningen av din programvara. Genom att automatiskt anpassa gränssnitt till individuella användarbehov skulle GenUI kunna göra digitala upplevelser mer inkluderande än någonsin tidigare. Föreställ dig gränssnitt som sömlöst justerar sig för att ge större text för yngre eller synskadade användare, eller förenklade layouter för personer med kognitiva funktionsnedsättningar, allt utan att kräva manuell konfiguration eller separata "tillgängliga" versioner av applikationer.

Personaliseringsmöjligheterna hos GenUI kommer sannolikt att driva ökad användarengagemang, tillfredsställelse och lojalitet över ett brett spektrum av digitala produkter. När gränssnitt blir mer anpassade till individuella preferenser och beteenden kommer användare att finna digitala upplevelser mer intuitiva och njutbara, vilket potentiellt leder till djupare och mer meningsfulla interaktioner med teknologi.

GenUI har också potential att förändra introduktionsprocessen för nya användare. Genom att skapa intuitiva, personaliserade användarupplevelser för förstagångsanvändare som snabbt anpassar sig till varje användares kompetensnivå, skulle GenUI kunna minska inlärningskurvan för nya applikationer avsevärt. Detta skulle kunna leda till snabbare adoptionsgrad och ökat användarförtroende i att utforska nya funktioner.

En annan spännande möjlighet är GenUI:s förmåga att upprätthålla en konsekvent

användarupplevelse över olika enheter och plattformar samtidigt som den optimerar för varje specifik användningskontext. Detta skulle kunna lösa den långvariga utmaningen att tillhandahålla sammanhängande upplevelser över ett alltmer fragmenterat enhetslandskap, från smartphones och surfplattor till stationära datorer och framväxande teknologier som AR-glasögon.

Den datadrivna naturen hos GenUI öppnar möjligheter för snabb iteration och förbättring inom UI-design. Genom att samla realtidsdata om hur användare interagerar med genererade gränssnitt kan designers och utvecklare få tidigare osedda insikter i användarbeteende och preferenser. Denna återkopplingsslinga skulle kunna leda till kontinuerliga förbättringar i UI-design, drivna av faktiska användningsmönster snarare än antaganden eller begränsad användartestning.

För att förbereda sig för denna förändring kommer designers behöva utveckla sina färdigheter och tankesätt. Fokus kommer att skifta från att skapa fasta layouter till att utveckla omfattande designsystem och riktlinjer som kan informera AI-driven gränssnittsgenering. Designers kommer behöva utveckla en djup förståelse för dataanalys, AI-teknologier och systemtänkande för att effektivt vägleda GenUI-system.

Dessutom, när GenUI suddar ut gränserna mellan design och teknologi, kommer designers behöva samarbeta närmare med utvecklare och datavetare. Detta tvärvetenskapliga tillvägagångssätt kommer vara avgörande för att skapa GenUI-system som inte bara är visuellt tilltalande och användarvänliga utan också tekniskt robusta och etiskt sunda.

De etiska konsekvenserna av GenUI kommer också att hamna i förgrunden i takt med att tekniken mognar. Designers kommer att spela en avgörande roll i utvecklingen av ramverk för ansvarsfull AI-användning inom gränssnittsdesign, för att säkerställa att personalisering förbättrar användarupplevelser utan att kompromissa med integriteten eller manipulera användarbeteende på oetiska sätt.

När vi blickar mot framtiden presenterar GenUI både spännande möjligheter och betydande utmaningar. Tekniken har potential att skapa mer intuitiva, effektiva och

tillfredsställande digitala upplevelser för användare över hela världen. Även om det kommer att kräva att designers anpassar sig och förvärvar nya färdigheter, erbjuder det också en aldrig tidigare skådad möjlighet att forma framtiden för människa-datorinteraktion på djupgående och meningsfulla sätt. Resan mot fullt utvecklade GenUI-system kommer utan tvekan att vara komplex, men de potentiella fördelarna i form av förbättrade användarupplevelser och digital tillgänglighet gör det till en framtid värd att sträva efter.

# Intelligent arbetsflödesorkestring

Inom området för applikationsutveckling spelar arbetsflöden en avgörande roll för att definiera hur uppgifter, processer och användarinteraktioner struktureras och utförs. I takt med att applikationer blir mer komplexa och användarnas förväntningar fortsätter att öka, blir behovet av intelligent och adaptiv arbetsflödesorkestring allt mer uppenbart.

Metoden “Intelligent arbetsflödesorkestring” fokuserar på att utnyttja AI-komponenter för att dynamiskt orkestrera och optimera komplexa arbetsflöden inom applikationer. Målet är att skapa applikationer som är mer effektiva, responsiva och anpassningsbara till realtidsdata och kontext.

I detta kapitel kommer vi att utforska de viktigaste principerna och mönstren som ligger till grund för den intelligenta arbetsflödesorkestreringsmetoden. Vi kommer att

undersöka hur AI kan användas för att intelligent dirigera uppgifter, automatisera beslutsfattande och dynamiskt anpassa arbetsflöden baserat på olika faktorer som användarbeteende, systemprestanda och affärsregler. Genom praktiska exempel och verkliga scenarier kommer vi att demonstrera AI:s transformativa potential för att effektivisera och optimera applikationers arbetsflöden.

Oavsett om du bygger företagsapplikationer med komplexa affärsprocesser eller konsumentinriktade applikationer med dynamiska användarresor, kommer mönstren och teknikerna som diskuteras i detta kapitel att ge dig kunskapen och verktygen för att skapa intelligenta och effektiva arbetsflöden som förbättrar den övergripande användarupplevelsen och driver affärsvärde.

## Affärsbehov

Traditionella metoder för arbetsflödeshantering förlitar sig ofta på fördefinierade regler och statiska beslutsträd, som kan vara stela, oflexibla och oförmögna att hantera moderna applikationers dynamiska natur.

Tänk dig ett scenario där en e-handelsapplikation behöver hantera en komplex orderhanteringsprocess. Arbetsflödet kan innefatta flera steg som ordervalidering, lagerkontroll, betalningshantering, leverans och kundmeddelanden. Varje steg kan ha sina egna regler, beroenden, externa integrationer och undantagshanteringsmekanismer. Att hantera ett sådant arbetsflöde manuellt eller genom hårdkodad logik kan snabbt bli besvärligt, felbenäget och svårt att underhålla.

När applikationen dessutom skalar upp och antalet samtidiga användare växer kan arbetsflödet behöva anpassa och optimera sig själv baserat på realtidsdata och systemprestanda. Under perioder med hög trafik kan applikationen till exempel behöva dynamiskt justera arbetsflödet för att prioritera vissa uppgifter, allokera resurser effektivt och säkerställa en smidig användarupplevelse.

Det är här metoden “Intelligent arbetsflödesorkestring” kommer in i bilden. Genom

att utnyttja AI-komponenter kan utvecklare skapa arbetsflöden som är intelligenta, adaptiva och självoptimerande. AI kan analysera stora mängder data, lära sig av tidigare erfarenheter och fatta välgrundade beslut i realtid för att orkestrera arbetsflödet effektivt.

## Huvudsakliga fördelar

1. **Ökad effektivitet**: AI kan optimera uppgiftsallokering, resursutnyttjande och arbetsflödesutförande, vilket leder till snabbare bearbetningstider och förbättrad övergripande effektivitet.
2. **Anpassningsbarhet**: AI-drivna arbetsflöden kan dynamiskt anpassa sig till förändrade förhållanden, såsom fluktuationer i användarefterfrågan, systemprestanda eller affärskrav, vilket säkerställer att applikationen förblir responsiv och motståndskraftig.
3. **Automatiserat beslutsfattande**: AI kan automatisera komplexa beslutsprocesser inom arbetsflödet, vilket minskar manuell intervention och minimerar risken för mänskliga fel.
4. **Personalisering**: AI kan analysera användarbeteende, preferenser och kontext för att personalisera arbetsflödet och leverera skräddarsydda upplevelser till enskilda användare.
5. **Skalbarhet**: AI-drivna arbetsflöden kan sömlöst skala för att hantera ökande volymer av data och användarinteraktioner, utan att kompromissa med prestanda eller tillförlitlighet.

