ИНСТИТУТ ЗА МАТЕМАТИКУ И ИНФОРМАТИКУ
ПРИРОДНО-МАТЕМАТИЧКИ ФАКУЛТЕТ
УНИВЕРЗИТЕТ У КРАГУЈЕВЦУ

ЗАВРШНИ РАД

РАЗВОЈ ИНТЕЛИГЕНТНОГ АСИСТЕНТА ЗА ПРОГРАМИРАЊЕ

Ментор
др Бранко Арсић

Студент
Марија Јоловић, 46/21

Новембар 2025.

Садржај

Предговор

У последњих неколико година, развој вештачке интелигенције, посебно у области обраде природног језика (*Natural Language Processing* - NLP), довео је до значајних иновација које су трансформисале многе индустрије. Појава великих језичких модела (*Large Language Models* - LLM) као што су GPT, LLaMA, StarCoder и други модели о којима ће касније бити више приче, показала је да ови модели могу успешно да разумеју, анализирају и генеришу текст на нивоу који се раније сматрао недостижним. Њихова примена проширила се од општих задатака у разумевању текста до веома специфичних области као што су генерисање кода, аутоматско довршавање функција, откривање потенцијалних грешака у коду, ревизије и тестирање кода.

Мотивација за овај рад произашла је из знатижеље како велики језички модели могу додатно унапредити алате за асистенцију у програмирању и на тај начин помоћи програмерима. Инспирацију представљају савремени системи попут *GitHub Copilot*-а, који показују да је могуће развити интелигентног асистента који разуме контекст кода и предлаже релевантне наставке. Међутим, ови системи често имају потешкоћа при раду са специфичним или доменски ограниченим кодом, што је подстакло истраживање могућности развоја прилагођеног модела обученог на циљаним подацима, способног да боље разуме контекст и стил одређеног окружења. Циљ овог рада је да се анализира на који начин претренирани језички модели, специфично обучени на подацима који садрже програмски код, могу користити за развој ефикасног асистента, интегрисаног у популарно развојно окружење, као што је *Visual Studio Code*.

Циљ рада је да се истражи и демонстрира процес развоја асистента за програмирање који користи модел дубоког учења као основу за предвиђање наредног дела кода на основу тренутног контекста. Рад обухвата:

1. анализу архитектуре трансформера и механизма пажње који представљају основу великих језичких модела,
2. избор и прилагођавање одабраних модела на специфичном скупу програмерских података применом LoRA техника финог подешавања,
3. имплементацију *Visual Studio Code* екстензије под називом *assisthtec,* која комуницира са обученим моделом и нуди предлоге у реалном времену,
4. евалуацију резултата на основу метрика као што су CodeBLEU, сличност синтаксног стабла и друге релевантне мере квалитета генерисаног кода

Рад је организован у шест поглавља. У првом поглављу, дат је преглед литературе и теоријских основа које укључују архитектуру трансформера, механизам пажње, као и концепте као што су LoRA и квантизација. Друго поглавље дефинише проблем и идејно решење, као и мотиве за развој сопственог асистента и екстензије. Треће поглавље бави се описом скупа података и изабраним моделима. Четврто поглавље представља методологију која обухвата процес припреме за фино подешавање, сам поступак додатне обуке, и фазу закључивања. Пето поглавље приказује опис метрика за евалуацију и резултате евалуације обучених модела. Шесто поглавље говори о имплементацији екстензије и интеграцији са моделом., након чега следи закључак са прегледом постигнутих резултата и могућим правцима даљег истраживања.

1. Преглед литературе

1.1. Обрада природног језика и велики језички модели

Природни језик представља основу комуникације међу људима. Од свакодневних разговора до сложених научних чланака, језик служи као средство за преношење идеја, размену информација и изградњу међусобног разумевања. Међутим, поставља се питање како обезбедити рачунарима да разумеју и користе природни језик на начин сличан људима. Ово питање је срж области обраде природног језика (*Natural Language Processing - NLP*).

NLP обухвата низ техника и алгоритама који омогућавају рачунарима да анализирају, интерпретирају и генеришу текстуални садржај. Кроз NLP, рачунари могу да обављају задатке попут превођења језика, генерисања одговора на питања, класификовања текста, препознавање емоција, па чак и предвиђања следеће речи у реченици. Ови задаци су значајни за развој напредних алата и апликација, као што су виртуелни асистенти, аутоматски преводиоци, системи за препоруку и интелигентни претраживачи.

Развој у области дубоког учења (*Deep Learning*), значајно је унапредио могућности NLP-а, тако што је довео до појаве моћних модела познатих као велики језички модели (*Large Language Models* - LLM). Модели попут GPT-3, BERT и StarCoder променили су начин на који рачунари приступају језику. Захваљујући великим количинама података, ресурсима на којима су тренирани и напредним архитектурама као што је трансформер, ови модели могу да разумеју контекст, сложене језичке структуре и да генеришу текст који стилски и семантички подсећа на људски.

У области софтверског инжењерства, LLM-ови су нашли посебну примену у креирању алата за асистенцију у програмирању. Пример за то је GitHub Copilot, систем који користи LLM да предлаже линије кода или целе функције у складу са контекстом тренутно отвореног документа. Овакви системи значајно повећавају продуктивност програмера и смањују број рутинских задатака.

Ипак, развој и примена оваквих модела захтева велике ресурсе - како у погледу рачунарске снаге, тако и у количини података потребних за обуку. Да би се модели прилагодили специфичним задацима, често је потребно применити технике финог подешавања (fine-tuning) или параметарски ефикасне адаптације, као што је LoRA (Low-Rank Adaptation). Ове технике омогућавају прилагођавање модела ограниченим ресурсима, задржавајући при томе његове способности за генерализацију.

Разумевање унутрашњих механизама великих језичких модела, посебно архитектуре трансформера и механизма пажње, представља основу за разумевање њиховог начина рада. Управо том темом бави се наредно поглавље.

1.2. Архитектура трансформера и основе механизма пажње

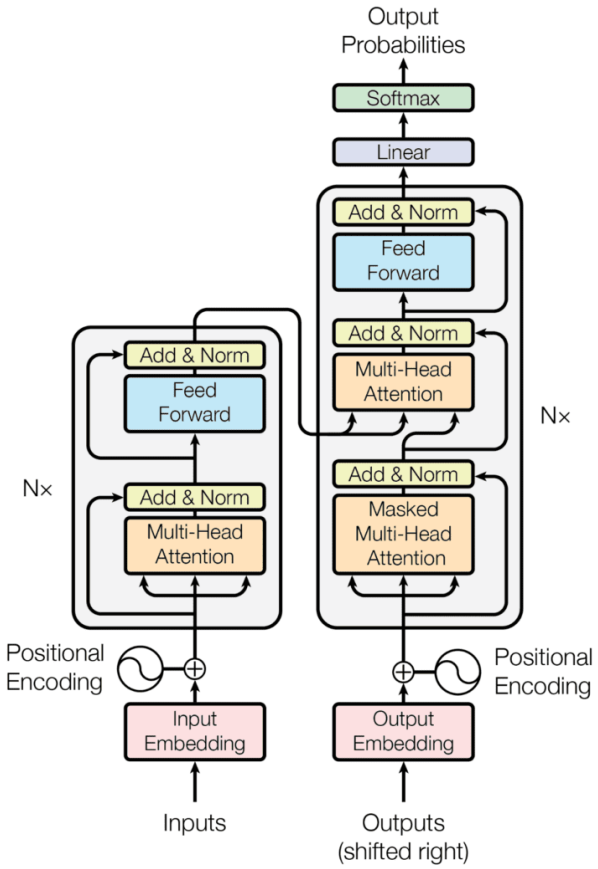
Архитектура трансформера, први пут представљена у раду „Attention is All You Need”, представља једну од најзначајнијих иновација у области обраде природног језика (NLP). За разлику од ранијих приступа као што су RNN (Recurrent Neural Networks) и LSTM (Long Short-Term Memory), трансформери у потпуности елиминишу рекурентност и уместо ње користе механизам пажње (self-attention), који омогућава да се сви токени у секвенци обрађују истовремено.

Овај приступ омогућава паралелну обраду података, што резултује знатно бржим тренирањем модела, бољом ефикасношћу и могућношћу скалирања на моделе са милијардама параметара. Основна идеја рада је да је пажња довољна за хватање односа између речи у реченици, без потребе да се токени обрађују секвенцијално. Трансформер се састоји од два главна дела:

1. Енкодер, који обрађује улазни текст и гради његову репрезентацију,
2. Декодер, који на основу те репрезентације генерише излаз (на пример, превод, одговор или наставак реченице).

Кључни елемент архитектуре је *Scaled Dot-Product Attention*, која омогућава моделу да „обрати пажњу“ на различите делове улаза при генерисању сваког излаза. Овај механизам израчунава релевантност сваког елемента реченице у односу на друге.

У раду се такође уводи концепт *Multi-Head Attention*, који омогућава моделу да истовремено учи различите типове релација међу токенима.



Слика 1 - трансформер архитектура

Увођењем трансформера, могуће је ефикасно тренирати велике језичке моделе као што су GPT, BERT и LLaMA, који се данас користе у бројним апликацијама, укључујући и ову екстензију описану у раду.

1.2.1. Енкодер

Енкодер у трансформер архитектури има задатак да прими улазну секвенцу и претвори је у континуалну репрезентацију - векторски приказ који садржи све релевантне информације из текста. Сваки енкодер се састоји од шест идентичних слојева, а сваки слој има два основна блока:

- *Multihead Self-Attention* блок, који омогућава моделу да „обрати пажњу“ на различите токене унутар улазне реченице.
- *Feed-Forward* блок, који служи за нелинеарну трансформацију података и учење комплекснијих односа између карактеристика, и представља једноставну потпуно повезано мрежу.

Излаз енкодера је низ вектора који представљају „смисао“ сваког токена у контексту целе реченице. За сваки подслој, користи се резидуална веза, праћена нормализацијом слоја. Сви под слојеви као слојеви за уметање производе излазе димензије 512.

1.2.2. Декодер

Декодер има задатак да на основу излаза енкодера генерише одговарајућу излазну секвенцу, било да је реч о преводу, одговору или наставку кода. И он се састоји од шест слојева, али уз додатни механизам пажње који омогућава да декодер у сваком кораку „гледа“ у одговарајуће делове излаза енкодера.

У сваком слоју има и трећи под-слој, који обавља *multi-head attention* над излазима енкодера. У декодеру се онемогућава праћење будућих позиција кроз маскирање, што заједно са помереним излазним уметањима, омогућава да предвиђања за позицију *i* зависе само од познатих излаза на позицијама мањим од *i*. Декодер се састоји из три основна блока:

- *Masked Self-Attention* блок, који обезбеђује да модел не користи будуће токене током тренинга (чувајући ауторегресивност).
- *Encoder-Decoder Attention* блок, који повезује улаз и излаз.
- *Feed-Forward* блок, који омогућава обраду и трансформацију вектора пажње.

1.2.3. Механизам пажње

Механизам пажње, познат као *Scaled Dot-Product Attention*, представља срце трансформер архитектуре. Он се може описати као функција која мапира упите (queries) и скуп парова кључ–вредност (key–value pairs) на излаз. Свака од наведених променљивих, представљена је векторима. Излаз се рачуна као пондерисани збир вредности, где тежина сваке вредности зависи од компатибилности између упита и одговарајућег кључа. Формално, овај механизам је дефинисан следећом једначином:

$$Attention\,(Q,\,K,\,V)\;=\;soft\max\left(\frac{Q\cdot K^{T}}{\sqrt{d_k}}\right)\cdot V$$

где су Q (queries), K (keys), и V (values) матрице добијене из улазних вектора, а d_k представља димензију кључева. Резултат ове операције је тежински збир вредности који представља нову репрезентацију токена у односу на контекст целе секвенце.

Овај принцип омогућава моделу да "обрати пажњу" на различите делове улазне реченице. На пример, приликом превођења речи "it", модел може да препозна на који претходни ентитет се та реч односи. Механизам пажње је, дакле, оно што омогућава трансформерима да на ефикасан и интуитиван начин обрађују језик, што их чини основом савремених великих језичких модела.

Поред тога, у раду се уводи концепт *Multi-Head Attention*, који омогућава моделу да истовремено учи више различитих типова односа између речи. Свака "глава" пажње фокусира се на други аспект зависности у реченици (нпр. граматички, семантички или позициони).

Повезаност са великим језичким моделима

Оваква архитектура омогућила је развој великих језичких модела као што су BERT, који представља модел базиран на енкодеру (*encoder-based*), и GPT, LLaMA, CodeLlama, StarCoder и DeepSeek, који представљају моделе базиране на декодеру (*decoder-based*). Сви они користе исте принципе механизма пажње, али се разликују у начину обраде улазног и излазног текста, при чему модели базирани на енкодеру, боље служе за разумевање текста, док модели базирани на декодеру предњаче у генерисању садржаја. Трансформер је тиме постао основа савремених AI система који обрађују природни језик и програмски код.

1.3. LoRA и квантизација модела QLoRA

Развој великих језичких модела захтева значајне рачунарске ресурсе, како током тренирања, тако и при фином подешавању. Традиционални приступ фином усавршавању подразумева да се унапред обучени модел поново тренира на новом скупу података, при чему се ажурирају све тежине у моделу. Међутим, за моделе велике величине, као што је LLaMa-70B (који има 70 милијарди параметара), овај процес постаје изузетно захтеван у погледу меморије и времена рачунања.

