Izrada web-aplikacije za čavrljanje s mogućnošću strujanja odgovora (engl. *streaming*) i generacijom poboljšanom dohvatom (engl. *RAG*) koristeći razvojni okvir *LangChain*

Sažetak

U sklopu ovog projekta razvijena je web-aplikacija za čavrljanje (engl. *chatbot*) s podrškom za strujanje odgovora u stvarnom vremenu (engl. *streaming*). Također, prikazana je i nadogradnja aplikacije do razine gdje više nije samo web-aplikacija za čavrljanje, već i za postavljanje i dobivanje odgovora o specifičnom setu podataka koristeći generaciju poboljšanu dohvatom (engl. *retrieveal augmented generation* – *RAG*).

Ključne riječi: web-aplikacija, čavrljanje, umjetna inteligencija, strujanje odgovora u stvarnom vremenu, rag, chatbot, langchain

Uvod

Radni okvir *LangChain*[1] koristi se za razvoj aplikacija koje pokreću veliki jezični modeli (engl. *large language models – LLMs*). Pojednostavlja rad s jezičnim modelima koristeći gotove komponente i integracije treće strane otvorenog koda.

Sastoji se od više biblioteka otvorenog koda:

- `@langchain/core` - bazne apstrakcije i izražajni jezik *LangChain Expression Language (LCEL)*
- `@langchain/community` - integracije trećih strana, npr. `@langchain/opeanai`, `@langchain/anthropic`
- `langchain` – lanci, agenti i strategije dohvata koje čine kognitivnu arhitekturu aplikacije

Ovaj dokument sadrži i određene programske kodove. Odabrani su oni koji prikazuju ključne funkcionalnosti korištenja. Kodovi koji u to ne spadju, poput kodova izgradnje grafičkog web-sučelja i sl, nisu uvršteni u dokument te su vidljivi u priloženoj kodnoj bazi.

Veliki jezični modeli

Veliki jezični modeli su napredni modeli strojnog učenja koji se ističu po širokom rasponu jezično povezanih zadataka kao što su generiranje teksta, prevođenje, sažimanje i odgovaranje na pitanja bez potrebe za podešavanje na razini svakog zadatka za svaki scenarij.

Moderni veliki jezični modeli su najčešće dostupni putem sučelja za čavrljanje koje prima listu poruka kao ulaz i vraća poruke kao izlaz.

Razvojni okvir *LangChain* pruža konzistentno sučelje za rad s jezičnim modelima različitih pružatelja pritom pružajući dodatne značajke za nadzor, otklanjanje pogrešaka i optimiziranje performansi aplikacija koje koriste velike jezične modele.

Moguće je koristiti format poruka radnog okvira ili format poruka organizacije *OpenAI*.

Standardni parametri za konfiguriranje modela su:

- `model` - identifikator modela - `gpt-3.5-turbo`, `gpt-4`
- `temperature` - faktor nasumičnosti izlaza modela gdje viša vrijednost znači viši stupanj kreativnosti, a niža vrijednost viši stupanj determinizma i fokusiranosti
- `timeout` - maksimalno vrijeme u sekundama dozvoljeno za čekanje odgovora modela prije otkazivanja zahtjeva
- `maxTokens` - gornja granica broja tokena, odnosno riječi i interpunkcije, u odgovoru, a služi za kontrolu duljine izlaza
- `stop` - definira zaustavne sekvence koje naznačuju kada model treba stati s generiranjem tokena, npr. posebne riječi koje signaliziraju kraj odgovora
- `maxRetries` - gornja granica broja pokušaja ponovnog slanja zahtjeva, npr. zbog problema s mrežom, ograničenje stope (engl. *rate-limit*)
- `apiKey` - ključ za autentifikaciju s aplikacijskim sučeljem pružatelja modela
- `baseUrl` - URL krajnje točke aplikacijskog sučelja gdje se šalju zahtjevi

Veliki jezični modeli imaju **kontekstni prozor** (engl. *context window*) koji predstavlja maksimalni ulaz koji model može procesirati u datom trenutku.

Upitni predlošci

Upitni predlošci (engl. *Prompt Templates*) pomažu prevoditi korisničke ulaze i parametre u upute za jezični model. Služe za navođenje odgovora modela, razumijevanje konteksta i generiranje relevantnih i koherentnih izlaza.

Predlošci kao ulaz uzimaju objekt gdje svaki ključ predstavlja varijablu predloška koju treba popuniti.

Kao izlaz vraćaju objekt tipa `PromptValue` – može ga se predati jezičnom ili razgovornom modelu ili se može pretvoriti u niz ili listu poruka.

Postoje tri vrste predložaka.

Tip predloška `String PromptTemplates`

Predložak tipa `String PromptTemplates` (*Kod 1*) služi za formatiranje jednog niza i koristi se za jednostavne ulaze.

Kao izlaz daje jednostavan objekt tipa `StringPromptValue` koji sadrži jedan ključ i vrijednost (*Kod 2*).

Kod 1. Primjer korištenja predloška tipa `String PromptTemplates`

```
const invokeFn = async (input: BaseLanguageModelInput) => {
  "use server";
  const response = await llm.invoke(input);
  console.log(response);
  return response.content.toString();
};
```

Kod 2. Primjer izlaza `StringPrmomptValue`

```
StringPromptValue {
 value: 'Tell me a joke about cats'
}
```

Tip predloška `ChatPromptTemplates`

Predložak tipa `ChatPromptTemplates` (*Kod 2*) služi za oblikovanje liste poruka koja se i sama sastoji od niza predložaka.

Kao izlaz daje objekt tipa `ChatPromptValue` koji sadrži niz poruka različitih tipova (*Kod 4*).

Kod 3. Primjer korištenja predloška tipa `ChatPromptTemplates`

```
import { ChatPromptTemplate } from "@langchain/core/prompts";

const promptTemplate = ChatPromptTemplate.fromMessages([
  ["system", "You are a helpful assistant"],
  ["user", "Tell me a joke about {topic}"],
]);
await promptTemplate.invoke({ topic: "cats" });
```

Kod 4. Primjer izlaza `ChatPromptValue`

```
ChatPromptValue {
  messages: [
    SystemMessage {
      "content": "You are a helpful assistant",
      "additional_kwargs": {}, "response_metadata": {}
    },
    HumanMessage {
      "content": "Tell me a joke about cats",
      "additional_kwargs": {}, "response_metadata": {}
    }
  ]
}
```

U *Kodu 2* prva poruka je poruka sustava tipa `SystemMessage` koja ne sadrži varijablu za oblikovanje, dok je druga poruka korisnička poruka tipa `HumanMessage` i oblikovana je varijablom `topic`.

Tip predloška `MessagesPlaceholder`

Predložak tipa `MessagesPlaceholder` (*Kod 5*) zaslužan je za dodavanje niza poruka na određeno mjesto i koristi se uz predložak tipa `ChatPromptTemplate`.

Kao izlaz daje objekt tipa `ChatPromptValue` (*Kod 6*) koji sadrži niz poruka različitih tipova, ali na mjestu određenom predloškom.

Kod 5. Primjer korištenja predloška tipa `MessagesPlaceholder`

```
import {
  ChatPromptTemplate,
  MessagesPlaceholder,
} from "@langchain/core/prompts";
import { HumanMessage } from "@langchain/core/messages";

const promptTemplate = ChatPromptTemplate.fromMessages([
  ["system", "You are a helpful assistant"], new MessagesPlaceholder("msgs"),
]);
await promptTemplate.invoke({ msgs: [new HumanMessage("hi!")] });
```

Kod 6. Primjer izlaza `ChatPromptValue` nakon korištenja predloška tipa `MessagesPlaceholder`

```
ChatPromptValue {
  messages: [
    SystemMessage {
      "content": "You are a helpful assistant",
      "additional_kwargs": {}, "response_metadata": {}
    },
    HumanMessage {
      "content": "hi!", "additional_kwargs": {}, "response_metadata": {}
    }
  ]
}
```

U *Kodu 6* stvara se niz dviju poruka, od kojih je jedna poruka sustava, a druga korisnička poruka postavljeno na mjesto predloška `MessagesPlaceholder`.

Poruka

Poruka je jedinica komunikacije u razgovornim modelima. Predstavlja ulaz i izlaz modela kao i dodatni kontekst i metapodatke asocirane s razgovorom.

Svaka poruka ima **ulogu**, **sadržaj** i dodatne metapodatke.