I följande avsnitt kommer vi att utforska de viktigaste mönstren och teknikerna som möjliggör implementering av intelligenta arbetsflöden och visa verkliga exempel på hur AI transformerar arbetsflödeshantering i moderna applikationer.

# Viktiga mönster

För att implementera intelligent arbetsflödesorkestring i applikationer kan utvecklare utnyttja flera viktiga mönster som tar vara på AI:s kraft. Dessa mönster ger ett strukturerat tillvägagångssätt för att designa och hantera arbetsflöden, vilket gör det möjligt för applikationer att anpassa, optimera och automatisera processer baserat på realtidsdata och kontext. Låt oss utforska några av de grundläggande mönstren inom intelligent arbetsflödesorkestring.

## Dynamisk uppgiftsdirigering

Detta mönster innebär att använda AI för att intelligent dirigera uppgifter inom ett arbetsflöde baserat på olika faktorer såsom uppgiftsprioritet, resurstillgänglighet och systemprestanda. AI-algoritmer kan analysera egenskaperna hos varje uppgift, ta hänsyn till systemets aktuella tillstånd och fatta välgrundade beslut för att tilldela uppgifter till de mest lämpliga resurserna eller bearbetningsvägarna. Dynamisk uppgiftsdirigering säkerställer att uppgifter distribueras och utförs effektivt, vilket optimerar den övergripande arbetsflödesprestandan.

```
class TaskRouter
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  attr_accessor :task

  # list of functions that can be called by the AI entirely at its
  # discretion depending on the task received

  function :analyze_task_priority do
    TaskPriorityAnalyzer.perform(task)
  end

  function :check_resource_availability, # ...
  function :assess_system_performance, # ...
```

```
16    function :assign_task_to_resource, # ...
17
18    DIRECTIVE = "You are a task router, responsible for intelligently
19     assigning tasks to available resources based on priority, resource
20     availability, and system performance..."
21
22    def initialize(task)
23      self.task = task
24      transcript << { system: DIRECTIVE }
25      transcript << { user: task.to_json }
26    end
27
28    def perform
29      while task.unassigned?
30        chat_completion
31
32        # todo: add max loop counter and break
33      end
34
35      # capture the transcript for later analysis
36      task.update(routing_transcript: transcript)
37    end
38  end
```

Observera loopen som skapas av `while`-uttrycket på rad 29, som fortsätter att fråga AI:n tills uppgiften är tilldelad. På rad 35 sparar vi transkriptet för uppgiften för senare analys och felsökning, om det skulle bli nödvändigt.

## Kontextbaserat beslutsfattande

Du kan använda mycket liknande kod för att fatta kontextmedvetna beslut inom ett arbetsflöde. Genom att analysera relevanta datapunkter som användares preferenser, historiska mönster och realtidsindata kan AI-komponenter bestämma den mest lämpliga handlingsvägen vid varje beslutspunkt i arbetsflödet. Anpassa ditt arbetsflödes beteende baserat på den specifika kontexten för varje användare eller scenario, vilket ger personaliserade och optimerade upplevelser.

### Adaptiv arbetsflödeskomposition

Detta mönster fokuserar på att dynamiskt komponera och justera arbetsflöden baserat på föränderliga krav eller förhållanden. AI kan analysera arbetsflödets aktuella tillstånd, identifiera flaskhalsar eller ineffektivitet, och automatiskt modifiera arbetsflödets struktur för att optimera prestandan. Adaptiv arbetsflödeskomposition tillåter applikationer att kontinuerligt utvecklas och förbättra sina processer utan att kräva manuella ingrepp.

## Undantagshantering och återhämtning

Undantagshantering och återhämtning är kritiska aspekter av intelligent arbetsflödesorkestrering. När man arbetar med AI-komponenter och komplexa arbetsflöden är det viktigt att förutse och hantera undantag på ett elegant sätt för att säkerställa systemets stabilitet och tillförlitlighet.

Här är några viktiga överväganden och tekniker för undantagshantering och återhämtning i intelligenta arbetsflöden:

1. **Undantagspropagering**: Implementera ett konsekvent tillvägagångssätt för att propagera undantag mellan arbetsflödeskomponenter. När ett undantag inträffar inom en komponent bör det fångas upp, loggas och propageras till orkestratorn eller en separat komponent som ansvarar för att hantera undantag. Idén är att centralisera undantagshanteringen och förhindra att undantag tystas ner i tysthet, samt öppna möjligheter för Intelligent Felhantering.
2. **Återförsöksmekanismer**: Återförsöksmekanismer hjälper till att förbättra arbetsflödets motståndskraft och hantera tillfälliga fel på ett elegant sätt. Se definitivt till att implementera återförsöksmekanismer för övergående eller återhämtningsbara undantag, såsom problem med nätverksanslutning eller resurstillgänglighet som automatiskt kan provas igen efter en angiven

fördröjning. Att ha en AI-driven orkestrator eller undantagshanterare innebär att dina återförsöksstrategier inte behöver vara mekaniska till sin natur och förlita sig på fasta algoritmer som exponentiell tillbakagång. Du kan överlåta hanteringen av återförsöket till AI-komponentens "omdöme" som ansvarar för att bestämma hur undantaget ska hanteras.

3. **Reservstrategier**: Om en AI-komponent misslyckas med att ge ett giltigt svar eller stöter på ett fel—något som är vanligt förekommande med tanke på dess experimentella natur—ha en reservmekanism på plats för att säkerställa att arbetsflödet kan fortsätta. Detta kan innebära att använda standardvärden, alternativa algoritmer eller en Människa i loopen för att fatta beslut och hålla arbetsflödet igång.
4. **Kompenserande åtgärder**: Orkestratorns direktiv bör inkludera instruktioner om kompenserande åtgärder för att hantera undantag som inte kan lösas automatiskt. Kompenserande åtgärder är steg som tas för att ångra eller mildra effekterna av en misslyckad operation. Till exempel, om ett betalningsprocessteg misslyckas, kan en kompenserande åtgärd vara att återställa transaktionen och meddela användaren. Kompenserande åtgärder hjälper till att upprätthålla datakonsistens och integritet när undantag uppstår.
5. **Undantagsövervakning och avisering**: Sätt upp övervaknings- och aviseringsmekanismer för att upptäcka och meddela relevanta intressenter om kritiska undantag. Orkestratorn kan göras medveten om tröskelvärden och regler för att utlösa varningar när undantag överskrider vissa gränser eller när specifika typer av undantag inträffar. Detta möjliggör proaktiv identifiering och lösning av problem innan de påverkar hela systemet.

Här är ett exempel på undantagshantering och återhämtning i en Ruby-arbetsflödeskomponent:

```
class InventoryManager
  def check_availability(order)
    begin
      # Perform inventory check logic
      inventory = Inventory.find_by(product_id: order.product_id)
      if inventory.available_quantity >= order.quantity
        return true
      else
        raise InsufficientInventoryError,
              "Insufficient inventory for product #{order.product_id}"
      end
    rescue InsufficientInventoryError => e
      # Log the exception
      logger.error("Inventory check failed: #{e.message}")

      # Retry the operation after a delay
      retry_count ||= 0
      if retry_count < MAX_RETRIES
        retry_count += 1
        sleep(RETRY_DELAY)
        retry
      else
        # Fallback to manual intervention
        NotificationService.admin("Inventory check failed: Order #{order.id}")
        return false
      end
    end
  end
end
```

I detta exempel kontrollerar komponenten `InventoryManager` tillgängligheten av en produkt för en given order. Om den tillgängliga kvantiteten är otillräcklig, genereras ett `InsufficientInventoryError`. Undantaget fångas upp, loggas, och en omförsöksmekanism implementeras. Om gränsen för omförsök överskrids, övergår komponenten till manuell intervention genom att meddela en administratör.

Genom att implementera robust undantagshantering och återhämtningsmekanismer kan du säkerställa att dina intelligenta arbetsflöden är motståndskraftiga, underhållbara

och kan hantera oväntade situationer på ett elegant sätt.