Да би се решио овај проблем, развијене су технике као што су LoRA (Low-Rank Adaptation) и квантизација, које омогућавају еифкасније коришћење ресурса без значајног губитка перформанси.

1.3.1. LoRA смањење меморијских захтева

LoRA представља технику адаптације модела коришћењем ниског ранга матрица. Уместо да се директно ажурирају све тежине модела, LoRA користи факторизацију матрица и матрично множење, чиме се смањује број параметара који се мењају током финог подешавања модела. На овај начин се избегава такозвано "катастрофално заборављање" (catastrophic forgetting), јер оригиналне тежине модела остају непромењене, а прилагођавање се врши кроз додатне матрице ниског ранга.

LoRA је нарочито корисна за велике моделе, чије су матрице тежина обично високог ранга. Коришћењем LoRA методе, тежине се не чувају у једној матрици пуног ранга, већ се декомпонују у више мањих матрица нижег ранга, чиме се постиже значајна уштеда меморије.

Иако LoRA омогућава значајно смањење меморијских захтева, постоје одређени компромиси у погледу квалитета моделирања. Матрице високог ранга садрже више независних информација у поређењу са матрицама нижег ранга, што значи да увођење LoRA адаптације може довести до одређеног губитка информација и благог пада у перформансама модела.

1.3.2. Квантизација као смањење презицности

Квантизација представља процес редукције прецизности нумеричке репрезентације параметара модела, укључујући тежине, пристрасности (biases) и активације. Стандардне неуронске мреже користе бројеве са покретним зарезом једноструке или двоструке прецизности (32-bit или 64-bit floating point) за представљање ових параметара. Међутим, ови типови података захтевају велику меморију и значајне рачунарске ресурсе, што може бити проблематично за примену LLM-а на хардверу са ограниченим ресурсима, као што су мобилни уређаји или edge уређаји.

Смањењем прецизности представљања података на lower precision формате, као што су FP16 (16-bit floating point) или INT8 (8-bit integer), могуће је драстично смањити величину модела и убрзати његову извршну ефикасност. Ова техника је нарочито корисна приликом инференције модела, јер омогућава брже рачунање уз мању потрошњу меморије.

Међутим, квантизација такође уводи одређене компромисе. Како је смањење прецизности у суштини процес апроксимације, могући су губици информација, што може довести до благог пада у тачности предикција модела.

1.3.3. QLoRA – комбинација квантизације и LoRA приступа

QLoRA (Quantized LoRA) представља оптимизовану верзију LoRA технике, која комбинује адаптацију ниског ранга са квантизацијом. Уместо да се LoRA примењује на модел са пуном прецизношћу, QLoRA прво квантизује модел, смањујући његове меморијске захтеве, а затим примењује LoRA адаптацију на овај компактнији модел. Примена QLoRA методе омогућава додатну уштеду меморије у односу на класичан LoRA, јер модел већ користи тип података ниже прецизности. Ово чини QLoRA-у посебно погодном за тренинг и фино подешавање великих модела на рачунарима са ограниченим GPU меморијским капацитетом.

Иако QLoRA омогућава значајно смањење рачунарских захтева, цена оваквог приступа је потенцијални губитак информација услед примене квантизације. Ипак, у пракси је показано да је овај губитак минималан, а добици у меморијској ефикасности често надмашују потенцијалне недостатке.

1.3.4. Закључак

LoRA, квантизација и њихова комбинација у виду QLoRA представљају кључне технике за оптимизацију великих језичких модела. Док LoRA омогућава ефикасно фино подешавање модела уз задржавање оригиналних тежина, квантизација смањује меморијске захтеве и убрзава закључивање. QLoRA комбинује предности обе методе, пружајући компромис између уштеде ресурса и задржавања квалитета моделирања. Примена ових метода омогућава коришћење LLM-а у окружењима са ограниченим ресурсима, као што су *edge* уређаји и мање GPU конфигурације, чиме се отвара могућност шире примене напредних језичких модела у индустрији и истраживању.

2. Дефинисање проблема и идеје решења

Савремени развој софтвера у великој мери зависи од алата који програмерима омогућавају брже и ефикасније писање кода. Са појавом алата заснованих на вештачкој интелигенцији као што су GitHub Copilot, ChatGPT и Amazon CodeWhisperer, процес програмирања је постао значајно унапређен. Ипак, већина ових система ослања се на *cloud* инфраструктуру, што подразумева да се улазни подаци корисника шаљу на спољне сервере ради обраде. Овакав приступ отвара питања приватности, безбедности и зависности од интернет конекције, као и повећане латенције током рада.

Основни проблем који се у овом раду разматра односи се на потребу за развојем локалног AI асистента за програмирање који би омогућио коришћење великог језичког модела без ослањања на спољне сервере и готова решења. Циљ је омогућити потпуну контролу над моделом, задржавајући све предности интелигентног генерисања кода и омогућити његово прилагођавање специфичном стилу писања кода.

2.1. Развој свог асистента, у виду екстензије за VS Code

Као решење наведеног проблема, развијена је екстензија за Visual Studio Code која функционише као интелигентни асистент за програмирање. Екстензија комуницира са локалним Python API-јем који користи један од фино подешених великих језичких модела, доступних на Hugging Face платформи. Након извршене компаративне анализе више модела, одабран је модел који је показао најбоље перформансе за задатак предвиђања програмског кода. Детаљнији опис одабраног модела, као и резултати анализе, биће представљени у наредним секцијама рада. Приликом активирања екстензије, последње линије кода из отвореног фајла шаљу се API-ју, који затим предвиђа наредни део кода и враћа резултат назад у VS Code окружење. На овај начин, модел се користи као аутоматски асистент за довршавање кода, слично као GitHub Copilot, али без cloud инфраструктуре.

Циљеви решења су следећи: обезбедити функционалност предвиђања и генерисања кода на основу контекста, смањити зависност од спољних сервиса, побољшати приватност података, убрзати процес обраде и показати могућности примене великог језичког модела у реалном развојном окружењу.

Очекује се да ће овај приступ омогућити већу продуктивност програмера, брже генерисање кода у контролисаним окружењима, без ослањања на јавне интернет сервисе, што доприноси већој безбедности рада са осетљивим подацима. Развијени систем такође пружа могућност прилагођавања и даљег развоја, што га чини погодним за употребу у различитим софтверским окружењима и едукативним контекстима.

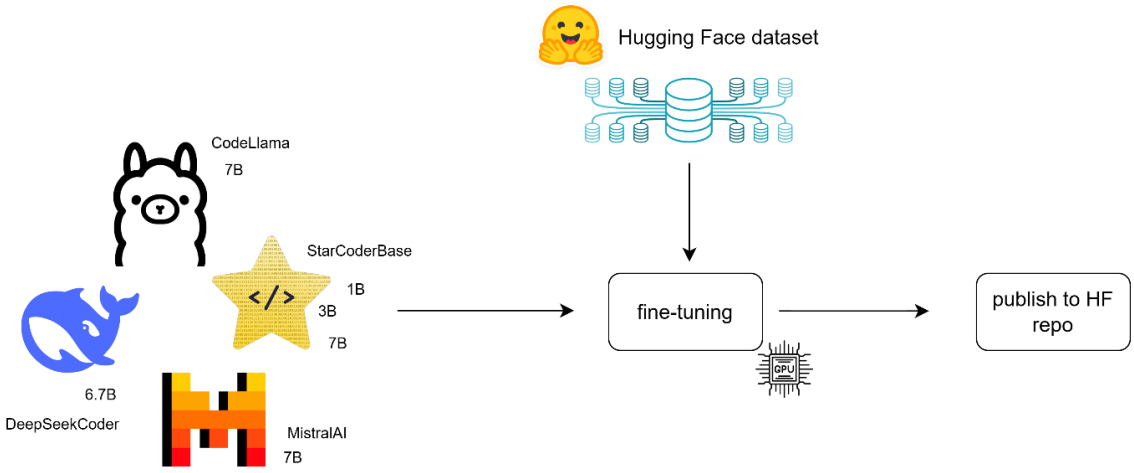
Развој оваквог система подразумева комбинацију више технологија - од архитектуре великог језичког модела и техника финог подешавања, до практичне интеграције са VS Code окружењем. У наредним поглављима биће описана методологија, архитектура екстензије и начин комуникације са моделом.

2.2. Ток решења проблема

На дијаграму испод, приказан је ток решења који обухвата цео процес обуке, закључивања и евалуације модела.

У почетној фази, изабрани LLM модели: StarCoder (1B, 3B, 7B), CodeLlama 7B, DeepseekCoder 6.7B и MistralAI 7B, представљају основу система. Сви модели користе заједнички Hugging Face датасет који садржи прилагођене узорке програмског кода.

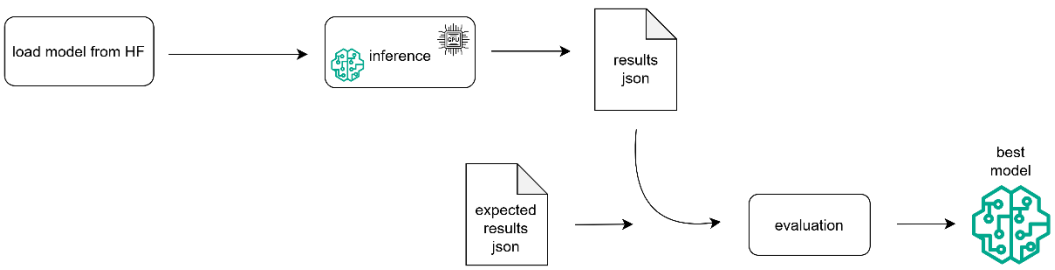
Након тога, врши се фино подешавање (fine-tuning) над базним моделима, како би се сваки модел адаптирао на специфичне обрасце кода из скупа података. Адаптери фино подешених модела се затим чувају на Hugging Face репозиторијуму, чиме постају доступни за даљу употребу и тестирање.



Дијаграм 1 - фино подешавање одабраних модела

У следећој фази, модели се учитавају са *Hugging Face*-а и користе се за закључивање, где на основу улазног текста (*prompt*-а) сваки модел генерише излазни код. Сви генерисани резултати се чувају у једно фајлу, у овом случају results.json, док очекивани излази (референце) стоје у expected_results.json.

На крају се спроводи евалуација, у којој се пореде добијени и очекивани резултати применом метрика као што су CodeBLEU, n-gram, weighted n-gram и syntax match. На основу тога се идентификује најбољи модел, односно онај који остварује највећу тачност и највећу структурну сличност са референтним кодом.



Дијаграм 2- закључивање и евалуација модела

Тај модел, даље се користи у екстензији која директно комуницира са корисником и прави предикције наредног дела кода.

3. Скуп података и одбрани модели

Ово поглавље описује извор података коришћен за фино подешавање великог језичког модела, као и процес избора модела за фино подешавање и интеграцију у развијену екстензију за Visual Studio Code. У овој секцији, приказан је поступак избора и анализе скупа података *smangrul/hf-stack-v1* са платформе Hugging Face, као и критеријуми и разлози који су довели до избора модела.

3.1. Опис скупа података

3.1.1. Порекло и структура података

За потребе финог подешавања модела коришћен је скуп података *smangrul/hf-stack-v1*, који представља филтрирану и структурисану верзију оригиналног скупа података *The Stack*, развијеног у оквиру BigCode иницијативе.

Скуп података садржи изворни код из великог броја јавних GitHub репозиторијума, прикупљених у складу са лиценцама отвореног кода. Посебна пажња посвећена је уклањању осетљивих и приватних информација као што су лозинке, API кључеви и приватни подаци. Садржи код у више програмских језика, при чему су најзаступљенији Python, JavaScript, C++, Java, и TypeScript. Највећи део скупа чине Python датотеке, што га чини посебно погодним за фино подешавање модела намењених генерисању и довршавању Python кода, као у случају развијене екстензије.

Сваки узорак у скупу података, садржи следеће компоненте:

- *content* - сам изворни код,
- *file_path* - путања до фајла унутар репозиторијума

Табела 1 - Пример узорка из скупа података

repo_id	file_path	content
“hf_public_repos”	“hf_public_repos/accelerate/setup.py”	“from setuptools import setup from setuptools import find_packages …”

3.1.2. Критеријуми за избор скупа података

Приликом избора скупа података разматрани су следећи аспекти:

1. **Релевантност**- висок удео кода из стварних софтверских пројеката, што обезбеђује природне шаблоне кода и коментара.
2. **Разноврсност**- присуство више програмских језика омогућава моделирање општих образаца програмирања.

3. **Квалитет и чистоћа**- подаци су прошли процес филтрирања и чишћења (уклањање бинарних фајлова, приватних података и дупликата).
4. **Лиценце**- задржани су само фајлови са компатибилним open-source лиценцама, у складу са *BigCode* политиком о транспарентности и етичком употребом кода.
5. **Практична величина**- скуп података омогућава рад на локалним GPU ресурсима, што је било пресудно за извођење финог подешавања у реалним условима.

3.1.3. Статистичка анализа

Након почетног прегледа и филтрирања, спроведена је основна статистичка анализа скупа *smangrul/hf-stack-v1,* ради бољег разумевања структуре и карактеристика података који ће се користити у процесу финог подешавања различитих модела.

Анализа је обухватила неколико кључних аспеката: величину и расподелу узорака, дужину кодних датотека, број линија и токена, присуство потенцијално осетљивих података и процену синтаксне исправности кода.