Uloga može npr. biti tipa `user`, `assistant` i sl.

Sadržaj može biti tekst, multimedijalni podaci kao što su slika, zvuk i video. Neki od tipova sadržaja su `AIMessage`, `HumanMessage` i `Multimodality`.

Dodatni metapodaci variraju ovisno o poslužitelju modela, a neki od njih su `ID`, `Name`, `Metadata`, `Tool Calls` i sl.

Razvojni okvir *LangChain* pruža **unificirani format poruka** koji se može koristiti preko više modela. Korisnicima omogućava rad s različitim modelima bez vođenja brige o specifičnostima formata poruka koje taj model koristi.

Redoslijed poruka kao ulaz u razgovorni model morao bi pratiti određenu strukturu koja osigurava generiranje valjanih odgovora iz modela. Struktura nalikuje redoslijedu:

1. poruka korisnika
2. poruka pomoćnika – razgovornog modela
3. poruka korisnika
4. poruka pomoćnika
5. ...

Radni okvir podržava šest tipova poruka.

Tip poruke `SystemMessage`

Poruka tipa `SystemMessage` ima ulogu `system` i služi za pripremu ponašanja modela te pružanja dodatnog konteksta.

Drugim riječima, modelu možemo reći da zauzme određeni tip ličnosti ili postaviti ton razgovora. Na primjer, kažemo mu da priča kao gusar.

Tip poruke `HumanMessage`

Poruka tipa `HumanMessage` ima ulogu `user` i predstavlja poruku korisnika koji komunicira s modelom.

Tip poruke `AIMessage` i `AIMessageChunk`

Poruka tipa `AIMessage` ima ulogu `assistant` i predstavlja odgovor modela. Tip poruke `AIMessageChunk` također ima ulogu `assistant`, ali predstavlja dio odgovora modela u načinu rada strujanja u stvarnom vremenu.

Tip poruke `ToolMessage`

Poruka tipa `ToolMessage` ima ulogu `tool` i predstavlja rezultat poziva nekog alata postavljenog u lancu.

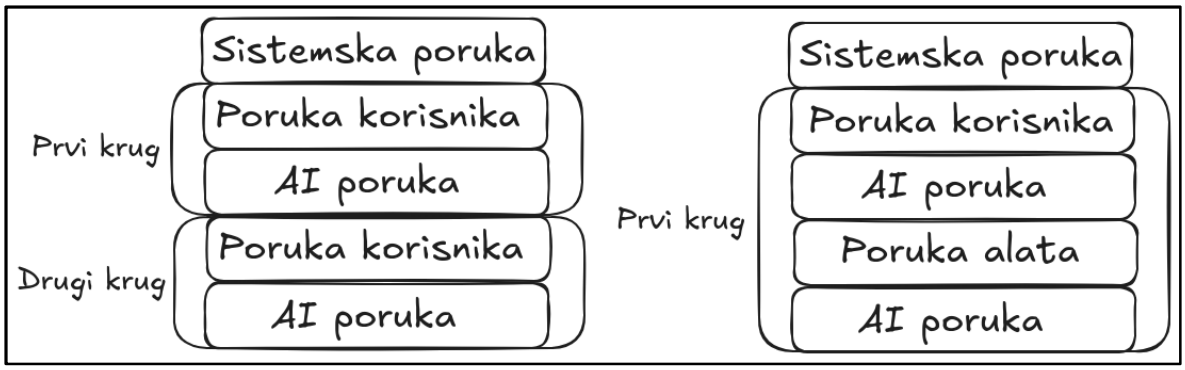
Tip poruke `RemoveMessage`

Poruka tipa `RemoveMessage` nema ulogu i predstavlja apstrakciju za upravljanje poviješću razgovora.

Povijest razgovora

Povijest razgovora je zapis razgovora između korisnika i razgovornog modela. Koristi se za održavanje konteksta i stanja kroz razgovor. Zapravo se radi o sekvenci poruka gdje je svaka povezana sa specifičnom ulogom (*Slika 1*).

Slika 1. Prikaz sheme sekvence poruka



Većina razgovora započinje sistemskom porukom koja priprema i postavlja kontekst razgovora. Obično slijedi korisnička poruka koju zatim slijedi poruka pomoćnika kao odgovor modela.

Razgovorni modeli imaju gornju granicu veličine ulaza pa je potrebno upravljati poviješću razgovora i kratiti ju po potrebi sa svrhom izbjegavanja prekoračenja kontekstnog prozora.

Izražajni jezik LCEL

Izražajni jezik *Language Chain Expression Language* (LCEL) uzima deklarativni pristup izgradnji novih sučelja tipa *Runnable* od postojećih sučelja istog tipa.

Deklarativni pristup znači da opisujemo **što** želimo da se dogodi umjesto **kako** želimo da se nešto dogodi. Taj pristup omogućava razvojnom okviru optimizaciju lanaca za vrijeme izvođenja (engl. *optimization during run-time execution*).

Runnable jest fundamentalno sučelje za rad s komponentama razvojnog okvira *LangChain*. Pristup sučelja definira standardno sučelje koje omogućuje komponenti tog tipa da bude:

- **prizvana** (engl. *invoked*) – jedan ulaz pretvoren je u jedan izlaz
- **skupna** (engl. *batched*) – više ulaza efikasno su pretvoreni u izlaze
- **u obliku toka** (engl. *streamed*) – izlazi dolaze kako se generiraju
- proučena (engl. *inspected*) – dostupne su shematske informacije o ulazu, izlazu i konfiguraciji jedne komponente sučljea *Runnable*

Komponenta sučelja *Runnable* stvorena koristeći LCEL često se naziva **lanac** (engl. *chain*). Drugim riječima, lanac jest komponenta sučelja *Runnable* i implementira cijelo njeno sučelje.

Postoje dvije glavne prednosti korištenja LCEL-a:

1. **optimizacija paralelnog izvođenja** – Moguće je izvoditi komponente sučelja *Runnable* u paraleli ili izvoditi više ulaza kroz dani lanac u paraleli koristeći skupno izvođenje. Paralelno izvođenje znatno umanjuje latenciju pošto obrada može biti izvedena u paraleli umjesto sekvencijalno.
2. **pojednostavljanje izvođenja u obliku toka** – LCEL lanci mogu bit izvedeni u obliku toka što omogućava inkrementalni izlaz kako se lanac izvodi. LCEL može značajno optimizirati tok izlaza smanjujući vrijeme vrijeme do prvog tokena, odnosno do prvog dijela izlaza.

LCEL je orkestracijsko rješenje, što znači da omogućava razvojnom okviru rukovanje lancima za vrijeme izvođenja na optimizirani način. Preporuča ga se koristiti za **jednostavnije** orkestracijske zadatke, iako nije neviđeno korištenje lanaca sa stotinama koraka u produkciji.

Međutim, kada aplikacija zahtjeva kompleksno vođenje stanja, grananje, cikluse ili više agenata, preporuča se korištenje orkestracijskog radnog okvira *LangGraph* koji omogućava definiranje grafova koji pak definiraju tok aplikacije. Moguće je koristiti LCEL unutar samih čvorova grafa, gdje su potrebni, a pritom olakšavajući definiranje kompleksne orkestralne logike na čitljiviji i održiviji način.

Slijede smjernice korištenja LCEL-a:

- ukoliko je potreban jedan poziv prema jezičnom modelu, utoliko LCEL **nije** potreban već se preporuča pozvati temeljni model **direktno**

- ukoliko je potreban jednostavan lanac, npr. upit + LLM + parser, utoliko je LCEL razuma odabir, ako se pritom iskorištavaju njegove prednosti
- ukoliko se gradi kompleksni lanac s npr. grananjem, ciklusima, više agenata i sl., utoliko koristiti LangGraph unutar kojeg se LCEL uvijek može koristiti unutar pojedinačnih čvorova grafa

Generacija poboljšana dohvatom (engl. *RAG*)

Jedno od najmoćnijih načina korištenja razgovornih modela su sofisticirane aplikacije za postavljanje i odgovaranje na pitanja (engl. *Q&A chatbots*). To su aplikacije koje mogu odgovoriti na pitanja o specifičnom izvoru podataka i koriste tehniku generacije poboljšane dohvatom (engl. *retrieval augmented generation – RAG*).