Dessa mönster utgör grunden för intelligent arbetsflödesorkesterering och kan kombineras och anpassas för att möta specifika krav i olika applikationer. Genom att utnyttja dessa mönster kan utvecklare skapa arbetsflöden som är flexibla, motståndskraftiga och optimerade för prestanda och användarupplevelse.

I nästa avsnitt ska vi utforska hur dessa mönster kan implementeras i praktiken, med hjälp av exempel från verkligheten och kodavsnitt för att illustrera integrationen av AI-komponenter i arbetsflödeshantering.

# Implementering av Intelligent Arbetsflödesorkesterering i Praktiken

Nu när vi har utforskat de viktigaste mönstren inom intelligent arbetsflödesorkesterering, låt oss fördjupa oss i hur dessa mönster kan implementeras i verkliga applikationer. Vi kommer att tillhandahålla praktiska exempel och kodavsnitt för att illustrera integrationen av AI-komponenter i arbetsflödeshantering.

## Intelligent Orderhanterare

Låt oss dyka in i ett praktiskt exempel på implementering av intelligent arbetsflödesorkesterering med hjälp av en AI-driven `OrderProcessor`-komponent i en Ruby on Rails e-handelsapplikation. `OrderProcessor` förverkligar konceptet Processhanterare Företagsintegration som vi först stötte på i Kapitel 3 när vi diskuterade Mångfald av Arbetare. Komponenten kommer att ansvara för att hantera orderuppfyllelsens arbetsflöde, fatta routingbeslut baserat på mellanliggande resultat och orkestrera utförandet av olika processteg.

Orderuppfyllelseprocessen involverar flera steg såsom ordervalidering, lagerkontroll, betalningshantering och leverans. Varje steg implementeras som en separat arbetsprocess som utför en specifik uppgift och returnerar resultatet till `OrderProcessor`. Stegen är inte obligatoriska och behöver inte ens nödvändigtvis utföras i en exakt ordning.

Här är ett exempel på implementering av `OrderProcessor`. Den innehåller två mixins från Raix. Den första (`ChatCompletion`) ger den förmågan att göra chattfärdighetsställning, vilket är det som gör detta till en AI-komponent. Den andra (`FunctionDispatch`) möjliggör funktionsanrop av AI:n, vilket låter den svara på en prompt med ett funktionsanrop istället för ett textmeddelande.

Arbetarfunktionerna (`validate_order`, `check_inventory`, med flera) delegerar till sina respektive arbetarklasser, som kan vara AI- eller icke-AI-komponenter, med det enda kravet att de returnerar resultaten av sitt arbete i ett format som kan representeras som en sträng.

Precis som med alla andra exempel i denna del av boken är denna kod i praktiken pseudo-kod och är endast avsedd att förmedla mönstrets innebörd och inspirera dina egna skapelser. Fullständiga beskrivningar av mönster och kompletta kodexempel finns i Del 2.

```ruby
class OrderProcessor
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are an order processor, tasked with..."

  def initialize(order)
    self.order = order
    transcript << { system: SYSTEM_DIRECTIVE }
    transcript << { user: order.to_json }
  end

  def perform
    # will continue looping until `stop_looping!` is called
    chat_completion(loop: true)
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "validate_order",
        description: "Invoke to check validity of order",
        parameters: {
          ...
      },
      ...
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_order
    OrderValidationWorker.perform(@order)
  end

  def check_inventory
    InventoryCheckWorker.perform(@order)
  end
```

```
43
44    def process_payment
45      PaymentProcessingWorker.perform(@order)
46    end
47
48    def schedule_shipping
49      ShippingSchedulerWorker.perform(@order)
50    end
51
52    def send_confirmation
53      OrderConfirmationWorker.perform(@order)
54    end
55
56    def finished_processing
57      @order.update!(transcript:, processed_at: Time.current)
58      stop_looping!
59    end
60  end
```

I exemplet initieras OrderProcessor med ett orderobjekt och upprätthåller en transkript av arbetsflödets utförande, i det typiska konversationsformat som är naturligt för stora språkmodeller. AI:n ges fullständig kontroll att orkestrera utförandet av olika processteg, såsom ordervalidering, lagerkontroll, betalningshantering och leverans.

Varje gång metoden `chat_completion` anropas skickas transkriptet till AI:n för att tillhandahålla en färdigställning som ett funktionsanrop. Det är helt upp till AI:n att analysera resultatet från föregående steg och bestämma lämplig åtgärd. Till exempel, om lagerkontrollen visar låga lagernivåer kan OrderProcessor schemalägga en påfyllningsuppgift. Om betalningshanteringen misslyckas kan den initiera ett nytt försök eller meddela kundsupport.

> Exemplet ovan har inte definierade funktioner för påfyllning eller för att meddela kundsupport, men det skulle det absolut kunna ha.

Transkriptet växer varje gång en funktion anropas och fungerar som en dokumentation av arbetsflödets utförande, inklusive resultaten från varje steg och AI-genererade instruktioner för nästa steg. Detta transkript kan användas för felsökning, granskning och för att ge insyn i orderhanteringsprocessen.

Genom att utnyttja AI i OrderProcessor kan e-handelsapplikationen dynamiskt anpassa arbetsflödet baserat på realtidsdata och hantera undantag på ett intelligent sätt. AI-komponenten kan fatta välgrundade beslut, optimera arbetsflödet och säkerställa smidig orderhantering även i komplexa scenarion.

Det faktum att det enda kravet på arbetsprocesserna är att returnera någon begriplig output för AI:n att överväga när den bestämmer nästa steg, kan få dig att inse hur detta tillvägagångssätt kan minska arbetet med input/output-mappning som vanligtvis krävs när olika system ska integreras med varandra.

## Intelligent innehållsmoderator

Sociala medier-applikationer kräver generellt åtminstone minimal innehållsmoderering för att säkerställa en säker och hälsosam gemenskap. Detta exempel på en ContentModerator-komponent utnyttjar AI för att intelligent orkestrera modereringsarbetsflödet, och fattar beslut baserat på innehållets egenskaper och resultaten från olika modereringssteg.

Modereringsprocessen involverar flera steg såsom textanalys, bildigenkänning, bedömning av användarens anseende och manuell granskning. Varje steg implementeras som en separat arbetsprocess som utför en specifik uppgift och returnerar resultatet till ContentModerator.