Величина и структура узорака

Скуп података *smangrul/hf-stack-v1* садржи укупно 5.905 узорака, при чему сваки запис представља један изворни фајл програмског кода.

Просечна дужина фајлова износи 15.348 карактера, док је медијана 6.893, што указује на то да већина узорака представља средње велике сегменте кода, док мањи број фајлова садржи изузетно обиман код. Деведесети перцентил износи 37.595 карактера, што значи да само 10% узорака прелази ову дужину.

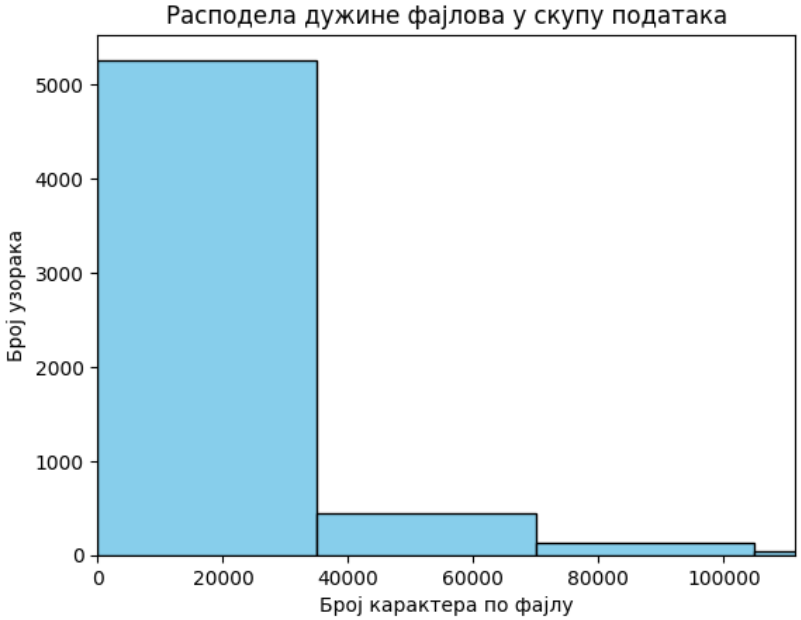


График 1 - расподела дужине фајлова

Анализа броја линија кода показује да просечан фајл садржи око 378 линија, док је медијана 169 линија, што је у складу са структуром мањих програмских модула или функција. Деведесети перцентил (842 линије) показује постојање мањег броја фајлова

са значајно већим обимом, који вероватно представљају комплексније класе или целе библиотеке.

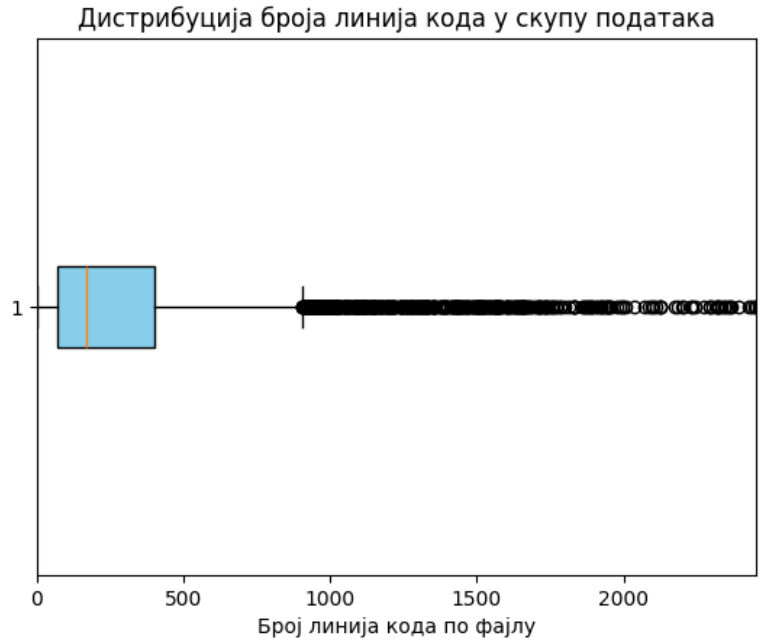


График 2 - расподела броја линија кода

Дијаграм зависности броја линија и укупне дужине фајла показује јасну линеарну везу, што је очекивано с обзиром да већи број линија подразумева већи број карактера.

Међутим, приметно је да постоји неколико издвојених примера са великом дужином, што су највероватније обимнији модули.

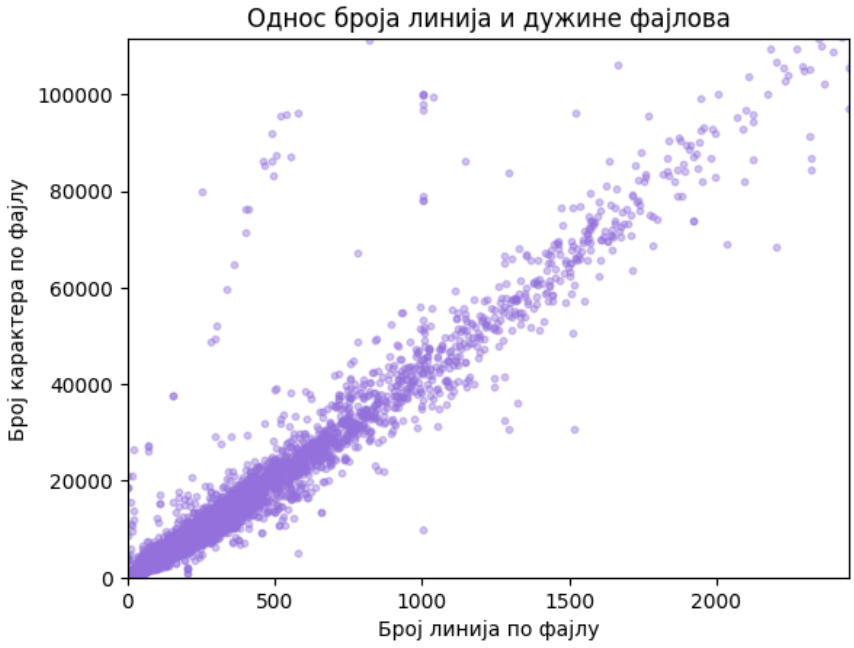


График 3 - Однос броја линија и дужине фајлова

Ови резултати показују да је скуп довољно разноврстан да покрије и једноставне и напредне структуре кода, што је посебно корисно за фино подешавање модела који треба да разуме и краће кодове који су једноставнији и дужи контекст у кодовима као што су функције за библиотеке.

Анализа броја токена по узорку

Да би се проценила комплексност и дужина кодова из изабраног скупа података, спроведена је анализа дистрибуције дужина у токенима, користећи токенизатор модела StarCoderBase-1B из *Transformers* библиотеке. Анализа је обављена на узорку од 500 фајлова, без скраћивања садржаја *(truncation=False*), како би се добио реалан увид у природну дужину примера.

График 4 - Расподела броја токена по узорку

Резултати показују да медијана броја токена износи 1.929, што значи да половина свих узорака садржи мање од две хиљаде токена. Деведесети перцентил (10.112 токена) показује да већина докумената има умерену дужину, док само мали број узорака садржи више од 30.000 токена, што представља горњу границу (99. перцентил). Ова дистрибуција указује на асиметрију, тј. скуп доминира краћим и средње дугачким кодним исечцима, док мали број дугачких фајлова представља „реп“ расподеле.

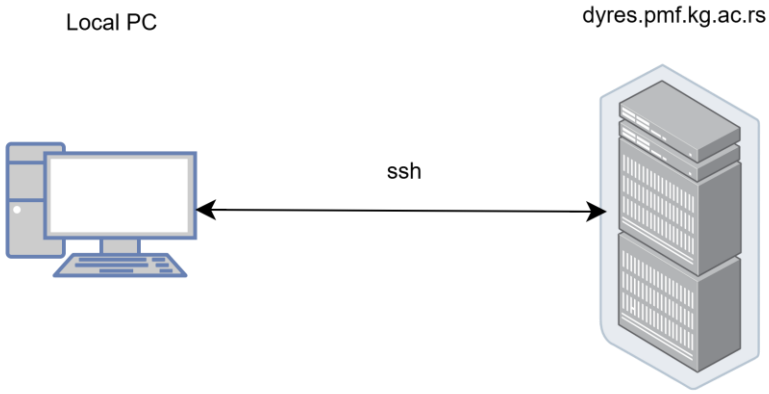
Овај резултат има практичну вредност приликом финог подешавања модела, јер омогућава избор оптималног контекстног прозора (*context window*). Будући да се већина узорака налази испод границе од 10.000 токена, модел са контекстом од 8–16 хиљада токена може ефикасно да обухвати највећи део скупа без губитка релевантног контекста. Осим тога, оваква дужина је погодна за примену *Fill-in-the-Middle* (FIM) приступа, где се токенизација дели на префикс, уметнути и суфикс део, јер довољна количина садржаја омогућава учење зависности између удаљенијих делова кода.

3.2. Избор модела – опис, карактеристике и могућности

Током рада разматрани су отворени LLM модели усмерени на програмски код: CodeLlama, Mistral, LLaMA, StarCoder и DeepSeekCoder. Сваки има различити однос између перформанси, величине, контекстуалног опсега и брзине закључивања. С обзиром на циљ рада, развој локалног асистента за VS Code, додатно је стављен акценат на брзину извршавања, ниску потрошњу ресурса и једноставну интеграцију преко Python API-ја.

3.2.1. Могућности и ограничења GPU

Тренинг је вршен на серверу *dyres.pmf.kg.ac.rs* који располаже са два NVIDIA GPU-а (по 15 GB VRAM-а).



Дијаграм 3 - веза се сервером

Командом *nvidia-smi* добијамо више информација о доступним GPU ресурсима на серверу, као и њиховим тренутним заузећем.