Uobičajena RAG aplikacija ima dvije komponente:

1. **indeksiranje** – cjevovod za procesiranje i indeksiranje podataka iz nekog izvora, obično u vanmrežnom radu
2. **dohvat i generiranje** – sami RAG lanac koji uzima korisnikov upit za vrijeme izvođenja i dohvaća relevantne podatke iz indeksa koje zatim predaje modelu

Indeksiranje se satoji od tri koraka:

1. **učitavanje** – podatke je prvo potrebno učitati koristeći **učitavače dokumenata** (engl. *Document loaders*)
2. **razdvajanje** – razdvajivači teksta (engl. *Text splitters*) razlamaju velike dokumente u manje dijeliće što je korisno i za indeksiranje i za prosljeđivanje modelu radi veličine kontekstnog prozora
3. **pohranjivanje** – potrebno je negdje pohraniti i indeksirati razdvojene dijeliće kako bi se kasnije mogli pretražiti

Dohvat i generiranje sastoji se od dva koraka:

1. **dohvat** – po korisničkom ulazu, relevantni dijelići se dohvaćaju iz pohrane pomoću **pretraživača** (engl. *Retriever*)
2. **generiranje** – razgovorni ili jezični model proizvodi odgovor koristeći upit koji sadrži i korisnikovo pitanje i dohvaćene dijeliće podataka

Za dohvat i generiranje može se ponovo koristiti graf biblioteke *LangGraph*.

Učitavači dokumenata (engl. *Document loaders*)

Učitavači dokumenata osmišljeni su za učitavanje objekata dokumenta. Razvojni okvir *LangChain* posjeduje integracije s raznim izvorima podataka poput platforma *Slack*, *Notion*, *Google Drive* i sl.

Izgradnja web-aplikacije

U ovom poglavlju prikazana je izgradnja web-aplikacije za čavrljanje koja koristi i ugrađuje sve prije spomenute tehnologije i mehanizme.

Odabrana arhitektura

Za razvoj web-aplikacije odabrana je biblioteka *React*[2] za izradu web-sučelja, odnosno, odabran je razvojni okvir *Next.js*[3] koji omogućava klijentsko i poslužiteljsko izvođenje *React* aplikacija. Poslužiteljsko izvođenje je u ovom kontekstu potrebno za komunikaciju s pružateljima modela putem razvojnog okvira *LangChain*.

Glavna značajka odabrane arhitekture je biblioteka *React* jer omogućuje dinamičku hidraciju stranice što znači da se vrijednosti varijabli, odnosno **stanja** kako ih naziva biblioteka, mogu mijenjati bez ponovnog učitavanja stranice. Ova značajka je iznimno korisna za razvoj aplikacija čiji se sadržaj često mijenja, a želi se izbjeći ponovno učitavanje stranice, kao što je cilj i u ovom projektu jer želimo ulaze i izlaze, odnosno odgovore, korisnika i modela prikazivati na web-sučelju u stvarnom vremenu.

Biblioteka *React* i radni okvir *Next.js* su tehnologije zasnivane na programskom jeziku *Javascript*[4], a često se koriste i preporučaju koristiti kroz nadogradnju programskog jezika *Javascript* zvanu *TypeScript*[5].

Razvojni okvir *Tailwind CSS*

Tailwind CSS[6] je CSS razvojni okvir pun preddefiniranih klasa koje se mogu slagati jedna na drugu tokom pisanja HTML koda za lakši, brži i jednostasvniji razvoj grafičkog web-sučelja.

U sklopu ovog projekta koristi se za prilagodbu grafičkog web-sučelja aplikacije.

Biblioteka *shadcn/ui*

Biblioteka *shadcn/ui*[7] je biblioteka otvorenog koda koja sadrži skup ponovo upotrebljivih komponenti za izradu klijentskog web-sučelja. Bazirana je na razvojnog okviru *Tailwind CSS* i pruža visoki stupanj prilagodljivosti.

U sklopu ovog projekta koristi se za prilagodbu grafičkog web-sučelja aplikacije.

Biblioteka *lucide-react*

Biblioteka *lucide-react*[8] je *React* biblioteka za popularni skup grafičkih ikonica *lucide*[9] za korištenje u razvoju aplikacija.

U sklopu ovog projekta koristi se za prilagodbu grafičkog web-sučelja aplikacije.

Biblioteka *zod*

Biblioteka *zod* je biblioteka za definiranje i provjeru valjanosti shema, primarno za jezik *TypeScript*. Koristi se najčešće za definiranje i provjeru valjanosti shema za web-obrasce.

U sklopu ovog projekta koristi se za validaciju polja za slanje poruka prema modelu.

Postavljanje biblioteka za rad s razvojnim okvirom *LangChain*

Potrebne su sljedeće biblioteke: `@langchain/core`, `@langchain/langgraph` i `uuid`.

Moguće ih je instalirati pomoću odabranog upravitelja paketa, npr. `npm i @langchain/mistralai`.

Početak

Na početku potrebno je odabrati koji model želimo koristiti. Za razvoj ove web-aplikacije odabran je model **Mistral AI** koristeći njihovu besplatnu razinu korištenja.

Potrebno se registrirati na pružateljevoj stranici, odabrati besplatnu razinu i generirati ključ za aplikacijsko sučelje, odnosno API ključ.

Zatim je potrebno instalirati *LangChain* biblioteku za rad s odabranim modelom, u ovom slučaju pomoću naredbe `npm i @langchain/mistralai`.

Moraju se postaviti i potrebne varijable okruženja, u ovom slučaju varijabla okruženja koja sadrži generirani ključ aplikacijskog sučelja `MISTRAL_API_KEY=generirani-ključ`.

Potom se uspostavlja model kako je prikazano u *Kodu 7*.

Kod 7. `/app/no-markdown/page.tsx` – Uspostavljanje modela.

```
import { ChatMistralAI } from "@langchain/mistralai";

const llm = new ChatMistralAI({
  model: "mistral-large-latest",
  temperature: 0,
});
```

Zatim je potrebno stvoriti funkciju koja će pozivati model. Ova funkcija se mora izvršavati na poslužitelju jer je to zahtjev razvojnog okvira *LangChain* i jer mora koristiti varijablu okruženja koja sadrži ključ za aplikacijsko sučelje. Taj ključ mora biti tajan i pohranjen je samo na poslužitelju.

Pozivanje funkcije na poslužitelju ostvarujemo koristeći mehanizam *Server Actions* razvojnog okvira *Next.js* koji omogućava jednostavno definiranje dijela koda koji se mora izvršavati na poslužitelju unutar datoteke u kojoj se inače može nalaziti i kod klijenta.

Funkcija mora biti asinkrona kako bi mogli pričekati odgovor modela, a kao ulaz prima objekt tipa `BaseLanguageModelInput`.

Sve opisano prikazano je u *Kodu 8*.

Kod 8. `/app/no-markdown/page.tsx` – Funkcija poziva modela.

```
import { BaseLanguageModelInput } from "@langchain/core/language_models/base";

const invokeFn = async (input: BaseLanguageModelInput) => {
  "use server";
  const response = await llm.invoke(input);
  console.log(response);
  return response.content.toString();
};
```

Sada je potrebna komponenta za prikaz sučelja koje će:

- korisniku omogućiti upis i slanje poruke
- prikazati korisnikovu poruku
- prikazati odgovor modela

Komponenta je nazvana `ChatbotNoMarkdown` (jer ćemo kasnije dodati podršku za format *Markdown*), a sadrži sljedeće značajke:

- *React* stanje `answer` pomoću kojeg ćemo prikazivati odgovor modela bez potrebe za ručnim osvježavanjem sučelja – *Kod 9*
- *React* stanje `input` pomoću kojeg ćemo prikazati korisnikov ulaz kao poruku u razgovoru – *Kod 10*
- *React* stanje `loading` koje ćemo postaviti na vrijednost `true` pri pozivu funkcije poziva modela, a postaviti na `false` kada funkcija završi. Služi nam za prikaz indikatora učitavanja kako bi korisnik znao da nešto čeka. – *Kod 11*
- Grafički stilizirani prikaz razgovora na web-sučelju aplikacije
- Web-obrazac `MessageForm` za slanje poruka, odnosno ulaza, prema modelu. Komponenta web-obrasca bit će obrađena kasnije. – *Kod 12*

Komponenta kao ulazni parametar prima prije spomenutu funkciju poziva modela `invoke` i predaje ju komponenti web-obrasca koji će tu funkciju zapravo i pozivati (*Kod 13*).