Här är ett exempel på implementering av ContentModerator:

```
class ContentModerator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a content moderator process manager,
    tasked with the workflow involved in moderating user-generated content..."

  def initialize(content)
    @content = content
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: content.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "analyze_text",
        # ...
      },
      {
        name: "recognize_image",
        description: "Invoke to describe images...",
        # ...
      },
      {
        name: "assess_user_reputation",
        # ...
      },
      {
```

```
        name: "escalate_to_manual_review",
        # ...
      },
      {
        name: "approve_content",
        # ...
      },
      {
        name: "reject_content",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_text
    result = TextAnalysisWorker.perform(@content)
    continue_with(result)
  end

  def recognize_image
    result = ImageRecognitionWorker.perform(@content)
    continue_with(result)
  end

  def assess_user_reputation
    result = UserReputationWorker.perform(@content.user)
    continue_with(result)
  end

  def escalate_to_manual_review
    ManualReviewWorker.perform(@content)
    @content.update!(status: 'pending', transcript: @transcript)
  end

  def approve_content
    @content.update!(status: 'approved', transcript: @transcript)
  end

  def reject_content
```

```
    @content.update!(status: 'rejected', transcript: @transcript)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

I detta exempel initieras `ContentModerator` med ett innehållsobjekt och upprätthåller ett modereringsprotokoll i konversationsformat. AI-komponenten har full kontroll över modereringsarbetsflödet och bestämmer vilka steg som ska utföras baserat på innehållets egenskaper och resultaten från varje steg.

De tillgängliga arbetarfunktionerna som AI:n kan anropa inkluderar `analyze_text`, `recognize_image`, `assess_user_reputation` och `escalate_to_manual_review`. Varje funktion delegerar uppgiften till en motsvarande arbetsprocess (`TextAnalysisWorker`, `ImageRecognitionWorker`, etc.) och lägger till resultatet i moduleringsprotokollet, med undantag för eskaleringsfunktionen som fungerar som ett sluttillstånd. Slutligen fungerar även funktionerna `approve_content` och `reject_content` som sluttillstånd.

AI-komponenten analyserar innehållet och fastställer lämplig åtgärd. Om innehållet innehåller bildreferenser kan den anropa arbetaren `recognize_image` för hjälp med en visuell granskning. Om någon arbetare varnar för potentiellt skadligt innehåll kan AI:n besluta att eskalera innehållet för manuell granskning eller helt enkelt avvisa det direkt. Men beroende på varningens allvarlighetsgrad kan AI:n välja att använda resultaten från bedömningen av användarrykte för att besluta hur den ska hantera innehåll som den annars är osäker på. Beroende på användningsfallet kan betrodda användare kanske ha mer spelrum i vad de kan publicera. Och så vidare…

Precis som i det föregående exemplet med processhanteraren fungerar modereringsprotokollet som en dokumentation av arbetsflödets utförande, inklusive

resultaten från varje steg och AI-genererade beslut. Detta protokoll kan användas för granskning, transparens och förbättring av modereringsprocessen över tid.

Genom att utnyttja AI i `ContentModerator` kan sociala medie-applikationen dynamiskt anpassa modereringsarbetsflödet baserat på innehållets egenskaper och hantera komplexa modereringsscenarier på ett intelligent sätt. AI-komponenten kan fatta välgrundade beslut, optimera arbetsflödet och säkerställa en säker och hälsosam gemenskapsupplevelse.

Låt oss utforska två exempel till som demonstrerar prediktiv uppgiftsschemaläggning och undantagshantering och återhämtning inom ramen för intelligent arbetsflödesorkesterering.

## Prediktiv uppgiftsschemaläggning i ett kundsupportssystem

I en kundsupportapplikation byggd med Ruby on Rails är effektiv hantering och prioritering av supportärenden avgörande för att kunna ge snabb hjälp till kunder. Komponenten `SupportTicketScheduler` utnyttjar AI för att prediktivt schemalägga och tilldela supportärenden till tillgängliga agenter baserat på olika faktorer som ärendets brådskande karaktär, agentens expertis och arbetsbelastning.

```
class SupportTicketScheduler
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
    tasked with intelligently assigning tickets to available agents..."

  def initialize(ticket)
    @ticket = ticket
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: ticket.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "analyze_ticket_urgency",
        # ...
      },
      {
        name: "list_available_agents",
        description: "Includes expertise of available agents",
        # ...
      },
      {
        name: "predict_agent_workload",
        description: "Uses historical data to predict upcoming workloads",
        # ...
      },
      {
        name: "assign_ticket_to_agent",
        # ...
```

```
      },
      {
        name: "reschedule_ticket",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_ticket_urgency
    result = TicketUrgencyAnalyzer.perform(@ticket)
    continue_with(result)
  end

  def list_available_agents
    result = ListAvailableAgents.perform
    continue_with(result)
  end

  def predict_agent_workload
    result = AgentWorkloadPredictor.perform
    continue_with(result)
  end

  def assign_ticket_to_agent
    TicketAssigner.perform(@ticket, @transcript)
  end

  def delay_assignment(until)
    until = DateTimeStandardizer.process(until)
    SupportTicketScheduler.delay(@ticket, @transcript, until)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

I detta exempel initieras `SupportTicketScheduler` med ett supportärende och upprätthåller ett planeringsprotokoll. AI-komponenten analyserar ärendeinformationen och schemalägger prediktivt ärendetilldelningen baserat på faktorer som ärendets brådskande karaktär, handläggarens expertis och förutsagd handläggararbetsbelastning.

De tillgängliga funktionerna som AI:n kan anropa inkluderar `analyze_ticket_urgency`, `list_available_agents`, `predict_agent_workload` och `assign_ticket_to_agent`. Varje funktion delegerar uppgiften till en motsvarande analys- eller prediktionskomponent och lägger till resultatet i planeringsprotokollet. AI:n har också möjlighet att fördröja tilldelningen genom funktionen `delay_assignment`.

AI-komponenten granskar planeringsprotokollet och fattar välgrundade beslut om ärendetilldelning. Den tar hänsyn till ärendets brådskande karaktär, tillgängliga handläggares expertis och den förutsagda arbetsbelastningen för varje handläggare för att avgöra vilken handläggare som är mest lämpad att hantera ärendet.

Genom att utnyttja prediktiv uppgiftsplanering kan kundsupportapplikationen optimera ärendetilldelningen, minska svarstiderna och förbättra den övergripande kundnöjdheten. Proaktiv och effektiv hantering av supportärenden säkerställer att rätt ärenden tilldelas rätt handläggare vid rätt tidpunkt.

## Undantagshantering och återställning i en datbehandlingskedja

Hantering av undantag och återhämtning från fel är avgörande för att säkerställa dataintegritet och förhindra dataförlust. Komponenten `DataProcessingOrchestrator` använder AI för att intelligent hantera undantag och orkestrera återställningsprocessen i en datbehandlingskedja

```
class DataProcessingOrchestrator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a data processing orchestrator..."

  def initialize(data_batch)
    @data_batch = data_batch
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: data_batch.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "validate_data",
        # ...
      },
      {
        name: "process_data",
        # ...
      },
      {
        name: "request_fix",
        # ...
      },
      {
        name: "retry_processing",
        # ...
      },
      {
        name: "mark_data_as_failed",
```

```
43          # ...
44        },
45        {
46          name: "finished",
47          # ...
48        }
49      ]
50    end
51
52    # implementation of functions that can be called by the AI
53    # entirely at its discretion, depending on the needs of the order
54
55    def validate_data
56      result = DataValidator.perform(@data_batch)
57      continue_with(result)
58    rescue ValidationException => e
59      handle_validation_exception(e)
60    end
61
62    def process_data
63      result = DataProcessor.perform(@data_batch)
64      continue_with(result)
65    rescue ProcessingException => e
66      handle_processing_exception(e)
67    end
68
69    def request_fix(description_of_fix)
70      result = SmartDataFixer.new(description_of_fix, @data_batch)
71      continue_with(result)
72    end
73
74    def retry_processing(timeout_in_seconds)
75      wait(timeout_in_seconds)
76      process_data
77    end
78
79    def mark_data_as_failed
80      @data_batch.update!(status: 'failed', transcript: @transcript)
81    end
82
83    def finished
84      @data_batch.update!(status: 'finished', transcript: @transcript)
```

```
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end

  def handle_validation_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end

  def handle_processing_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end
end
```

I detta exempel initieras `DataProcessingOrchestrator` med ett databatchobjekt och upprätthåller en bearbetningslogg. AI-komponenten orkesterar datakörningsflödet, hanterar undantag och återhämtar sig från fel vid behov.

De tillgängliga funktionerna som AI:n kan anropa inkluderar `validate_data`, `process_data`, `request_fix`, `retry_processing` och `mark_data_as_failed`. Varje funktion delegerar uppgiften till en motsvarande databehandlingskomponent och lägger till resultatet eller undantagsdetaljerna i bearbetningsloggen.

Om ett valideringsundantag inträffar under `validate_data`-steget, lägger funktionen `handle_validation_exception` till undantagsdata i loggen och lämnar över kontrollen tillbaka till AI:n. På samma sätt, om ett bearbetningsundantag inträffar under `process_data`-steget, kan AI:n bestämma återhämtningsstrategin.

Beroende på typen av undantag som påträffas kan AI:n efter eget gottfinnande besluta att anropa `request_fix`, som delegerar till en AI-driven `SmartDataFixer`-komponent (se kapitlet om Självläkande Data). Datafixaren får en enkel beskrivning på engelska om

hur den ska modifiera `@data_batch` så att bearbetningen kan göras om. Kanske skulle en lyckad omförsök innebära att ta bort poster från databatchen som har misslyckats med valideringen och/eller kopiera dem till ett annat bearbetningsflöde för manuell granskning? Möjligheterna är nästan oändliga.