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.100                Driver Version: 550.100        CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:01:00.0 Off |                    0 |
| N/A   56C    P0            29W /   70W |   14295MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  Tesla T4                       Off |   00000000:02:00.0 Off |                    0 |
| N/A   43C    P8            15W /   70W |       3MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A    209924      C   ...jolovic/assisthtec/venv/bin/python3      14292MiB |
+-----------------------------------------------------------------------------------------+
```

Слика 2 - доступни ресурси

Најчешће је коришћен само један GPU по процесу - један за тренирање једног модела, а други за тренирање другог модела или закључивање, како би се избегло дељење меморије и омогућило паралелно више процеса.

Могућности:

1. Подршка за FP16 и 4-битну квантовање (QLoRA) - омогућава тренинг великих модела (са по 7B параметара) у оквиру расположивих 15 GB.
2. *Gradient checkpointing* – значајно смањује VRAM трошак (≈ -30 %), уз минимално успоравање.
3. *PagedAdamW* (*bitsandbytes*) - оптимизатор који чува меморију приликом ажурирања тежина.
4. Мулти-GPU окружење - омогућава паралелно извођење више експеримената.

Ограничења:

1. Веће секвенце доводе до OOM (*Out-Of-Memory*) грешака.
2. Трајање тренинга: фино подешавање 7 B модела у зависности од броја корака, ако узмемо 500 корака, траје приближно 5 h на једном GPU-у.

3.2.2. Карактеристике модела и захтеви за меморију

У оквиру овог истраживања разматрано је више отворених великих језичких модела који су прилагођени задацима разумевања и генерисања програмског кода. Сви модели су засновани на *Transformer* архитектури, описаној у поглављу 1, и примењују механизам *self-attention* за моделирање односа између токена у улазној секвенци.

Пошто је циљ развоја система био креирање локалног *AI* асистента који ради без зависности од *cloud* инфраструктуре, посебна пажња је посвећена најпознатијим моделима мање величине (до неколико милијарди параметара) који се могу извршавати у ограниченим меморијским условима.

Разматрани модели обухватају варијанте из породица CodeLlama, Mistral, LLaMA, StarCoder и DeepSeekCoder. Сви су фино подешени над скупом *smangrul/hf-stack-v1,* описаног у делу које се бави анализом скупа података, али се међусобно разликују по броју параметара, потребној количини VRAM-а, величини контекстног прозора и времену инференције. Што се тиче структурних карактеристика, модели DeepSeekCoder и StarCoder су у старту тренирани на великом броју програмских језика, што их чини погодним за мултијезичке сценарије у VS Code окружењу.

Приликом експеримента примењени су LoRA и QLoRA приступи ради смањења потребне меморије током финог подешавања. Фино подешавање модела са 1B параметара захтева просечно 10–14 GB VRAM-а, зависно од дужине контекста и batch величине. Модели величине 3B–7B параметара захтевају знатно више, у просеку од 20 до 32 GB VRAM-а, што је чинило неке варијанте мање погодним за локално извршавање.

Табела 2 - Поређење модела и процењене меморије током тренирања

Модел	**Параметара**	**Прецизност**	**Процењен VRAM током тренинга**	**Напомене**
StarCoder - 7B	$7 \cdot 10^9$	QLoRA 4bit / FP16 compute	~13GB	Подржава FIM, коришћен fim_rate=0.5
Mistral-7B	$7 \cdot 10^9$	QLoRA 4bit / FP16 compute	~12GB	Без FIM подршке
CodeLlama-7B	$7 \cdot 10^9$	QLoRA 4bit	~12GB	FIM токени <PRE>, <MID>, <SUF>
Llama-2-7B	$7 \cdot 10^9$	FP16	>28GB	Није могао бити трениран без квантизације
DeepSeekCoder-6.7B	$6.7 \cdot 10^9$	QLoRA 4bit / FP16 compute	~14 GB	Подржава FIM токене (<PRE>, <MID>, <SUF>)

Коришћењем квантизације (INT8 или INT4) током закључивања, меморијски захтеви су смањени за 40–70%, без значајног губитка у перформансама. На овај начин је постигнут баланс између тачности модела, времена одзива, и ресурсне ефикасности.

Табела 3 - Утицај квантизације на потребан меморијски простор

Број параметара	Оригинална величина fp16 на диску	LoRA (fp16) - потребан VRAM	QLoRA (4-bit) - потребан VRAM
1B	2GB	4GB	2.5GB
3B	6GB	9GB	5GB
5B	10GB	13GB	8GB
7B	14GB	20GB	12GB
13B	26GB	40GB	22GB

3.2.3. StarCoder Base 1B

bigcode/StarCoderbase-1b

StarCoder-1B је најмања варијанта модела из породице StarCoder, развијене у оквиру заједничког истраживачког пројекта *BigCode* који су покренули *Hugging Face* и *ServiceNow Research* 2023. године. Циљ пројекта био је креирање отворених великих језичких модела намењених за разумевање, анализу и генерисање програмског кода уз поштовање лиценци и етичких смерница.

Са становишта архитектуре, StarCoder-1B се заснива на ауторегресивном Transformer моделу са causal self-attention механизмом. Сваки улазни токен условљен је претходним токенима у секвенци, што омогућава моделу да предвиђа наредни елемент кода током процеса инференције. Модел користи multi-head attention са ротирајућим позиционим вектором представљања (*Rotary Positional Embedding* – RoPE), што побољшава стабилност током учења и омогућава боље представљање дугорочних зависности између токена.

Модел је иницијално обучен на скупу података *The Stack*, који садржи више од 3 трилиона токена из преко 80 програмских језика. Податак је прикупљен из јавних GitHub репозиторијума са компатибилним лиценцама, а осетљиве информације као што су лозинке и API кључеви су филтриране током припреме. У оквиру процеса претобраде коришћен је *Byte Pair Encoding* (BPE) токенизатор специјализован за код, који ефикасно обрађује структурне елементе попут увлачења, заграда и кључних речи.

StarCoder-1B подржава и режим *Fill-in-the-Middle* (FIM), што омогућава да модел не само додаје нови код на крају, већ и да попуњава или проширује постојеће делове кода унутар функција, класа или блокова. Контекстни прозор износи 2048 токена, што је довољно за рад у реалним програмерским сценаријима.

Захваљујући релативно малом броју параметара ($\approx 1 \times 10^9$), овај модел се може ефикасно извршавати на графичким картицама средњег ранга, са VRAM-ом од око 10–12 GB у FP16 режиму, или чак мање у INT8/INT4 квантизованим верзијама. Иако има мањи капацитет у поређењу са већим варијантама, StarCoder-1B задржава одличну способност синтаксне и семантичке анализе кода и често се користи као основа за истраживања у области фино подешених LLM-ова за програмски језик *Python*.

Табела 4 - Заузетост GPU 1 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	36°C	9W/ 70W	91 / 15360	0%	-	-
1	Tesla T4	71°C	71W / 70W	4407 / 15360	81%	41908	python

3.2.4. StarCoder Base 3B

bigcode/StarCoderbase-3b

StarCoder-3B представља средњу варијанту у оквиру породице StarCoder модела, дизајнирану као компромис између перформанси, величине и ефикасности током извршавања. Модел садржи приближно три милијарде параметара, што га сврстава у категорију компактних LLM-ова који могу да се користе у локалним окружењима без значајног оптерећења меморије.

Када је реч о архитектури модела, StarCoder-3B је заснован на ауторегресивном Transformer моделу са causal self-attention механизмом, где сваки токен предвиђа следећи у секвенци на основу контекста који му претходи. Састоји се од више слојева multi-head attention механизама, у комбинацији са нормализацијом и feed-forward неуронским блоковима који омогућавају моделирање комплексних релација у коду.

Модел користи *Rotary Positional Embeddings* (*RoPE*) као замену за класичне синусоидне позиционе енкодере, што омогућава стабилније представљање токена при већим дужинама контекста. Контекстни прозор износи 2048 токена, што омогућава моделу да обухвати више логичких целина унутар истог сегмента кода.

StarCoder-3B подржава и *Fill-in-the-Middle* (FIM) режим, који омогућава уметање кода у средини постојеће структуре, а не само на крају датотеке. Ово га чини посебно погодним за задатке као што су аутоматско довршавање кода, генерисање функција и сугестије унутар едитора.

Као и остале варијанте у овој породици, модел користи Byte Pair Encoding (BPE) токенизацију специјализовану за програмски код, при чему се посебна пажња посвећује очувању синтаксичких елемената као што су увлачења, заграде и оператори.

Захваљујући својој умереној величини и доброј ефикасности, StarCoder-3B се често користи у применама које захтевају разумевање ширег контекста без потребе за екстремно великим меморијским ресурсима. Просечни VRAM захтеви за закључивање у FP16 режиму износе око 16 GB, док квантизоване верзије могу радити и на графичким картицама са 8–12 GB меморије.

Табела 5 - заузетост GPU 0 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	67°C	67W/ 70W	8301 / 15360	86%	104283	python
1	Tesla T4	46°C	15W / 70W	67 / 15360	0%	-	-

3.2.5. StarCoder Base 7B

bigcode/StarCoderbase-7b

StarCoder-7B је највећа и најмоћнија варијанта у оквиру StarCoder породице модела, са укупно око седам милијарди параметара. Модел је по својој архитектури заснован на истој *Transformer* структури као и мање варијанте (1B и 3B), али са већим бројем слојева, ширим attention главама и повећаном унутрашњом димензијом вектора представљања.

Захваљујући већем капацитету, модел поседује знатно бољу могућност апстрактног разумевања и обраде дугорочних зависности у коду, као и способност да генерише функционалније и контекстуално доследније предлоге.

Модел подржава Fill-in-the-Middle (FIM) режим као и остале варијанте, али остварује боље резултате у задацима попут довршавања дужих функција, уметања кода унутар постојећих структура и аутоматског исправљања сложенијих грешака. Захваљујући већем броју attention глава и дубљој архитектури, StarCoder-7B има виши степен експресивности и боље разликовање програмских образаца, али и веће хардверске захтеве. Током закључивања у FP16 режиму потребно је приближно 20–24 GB VRAM-а, док квантизоване верзије у INT8 или INT4 формату омогућавају покретање и на системима са 12–16 GB меморије.

Често се користи као референтна тачка за евалуацију мањих StarCoder варијанти модела, јер представља пунију верзију архитектуре са којом се могу поредити компромиси у тачности и брзини. Што се тиче стабилности и прецизности у предвиђању, сматра се једним од најпоузданијих *open-source* модела за анализу и генерацију програмског кода.

Табела 6 - Заузетост GPU 0 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	59°C	68W/ 70W	14009 / 15360	100%	10519	python
1	Tesla T4	45°C	15W / 70W	67 / 15360	0%	-	-

3.2.6. Mistral AI 7B

mistralai/Mistral-7B-v0.1

Mistral-7B је модел који је развила компанија Mistral AI 2023. године и представља једну од најефикаснијих имплементација Transformer архитектуре у оквиру модела са мање од 10 милијарди параметара.

За разлику од класичних LLM решења, Mistral је осмишљен тако да постигне оптималан однос између перформанси, брзине и меморијске ефикасности, захваљујући увођењу неколико иновативних техника унутар attention механизма.

Кључна новина у овом моделу је Grouped-Query Attention (GQA), механизам који омогућава груписање више query токена који деле исти key/value простор. Ово смањује потребу за дуплирањем израчунавања током фазе закључивања и значајно убрзава рад модела без губитка квалитета резултата. Због овог приступа, Mistral-7B може да обрађује дужи контекст са мањим хардверским оптерећењем у поређењу са класичним *attention* моделима.

Модел користи *Sliding Window Attention* (SWA), који омогућава постепену обраду токена у сегментима, што додатно убрзава закључивање и омогућава обраду секвенци дужих од стандардног контекстног прозора. Контекстни прозор у основној конфигурацији износи 8192 токена, што је четири пута више него код већине StarCoder варијанти, што модел чини погодним за анализу и разумевање већих програмских фајлова или комплексних функција.

Са становишта архитектуре, Mistral-7B користи *Rotary Positional Embeddings* (RoPE) и LayerNorm пре attention блокова, што побољшава стабилност током учења и закључивања. Због своје компактне структуре и оптимизације, модел постиже перформансе упоредиве са већим моделима као што су LLaMA 13B, али са мањом потрошњом меморије и бржим временом одзива.

Током закључивања, Mistral-7B захтева око 14–16 GB VRAM-а у FP16 режиму, док се у INT8 или INT4 квантизованим варијантама може ефикасно извршавати и на графичким картицама са 8–12 GB меморије.

Модел је дизајниран тако да подржава примене у реалном времену, укључујући системе за препоруку, генерацију текста, као и интелигентне асистенте за писање кода.

Табела 7 - Заузетост GPU 1 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	67°C	65W/ 70W	9345 / 15360	100%	49672	python
1	Tesla T4	49°C	27W / 70W	7021 / 15360	5%	49988	python

3.2.7. DeepSeek Coder 6.7B

deepseek-ai/deepseek-coder-6.7b-base

DeepSeekCoder-6.7B је модел који припада новијој генерацији великих језичких модела оптимизованих за обраду програмског кода. Развијен је у оквиру истраживачке иницијативе DeepSeek AI, са циљем да се постигне што бољи баланс између перформанси и рачунске ефикасности при раду са сложеним кодним структурама.

Садржи око 6.7 милијарди параметара и заснован је на класичној Transformer архитектури, али са неколико кључних унапређења у attention механизму и начину представљања позиционих односа токена. Као и већина модерних LLM-ова, користи Rotary Positional Embeddings (RoPE), који омогућавају боље моделирање дугорочних зависности у коду.

Једна од карактеристика која издваја DeepSeekCoder је подршка за Fill-in-the-Middle (FIM) токене, означене као <PRE>, <MID> и <SUF>, што омогућава моделу да попуњава празнине унутар постојећег кода, а не само да наставља генерацију на крају секвенце. Ово га чини посебно погодним за задатке као што су допуна кода, рефакторисање и аутоматско исправљање грешака.

DeepSeekCoder је оптимизован тако да подржава дугачке контекстне прозоре (до 4096 токена) и стабилну фазу закључивања, чак и при већим дужинама улаза. У имплементацији attention механизма користи Query-Key Normalization (QKN) технику, која побољшава стабилност при раду са квантизованим тежинама, што је од посебног значаја за примене на хардверима ограничених ресурса. Због тога, показује високу отпорност на грешке у контексту, као и боље разумевање логичких структура кода.

Просечни захтеви за меморијом током закључивања у FP16 режиму износе око 14–16 GB VRAM-а, док се у INT8 или INT4 варијантама може покретати и на системима са мањим капацитетом меморије, уз минималан губитак прецизности.

Пример је једног од модерних LLM решења која комбинују архитектонску сложеност са практичном применљивошћу, а погодан је специфично за интеграцију у интелигентне асистенте и IDE проширења намењена анализи и писању програмског кода.

Табела 8 - Заузетост GPU 0 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	67°C	72W/ 70W	9084 / 15360	100%	49672	python
1	Tesla T4	46°C	27W / 70W	67 / 15360	0%	-	-

3.2.8. CodeLlama 7B

codellama/CodeLlama-7b-hf

CodeLlama-7B је модел који је развио тим Meta AI као проширење постојеће архитектуре LLaMA 2, са идејом да се направи систем који може да разуме и генерише програмски код. Модел има око седам милијарди параметара и припада групи мањих, ефикаснијих LLM система који се могу користити и ван великих рачунарских окружења.

Архитектура модела заснива се на Transformer концепту са causal self-attention механизмом. У односу на класични LLaMA, унапређен је тако да боље препознаје структуру кода и логичке везе између различитих делова програма. Модел користи Rotary Positional Embeddings (RoPE) како би могао да прати редослед и позицију токена у дужим секвенцама.

Контекстни прозор обухвата до 4096 токена, што омогућава да се истовремено обрађује више функција или дужих делова кода. CodeLlama подржава и Fill-in-the-Middle (FIM) начин рада, што значи да може да убацује делове кода у постојеће структуре, а не само да наставља писани садржај на крају.

Поред тога, модел је прилагођен да ради са више програмских језика и да очува исправну синтаксу приликом предвиђања. У пракси, често се користи за задатке као што су довршавање кода, објашњавање функција или предлагање алтернативних решења.

Током извршавања, модел захтева око 14 до 16 GB VRAM-а у FP16 режиму, док се у квантизованој верзији може користити и на системима са мање меморије. Поуздан је за примене у развојним окружењима и алатима који подржавају аутоматске предлоге кода.

Табела 9 - Заузетост GPU 1 током тренирања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	37°C	9W/ 70W	69 / 15360	0%	-	-
1	Tesla T4	72°C	70W / 70W	7369 / 15360	100%	60935	python

4. Методологија финог подешавања и анализе модела

У овом поглављу описана је методологија примене великог језичког модела за задатак генерисања програмског кода и његовог финог подешавања у циљу побољшања перформанси. Приказани су кораци припреме података, конфигурација окружења, процес обуке модела применом LoRA и QLoRA техника, као и поступак закључивања и евалуације резултата.

Посебна пажња посвећена је параметрима који утичу на стабилност и ефикасност тренинга, као и поређењу различитих базних модела (StarCoder, CodeLlama, MistralAI и DeepSeek). Циљ овог поглавља је да се прикаже комплетан ток процеса, од припреме података до анализе резултата, уз објашњење избора метода и хиперпараметара који су довели до оптималног решења.

4.1. Припрема окружења и података за фино подешавање

4.1.1. Учитавање скупа података

За фино подешавање модела, коришћен је јавно доступан скуп података који је описан у поглављу 3. Скуп података се учитава преко библиотеке *datasets* уз *streaming* режим.

Учитавање скупа података