Kod 9. `/components/chatbot-no-markdown.tsx` – *React* stanje `answer`

```
...
const [answer, setAnswer] = useState<string>();
...
<div className="flex flex-col gap-4">
  ...
  {answer &&
    <Card className="p-4 bg-orange-500 text-white max-w-[90%] self-
      {answer}
    </Card>
```

```
  )}
</div>
...
```

Kod 10. /components/chatbot-no-markdown.tsx – *React* stanje input

```
...
const [input, setInput] = useState<string>();
...
<div className="flex flex-col gap-4">
  {input && (
    <Card className="p-4 bg-slate-200 max-w-[90%] self-end">{input}</Card>
  )}
  ...
</div>
...
```

Kod 11. /components/chatbot-no-markdown.tsx – *React* stanje loading

```
...
const [loading, setLoading] = useState<boolean>(false);
...
<div className="flex flex-col gap-4">
  ...
  {loading && (
    <Card className="p-4 bg-orange-500 text-white self-start max-w-[90%]">
      <span className="animate-ping h-2 inline-flex w-2 rounded-full bg-
white"></span>
    </Card>
  )}
  ...
</div>
...
```

Kod 12. /components/chatbot-no-markdown.tsx – Web-obrazac MessageForm

```
import MessageForm from "./message-form";
...
<MessageForm
  invoke={invoke}
  setAnswer={setAnswer}
  setInput={setInput}
  setLoading={setLoading}
/>
...
```

Kod 13. /components/chatbot-no-markdown.tsx – Ulazni parametar invoke

```
interface ChatbotProps {
  invoke: (input: BaseLanguageModelInput) => Promise<string>;
}
const ChatbotNoMarkdown = ({ invoke }: ChatbotProps) => {
  ...
};
```

Web-obrazac MessageForm kao ulazne parametre (*Kod 14*) prima:

- invoke – funkcija poziva modela
- setAnswer – funkcija postavljanja nove vrijednosti stanja answer

- `setInput` – funkcija postavljanja nove vrijednosti stanja `input`
- `setLoading` – funkcija postavljanja nove vrijednosti stanja `loading`

Kod 14. `/components/message-form.tsx` – Ulazni parametri web-obrasca `MessageForm`

```
interface MessageFormProps {
  invoke: (
    input: BaseLanguageModelInput,
    options?: ChatMistralAICallOptions | undefined
  ) => Promise<string>;
  setAnswer: Dispatch<SetStateAction<string | undefined>>;
  setInput: Dispatch<SetStateAction<string | undefined>>;
  setLoading: Dispatch<SetStateAction<boolean>>;
}
```

Na početku komponente web-obrasca koristi se biblioteka *zod* za definiranje sheme validacije samog obrasca (*Kod 15*). Zahtjeva se upis u obliku običnog teksta s duljinom od barem jednog znaka.

Kod 15. `/components/message-form.tsx` – Definicija sheme validacije obrasca

```
import { z } from "zod";

const schema = z.object({
  message: z.string().min(1),
});
```

Zatim se stvara objekt obrasca koristeći definiranu shemu (*Kod 16*) i postavljaju se početne vrijednosti.

Kod 16. `/components/message-form.tsx` – Objekt obrasca koji koristi shemu validacije

```
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";

const form = useForm<z.infer<typeof schema>>({
  resolver: zodResolver(schema),
  defaultValues: {
    message: "",
  },
});
```

Potom je potrebno definirati funkciju predaje obrasca (*Kod 17*). Ona mora biti asinkrona jer se u njoj poziva funkcija poziva modela i poželjno je pričekati njeno izvršavanje.

Na početku poništava trenutni odgovor, postavlja novu vrijednost stanja `input` i postavlja stanje `loading` te poništava upisane podatke u obrascu.

Zatim pokušava izvršiti poziv funkcije modela i ako ona uspješno završi, gasi učitavanje, postavlja novu vrijednost stanja `answer` i završava s radom.

Kod 17. `/components/message-form.tsx` – Funkcija predaje obrasca

```
const onSubmit = async (data: z.infer<typeof schema>) => {
  setAnswer(undefined);
  setInput(data.message);
  setLoading(true);
  form.reset();
  try {
    const invokePromise = invoke(data.message);
    const response = await invokePromise;
    console.log("response", response);
    setLoading(false);
    setAnswer(response);
  } catch (err) {
    console.error(err);
    setLoading(false);
    setAnswer("Greška pri dohvaćanju odgovora");
  }
};
```

Na posljetku je potrebno formu prikazati korištenjem biblioteka za izradu grafičkog web-sučelja.

U primjeru izlaza dobivenog nakon poziva modela, vidljivo je da model vraća odgovore oblikovane formatom *Markdown*[10] (*Kod 18*) (*Slika 2*).

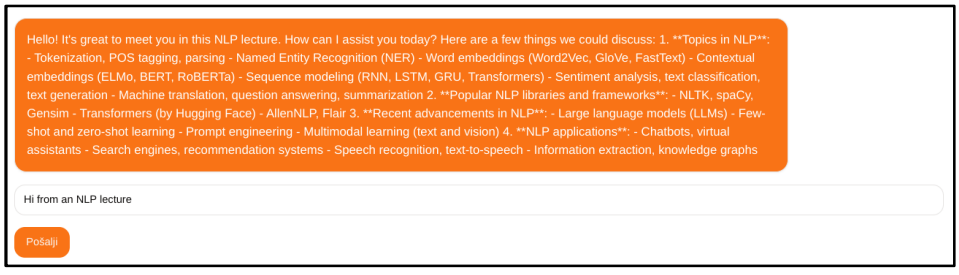
Kod 18. Primjer izlaza modela

```
Hello! It's great to meet you in this NLP lecture. How can I assist you today?
Here are a few things we could discuss:

1. **Topics in NLP**:
   - Tokenization, POS tagging, parsing
   - Named Entity Recognition (NER)
   - Word embeddings (Word2Vec, GloVe, FastText)
   - Contextual embeddings (ELMo, BERT, RoBERTa)
   - Sequence modeling (RNN, LSTM, GRU, Transformers)
   - Sentiment analysis, text classification, text generation
   - Machine translation, question answering, summarization
```

Markdown je popularan **označni jezik** (engl. *markup language*) za brzo oblikovanje teksta i često se koristi na web-mjestima.

Slika 2. `/app/no-markdown/page.tsx` - Prikaz odgovora modela bez podrške za format *Markdown*



Podrška za format *Markdown*

Podrška za format *Markdown* ugrađena je pomoću biblioteke *react-markdown* i njene komponente `Markdown`.

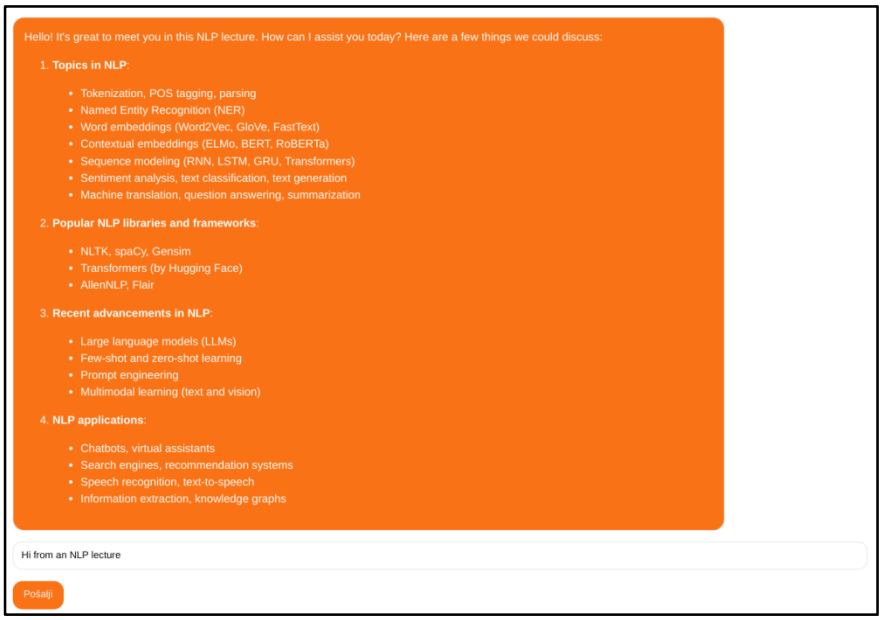
Promjene u odnosu na prethodni korak počinju tek od komponente za prikaz sučelja koja sada postaje `ChatbotMarkdown`. Stanja `answer` i `input` više ne prikazujemo kako jesu, već ih omotavamo u komponentu `Markdown` (*Kod 19*) koja preuzima cijeli posao pretvaranja formata *Markdown* u ekvivalentne HTML elemente.