Genom att införliva AI-driven undantagshantering och återhämtning blir databehandlingsapplikationen mer motståndskraftig och feltolerant. `DataProcessingOrchestrator` hanterar intelligent undantag, minimerar dataförluster och säkerställer smidig körning av databehandlingsflödet.

# Övervakning och Loggning

Övervakning och loggning ger insyn i framsteg, prestanda och hälsa hos AI-drivna arbetsflödeskomponenter, vilket gör det möjligt för utvecklare att spåra och analysera systemets beteende. Att implementera effektiva övervaknings- och loggningsmekanismer är avgörande för felsökning, granskning och kontinuerlig förbättring av intelligenta arbetsflöden.

## Övervakning av Arbetsflödets Framsteg och Prestanda

För att säkerställa smidig körning av intelligenta arbetsflöden är det viktigt att övervaka framsteg och prestanda för varje arbetsflödeskomponent. Detta innebär att spåra viktiga mätvärden och händelser genom hela arbetsflödets livscykel.

Viktiga aspekter att övervaka inkluderar:

1. **Arbetsflödets Körningstid**: Mät tiden som varje arbetsflödeskomponent tar för att slutföra sin uppgift. Detta hjälper till att identifiera prestandaflaskhalsar och optimera arbetsflödets övergripande effektivitet.

2. **Resursanvändning**: Övervaka användningen av systemresurser, såsom CPU, minne och lagring, av varje arbetsflödeskomponent. Detta hjälper till att säkerställa att systemet fungerar inom sin kapacitet och effektivt kan hantera arbetsbelastningen.

3. **Felfrekvenser och Undantag**: Spåra förekomsten av fel och undantag inom arbetsflödeskomponenter. Detta hjälper till att identifiera potentiella problem och möjliggör proaktiv felhantering och återhämtning.

4. **Beslutspunkter och Resultat**: Övervaka beslutspunkterna inom arbetsflödet och resultaten av AI-drivna beslut. Detta ger insikter i beteendet och effektiviteten hos AI-komponenterna.

Data som samlas in genom övervakning kan visas i instrumentpaneler eller användas som indata till schemalagda rapporter som informerar systemadministratörer om systemets hälsa.

Övervakningsdata kan matas till en AI-driven systemadministratörsprocess för granskning och potentiell åtgärd!

## Loggning av Viktiga Händelser och Beslut

Loggning är en viktig praxis som innebär att fånga och lagra relevant information om viktiga händelser, beslut och undantag som inträffar under arbetsflödets körning.

Viktiga aspekter att logga inkluderar:

1. **Arbetsflödets Initiering och Slutförande**: Logga start- och sluttider för varje arbetsflödesinstans, tillsammans med relevant metadata såsom indata och användarkontext.

2. **Komponentkörning**: Logga körningsdetaljer för varje arbetsflödeskomponent, inklusive inparametrar, utdataresultat och eventuella mellanliggande data som genereras.

3. **AI-beslut och Resonemang**: Logga beslut som fattas av AI-komponenter, tillsammans med underliggande resonemang eller konfidensnivåer. Detta ger transparens och möjliggör granskning av AI-drivna beslut.

4. **Undantag och Felmeddelanden**: Logga eventuella undantag eller felmeddelanden som påträffas under arbetsflödets körning, inklusive stackspårning och relevant kontextinformation.

Loggning kan implementeras med olika tekniker, såsom att skriva till loggfiler, lagra loggar i en databas eller skicka loggar till en centraliserad loggningstjänst. Det är viktigt att välja ett loggningsramverk som ger flexibilitet, skalbarhet och enkel integration med applikationens arkitektur.

Här är ett exempel på hur loggning kan implementeras i en Ruby on Rails-applikation med klassen `ActiveSupport::Logger`:

```ruby
class WorkflowLogger
  def self.log(message, severity = :info)
    @logger ||= ActiveSupport::Logger.new('workflow.log')
    @logger.formatter ||= proc do |severity, datetime, progname, msg|
      "#{datetime} [#{severity}] #{msg}\n"
    end
    @logger.send(severity, message)
  end
end

# Usage example
WorkflowLogger.log("Workflow initiated for order ##{@order.id}")
WorkflowLogger.log("Payment processing completed successfully")
WorkflowLogger.log("Inventory check failed for item ##{item.id}", :error)
```

Genom att strategiskt placera loggningssatser genom arbetsflödets komponenter och AI:ns beslutspunkter kan utvecklare samla värdefull information för felsökning, granskning och analys.

## Fördelar med övervakning och loggning

Implementering av övervakning och loggning i intelligent arbetsflödesorkestrering erbjuder flera fördelar:

1. **Felsökning och problemlösning**: Detaljerade loggar och övervakningsdata hjälper utvecklare att snabbt identifiera och diagnostisera problem. De ger insikter i arbetsflödets exekvering, komponentinteraktioner och eventuella fel eller undantag som påträffats.

2. **Prestandaoptimering**: Övervakning av prestandamått låter utvecklare identifiera flaskhalsar och optimera arbetsflödets komponenter för bättre effektivitet. Genom att analysera exekveringstider, resursutnyttjande och andra mätvärden kan utvecklare fatta välgrundade beslut för att förbättra systemets övergripande prestanda.

3. **Granskning och efterlevnad**: Loggning av viktiga händelser och beslut skapar en granskningslogg för regelefterlevnad och ansvarsskyldighet. Det gör det möjligt för organisationer att spåra och verifiera åtgärder som vidtagits av AI-komponenter och säkerställa efterlevnad av affärsregler och juridiska krav.

4. **Kontinuerlig förbättring**: Övervaknings- och loggningsdata fungerar som värdefulla ingångsvärden för kontinuerlig förbättring av intelligenta arbetsflöden. Genom att analysera historiska data, identifiera mönster och mäta effektiviteten i AI-beslut kan utvecklare iterativt förfina och förbättra arbetsflödesorkestreringslogiken.

## Överväganden och bästa praxis

När man implementerar övervakning och loggning i intelligent arbetsflödesorkestrering bör man överväga följande bästa praxis:

1. **Definiera tydliga övervakningsmått**: Identifiera de viktigaste mätvärdena och händelserna som behöver övervakas baserat på arbetsflödets specifika krav. Fokusera på mätvärden som ger meningsfulla insikter i systemets prestanda, hälsa och beteende.

2. **Implementera detaljerad loggning**: Säkerställ att loggningssatser placeras på lämpliga punkter inom arbetsflödets komponenter och AI-beslutspunkter. Fånga relevant kontextinformation, såsom inparametrar, utdata och eventuella mellanliggande data som genereras.

3. **Använd strukturerad loggning**: Använd ett strukturerat loggningsformat för att underlätta enkel parsning och analys av loggdata. Strukturerad loggning möjliggör bättre sökbarhet, filtrering och aggregering av loggposter.

4. **Hantera loggbevarande och rotation**: Implementera policyer för loggbevarande och rotation för att hantera lagring och livscykel för loggfiler. Bestäm lämplig bevarandeperiod baserat på juridiska krav, lagringsbegränsningar och analysbehov. Om möjligt, lägg ut loggningen på en tredjepartstjänst som Papertrail.

5. **Säkra känslig information**: Var försiktig vid loggning av känslig information, såsom personligt identifierbar information (PII) eller konfidentiell affärsdata. Implementera lämpliga säkerhetsåtgärder, såsom datamaskering eller kryptering, för att skydda känslig information i loggfiler.

6. **Integrera med övervaknings- och varningsverktyg**: Utnyttja övervaknings- och varningsverktyg för att centralisera insamling, analys och visualisering av övervaknings- och loggningsdata. Dessa verktyg kan ge realtidsinsikter, generera varningar baserat på fördefinierade tröskelvärden och underlätta proaktiv problemupptäckt och -lösning. Mitt favoritverktyg bland dessa är Datadog.