```
dataset = load_dataset("smangrul/hf-stack-v1", data_dir="data",
split="train", streaming=True)
```

Стримовање омогућава рад са великим скуповима података без њиховог потпуног преузимања на диск, што је олакшавајуће приликом рада на GPU машинама са ограниченим капацитетом меморије.

Главно поље у скупу података, над којим вршимо тренирање је поље са садржајем кодова. Токенизација се врши над подацима у пакету, уз скраћивање и попуњавање до фиксне дужине секвенце задате у конфигурацији. Коришћењем токенизације, обезбеђујемо да се уједначи дужина улазних секвенци и да се меморија машине троши стабилно током тренинга.

Подешавање токенизације

```
def tokenize(examples):
    return tokenizer(
        examples["content"],
        truncation=True,
        padding="max_length",
        max_length=cfg.get("sequence_length", 512),
    )
tokenized = dataset.map(tokenize, batched=True,
remove_columns=dataset.column_names)
```

4.1.2. Учитавање конфигурационих параметара

Пре почетка самог процеса тренирања, неопходно је припремити окружење и подесити све параметре који ће управљати радом модела. Овај корак омогућава да се експерименти лако понављају и да се различите поставке испробавају без измене изворног кода.

Скрипта за фино подешавање је осмишљена тако да конфигурационе податке преузима из два различита извора: из фајл окружења и из YAML конфигурационог записа.

Фајл окружења (*.env*) садржи осетљиве и променљиве податке, попут приступног токена за Hugging Face платформу и назива базног модела који се фино подешава. У конфигурационим YAML фајловима, налазе се технички параметри самог тренинга, као што су број слојева који ће бити активни у адаптацији, дужина секвенци током уноса података, брзина учења, и други хиперпараметри.

Учитавање фајла окружења врши се помоћу библиотеке dotenv, која омогућава да се вредности из .env фајла аутоматски учитају у системске променљиве. На тај начин се избегава директно уписивање поверљивих података у код.

Учитавање фајла окружења

```
load_dotenv()
HF_TOKEN = os.getenv("HUGGINGFACE_HUB_TOKEN", None)
MODEL_ID = os.getenv("MODEL_ID", None)
```

Овим приступом се обезбеђује да токен за приступ Hugging Face платформи и назив модела буду доступни програму, без потребе да се уносе ручно при сваком покретању. Поред тога, остали параметри експеримента читају се из YAML датотеке. Овај формат је одабран јер је прегледан, лак за уређивање и погодан за чување сложених подешавања. Учитавање се врши помоћу помоћне функције.

Учитавање конфигурационих фајлова

```
def load_yaml_config(path):
    with open(path, "r", encoding="utf-8") as f:
        return yaml.safe_load(f)
cfg = load_yaml_config(config_path)
```

Садржај конфигурационих фајлова, може се видети у наредном поглављу у оквиру секције која говори о специфичним параметрима.

4.2. Фино подешавање модела

4.2.1. Параметри

Вредности параметара из табеле изабране су експериментално, тако да обезбеде стабилан рад модела уз оптималну потрошњу меморије, и складу са потребом да се што већи језички модели извршавају на доступним ресурсима од 15GB VRAM-а. Дужина секвенце постављена је на 512 токена како би се задржао контекст потребан за разумевање структуре кода, док је стопа учења подешена на $1 * 10^{-4}$, што је препоручена вредност за LoRA/QLoRA фино подешавање великих језичких модела. Ограничење броја корака на 500 омогућава да се тренинг изврши у разумном временском оквиру, уз постепену конвергенцију функције губитка. Сви коришћени параметри се могу видети у Табела 10 испод.

Табела 10 - Основни параметри током тренирања модела

Назив параметра	Вредност	Значење параметра
seed	42	Почетна вредност генератора случајних бројева која омогућава репродуктивност резултата.
sequence_length	512	Максимална дужина улазне секвенце токена коју модел обрађује током тренинга.
pack_sequences	true	Омогућава паковање више кратких секвенци у један улаз ради ефикаснијег коришћења меморије.
max_steps	500	Укупни број корака током тренинга, након чега се процес зауставља.
train_batch_size	4	Број примера (batch) који се обрађује у једној итерацији по графичкој картици.
gradient_accumulation_steps	4	Број корака након којих се градијенти акумулирају пре ажурирања тежина — симулира већи batch.
learning_rate	1e-4	Почетна стопа учења која контролише брзину прилагођавања тежина модела,
warmup_steps	30	Број почетних корака током којих се стопа учења постепено повећава до задате вредности.
lr_scheduler_type	cosine	Тип распореда промене стопе учења током тренинга (у овом случају косинусна функција).
bf16	false	Подешава да ли се користи 16-битна прецизност у bfloat формату.
fp16	true	Подешава да ли се користи 16-битна прецизност у класичном формату полупрецизног броја са покретним зарезом.
eval_freq	100	Учесталост евалуације, тј. број корака после којих се проверава тачност модела.
save_freq	100	Учесталост чувања модела током тренинга.
log_freq	25	Број корака после којих се бележе (логирају) метрике тренинга.

4.2.2. FIM (Fill-in-the-middle)

Техника *Fill-in-the-Middle* (FIM) омогућава моделу да „попуни“ средишњи део кода, на основу задатог префикса и суфикса. Такав приступ је користан у системима за аутоматске сугестије кода, јер имитира начин на који програмери допуњују постојеће функције.

Тренинг је заснован тако да подржава и класичан *Causal LM* циљ (предвиђање наредног дела кода), без експлицитне примене FIM токена али и омогућава примене FIM токена, додавањем специјалних ознака <*prefix*>, <*infill*>, <*suffix*>.

Утицај *FIM* технике на процес закључивања, приказан је у **Пример 2 – попуњавање кода у средини**.

Табела 11 - Параметри за FIM

Назив параметра	Вредност		Значење параметра
	StarCoder CodeLlama Deepseek	MistralAI	
use_fim	true	false	Одређује да ли је током тренинга активиран Fill-in-the-Middle режим.
fim_rate	0.5	0.0	Проценат узорака код којих се примењује FIM форматирање током тренинга.
fim_spm_rate	0.5	0.0	Стопа примене FIM-а на нивоу појединачних токена унутар секвенце.

4.2.3. PEFT (*Parameter-Efficient Fine-Tuning*)

PEFT представља концепт параметарски ефикасног финог подешавања, при чему се већина тежина базног модела „замрзава“, а тренира се само мањи број додатних параметара (адаптера).

PEFT се овде реализује преко LoRA механизма, чиме се постиже значајно смањење потрошње меморије, брже тренирање и једноставно дељење модела у виду LoRA адаптера, без потребе за дистрибуцијом целокупних тежина.

PEFT модел

```
1  model = get_peft_model(model, lora_cfg)
```

Поред LoRA механизма, PEFT обухвата и друге приступе као што су *Prefix-Tuning*, *Prompt-Tuning* и *Adapter-Tuning*, који се разликују по начину додавања и обучавања додатних параметара у односу на базни модел.

4.2.4. LoRA

LoRA (Low-Rank Adaptation) уводи додатне матрице са ниским рангом које уче промене у параметрима базног модела током fine-tuning-а. У овом случају, прилагођавају се само одабрани модули (најчешће слојеви пажње и пројекције), који се разликују од модела који се користи и могу се видети у Табела 12, заједно са осталим типичним вредностима.

Табела 12 - коришћене вредности LoRA параметара

Назив параметра	**Вредност**		**Значење параметра**
	StarCoder, CodeLlama, DeepSeek	**MistralAI**	
lora_r	16	16	Одређује број додатних параметара који се тренирају.
lora_alpha	64	64	Скалар који контролише обим прилагођавања LoRA тежина у односу на базни модел.
lora_dropout	0.05	0.05	Вероватноћа искључивања неких неурона током тренинга ради смањења *overfitting*-а.
lora_target_modules	[q_attn, c_attn, c_proj, c_fc]	["q_proj","k_proj","v_proj","o_proj","gate_proj","up_proj","down_proj"]	Делови модела (модули) на које се примењује LoRA адаптација, углавном компоненте механизма пажње.
lora_bias	none	none	Одређује да ли се LoRA примењује и на bias параметре.
lora_task_type	CASUAL_LM	CASUAL_LM	Тип задатка - класично моделирање језика, где се предвиђа следећи токен у низу.

На крају процеса модел чува само адаптер, што је погодно за једноставно дељење и поновно учитавање адаптера током закључивања, без потребе за поновним тренирањем базног модела.

Чување модела

```
1  model.save_pretrained(output_dir)
```

4.2.5. QLoRA – квантизована LoRA

Када је у конфигурацији омогућено *use_qlora=True*, модел се учитава у 4-битном режиму користећи библиотеку *bitsandbytes* и конфигурацију BitsAndBytesConfig:

QLoRA

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=cfg.get("bnb_4bit_use_double_quant", True),
    bnb_4bit_quant_type=cfg.get("bnb_4bit_quant_type", "nf4"),
    bnb_4bit_compute_dtype=torch.float16
)
model = prepare_model_for_kbit_training(model)
```

Овај приступ значајно смањује потрошњу GPU меморије, тако да се тренинг може изводити и на појединачним графичким картицама (нпр. 12–16 GB VRAM-а). У комбинацији са *gradient_checkpointing_enable*(), *fp16=True* и оптимизатором *paged_adamw_32bit*, QLoRA омогућава стабилан и меморијски ефикасан тренинг без значајног губитка квалитета резултата.

Табела 13 - Коришћене вредности QLoRA параметара

Назив параметра	Вредност	Значење параметра
use_qlora	true	Одређује да ли ће модел бити трениран у четворобитном (4-bit) режиму.
use_nested_quant	true	Активира двоструку квантизацију ради даљег смањења меморијског оптерећења.
bnb_4bit_quant_type	nf4	Тип квантизације који се користи у bitsandbytes библиотеци; NF4 је најпрецизнији.
bnb_4bit_use_double_quant	true	Додатно смањује величину тежина применом друге фазе квантизације.
bnb_compute_dtype	torch.float16	Тип података који се користи у рачунању током тренинга (полупрецизни флоат).

Након учитавања модела, да бисмо активирали постепено чување градијената, довољно је да се позове:

Делимично чување градијената:

```
model.gradient_checkpointing_enable()
```

4.3. Закључивање

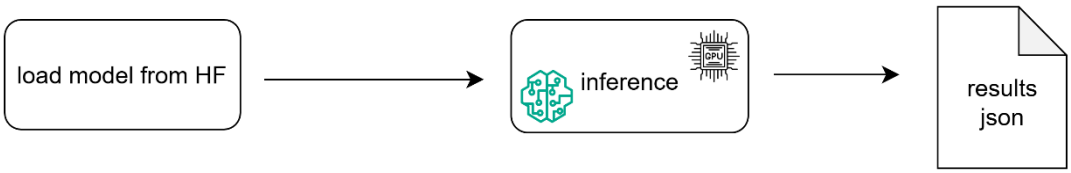
Да бисмо користили наше додатно трениране моделе, фино подешене на одабраном скупу података, морамо да покренемо закључивање. Закључивање се врши тако што учитамо тренирани модел које смо претходно сачували на HuggingFace репозиторијуму. Учитаћемо и базни модел и фино подешени, за сваки од претходно поменутих модела са којима смо радили.

Улаз у те моделе биће неки код или кориснички упит. Излаз зависи од предефинисаног броја токена у конфигурационом фајлу, и представља предикцију модела за наставак кода/упита који је представљао улаз.

Табела 14 - Параметри закључивања

Назив параметра	Вредност	Значење параметра
max_new_tokens	128	Максималан број нових токена које модел може да генерише током закључивања.
temperature	0.2	Контрола случајности у генерисању - мање вредности дају прецизније, али мање разноврсне излазе.
top_p	0.95	Параметар за nucleus sampling - модел бира токене који укупно чине 95% вероватноће.
repetation_penalty	1.0	Казна за понављање речи или токена у излазу. Вредности >1 смањују понављања.
use_fim	false/true	Одређује да ли ће закључивање користити FIM (попуњавање средине кода) или класично предвиђање наставка.

Током процеса закључивања генеришу се излазни резултати за оба модела - базни претренирани и фино подешени. Сваки модел производи сопствени излаз који се чува у засебном фајлу, што омогућава накнадну евалуацију и поређење њихових перформанси.



Дијаграм 4 - процес закључивања

4.3.1. Пример 1 – предвиђање наредног дела кода

У овом примеру приказано је класично предвиђање наставка кода, које представља основни облик закључивања у моделима типа Causal Language Model. Модел као улаз добија почетни део функције и покушава да предвиди логичан наставак који следи након префикса.