Kod 19. `/components/chatbot-markdown.tsx` – Korištenje komponente `Markdown` za prikaz formatiranog teksta

```
import Markdown from "react-markdown";
...
{input && (
  <Card className="p-4 bg-slate-200 max-w-5/6 self-end">
  <Markdown>{input}</Markdown>
  </Card>
)}
...
{answer && (
  <Card className="p-4 bg-orange-500 text-white w-5/6 self-start">
  <Markdown>{answer}</Markdown>
  </Card>
)}
...
```

Ostatak komponenti i načina korištenja ostaje isti, a kako sada izgleda odgovor modela vidljivo je na *Slici 3*.

Slika 3. `/app/no-markdown/page.tsx` - Prikaz odgovora modela s podrškom za format *Markdown*



Trenutno aplikacija nema podršku za perzistenciju poruka pa model ne može koristiti informacije do sad prisutne u razgovoru niti korisniku pokazuje što je sve modelu slao i što mu je sve model zatim odgovorio.

Podrška za perzistenciju poruka

Perzistencija poruka ostvarena je bibliotekom *LangGraph* pomoću ugrađenog sloja perzistencije.

Aplikaciju je potrebno omotati u minimalni sloj biblioteke *LangGraph* i to će biti dovoljno za automatsko omogućavanje perzistencije poruka.

Potrebno je napisati novu pozivajuću funkciju koja će pozivati prijašnju funkciju poziva modela (*Kod 20*).

Kod 20. `/app/message-persistance/page.tsx` – Funkcija pomoću koje graf poziva funkciju poziva modela

```
import { MessagesAnnotation } from "@langchain/langgraph";

const callModel = async (state: typeof MessagesAnnotation.State) => {
  const response = await llm.invoke(state.messages);
  return { messages: response };
};
```

Zatim je potrebno definirati graf i u njemu definirati potrebne čvorove (*Kod 21*). Obzirom da je perzistencija poruka ugrađena funkcionalnost, nije potrebno definirati napredni graf pa su jedini čvorovi čvor modela, početka i kraja grafa. Čvoru modela predajemo funkciju `callModel` iz *Koda 20*.

Kod 21. `/app/message-persistance/page.tsx` – Definicija grafa

```
import { START, END, MessagesAnnotation, StateGraph } from
"@langchain/langgraph";

const workflow = new StateGraph(MessagesAnnotation)
  .addNode("model", callModel)
  .addEdge(START, "model")
  .addEdge("model", END);
```

Kod izgradnje grafa, potrebno je definirati mjesto pohrane koje će koristiti za perzistenciju poruka (*Kod 22*). Biblioteka *LangGraph* i za to nudi gotovu metodu `MemorySaver()`.

Kod 22. `/app/message-persistance/page.tsx` – Definicija i postavljanje mjesta pohrane poruka

```
import { MemorySaver } from "@langchain/langgraph";

const memory = new MemorySaver();
const app = workflow.compile({ checkpointer: memory });
```

Potrebno je također izmijeniti funkciju poziva modela (*Kod 23*) koja uz dosadašnji ulaz prima i identifikator dretve za potrebe razlikovanja razgovora. Ovaj identifikator se u

web-aplikaciji koristi za poništavanje trenutnog razgovora, tako da se preda novi identifikator i time započne novi razgovor.

Također, na drugačiji način se sami ulaz predaje samom model, a odgovor modela je sada polje svih poruka s dodanom novom porukom, tako da kao izlaz funkcije predajemo samo zadnju poruku u polju, koja je nova poruka.

Kod 23. `/app/message-persistance/page.tsx` – Izmjena funkcije poziva modela

```
const invokeFn = async (message: string, threadId: string) => {
  "use server";
  const input = [
    {
      role: "user",
      content: message,
    },
  ];
  const config = { configurable: { thread_id: threadId } };
  const output = await app.invoke({ messages: input }, config);
  console.log("output", output);
  return output.messages[output.messages.length - 1].content.toString();
};
```

Komponenta prikaza sučelja također sadrži značajne promjene te se sada radi o komponenti `ChatbotMessagePersistence` (*Kod 24*).

Više nije potrebno stanje input, a stanje answer postaje stanje answers koje je sada polje tipa string koje pohranjuje i korisničke upite i odgovore modela. U izradi grafičkog sučelja, iterira se kroz polje i ovisno o parnom ili neparnom indeksu elementa, prikazuje se korisnički, ili odgovor modela.

Kod 24. `/components/chatbot-message-persistence.tsx` – *React* stanje `answers`

```
const [answers, setAnswers] = useState<string[]>();
...
{answers &&
  answers.map((a, i) => (
    ...
))}
```

Izmijenjena je i komponenta web-obrasca koja sada postaje MessageFormMessagePersistence (*Kod 25*) i uz to što na drugačiji način ažurira stanje answers (prije answer), zadužena je i za generiranje, predaju i ponovno postavljanje identifikatora dretve razgovora.

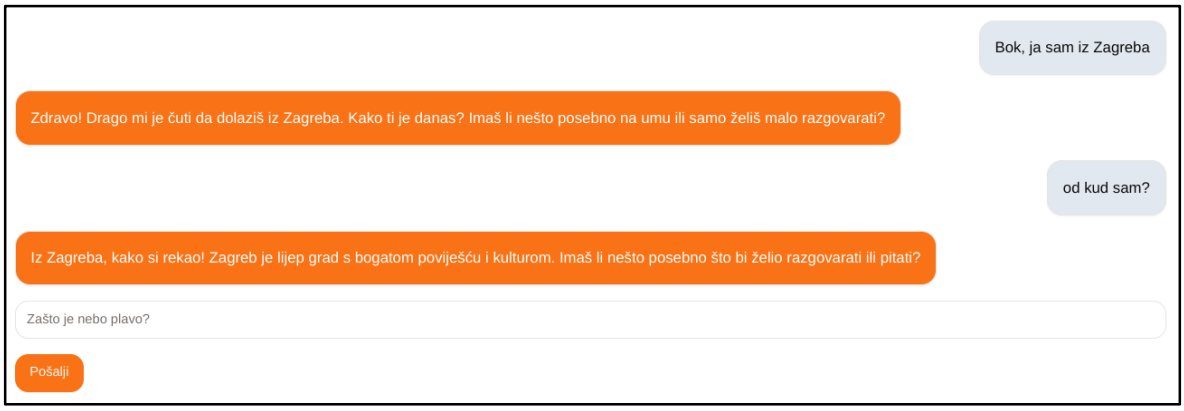
Kod 25. `/components/message-form-message-persistence.tsx` – Komponenta web-obrasca s podrškom za perzistenciju razgovora

```
import { v4 as uuidv4 } from "uuid";
interface MessageFormProps {
  ...
  setAnswers: Dispatch<SetStateAction<string[] | undefined>>;
  ...
}
...
const [threadId, setThreadId] = useState<string>(uuidv4());
const handleThreadIdReset = () => { setThreadId(uuidv4()); setAns-
```

```
wers(undefined); };
...
const onSubmit = async (data: z.infer<typeof schema>) => {
  setAnswers((prev) => {
    if (prev) { return [...prev, data.message]; }
    return [data.message];
  });
  ...
  try {
    const invokePromise = invoke(data.message, threadId);
    ...
    setAnswers((p) => { if (p) { return [...p, response]; } return [response];
});
  } catch (err) { ... }
};
...
```

Model sada može pamtiti informacije unesene u razgovor i korisnik ima prikaz svih razmijenjenih poruka unutar razgovora (*Slika 4*).

Slika 4. `/app/message-persistence/page.tsx` – Prikaz razgovora s perzistencijom poruka



Idući korak je korištenje upitnih predložaka.

Korištenje upitnih predložaka

Pomoću upitnih predložaka moguće je postaviti inicijalni kontekst i reći modelu kako da razgovara s korisnikom.

Nisu korištene sve napredne mogućnosti predložaka, ali postavljeno je da model s korisnikom priča samo na hrvatskom jeziku i da svaki odgovor završi s kratkim vicom (*Kod 26*).