Genom att implementera omfattande övervaknings- och loggningsmekanismer kan utvecklare få värdefulla insikter i beteendet och prestandan hos intelligenta arbetsflöden. Dessa insikter möjliggör effektiv felsökning, optimering och kontinuerlig förbättring av AI-drivna arbetsflödesorkestreringsystem.

## Skalbarhet och prestandaöverväganden

Skalbarhet och prestanda är kritiska aspekter att överväga vid design och implementering av intelligenta arbetsflödesorkestreringsystem. När volymen av samtidiga arbetsflöden och komplexiteten hos AI-drivna komponenter ökar blir det viktigt att säkerställa att systemet kan hantera arbetsbelastningen effektivt och skala sömlöst för att möta växande krav.

## Hantering av stora volymer samtidiga arbetsflöden

Intelligenta arbetsflödesorkestreringsystem behöver ofta hantera ett stort antal samtidiga arbetsflöden. För att säkerställa skalbarhet, överväg följande strategier:

1. **Asynkron bearbetning**: Implementera asynkrona bearbetningsmekanismer för att frikoppla exekveringen av arbetsflödeskomponenter. Detta låter systemet hantera flera arbetsflöden samtidigt utan att blockera eller vänta på att varje komponent ska slutföras. Asynkron bearbetning kan uppnås med hjälp av meddelandeköer, händelsedrivna arkitekturer eller ramverk för bakgrundsjobb som Sidekiq.

2. **Distribuerad arkitektur**: Designa systemarkitekturen för att använda serverlösa komponenter (som AWS Lambda) eller helt enkelt distribuera arbetsbelastningen över flera noder eller servrar tillsammans med din huvudapplikationsserver. Detta möjliggör horisontell skalbarhet, där ytterligare noder kan läggas till för att hantera ökade arbetsflödesvolymer.

3. **Parallell exekvering**: Identifiera möjligheter för parallell exekvering inom arbetsflöden. Vissa arbetsflödeskomponenter kan vara oberoende av varandra och kan exekveras samtidigt. Genom att utnyttja tekniker för parallell bearbetning, såsom multithreading eller distribuerade uppgiftsköer, kan systemet optimera resursutnyttjandet och minska den totala exekveringstiden för arbetsflöden.

## Optimering av AI-drivna komponenter

AI-drivna komponenter, såsom maskininlärningsmodeller eller system för naturlig språkbehandling, kan vara beräkningsintensiva och påverka den övergripande prestandan i arbetsflödesorkesteringssystemet. För att optimera prestandan hos AI-komponenter, överväg följande tekniker:

1. **Cachning**: Om din AI-behandling är rent generativ och inte involverar realtidssökningar eller externa integrationer för att generera chattresponser, kan

du undersöka cachningsmekanismer för att lagra och återanvända resultat från ofta använda eller beräkningstunga operationer.

2. **Modelloptimering**: Optimera kontinuerligt hur du använder AI-modellerna i arbetsflödeskomponenterna. Detta kan innefatta tekniker som *Prompt-destillering* eller helt enkelt vara en fråga om att testa nya modeller när de blir tillgängliga.

3. **Satsvis bearbetning**: Om du arbetar med GPT-4-klassmodeller, kan du eventuellt utnyttja tekniker för satsvis bearbetning för att behandla flera datapunkter eller förfrågningar i en enda batch, istället för att bearbeta dem individuellt. Genom att bearbeta data i satser kan systemet optimera resursutnyttjandet och minska overheaden från upprepade modellförfrågningar.

## Övervakning och prestandaprofilering

För att identifiera prestandaflaskhalsar och optimera skalbarheten hos det intelligenta arbetsflödesorkesteringssystemet är det avgörande att implementera övervaknings- och profileringsmekanismer. Överväg följande tillvägagångssätt:

1. **Prestandamått**: Definiera och spåra viktiga prestandamått, såsom svarstid, genomströmning, resursutnyttjande och latens. Dessa mått ger insikter i systemets prestanda och hjälper till att identifiera områden för optimering. Den populära AI-modellaggregatorn OpenRouter inkluderar värd[1]- och hastighets[2]-mått i varje API-svar, vilket gör det enkelt att spåra dessa nyckelmått.

2. **Profileringsverktyg**: Använd profileringsverktyg för att analysera prestandan hos enskilda arbetsflödeskomponenter och AI-operationer. Profileringsverktyg kan hjälpa till att identifiera prestandahotspots, ineffektiva kodvägar eller resurskrävande operationer. Populära profileringsverktyg inkluderar New Relic, Scout, eller inbyggda profilerare som tillhandahålls av programmeringsspråket eller ramverket.

[1] Värd är tiden det tog att ta emot den första byten av den streamade genereringen från modellvärden, även känt som "tid till första byte."

[2] Hastighet beräknas som antalet slutförda tokens dividerat med total genereringstid. För icke-streamade förfrågningar räknas latensen som en del av genereringstiden.

3. **Belastningstestning**: Genomför belastningstester för att utvärdera systemets prestanda under olika nivåer av samtidig arbetsbelastning. Belastningstestning hjälper till att identifiera systemets skalbarhetsgränser, upptäcka prestandaförsämringar och säkerställa att systemet kan hantera den förväntade trafiken utan att kompromissa med prestandan.

4. **Kontinuerlig övervakning**: Implementera mekanismer för kontinuerlig övervakning och varning för att proaktivt upptäcka prestandaproblem och flaskhalsar. Sätt upp övervakningspaneler och varningar för att spåra viktiga prestandaindikatorer (KPI:er) och ta emot meddelanden när fördefinierade tröskelvärden överskrids. Detta möjliggör snabb identifiering och lösning av prestandaproblem.

## Skalningsstrategier

För att hantera ökande arbetsbelastningar och säkerställa skalbarheten hos det intelligenta arbetsflödesorkesteringssystemet, överväg följande skalningsstrategier:

1. **Vertikal skalning**: Vertikal skalning innebär att öka resurserna (t.ex. CPU, minne) för enskilda noder eller servrar för att hantera högre arbetsbelastningar. Detta tillvägagångssätt är lämpligt när systemet kräver mer beräkningskraft eller minne för att hantera komplexa arbetsflöden eller AI-operationer.

2. **Horisontell skalning**: Horisontell skalning innebär att lägga till fler noder eller servrar till systemet för att distribuera arbetsbelastningen. Detta tillvägagångssätt är effektivt när systemet behöver hantera ett stort antal samtidiga arbetsflöden eller när arbetsbelastningen enkelt kan fördelas över flera noder. Horisontell skalning kräver en distribuerad arkitektur och lastbalanseringsmekanismer för att säkerställa jämn fördelning av trafiken.

3. **Automatisk skalning**: Implementera mekanismer för automatisk skalning för att automatiskt justera antalet noder eller resurser baserat på arbetsbelastningens behov. Automatisk skalning låter systemet dynamiskt skala upp eller ner

beroende på inkommande trafik, vilket säkerställer optimalt resursutnyttjande och kostnadseffektivitet. Molnplattformar som Amazon Web Services (AWS) eller Google Cloud Platform (GCP) tillhandahåller funktioner för automatisk skalning som kan utnyttjas för intelligenta arbetsflödesorkesteringssystem.

## Prestandaoptimeringstekniker

Utöver skalningsstrategierna, överväg följande prestandaoptimeringstekniker för att förbättra effektiviteten hos det intelligenta arbetsflödesorkesteringssystemet:

1. **Effektiv datalagring och -hämtning**: Optimera mekanismerna för datalagring och -hämtning som används av arbetsflödeskomponenterna. Använd effektiv databasindexering, optimering av frågor och datacachning för att minimera latensen och förbättra prestandan för dataintensiva operationer.

2. **Asynkron I/O**: Använd asynkrona I/O-operationer för att förhindra blockering och förbättra systemets responstid. Asynkron I/O låter systemet hantera flera förfrågningar samtidigt utan att behöva vänta på att I/O-operationer ska slutföras, vilket maximerar resursutnyttjandet.