Пример улаза

```
1  def factorial(n):
2
```

Пример излаза

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

У приказаном случају модел је успешно препознао да функција „*factorial*“ представља класичан пример рекурзивне имплементације и генерисао је исправан наставак кода који рачуна факторијел броја. Овај тип закључивања симулира ситуације у којима програмер започиње функцију, а модел нуди могући наставак или довршетак кода.

4.3.2. Пример 2 – попуњавање кода у средини

Други пример илуструје *Fill-in-the-Middle* (FIM) приступ, у којем модел не предвиђа само наставак кода, већ попуњава недостајући део између познатог почетка (префикса) и краја (суфикса). У овом случају моделу је задат почетак функције и излазна наредба, док је централни део означен специјалним токеном <<<*SUFFIX*>>>. Током закључивања модел попуњава овај простор и генерише код који логички повезује улаз и излаз.

Пример улаза

```
def add(a, b):
    <<<SUFFIX>>>
print(add(2, 3))
```

Пример излаза

```
def add(a, b):
    return a + b
print(add(2, 3))
```

Након закључивања, добијени излази се упоређују са очекиваним резултатима помоћу метрика као што је CodeBLEU, која мери сличност између генерисаног и референтног кода узимајући у обзир не само текстуалну, већ и синтаксну структуру. На овај начин могуће је објективно проценити колико је додатно тренирани модел побољшао разумевање програмског језика у односу на базну верзију.

Потрошња меморије током закључивања зависи од самог модела који се користи, али је углавном била у опсегу од 10-14GB VRAM-а, за сваки од коришћених модела.

Табела 15 - Заузетост GPU 0 током закључиавања

GPU	Name	Temp	Power Usage	Memory Usage (MB)	GPU Util	Process (PID)	Process Name
0	Tesla T4	63°C	56W/ 70W	13883 / 15360	32%	77571	python
1	Tesla T4	45°C	15W / 70W	67 / 15360	0%	-	-

5. Метрике и анализа модела

Код евалуације модела природног језика користе се различите метрике које мере колико је генерисани текст (природно језик, а у нашем случају програмерски код) сличан референтним решењима. Ове метрике се углавном ослањају на поређење са златним стандардима и мере различите аспекте квалитета.

5.1. Метрике за евалуацију генерисаног кода

5.1.1. BLEU (Bilingual Evaluation Understudy)

BLEU је једна од најчешће коришћених метрика за евалуацију машинског превођења, али се такође користи и за процену генеративних модела који производе текст или код. BLEU се заснива на упаривању n-gram секвенци између генерисаног текста и референтног текста, рачунајући прецизност. Важан аспект BLEU метрике је brevity penalty, који кажњава прекратке предикције.

Формула за BLEU са n-gram прецизношћу:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \cdot \log p_n\right)$$

где је BP (brevity penalty) казна за прекратке секвенце, а p_n представља n-gram прецизност.

BLEU се често користи у задацима аутоматског довршавања кода или генерације коментара. Недостатак BLEU метрике је што не узима у обзир семантичко значење генерисаног текста, већ само статистичку сличност са референтним подацима, због чега код посматра као обичан текст и не гледа његово значење.

5.1.2. CodeBLEU

CodeBLEU представља проширење BLEU метрике које је посебно дизајнирана за евалуацију генерације програмског кода. Ова метрика комбинује BLEU прецизност, проверу лексичке сличности и садржи AST проширење. Представља побољшање у односу на BLEU, када се ради о евалуацији кода, јер узима у обзир структуру кода и семантичке аспекте. Користи се у задацима као што су аутоматска генерација функција, комплетирање кода и машинско превођење програмских језика. Састоји се од четири компоненте, које су приказане у Табели 16.

Табела 16 - метрике за CodeBLEU и њихово значење

Компонента	Шта мери	Значење
n-gram match	Текстуалну сличност токена	Колико су речи (токени) исте
weighted n-gram match	Текстуалну сличност али кључне речи теже више	Колико су важније речи (if, return,..) исте
syntax match	Структуру кода кроз AST	Да ли су блокови и изрази слични
data-flow match	Логику кроз ток променљивих	Да ли код ради исто, иако изгледа другачије

Data-flow у CodeBLEU мери колико је логична повезаност података, тј. променљивих у генерисаном коду слична оној у референтном коду. Другим речима, да ли модел користи исте променљиве, у сличном редоследу и односима, као оригинални код. На пример, уколико имамо референтни код:

Пример референтног кода:	Пример генерисаног кода:
`1 a = b + c` `2 print(a)`	`1 x = y + z` `2 print(x)`

У овом примеру, текстуално су различите променљиве, али data-flow види исту структуру: постоји променљива која настаје сабирањем две друге, и затим се она штампа. Зато ће овај део метрике имати вредност приближно 1, док би у оваквим случајевима обичан BLEU био низак.

Коначни CodeBLEU скор, рачуна се као:

$$CodeBLEU = w_1 * n_{gram} + w_2 * weighted + w_3 * syntax + w_4 * dataflow,$$

где су тежине аутоматски подешене на 0.25, 0.25, 0.25, 0.25, редом, уколико се не нагласи другчије. Ова метрика је показала већу корелацију са људском, мануелном евалуацијом него BLEU и метрике тачности.

5.1.3. XLCoST

Библиотека XLCoST има сопствену, модернизовану имплементацију CodeBLEU-а (компатибилну са вишејезичним dataset-има и *Hugging Face* форматима), па се користи лакше и без потребе за ручним увозом свих CodeXGLUE модула. Она омогућава аутоматско поређење генерисаног кода са референтним имплементацијама кроз више програмских језика (нпр. Python, Java, C++), чиме се поједностављује евалуација вишејезичних модела. XLCoST такође користи унапређене семантичке анализе ради бољег препознавања функционалних еквивалената између различитих верзија истог алгоритма.

5.1.4. ROUGE (*Recall-Oriented Understudy for Gisting Evaluation*)

ROUGE је породица метрика првобитно развијена за процену аутоматског резимирања текста. У контексту генерације програмског кода, ROUGE може да се користи за процену колико је генерисани код сличан референтном решењу у смислу садржаних реченица или речи. Најчешће варијанте у којима се јавља:

- ROUGE-N – мерење n-gram сличности (слично BLEU),
- ROUGE-L – мерење најдуже заједничке подсеквенце (Longest Common Subsequence – LCS),
- ROUGE-W – тежински ROUGE који даје већу тежину дужим подударним секвенцама.

ROUGE је користан за процену генерације коментара или документације на основу кода, док је за саму процену кода чешће пожељан CodeBLEU.

5.1.5. Treesitter

Tree-Sitter није метрика сам по себи, већ алат који омогућава парсирање и анализу апстрактног синтаксног стабла (AST) кода. Користи се у метрикама као што су CodeBLEU и XLCoST, јер омогућава прецизну идентификацију структурних сличности и разлика између генерисаног и референтног кода. На тај начин омогућава евалуацију не само на текстуалном, већ и на структуралном и логичком нивоу исправности кода. У овом раду, коришћен је у оквиру CodeBLEU метрике, за специфичне програмске језике које његов парсер стабла подржава.

5.2. Компаративна анализа модела и резултата

С обзиром на брзи развој великих језичких модела (LLM) у домену обраде програмског кода, у овом истраживању извршена је компаративна анализа четири актуелна модела у овој области: CodeLlama, MistralAI, StarCoder и DeepSeek.

Циљ анализе је да се процени њихова структурна сложеност, број параметара, дужина векторског представљања, као и специфичне техничке иновације у односу на класичну *Transformer* архитектуру представљену у раду „*Attention is All You Need*".

Иако скоро сви модели припадају истој класи по броју параметра, око 7 милијарди параметара, они се разликују у структури архитектуре, начину примене механизма пажње и оптимизацији ресурса.

Табела 17 - Анализа различитих модела

Модел	**Број параметара**	**Дужина вектора представљања**	**Број слојева**	**Макс. контекст**	**Карактеристике**
StarCoder-7B	$7 \cdot 10^9$	4096	32	8К токена	Подршка за више програмских језика, FIM техника
Mistral-7B	$7 \cdot 10^9$	4096	32	8К токена	Ротациони позициони енкодинг (RoPE), побољшана токенизација
CodeLlama-7B	$7 \cdot 10^9$	4096	32	16К токена	Груписана пажња (GQA), клизећи прозор пажње (Sliding Window Attention)
DeepSeekCoder-6.7B	$6.7 \cdot 10^9$	3584	28	8К токена	Ефикасни блокови пажње, компресовани вектори представљања

Ове разлике директно утичу на брзину тренирања, ефикасност закључивања и, најзад, на квалитет генерисаног програмског кода, што ће бити приказано у наредним поглављима кроз визуелно поређење и метрике за евалуацију модела.

Прво ћемо за пример узети када имамо једну класу и очекујемо од асистента да имплементира конструктор и методе везано за рачун у банци, дато у *Java* програмском језику.

Упит 1: java_class.java

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;

```

Наредни пример је када је дата функција коју наш модел треба да настави, дата у *C++* програмском језику.

Упит 2: cpp_function.cpp

```
// Implement a function that finds the longest increasing subsequence in a
given vector of integers.
// The function should return the length of the longest increasing
subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
 // TODO: implement
```

И на крају, имамо пример једне линије кода која представља дефиницију функције, дату у *javascript* програмском језику, али се очекује од модела да зна да разуме контекст и настави одакле смо почели.

Упит 3: js_snippet.js