Kod 26. `/app/prompt-templates/page.tsx` – Definicija upitnog predloška

```
const promptTemplate = ChatPromptTemplate.fromMessages([
  [
    "system",
    "Pričaj samo na Hrvatskom jeziku. Odgovor završi s kratkim vicom.",
  ],
  ["placeholder", "{messages}"],
]);
```

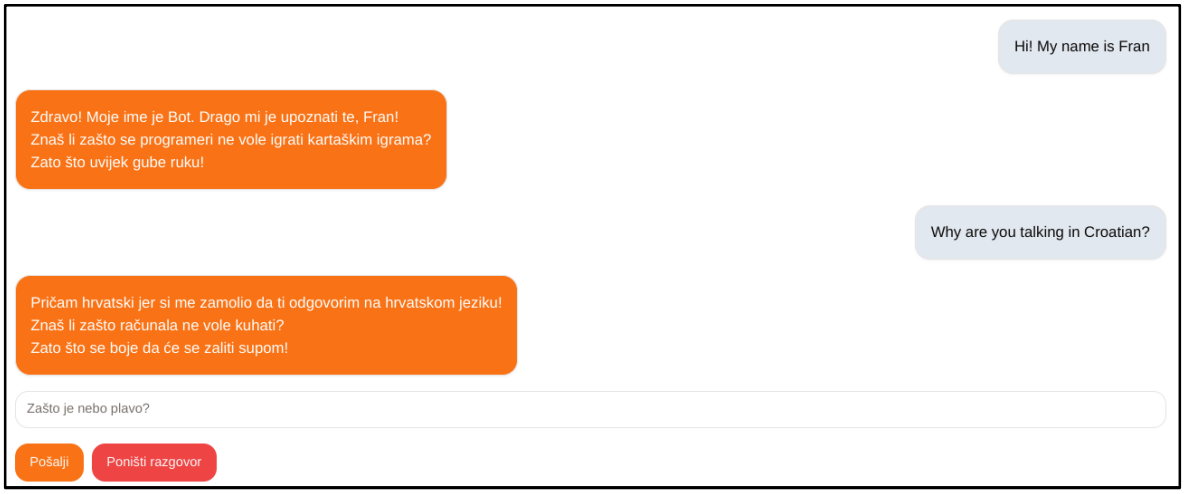
Potrebno je izmijeniti i funkciju poziva koja se predaje grafu izvršavanja (*Kod 27*) jer se više ne predaje ulaz, već se ulaz provodi kroz upit i modelu se zatim predaje upit.

Kod 27. `/app/prompt-templates/page.tsx` – Izmijena grafu predane funkcije poziva da koristi upitne predloške

```
const callModel = async (state: typeof MessagesAnnotation.State) => {
  const prompt = await promptTemplate.invoke(state);
  const response = await llm.invoke(prompt);
  return { messages: [response] };
};
```

Ostatak programskog koda ostaje isti kao u prethodnom koraku, a na *Slici 5* vidljivo je kako model odgovara na hrvatskom jeziku iako mu korisnik piše na engleskom i da svaki odgovor završava s kratkim vicom.

Slika 5. Prikaz razgovora s korištenjem upitnog predloška i sistemskom porukom



Upravljanje poviješću razgovora

Ranije je spomenuta važnost upravljanja poviješću razgovora kako kontekstni prozor modela ne bi bio prekoračen.

Razvojni okvir *LangChain* pruža ugrađene metode za upravljanje listom poruka, jedna od kojih je `trimMessages()`. Metoda prima nekoliko konfiguracijskih opcija, neke od kojih su:

- `maxTokens` – maksimalni dozvoljeni broj tokena, višak se krati

- `strategy` – od kuda broji tokene – ako postavimo `last` onda ostavlja zadnje tokene, a briše najprije dodane
- `tokenCounter` – definira što je token i kako se broji, u ovoj aplikaciji token je definiran kao duljina liste poruka pa smije biti najviše 6 poruka u listi
- `includeSystem` – ostaviti sistemsku poruku ili ne u listi poruka
- `allowPartial` – smije li kratiti i dijelove poruka
- `startOn` – od kojeg tipa poruka da krene brojati

Potrebno je definirati objekt `trimmer` kojim će se kasnije kratiti povijest razgovora pozivajući spomenutu metodu (*Kod 28*).

Kod 28. `/app/manage-history/page.tsx` – Definicija objekta za upravljanje poviješću razgovora

```
import { trimMessages } from "@langchain/core/messages";

const trimmer = trimMessages({
  maxTokens: 6,
  strategy: "last",
  tokenCounter: (msgs) => msgs.length,
  includeSystem: true,
  allowPartial: false,
  startOn: "human",
});
```

Ovom definicijom koristi se zadnjih 6 poruka u našoj povijesti, uključujući i početnu sistemsku poruku te brojanje započinje tek od korisničke poruke.

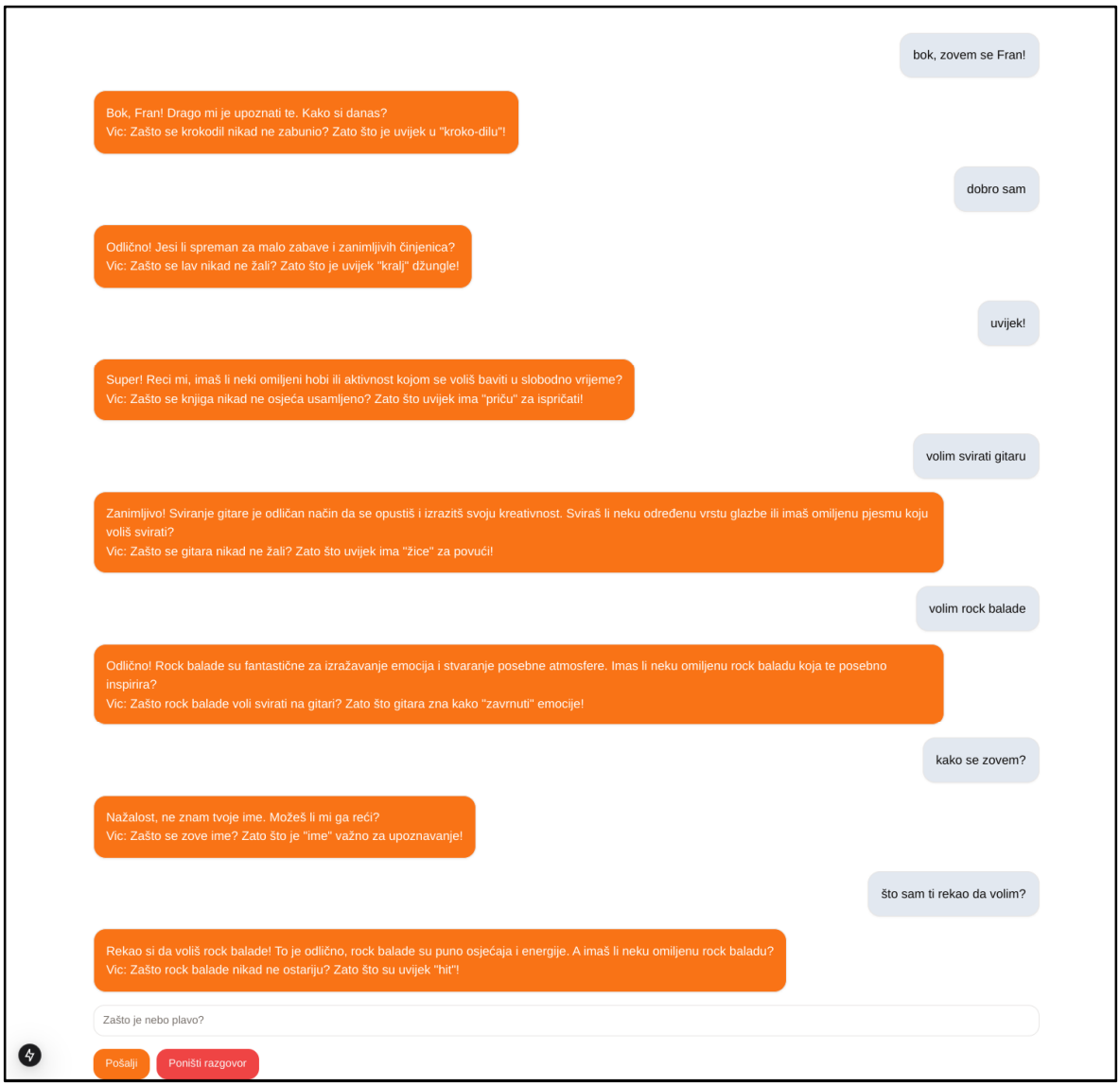
Iduće je potrebno koristiti ovaj objekt tako da se modificira funkcija poziva koja se predaje grafu (*Kod 29*). Upitnom predlošku sada predajemo *odrezane* poruke.