3. **Effektiv serialisering och deserialisering**: Optimera serialiserings- och deserialiseringsprocesserna som används för datautbyte mellan arbetsflödeskomponenter. Använd effektiva serialiseringsformat som Protocol Buffers eller MessagePack för att minska överbelastningen vid dataserialisering och förbättra prestandan i kommunikationen mellan komponenter.

För Ruby-baserade applikationer, överväg att använda Universal ID. Universal ID utnyttjar både MessagePack och Brotli (en kombination byggd för hastighet och förstklassig datakomprimering). När dessa bibliotek kombineras är de upp till 30% snabbare och ligger inom 2-5% komprimeringsgrad jämfört med Protocol Buffers.

4. **Komprimering och kodning**: Tillämpa komprimerings- och kodningstekniker för att minska storleken på data som överförs mellan arbetsflödeskomponenter. Komprimeringsalgoritmer som gzip eller Brotli kan avsevärt minska användningen av nätverksbandbredd och förbättra systemets övergripande prestanda.

Genom att beakta skalbarhet och prestandaaspekter under design och implementering av intelligenta arbetsflödesorkestreringssystem, kan du säkerställa att ditt system kan hantera stora volymer av samtidiga arbetsflöden, optimera prestandan för AI-drivna komponenter och skala sömlöst för att möta växande behov. Kontinuerlig övervakning, profilering och optimeringsinsatser är avgörande för att upprätthålla systemets prestanda och responstid när arbetsbelastning och komplexitet ökar över tid.

# Testning och validering av arbetsflöden

Testning och validering är kritiska aspekter vid utveckling och underhåll av intelligenta arbetsflödesorkestreringssystem. Med tanke på den komplexa naturen hos AI-drivna arbetsflöden är det viktigt att säkerställa att varje komponent fungerar som förväntat, att det övergripande arbetsflödet beter sig korrekt och att AI-besluten är korrekta och tillförlitliga. I detta avsnitt kommer vi att utforska olika tekniker och överväganden för testning och validering av intelligenta arbetsflöden.

## Enhetstestning av arbetsflödeskomponenter

Enhetstestning innebär att testa enskilda arbetsflödeskomponenter isolerat för att verifiera deras korrekthet och robusthet. När du enhetstestar AI-drivna arbetsflödeskomponenter, överväg följande:

1. **Indatavalidering**: Testa komponentens förmåga att hantera olika typer av indata, inklusive giltig och ogiltig data. Verifiera att komponenten hanterar gränsfall på ett elegant sätt och tillhandahåller lämpliga felmeddelanden eller undantag.

2. **Utdataverifiering**: Bekräfta att komponenten producerar den förväntade utdatan för en given uppsättning indata. Jämför den faktiska utdatan med de förväntade resultaten för att säkerställa korrekthet.

3. **Felhantering**: Testa komponentens felhanteringsmekanismer genom att simulera olika felscenarier, såsom ogiltig indata, otillgängliga resurser eller oväntade undantag. Verifiera att komponenten fångar upp och hanterar fel på lämpligt sätt.

4. **Gränsvillkor**: Testa komponentens beteende under gränsvillkor, såsom tom indata, maximal indatastorlek eller extrema värden. Säkerställ att komponenten hanterar dessa villkor elegant utan att krascha eller producera felaktiga resultat.

Här är ett exempel på ett enhetstest för en arbetsflödeskomponent i Ruby med hjälp av testramverket RSpec:

```
RSpec.describe OrderValidator do
  describe '#validate' do
    context 'when order is valid' do
      let(:order) { build(:order) }

      it 'returns true' do
        expect(subject.validate(order)).to be true
      end
    end

    context 'when order is invalid' do
      let(:order) { build(:order, total_amount: -100) }

      it 'returns false' do
        expect(subject.validate(order)).to be false
      end
    end
  end
end
```

I detta exempel testas komponenten `OrderValidator` med hjälp av två testfall: ett för en giltig order och ett annat för en ogiltig order. Testfallen verifierar att metoden `validate` returnerar det förväntade booleska värdet baserat på orderns giltighet.

## Integrationstestning av arbetsflödesinteraktioner

Integrationstestning fokuserar på att verifiera interaktioner och dataflöden mellan olika arbetsflödeskomponenter. Det säkerställer att komponenterna fungerar sömlöst tillsammans och producerar de förväntade resultaten. När man utför integrationstestning av intelligenta arbetsflöden bör man beakta följande:

1. **Komponentinteraktion**: Testa kommunikationen och datautbytet mellan arbetsflödeskomponenter. Verifiera att utdata från en komponent korrekt överförs som indata till nästa komponent i arbetsflödet.

2. **Datakonsistens**: Säkerställ att data förblir konsistent och korrekt när den flödar genom arbetsflödet. Verifiera att datatransformationer, beräkningar och aggregeringar utförs korrekt.

3. **Undantagspropagering**: Testa hur undantag och fel propageras och hanteras mellan arbetsflödeskomponenter. Verifiera att undantag fångas upp, loggas och hanteras på lämpligt sätt för att förhindra störningar i arbetsflödet.

4. **Asynkront beteende**: Om arbetsflödet innehåller asynkrona komponenter eller parallell exekvering, testa koordinerings- och synkroniseringsmekanismerna. Säkerställ att arbetsflödet beter sig korrekt under samtidiga och asynkrona scenarier.

Här är ett exempel på ett integrationstest för ett arbetsflöde i Ruby med hjälp av testramverket RSpec:

```
RSpec.describe OrderProcessingWorkflow do

  let(:order) { build(:order) }

  it 'processes the order successfully' do
    expect(OrderValidator).to receive(:validate).and_return(true)
    expect(InventoryManager).to receive(:check_availability).and_return(true)
    expect(PaymentProcessor).to receive(:process_payment).and_return(true)
    expect(ShippingService).to receive(:schedule_shipping).and_return(true)

    workflow = OrderProcessingWorkflow.new(order)
    result = workflow.process

    expect(result).to be true
    expect(order.status).to eq('processed')
  end

end
```

I detta exempel testas `OrderProcessingWorkflow` genom att verifiera interaktionerna mellan olika arbetsflödeskomponenter. Testfallet ställer upp förväntningar för varje komponents beteende och säkerställer att arbetsflödet behandlar ordern framgångsrikt och uppdaterar orderstatus på lämpligt sätt.

## Testning av AI-beslutspunkter

Att testa AI-beslutspunkter är avgörande för att säkerställa noggrannheten och tillförlitligheten hos AI-drivna arbetsflöden. När man testar AI-beslutspunkter bör man beakta följande:

1. **Beslutsprecision**: Verifiera att AI-komponenten fattar korrekta beslut baserat på indata och den tränade modellen. Jämför AI-besluten med förväntade resultat eller referensdata.

2. **Kantfall**: Testa AI-komponentens beteende under kantfall och ovanliga scenarier. Verifiera att AI-komponenten hanterar dessa fall elegant och fattar rimliga beslut.

3. **Partiskhet och rättvisa**: Utvärdera AI-komponenten för potentiell partiskhet och säkerställ att den fattar rättvisa och opartiska beslut. Testa komponenten med olika typer av indata och analysera resultaten för att upptäcka eventuella diskriminerande mönster.

4. **Förklarbarhet**: Om AI-komponenten tillhandahåller förklaringar eller resonemang för sina beslut, verifiera att förklaringarna är korrekta och tydliga. Säkerställ att förklaringarna överensstämmer med den underliggande beslutsprocessen.

Här är ett exempel på hur man testar en AI-beslutspunkt i Ruby med hjälp av RSpec-testramverket:

```
RSpec.describe FraudDetector do
  describe '#detect_fraud' do
    context 'when transaction is fraudulent' do
      let(:tx) do
        build(:transaction, amount: 10_000, location: 'High-Risk Country')
      end

      it 'returns true' do
        expect(subject.detect_fraud(tx)).to be true
      end
    end

    context 'when transaction is legitimate' do
      let(:tx) do
        build(:transaction, amount: 100, location: 'Low-Risk Country')
      end

      it 'returns false' do
        expect(subject.detect_fraud(tx)).to be false
      end
    end
  end
end
```

I detta exempel testas AI-komponenten `FraudDetector` med två testfall: ett för en bedräglig transaktion och ett annat för en legitim transaktion. Testfallen verifierar att metoden `detect_fraud` returnerar det förväntade booleska värdet baserat på transaktionens egenskaper.