```
async function getUserData(userId) {
```

5.2.1. Добијене метрике

Комплетни излази генерисаног кода за све коришћене моделе, на упитима 1-3, описаним у претходном делу, налазе се у одељку *Додатак,* на крају овог рада. Они овде нису приказани како не би одвраћали пажњу са анализе метрика и поређења перформанси модела.

Компонента *dataflow match* није коришћена, јер је у овом случају евалуација спровођена искључиво над наставцима (суфиксима) кода, без комплетне логичке структуре програма. Уместо стандардних CodeBLEU параметара (0.25, 0.25, 0.25, 0.25), примењене су тежине (0.3, 0.3, 0.4, 0.0), чиме је већи нагласак стављен на синтаксну компоненту, док је *dataflow* изостављен.

У првом задатку, где је моделу дата дефиниција класе у програмском језику *Java*, највишу вредност CodeBLEU метричке оцене постигао је StarCoder 7B са скором од 0.98, што је уједно и највиша вредност *syntax match* компоненте (1.0). Одмах иза њега

налази се DeepSeek 6.7B са CodeBLEU резултатом од 0.76, док су CodeLlama 7B и StarCoder 3B остварили стабилне и уједначене резултате у средњем опсегу.

Најнижи резултат бележи StarCoder 1B, што се може приписати мањој архитектури и ограниченој способности разумевања сложенијих структурних образаца у Јави.

Табела 18 – Резултати модела на упиту 1

Модел	**CodeBLEU**	**n_gram**	**weighted n_gram**	**syntax match**
StarCoder 1B	0.55	0.51	0.65	0.5
StarCoder 3B	0.63	0.73	0.72	0.5
StarCoder 7B	0.98	0.97	0.96	1.0
DeepSeek 6.7B	0.76	0.93	0.95	0.5
CodeLlama 7B	0.89	0.84	0.83	1.0
MistralAI 7B	0.63	0.73	0.71	0.5

Модел StarCoder 7B се показао најуспешнијим у разумевању и генерисању Јава класе, демонстрирајући одличну синтаксну усаглашеност и прецизност генерисаног кода.

У другом задатку, где је моделу дата дефиниција функције у програмском језику C++, поново се као најпрецизнији показао StarCoder 7B, који је остварио CodeBLEU скор од 0.98 и *syntax match* од 1.0, што указује на готово потпуну синтаксну и семантичку поклапање са референтним кодом. Од осталих модела, DeepSeek 6.7B и StarCoder 3B такође постижу солидне резултате (0.65 и 0.66), док су мањи модели (StarCoder 1B, CodeLlama 7B, Mistral 7B) показали нешто слабију конзистентност.

Табела 19 - Резултати модела на упиту 2

Модел	**CodeBLEU**	**n_gram**	**weighted n_gram**	**syntax match**
StarCoder 1B	0.53	0.55	0.55	0.5
StarCoder 3B	0.66	0.65	0.89	0.5
StarCoder 7B	0.98	0.98	0.95	1.0
Deepseek 6.7B	0.65	0.59	0.92	0.5
CodeLlama 7B	0.62	0.70	0.70	0.5
MistralAI 7B	0.66	0.73	0.79	0.5

Модел StarCoder 7B је поново показао најбоље резултате, овај пут у задатку генерисања функционалног C++ кода, потврђујући робусност у синтакси и на нивоу токена, док је Mistral 7B показао стабилан, али нешто нижи скор, вероватно због оријентације ка задацима природног језика.

Трећи задатак представљао је најкраћи и уједно најзахтевнији сценарио, односно генерисање једне линије кода у програмском језику JavaScript, где су разлике између модела најизраженије.

Табела 20 - Резултати модела на упиту 3

Модел	CodeBLEU	n_gram	weighted n_gram	syntax match
StarCoder 1B	0.29	0.14	0.15	0.5
StarCoder 3B	0.36	0.24	0.29	0.5
StarCoder 7B	0.36	0.24	0.30	0.5
DeepSeek 6.7B	0.25	0.07	0.08	0.5
CodeLlama 7B	0.22	0.02	0.02	0.5
MistralAI 7B	0.74	0.56	0.56	1.0

Највиши CodeBLEU скор остварио је Mistral 7B са вредношћу од 0.74, док су сви остали модели постигли знатно ниже резултате (испод 0.4). Овакви резултати показују да Mistral 7B, упркос нижем syntax match резултату у претходним задацима, боље генерализује у кратким и изолованим фрагментима кода. што је у складу са његовим ширим пре-тренирањем на мешовитим доменима (код и текст).

У кратким задацима, где нема контекста ни сложене структуре, Mistral 7B показује супериорну способност препознавања синтаксичких образаца *JavaScript* језика.

Табела 21 - Сумирани резултати по језицима

Задатак	Програмски језик	Најбољи модел	CodeBLEU
1	Java	StarCoder 7B	0.98
2	C++	StarCoder 7B	0.98
3	JavaScript	Mistral 7B	0.74

5.2.2. Поређење модела на тестном скупу

Ако узмемо сада тестни скуп од 500 узорака, и покренемо закључивање (*inference*) над њима, можемо да израчунамо пондерисани CodeBLEU за сваки модел на основу њиховог учешћа по програмским језицима у скупу података.

Пошто је у структури тестног скупа *Python* био најзаступљенији (≈ 45%), а остали језици у знатно мањој мери, евалуација је спроведена у контексту *Python* кода, са циљем да се добије стабилна и репрезентативна вредност метрике.

Приликом закључивања сваки модел је добио исти упит (првих 256 токена) из оригиналног кода, и генерисао наставак од 128 токена. Затим је исти број токена издвојен и из референтног кода, тако да се добију упоредиви суфикс сегменти. На тај начин се мери квалитет наставка кода.

За сваку генерисану пару (*pred*, *ref*) израчуната је CodeBLEU метрика на следећи начин: Пошто се евалуација спроводи над парцијалним наставцима кода (до 128 токена), модели често не завршавају синтаксно комплетне функције.

Да би се избегла дисторзија резултата изазвана syntax грешкама, у коначном CodeBLEU обрачуну је смањен удео syntax компоненте, док је *dataflow* у потпуности искључен. Уместо стандардне равномерне расподеле (0.25, 0.25, 0.25, 0.25), примењене су тежине (0.4, 0.4, 0.2, 0.0), чиме је нагласак стављен на лексичко поклапање токена, али је syntax делимично задржан ради мерења структурне усаглашености у примерима где је код валидан.

Табела 22 - Резултати модела на тестном скупу

Модел	Варијанта	CodeBLEU	n_gram	weighted n_gram	syntax
StarCoder 1B	base	0.5	0.61	0.62	0.04
	finetuned	0.584	0.71	0.72	0.06
StarCoder 3B	base	0.54	0.65	0.67	0.06
	finetuned	0.596	0.72	0.73	0.09
StarCoder 7B	base	0.744	0.90	0.90	0.12
	finetuned	0.8	0.97	0.97	0.14
Mistral 7B	base	0.72	0.87	0.87	0.12
	finetuned	0.78	0.94	0.94	0.13
CodeLlama 7B	base	0.596	0.71	0.72	0.12
	finetuned	0.62	0.74	0.74	0.14
DeepSeek 6.7B	base	0.726	0.87	0.88	0.13
	finetuned	0.792	0.95	0.96	0.14

Анализа резултата

Резултати показују да постоји разлика у квалитету излаза између мањих и већих модела, као и да је ефекат финог подешавања израженији код модела мањег капацитета.

Код StarCoder 1B и StarCoder 3B, базне варијанте бележе ниже резултате, што указује да мањи број параметара ограничава способност модела да правилно разуме контекст и структуру програмског кода. Након финог подешавања, њихов CodeBLEU расте, што сугерише бољу репродукцију семантичких образаца и стабилнију генерацију кода. С друге стране, већи модели (7B и 6.7B варијанте) показују знатно већи степен компетентности већ у базној форми. Модели StarCoder 7B, Mistral 7B, CodeLlama 7B и DeepSeek 6.7B остварују CodeBLEU у распону од 0.72 до 0.74 већ без fine-tuninga, што указује на њихову изузетну способност разумевања синтаксе и логике кода.

Фино подешавање код ових модела више делује као средство унапређења стила писања кода, јер се побољшава ток и читљивост кода, али не мења се драстично укупни резултат. Најбољи резултати се постижу код StarCoder 7B (finetuned), који бележи CodeBLEU = 0.80, док је DeepSeek 6.7B (finetuned) одмах иза са 0.792. Оба

модела такође имају највише вредности n_gram и weighted n_gram, што указује на висок степен сличности између генерисаног и референтног кода.

Објашњење изабраног модела за даљи рад

На основу анализе, StarCoder 7B (finetuned) је изабран као главни модел за интеграцију у AssistHTEC екстензију. У односу на друге, над овим тестовима који су вршени, он нуди најбољи баланс између тачности, стабилности и синтаксичке исправности генерисаног кода. У поређењу са осталим моделима, показује највиши CodeBLEU, али и најконзистентније резултате кроз све метрике које представљају део CodeBLEU (n_gram, weighted n_gram, syntax).

6. Имплементација

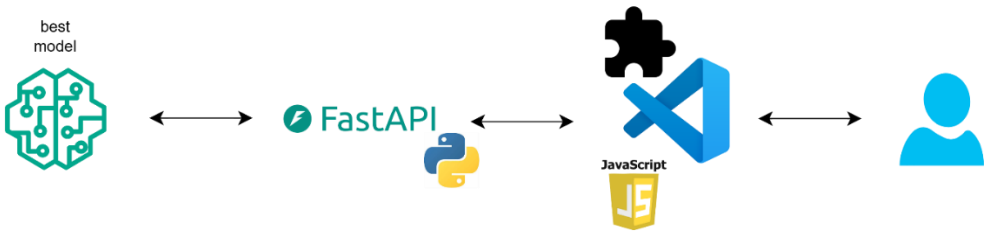
6.1. Креирање сервиса за предвиђање који користи модел

Сервис за предвиђање представља основу нашег система и реализован је у Python окружењу користећи FastAPI оквир који омогућава лако дефинисање REST API ендпоинта и асинхрону обраду захтева. API сервис је задужен за комуникацију са фино подешеним LLM моделом који је претходно трениран и објављен као PEFT адаптер на платформи Hugging Face.

Модел се приликом иницијализације сервиса учитава у меморију, након чега се може користити за обраду упита. Основни ендпоинт */predict* прихвата JSON захтев који садржи текстуални садржај (код) или чист текст, и враћа предикцију наставка у облику низа карактера.

Унутар сервиса се користи tokenizer из библиотеке transformers, а сам модел користи параметре за закључивање као што су max_new_tokens, temperature, top_p и repetition_penalty, чиме се контролише дужина и креативност генерисаног излаза, о којима је било приче у ранијим поглављима, у дели Параметри.

Сервис се извршава локално на серверу *dyres.pmf.kg.ac.rs* на адреси http://127.0.0.1:8000, али је током тестирања омогућено и да му се приступи споља преко тунела (на пример, *ssh -R 8000:localhost:8000 user@server*) на локалу корисника, како би VS Code екстензија могла да шаље захтев, чак и када се покреће са удаљене машине.



Дијаграм 5 - Комуникација корисник - модел

6.2. Развој екстензије за VS Code

Развој екстензије реализован је коришћењем JavaScript-а у оквиру *Visual Studio Code Extension Development Environment*-а, који омогућава креирање, тестирање и генерисање инсталационих пакета екстензија за VS Code. Коришћена је почетна структура коју нуди алат *yo code*, као и VS Code Extension API, који омогућава приступ садржају тренутно отворених датотека, обраду текста, као и додавање прилагођених команди у већ постојећу командну палету.

Екстензија је реализована тако да након активације од стране корисника, аутоматски приступа садржају отвореног фајла и екстрактује написане линије кода у циљу припреме улазних података за фино подешени LLM модел.

Активира се путем команде *assisthtec.triggerInline*, дефинисане у package.json фајлу, а након активације:

1. приступа садржају тренутно отвореног фајла,
2. издваја линије кода које служе као улаз моделу,
3. шаље их као JSON захтев API сервису,
4. прима предикцију и убацује је директно у документ
5. уколико корисник потврди са Enter или Tab, код се званично уписује у едитор,
6. уколико корисник поништи предлог, код се брише

На овај начин се добија ефекат интерактивног асистента за кодирање који делује у реалном времену, на сваку промену коју корисник унесе.

6.3. Архитектура екстензије и интеграција са моделом

Са становишта архитектуре система, екстензија се ослања на клијент-сервер комуникацију између VS Code окружења и локално покренутог API сервиса. API је развијен у Python-у, коришћењем библиотеке FastAPI , и покреће се као сервис на *localhost* адреси.

Модел, који је фино подешен и унапред учитан у овом API-ју, прихвата текстуални улаз (линије кода), врши обраду и враћа предикцију у виду наставка кода. VS Code екстензија затим узима тај резултат и убацује га директно у фајл почевши од позиције курсора, симулирајући понашање аутоматског асистента.

Екстензија користи Axios библиотеку за слање захтева, а VS Code API за уметање добијеног текста у фајл. Унутар кода екстензије дефинисан је максимални број карактера контекста (*max_context*), као и параметри за раздвајање почетка и краја фајла (*head_budget, tail_budget*) који се користе како би се моделу послао релевантан део кода.

6.4. Функционалности екстензије (предикција наредног дела кода)

Кључна функционалност екстензије је предикција наставка кода на основу контекста. Корисник може, путем командне палете у VS Code-у, активирати команду која иницира процес предикције. Екстензија прикупља написане линије кода тренутно отвореног фајла, прослеђује их Python API-ју, који затим, уз помоћ LLM модела, генерише могући наставак.

Генерисани код се затим убацује на позицију курсора унутар фајла, без потребе за ручним копирањем или додатним корацима. Ово омогућава кориснику флуидан радни ток, сличан искуству које нуде комерцијални алати као што је GitHub Copilot, али без потребе за сталном интернет конекцијом и зависношћу од спољних услуга.

6.5. Корисничко искуство екстензије

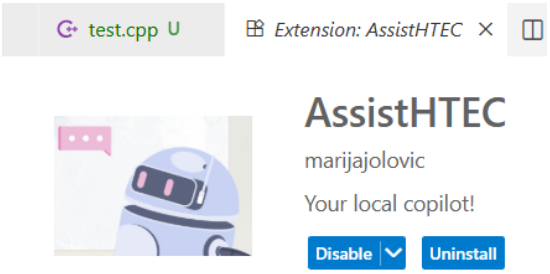
Екстензија је осмишљена тако да се природно уклапа у VS Code окружење, без нарушавања уобичајеног начина рада. Иако постоји благо кашњење у приказу предлога, систем реагује довољно брзо да не ремети ток писања. На овај начин кориснику је омогућено несметано програмирање, док модел у позадини динамички генерише релевантне наставке кода.

На крају развоја, екстензија је упакована у *.vsix* формат, који представља званични пакетни формат *Visual Studio Code* екстензија. Паковање је реализовано коришћењем алата *vsce package*, који креира дистрибутивни фајл *assisthtec-1.0.0.vsix*, који представља прву верзију (зависи од конфигурација у *json* фајлу).

Паковање екстензије