Kod 29. `/app/manage-history/page.tsx` – Definicija objekta za upravljanje poviješću razgovora

```
const callModel = async (state: typeof MessagesAnnotation.State) => {
  const trimmed = await trimmer.invoke(state.messages);
  const prompt = await promptTemplate.invoke({
    ...state,
    messages: trimmed,
  });
  ...
};
```

Na *Slici 6* vidljivo je da model na početku zna kako se korisnik zove, ali kasnije to više ne zna jer mu ta poruka više nije u danoj povijesti već je odrezana. Međutim, model i dalje zna da korisnik voli rock balade jer mu ta poruka još stoji u predanoj povijesti.

Slika 6. Prikaz razgovora s upravljanjem poviješću



Razgovor u obliku toka

Poželjno je interakciju s robotom napraviti responzivnijom kako korisnik nebi imao osjećaj nešto dugo čeka, već da pred njim dolaze odgovori modela onako kako ih on i stvara. To je moguće napraviti korištenjem ugrađene funkcionalnosti tokova.

Prvo je potrebno izmijeniti graf tako da se drugačija definira čvor početka (*Kod 30*), a zatim je potrebno značajno izmijeniti samu funkciju poziva modela.

Kod 30. `/app/stream/page.tsx` – Definicija grafa za rad s tokovima

```
const workflow = new StateGraph(MessagesAnnotation)
  .addNode("model", callModel)
  .addEdge("__start__", "model")
  .addEdge("model", END);
```

U funkciji poziva modela prvo je potrebno izmijeniti konfiguraciju koja se predaje pozivu modela tako da se doda parametar `streamMode` koji definira način rada toka (*Kod 31*).

Kod 31. `/app/stream/page.tsx` – Izmijena konfiguracije u funkciji poziva modela

```
const config = {
  streamMode: "messages",
  configurable: {
    thread_id: threadId,
  },
};
```

Zatim je cijelu funkciju potrebno pretvoriti iz asinkrone u asinkronu generatrosku[11] (*Kod 32*). Asinkrone funkcije vraća jedan `Promise` ključnm riječi `return` i završava s radom, dok asinkrona generatorska funkcija može generirati više objekata tipa `Promise` ključnom rijčei `yield`, svaki put čekajući dok se ne zatraži iduća vrijednost. Njeni rezultati traže se i čekaju pomoću asinkronih petlji `for await ... of`.

Ovdje je prvo potrebno stvoriti takvu petlju jer za tok pozivamo ugrađenu funkciju `.stream()` koja je i sama asinkrona generatorska funkcija, a zatim dobivene dijeliće odgovora vraćamo pozivatelju na klijentu (*Kod 35*) koji će također asinkronom petljom dohvaćati poslane dijeliće odgovora. Ovaj dio se možda čini redundantan, ali je potreban jer ne možemo `.stream()` metodu pozvati na klijentu već na poslužitelju.

Kod 32. `/app/stream/page.tsx` – Pretvorba funkcije poziva modela u asinkronu generatorsku funkciju i slanje dobivenih dijelova toka

```
const invokeFn = async function* (message: string, threadId: string) {
  ...
  for await (const [message] of await app.stream(
    { messages: input },
    config
  )) {
    yield message.content as string;
  }
};
```

Iduće izmjene su u komponenti prikaza sučelja koja sada postaje `ChatbotStream`. Potrebno je izmijeniti definicju tipa funkcije poziva modela koja se predaje komponenti (*Kod 33*) i dodajemo novo stanje `newAnswer` (*Kod 34*) koje se ažurira s novim dijelićima novog odgovora modela kako on pristiže, a na poslijetku se dodaje u prijašnje stanje svih odgovora `answers` i čisti se `newAnswer`.

Kod 33. `/components/chatbot-stream.tsx` – Izmijena definicije funkcije poziva modela

```
interface ChatbotProps {
  invoke: (
    message: string,
    threadId: string
  ) => AsyncGenerator<string, void, unknown>;
}
```

Kod 34. `/components/chatbot-stream.tsx` – Novo stanje `newAnswer` za prikaz dijelića odgovora kako pristižu u toku

```
const [newAnswer, setNewAnswer] = useState<string>();
{newAnswer && (
  <Card className="..."> <Markdown>{newAnswer}</Markdown> </Card>
)}
```

Iduća izmijena je u komponenti web-obrasca gdje je potrebno značajno izmijeniti funkciju predaje obrasca (*Kod 35*) obzirom da ona sada mora komunicirati s asinkronom generatorskom funkciju, umjesto samo asinkronom.

Kod 35. `/components/message-form-stream.tsx` – Izmijena funkcije predaje obrasca za komunikaciju s generatorskom funkcijom toka

```
const onSubmit = async (data: z.infer<typeof schema>) => {
  try {
    const invokeGen = invoke(data.message, threadId);
    let newAnswer = "";
    for await (const chunk of await invokeGen) {
      setLoading(false);
      setNewAnswer((prev) => {
        if (!prev) return chunk;
        return prev.concat(chunk);
      });
      newAnswer = newAnswer.concat(chunk);
    }
    setAnswers((prev) => {
      if (prev) {
        return [...prev, newAnswer];
      }
      return [newAnswer];
    });
    setNewAnswer(undefined);
    setDisabled(false);
  } catch (err) { ... }
};
```

Sada korisniku izgleda kao da model zapravo *piše* svoj odgovor umjesto da čeka duže vrijeme dok ne dobije cijeli odgovor kao blok teksta odjednom.

Generacija poboljšana dohvatom (engl. *RAG*)

Idući korak nije toliko nadogradnja, koliko je primjer drugačijeg, ali izrazito čestog, načina korištenja razgovornih modela.

Na novoj ruti napravljeno je sučelje za postavljanje i odgovaranje na pitanja o specifičnoj temi. Konkretno, koristimo obavijest s web-stranice FER-a u kojoj docent Marko Horvat priča o energetskoj potrošnji umjetne inteligencije[12].

Prvo je potrebno definirati ugradbeni model odabranog pružatelja jezičnog modela i njime stvoriti pohranu vektora (*Kod 36*).

Kod 36. /app/rag/page.tsx – Stvaranje pohrane vektora

```
import { MemoryVectorStore } from "langchain/vectorstores/memory";
import { MistralAIEmbeddings } from "@langchain/mistralai";

const embeddings = new MistralAIEmbeddings({
  model: "mistral-embed",
});
const vectorStore = new MemoryVectorStore(embeddings);
```

Iduće je potrebno dohvatiti podatke i razlomiti ih na djeliće (*Kod 37*). Za dohvat se koristi ugrađena integracija razvojonog okvira *LangChain* s bibliotekom *Cheerio*[13], popularna biblioteka za parsiranje i manipuliranje HTML i XML podacima. Razlamanje dokumenta na djeliće moguće je jednostavno obaviti pomoću ugrađene metode `RecursiveCharacterTextSplitter` koja se prvo konfigurira određivanjem veličine dijelića i koliko je dozvoljeno preklapanje, a zatim joj se predaje sami dokument.

Kod 37. /app/rag/page.tsx – Dohvat i razlamanje dokumenta na djeliće

```
import { CheerioWebBaseLoader } from
"@langchain/community/document_loaders/web/cheerio";
import { RecursiveCharacterTextSplitter } from "@langchain/textsplitters";

const cheerioLoader = new CheerioWebBaseLoader(
  "https://www.fer.unizg.hr/novosti/istrazivanja?@=30mua#news_96801",
);
const docs = await cheerioLoader.load();
const splitter = new RecursiveCharacterTextSplitter({
  chunkSize: 1000,
  chunkOverlap: 200,
});
const allSplits = await splitter.splitDocuments(docs);
```

Zatim je potrebno dobivene djeliće indeksirati pomoću što se postiže dodavanjem dijelića u vektorsku pohranu koja taj dio odrađuje automatski (*Kod 38*).