## End-to-end-testning

End-to-end-testning innebär att testa hela arbetsflödet från början till slut, genom att simulera verkliga scenarier och användarinteraktioner. Det säkerställer att arbetsflödet fungerar korrekt och ger önskade resultat. När man utför end-to-end-testning för intelligenta arbetsflöden bör man beakta följande:

1. **Användarscenarier**: Identifiera vanliga användarscenarier och testa arbetsflödets beteende under dessa scenarier. Verifiera att arbetsflödet hanterar användarindata korrekt, fattar lämpliga beslut och producerar förväntade resultat.

2. **Datavalidering**: Säkerställ att arbetsflödet validerar och sanerar användarindata för att förhindra datainkonsekvenser eller säkerhetssårbarheter. Testa arbetsflödet med olika typer av indata, inklusive giltig och ogiltig data.

3. **Felåterställning**: Testa arbetsflödets förmåga att återhämta sig från fel och undantag. Simulera felscenarier och verifiera att arbetsflödet hanterar dem elegant, loggar felen och vidtar lämpliga återställningsåtgärder.

4. **Prestanda och skalbarhet**: Utvärdera arbetsflödets prestanda och skalbarhet under olika belastningsförhållanden. Testa arbetsflödet med en stor volym samtidiga förfrågningar och mät svarstider, resursanvändning och övergripande systemstabilitet.

Här är ett exempel på ett end-to-end-test för ett arbetsflöde i Ruby som använder testramverket RSpec och biblioteket Capybara för att simulera användarinteraktioner:

```
RSpec.describe 'Order Processing Workflow' do
  scenario 'User places an order successfully' do
    visit '/orders/new'
    fill_in 'Product', with: 'Sample Product'
    fill_in 'Quantity', with: '2'
    fill_in 'Shipping Address', with: '123 Main St'
    click_button 'Place Order'

    expect(page).to have_content('Order Placed Successfully')
    expect(Order.count).to eq(1)
    expect(Order.last.status).to eq('processed')
  end
end
```

I detta exempel simulerar end-to-end-testet en användare som lägger en beställning via webbgränssnittet. Det fyller i de obligatoriska formulärfälten, skickar in beställningen och verifierar att ordern behandlas framgångsrikt genom att visa lämpligt bekräftelsemeddelande och uppdatera orderstatus i databasen.

## Kontinuerlig integrering och driftsättning

För att säkerställa tillförlitligheten och underhållbarheten av intelligenta arbetsflöden rekommenderas att integrera testning och validering i den kontinuerliga integrerings- och driftsättningsprocessen (CI/CD)-pipeline. Detta möjliggör automatiserad testning och validering av förändringar i arbetsflödet innan de driftsätts i produktion. Överväg följande metoder:

1. **Automatiserad testkörning**: Konfigurera CI/CD-pipelinen för att automatiskt köra testsviten närhelst ändringar görs i arbetsflödets kodbas. Detta säkerställer att eventuella regressioner eller fel upptäcks tidigt i utvecklingsprocessen.

2. **Övervakning av testtäckning**: Mät och övervaka testtäckningen av arbetsflödets komponenter och AI-beslutspunkter. Sträva efter hög testtäckning för att säkerställa att kritiska vägar och scenarier testas grundligt.

3. **Kontinuerlig återkoppling**: Integrera testresultat och kodkvalitetsmetrik i utvecklingsarbetsflödet. Ge kontinuerlig återkoppling till utvecklare om teststatus, kodkvalitet och eventuella problem som upptäcks under CI/CD-processen.

4. **Stagingmiljöer**: Driftsätt arbetsflödet i stagingmiljöer som efterliknar produktionsmiljön så mycket som möjligt. Utför ytterligare tester och validering i stagingmiljön för att fånga upp eventuella problem relaterade till infrastruktur, konfiguration eller dataintegration.

5. **Återställningsmekanismer**: Implementera återställningsmekanismer för händelse av driftsättningsfel eller kritiska problem som upptäcks i produktion. Säkerställ att arbetsflödet snabbt kan återställas till en tidigare stabil version för att minimera driftstopp och påverkan på användarna.

Genom att införliva testning och validering genom hela utvecklingslivscykeln för intelligenta arbetsflöden kan organisationer säkerställa tillförlitligheten, noggrannheten och underhållbarheten av sina AI-drivna system. Regelbunden testning och validering hjälper till att fånga upp buggar, förhindra regressioner och bygga förtroende för arbetsflödets beteende och resultat.

# Del 2: Mönstren

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Promptkonstruktion

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Tankekedjemetoden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Innehållsgenerering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Strukturerad Entitetsskapande

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Vägledning för LLM-agenter

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar och överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Lägesväxling

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När den ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Rolltilldelning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Promptobjekt

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Promptmall

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar och överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas:

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Strukturerad IO

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Skalning av Structured IO

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar och överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Promptkedjning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel: Olympias introduktionsprocess

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Promptomskrivare

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Response Fencing

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar och överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Felhantering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Frågeanalysator

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Implementering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Part-of-Speech (POS) Tagging och Named Entity Recognition (NER)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Avsiktsklassificering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Nyckelordsextrahering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Frågeomskrivare

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Fördelar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Buktalare

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Diskreta komponenter

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Predikat

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# API-fasad

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Huvudfördelar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När den ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Autentisering och Auktorisering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Hantering av förfrågningar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Formatering av svar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Felhantering och kantfall

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Skalbarhet och prestandaöverväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Jämförelse med andra designmönster

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Resultattolk

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Virtuell maskin

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## När det ska användas

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Bakom magin

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Specifikation och Testning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Specificera Beteendet

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Skriva Testfall

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel: Testa Översättningskomponenten

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Uppspelning av HTTP-interaktioner

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Human In The Loop (HITL)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Övergripande mönster

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Hybridintelligens

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Adaptiv respons

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Människa-AI-rollväxling

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Eskalering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Viktiga Fördelar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Verklig tillämpning: Sjukvård

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Återkopplingsslinga

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Tillämpningar och exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Avancerade tekniker för integration av mänsklig feedback

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Passiv informationsstrålning



## Hur det fungerar



### Kontextuell informationsvisning



### Proaktiva aviseringar



### Förklarande insikter



### Interaktiv utforskning

## Viktiga fördelar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Tillämpningar och exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Kollaborativt beslutsfattande (CDM)

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Kontinuerligt lärande

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Tillämpningar och exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Etiska överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## HITL:s roll i att minska AI-risker

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Teknologiska framsteg och framtidsutsikter

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Utmaningar och begränsningar med HITL-system

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Intelligent felhantering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Traditionella felhanteringsmetoder

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Kontextuell feldiagnos

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Promptkonstruktion för kontextuell feldiagnos

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### Hämtningsförstärkt generering för kontextuell feldiagnos

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Intelligent felrapportering

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Prediktiv felförebyggande

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Smart feråterhämtning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Personaliserad felkommunikation

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Adaptivt felhanteringsarbetsflöde

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Kvalitetskontroll

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Eval

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Problem

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Lösning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Förståelse av gyllene referenser

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur referensfria utvärderingar fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Skyddsmekanism

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Problem

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Lösning

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Hur det fungerar

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Exempel

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Överväganden

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Guardrails och Evals: Två sidor av samma mynt

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Utbytbarheten mellan Guardrails och referensfria Evals

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Implementering av dubbelfunktionella Guardrails och Evals

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Ordlista

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Ordlista

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### A

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### B

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

### C

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## D

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## E

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## F

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## G

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## H

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## I

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## J

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## K

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## L

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## M

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## N

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## O

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## P

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Q

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## R

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## S

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## T

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## U

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## V

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## W

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

## Z

Detta innehåll är inte tillgängligt i provboken. Boken kan köpas på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-sv.

# Index