Kod 38. /app/rag/page.tsx – Pohrana i indeksiranje dijelića

```
await vectorStore.addDocuments(allSplits);
```

Potrebno je definirati i upitni predložak za rad s RAG-om (*Kod 39*) kao i potrebna stanja koja graf mora koristiti i pratiti (*Kod 40*).

Kod 39. /app/rag/page.tsx – Upitni predložak za RAG

```
const promptTemplate = await pull<ChatPromptTemplate>("rlm/rag-prompt");
```

Kod 40. /app/rag/page.tsx – Potrebna stanja za rad s RAG-om u grafu

```
const InputStateAnnotation = Annotation.Root({
  question: Annotation<string>,
});

const StateAnnotation = Annotation.Root({
  question: Annotation<string>,
  context: Annotation<Document[]>,
  answer: Annotation<string>,
});
```

Slijedi definicija funkcija za sami dohvat (*Kod 41*) i generaciju (*Kod 42*). Funkcija za dohvat iz vektorske pohrane dohvaća relevantne dijeliće koji odgovaraju korisničkom upitu, a funkcija za generaciju na temelju dobivenog konteksta stvara svoj odgovor.

Kod 41. `/app/rag/page.tsx` – Funkcija za dohvat

```
const retrieve = async (state: typeof InputStateAnnotation.State) => {
  const retrievedDocs = await vectorStore.similaritySearch(state.question);
  return { context: retrievedDocs };
};
```

Kod 42. `/app/rag/page.tsx` – Funkcija za generaciju

```
const generate = async (state: typeof StateAnnotation.State) => {
  const docsContent = state.context.map((doc) => doc.pageContent).join("\n");
  const messages = await promptTemplate.invoke({
    question: state.question,
    context: docsContent,
  });
  const response = await llm.invoke(messages);
  return { answer: response.content };
};
```

Zatim je potrebno izmijeniti graf tako da koristi nove funkcije (*Kod 43*).

Kod 43. `/app/rag/page.tsx` – Izmijena grafa

```
const workflow = new StateGraph(StateAnnotation)
  .addNode("retrieve", retrieve)
  .addNode("generate", generate)
  .addEdge("__start__", "retrieve")
  .addEdge("retrieve", "generate")
  .addEdge("generate", "__end__");
const app = workflow.compile();
```

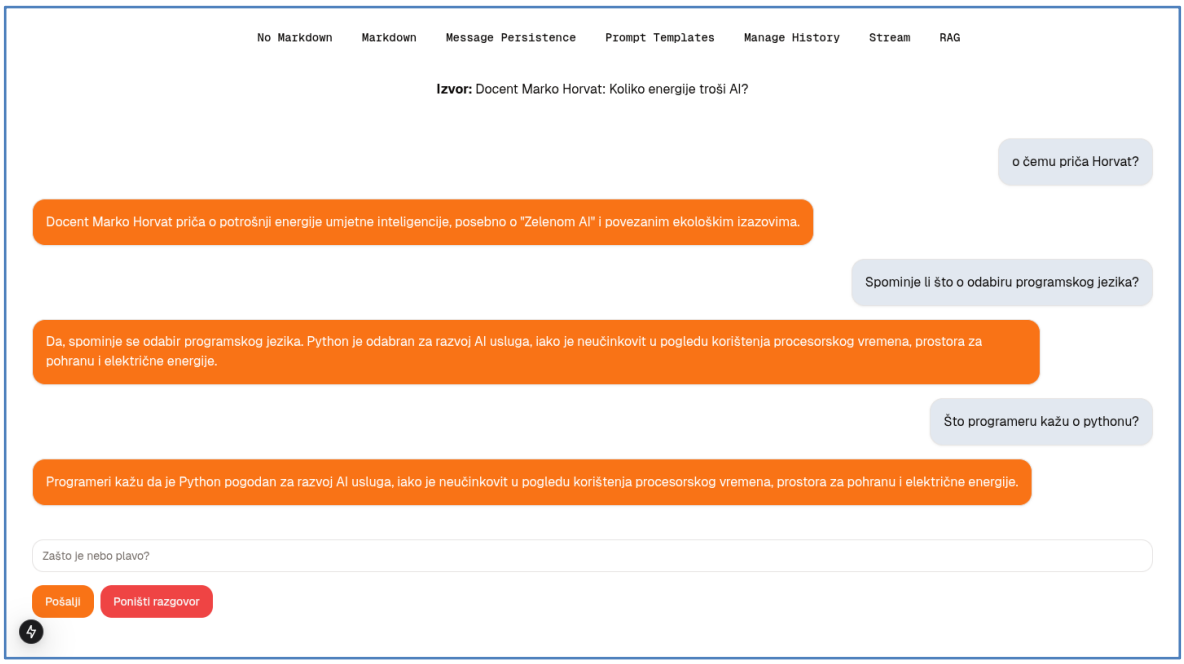
Na posljetku je potrebna mala izmjena u funkciji poziva model gdje je potrebno izmijeniti objekt ulaza (*Kod 44*).

Kod 44. `/app/rag/page.tsx` – Izmijena funkcije poziva modela

```
const input = { question: message };
```

Ostatak programskog koda ostaje isti, a upita li se sada razgovorni model o informacijama koje nisu opće poznato već su dostupne u prije spomenutoj obavijesti, znat će odgovoriti s točnim informacijama (*Slika 7)*.

Slika 7. Prikaz razgovora s generacijom poboljšanom dohvatom



Zaključak

Razvojni okvir *LangChain* uvelike olakšava izgradnju aplikacija koje pokreću veliki jezični i razgovorni modeli. Pruža velik broj standardnih sučelja i metoda koje su neovisne o samom korištenom modelu. To je od velikog značaja u slučaju da određeni pružatelj modela promijeni uvjete pružanja ili cijenu korištenja jer je moguće samo zamijeniti korišteni model bez ili uz minimalne promjene ostatka programskog koda.

Dokumentacija je vrlo dobra s visokim stupnjem pružanja detalja, ali nije savršena i zahtjeva određeni stupanj znanja i snalažljivosti od razvojnog inženjera.

Moguće nadogradnje razvijene web-aplikacije su podrška za spremanje razgovora i mogućnost odabira u koji razgovor korisnik želi ući. Dodatna nadogradnja za generiranje poboljšano dohvatom može bit implementacija i te funkcionalnosti i upravljanje poviješću s perzistencijom razgovora.

Razvojni okvir svakako je nešto za preporučiti kod izrade aplikacija koje rade s jezičnim i razgovornim modelima. Omogućava čak i razvojnim inženjerima bez iskustva i znanja o jezičnim modelima izgradnju aplikacija koje u potpunosti iskorištavaju njihove moći za praktične primjene.

O autoru

Fran Markulin student je zadnje godine diplomskog studija Programsko inženjerstvo i informacijski sustavi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Ovaj rad napisan je u sklopu kolegija Obrada prirodnog jezika.

Literatura

1. LangChain. *LangChain*. https://www.langchain.com/, zadnje pristupljeno 05.02.2025.
2. Meta Open Source. *React*. https://react.dev/, zadnje pristupljeno 05.02.2025.
3. Vercel. *Next.js*. https://nextjs.org/, zadnje pristupljeno 05.02.2025.
4. Ecma International. ECMAScript® 2024 Language Specification. https://262.ecma-international.org/, zadnje pristupljeno 05.02.2025.
5. Microsoft. *TypeScript*. https://www.typescriptlang.org/, zadnje pristupljeno 05.02.2025.
6. Tailwind Labs. *Tailwind*. https://tailwindcss.com/, zadnje pristupljeno 08.02.2025.
7. shadcn. *shadcn/ui*. https://ui.shadcn.com/, 08.02.2025.
8. Lucide Contributors. *lucide-react*. https://lucide.dev/guide/packages/lucide-react, zadnje pristupljeno 08.02.2025.
9. Lucide Contributors. *lucide*. https://lucide.dev/, zadnje pristupljeno 08.02.2025.
10. Wikipedia. *Markdown*. https://en.wikipedia.org/wiki/Markdown, zadnje pristupljeno 12.02.2025.
11. Mozilla. *async function**. mdn web docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function*, zadnje pristupljeno 13.02.2025.
12. Škaberna, P. *Docent Marko Horvat: Koliko energije troši AI?*. FER. https://www.fer.unizg.hr/novosti/istrazivanja?@=30mua#news_96801, zadnje pristupljeno 13.02.2025.
13. Cheerio doprinositelji. *cheerio*. https://cheerio.js.org/, zadnje pristupljeno 13.02.2025.