```
cd vscode-extension
vsce package
```

Овако припремљена екстензија се може лако делити и инсталирати на било ком VS Code окружењу, где је довољно изабрати опцију *„Install from VSIX…”* у менију Extensions.



Слика 3 - AssistHTEC екстензија

Ово омогућава да екстензију користе и други корисници, под условом да:

- поседују сопствени модел (или адаптер) фино подешен за кодирање,
- или имају конекцију ка API сервису који излаже /predict ендпоинт у складу са очекиваним форматима.

На тај начин, екстензија није везана искључиво за један модел, већ представља флексибилан интерфејс који може да се повеже са било којим LLM сервисом за предвиђање кода, под условом коришћења истих формата.

6.6. Поређење екстензије са GitHub Copilot-ом

Ради евалуације функционалности, извршено је поређење између предложене екстензије и GitHub Copilot-а на реалистичном, домен-специфичном задатку:

- коришћен је идентичан улаз,
- анализирани су CodeBLEU резултати као мера сличности са референтним решењима

Коришћен је један изоловани .c изворни фајл из AMD Kernel кода. Овај домен је намерно одабран јер садржи високо специјализоване *C* елементе кода. Због те

специфичности, минимална је вероватноћа да су конкретни кодови били у тренинг подацима базних модела који су тестирани, па самим тим, можемо боље да измеримо генерализацију модела.

Сви параметри за максимални број нових токена, температура, праг вероватноће и метрике за евалуацију, су остали исти као и у претходним спроведеним закључивањима и евалуацији.

Табела 23 - Анализа AssistHTEC и GitHub Copilot

Модел	**CodeBLEU**	**n_gram**	**weighted n_gram**	**syntax**
AssistHTEC (StarCoder 7B - finetuned)	0.424	0.398	0.412	0.5
StarCoder 7B - base	0.378	0.345	0.349	0.5
GitHub Copilot	0.447	0.421	0.446	0.5

Резултати показују да StarCoder-7B (*finetuned*), који је наш изабрани модел, остварује благо боље резултате у односу на базни модел. Иако је GitHub Copilot постигао виши скор, треба имати у виду да му је дат само један фајл без додатног контекста о повезаним фајловима, што може да објасни мало лошије резултате него очекиване када се ради о Copilot-у. Додатним финим подешавањем на скупу домен-специфичних података као што су AMD кодови писани у C програмском језику, модел би се вероватно још боље прилагодио специфичним обрасцима тог домена, што би резултовало побољшањем перформанси.

Закључак

Сумирање кључних резултата

У овом раду реализован је систем за аутоматско генерисање и евалуацију програмског кода заснован на претрениранim великим језичким моделима (LLM).

Фокус је био на процесу тренирања различитих модела, применом PEFT технике и QLoRA метод за прилагођавање већих модела да се извршавају у оквиру доступних ресурса, као и на изради VS Code екстензије која омогућава локално генерисање кода у реалном окружењу.

Систем је тестиран на прилагођеном скупу од 500 узорака, при чему је евалуација обављена помоћу CodeBLEU метрике, која комбинује лексичке, синтаксне и семантичке аспекте генерисаног кода. Резултати су показали да фино подешени модел у просеку остварује више CodeBLEU резултате у односу на базни модел, посебно у сегментима који се односе на синтаксну исправност и конзистентност програмске структуре.

Највеће побољшање уочено је код модела StarCoder 7B и StarCoder 3B, који су показали стабилније перформансе у Python и Java задацима.

Ограничење рада

Иако су резултати охрабрујући, рад има неколико ограничења.

Прво, величина и разноврсност скупа података ограничена је на 500 узорака, при чему Python доминира као примарни језик, што утиче на генерализацију резултата. Друго, због параметра max_new_tokens = 128, модел није увек завршавао целу функцију, што је утицало на ниже вредности *syntax* и *dataflow* компоненти CodeBLEU метрике. Треће, ограничени хардверски ресурси онемогућили су тестирање већих модела и дуготрајнији fine-tuning.

У погледу метрика, CodeBLEU је био основна мера за поређење модела, али ни он није савршен индикатор стварне структурне и семантичке сличности између кода који је модел генерисао и референтног решења. Да би се умањио утицај syntax грешака у непотпуним наставцима, примењене су прилагођене тежине (0.4, 0.4, 0.2, 0.0), чиме је нагласак био стављен на лексичко поклапање токена.

Поред CodeBLEU-а, у литературама се помињу и напредније метрике као што је TSED (Tree Structural Edit Distance), која мери разлику између апстрактних синтаксних стабала (AST) два фрагмента кода. Аутори ове метрике предлажу тежину од 0.8 за Insert операцију и 1.0 за Delete и Rename операције, а емпиријске студије показују да TSED позитивно корелира са BLEU резултатима, али нуди већу прецизност у процени стварне структурне и семантичке сличности. Ипак, у тренутној имплементацији постоји ограничење у нормализацији резултата, када се пореди празно AST стабло са непразним, метрика може вратити не-нулту сличност, што је контраинтуитивно и потенцијално доводи до погрешног тумачења резултата.

Све ово указује да евалуација генерисаног кода и даље представља отворен истраживачки изазов, јер ниједна постојећа метрика не пружа потпуно поуздану

корелацију између синтаксне исправности, логичке тачности и стварне употребљивости кода.

Предлози за унапређење и даља истраживања

Даљи развој система могао би бити усмерен ка интеграцији са постојећим open-source екстензијама и алатима за развој кода, што би омогућило лакше тестирање и шире прихватање решења у заједници програмера. Посебан значај има унапређење евалуације, увођењем комбинованих метрика као што су TSED, CodeBERTScore и Execution Accuracy, које би омогућиле свеобухватнију процену квалитета генерисаног кода, не само на нивоу синтаксе и лексике, већ и у погледу функционалне исправности и логичке доследности решења. Поред тога, требало би ставити нагласак на проширење скупа података који би обухватио већи број програмских језика и различите категорије задатака, чиме би се унапредила генерализација модела и реалнија процена његових способности. Унапређење би било могуће кроз примену обимнијих рачунарских ресурса, што би омогућило дужи процес додатне обуке и експериментисање са различитим конфигурацијама параметара током поступка финог подешавања.

Закључно, рад показује да параметарски ефикасно фино подешавање LLM модела представља исплатив и применљив приступ за задатке генерисања програмског кода. Иако постоје ограничења у погледу евалуације и обима модела, добијени резултати показују напредак у квалитету генерисаног кода и отварају простор за даља истраживања у правцу прецизнијег вредновања и примене LLM система у развојним окружењима.

Додатак

У овом поглављу приказани су резултати закључивања за шест одабраних модела: StarCoder 1B, StarCoder 3B, StarCoder 7B, MistralAI 7B, CodeLlama 7B и DeepSeekCoder 6.7B, на три различита програмска упита (prompt-а) који представљају различите језике и типове задатака, о којима је било речи раније у поглављу Методологије.

StarCoder 1B

Излаз 1: StarCoder-1B (base)

```
// Implement a simple class "BankAccount" with fields: accountNumber, balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber) {
            this.accountNumber = accountNumber;
            this.balance = 0;
       }

       public void deposit(double amount) {
            if (amount < 0) {
                  throw new IllegalArgumentException("Amount must be positive");
            }
            this.balance += amount;
       }

       public void withdraw(double amount) {
            if (amount < 0) {
                  throw new IllegalArgumentException("Amount must be positive");
            }
            if (this.balance < amount) {
                 throw new IllegalArgumentException("
```

Излаз 1: StarCoder-1B (finetuned)

```
// Implement a simple class "BankAccount" with fields: accountNumber, balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
```

```
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        if (this.balance < amount) {
            throw new IllegalArgumentException("Insufficient funds");
        }
        this.balance -= amount;
    }
}
```

Излаз 2: StarCoder-1B (base)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement me
    int max_length = 0;
    int max_index = 0;
    for (int i = 0; i < nums.size(); i++) {
        int current_length = 1;
        for (int j = i + 1; j < nums.size(); j++) {
            if (nums[j] > nums[i]) {
                current_length++;
            } else {
                break;
            }
        }
        if (current_length > max_length) {
            max_length = current_length;
            max_index = i;
        }
    }
```

Излаз 2: StarCoder-1B (finetuned)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement this function
    int len = 0;
    int max_len = 0;
    for (int i = 0; i < nums.size(); i++) {
        if (nums[i] > nums[i+1]) {
            max_len = max(max_len, i-nums[i]+1);
        }
    }
    return max_len;
}

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << longestIn
```

Излаз 3: StarCoder-1B (base)

```
async function getUserData(userId) {
    const user = await User.findById(userId);
    if (!user) {
        throw new Error('User not found');
    }
    return user;
}

async function getUsers(limit, offset) {
    const users = await User.find({}).limit(limit).skip(offset);
    return users;
}

async function getUser(userId) {
    const user = await User.findById(userId);
    if (!user) {
        throw new Error('User not found');
    }
    return user;
}

async function updateUser(userId, user) {
    const user
```

Излаз 3: StarCoder-1B (finetuned)

```
async function getUserData(userId) {
    const response = await
fetch(`${process.env.REACT_APP_API_URL}/users/${userId}`);
    const data = await response.json();
    return data;
}

async function getUsers() {
    const response = await fetch(`${process.env.REACT_APP_API_URL}/users`);
    const data = await response.json();
    return data;
}

async function getUsersByRole(role) {
    const response = await
fetch(`${process.env.REACT_APP_API_URL}/users/role/${role}`);
    const data = await response
```

Модел StarCoder 1B често генерише делимично исправан, али непотпун код, без унутрашње логичке провере или адекватне обраде грешака. Фино подешавање (fine-tuning) је имало највидљивији ефекат код мањих модела као што је 1B. Код StarCoder 1B, fine-tuned верзија исправља логичке грешке и допуњује недостајуће блокове, као што су изузеци и провере негативних вредности у Java коду.

StarCoder 3B

Излаз 1: StarCoder-3B (base)

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && this.balance >= amount) {
            this.balance -= amount;
        }
    }

    public double getBalance() {
```

```
        return this.balance;
    }
}
```

Излаз 1: StarCoder-3B (finetuned)

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
      }

      public void deposit(double amount) {
        if (amount >= 0) {
            this.balance += amount;
        }
      }

      public void withdraw(double amount) {
        if (amount >= 0 && this.balance >= amount) {
            this.balance -= amount;
        }
      }

      public double getBalance() {
        return this.balance;
      }
}
```

Излаз 2: StarCoder-3B (base)

```
// Implement a function that finds the longest increasing subsequence in a
given vector of integers.
// The function should return the length of the longest increasing
subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement me
    return 0;
}
```

Излаз 2: StarCoder-3B (finetuned)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement
    return 0;
}

int main() {
    vector<int> nums = {10, 22, 9, 33, 21, 50, 41, 60};
    cout << longestIncreasingSubsequence(nums) << endl;
    return 0;
}
```

Излаз 3: StarCoder-3B (base)

```
async function getUserData(userId) {
  const response = await fetch(`/api/users/${userId}`);
  const data = await response.json();
  return data;
}

async function getPostData(postId) {
  const response = await fetch(`/api/posts/${postId}`);
  const data = await response.json();
  return data;
}

async function getPostsByUser(userId) {
  const response = await fetch(`/api/posts/user/${userId}`);
  const data = await response.json();
  return data;
}

async function getPostsByTag(tagId) {
  const response =
```

Излаз 3: StarCoder-3B (finetuned)

```
async function getUserData(userId) {
 const response = await fetch(
 `https://api.github.com/users/${userId}?client_id=${process.env.GITHUB_CLIENT_ID}&client_secret=${process.env.GITHUB_CLIENT_SECRET}`
  );
  const data = await response.json();
  return data;
}

```

```
async function getRepos(userId) {
  const response = await fetch(

`https://api.github.com/users/${userId}/repos?client_id=${process.env.GITHUB_CLIENT_ID}&client_secret=${process.env.GITHUB_CLIENT_SECRET}`
```

Код StarCoder 3B, побољшана је структурна конзистентност и употреба функција као што су main() или пример иницијализације података. У односу на 1B модел, видљиво је боље разумевање логичке структуре, као и употреба услова (if/else) и иницијализације објеката.

StarCoder 7B

Када причамо о првом упиту, у овом случају и базни претренирани и ифно подешен модел враћају исто:

Излаз 1: StarCoder-7B

```
// Implement a simple class "BankAccount" with fields: accountNumber, balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
            this.accountNumber = accountNumber;
            this.balance = 0;
    }

    public void deposit(double amount) {
            this.balance += amount;
    }

    public void withdraw(double amount) {
            if (amount > 0) {
                    this.balance -= amount;
            }
    }

    public double getBalance() {
            return this.balance;
    }
}
```

Што се тиче другог упита, и у овом случау и базни и фино подешени модел враћају исто:

Излаз 2: StarCoder-7B

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

```

```
#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement
    return 0;
}

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << longestIncreasingSubsequence(nums) << endl;
    return 0;
}
```

Код последњег упита, StarCoder 7B враћа различите излазе у зависности од варијанте модела коју користимо, тј. да ли користимо базни или фино подешен модел.

Излаз 3: StarCoder-7B (base)

```
async function getUserData(userId) {
  const user = await User.findById(userId);
  return user;
}

async function getUsers(userId) {
  const users = await User.find({ _id: { $ne: userId } });
  return users;
}

async function updateUser(userId, data) {
  const user = await User.findByIdAndUpdate(userId, data, {
    new: true,
  });
  return user;
}

async function deleteUser(userId) {
  const user = await User.findByIdAndDelete(userId);
  return user;
}

module.exports = {

```

Излаз 3: StarCoder-7B (finetuned)

```
async function getUserData(userId) {
    const user = await User.findOne({
    where: {
      id: userId,
    },
    include: [
      {
        model: User,
        as: "following",
        attributes: ["id", "username"],
      },
```

```
      {
        model: User,
        as: "followers",
        attributes: ["id", "username"],
      },
    ],
  });

  if (!user) {
    throw new Error("User not found");
  }

  return user;
}

module.exports = {
  getUserData,
};

```

StarCoder 7B показује највећу стабилност — у више случајева фино подешени и базни модел враћају идентичан, потпуно исправан код, што указује да је већ основни модел добро обучен за задатке програмирања, док је fine-tuning додатно стабилизовао предвиђања.

MistralAI 7B

Излаз 1: MistralAI-7B (base)

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > balance) {
            System.out.println("Insufficient funds");
        } else {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

```
29
30 public class BankAccountTest {
31     public static void main
```

Излаз 1: MistralAI-7B (finetuned)

```
1  // Implement a simple class "BankAccount" with fields: accountNumber,
   balance.
2  // Provide methods: deposit(double amount), withdraw(double amount), and
3  // getBalance().
4  // Withdraw should not allow negative balance.
5
6  public class BankAccount {
7      private String accountNumber;
8      private double balance;
9      public BankAccount(String accountNumber) {
10         this.accountNumber = accountNumber;
11         this.balance = 0.0;
12     }
13
14     public void deposit(double amount) {
15         if (amount >= 0) {
16             this.balance += amount;
17         }
18     }
19
20     public void withdraw(double amount) {
21         if (amount >= 0 && this.balance >= amount) {
22             this.balance -= amount;
23         }
24     }
25
26     public double getBalance() {
27         return this.balance
```

Излаз 2: MistralAI-7B (base)

```
1  // Implement a function that finds the longest increasing subsequence in a
   given vector of integers.
2  // The function should return the length of the longest increasing
   subsequence.
3
4  #include <vector>
5  using namespace std;
6
7  int longestIncreasingSubsequence(const vector<int>& nums) {
8      // TODO: implement your solution here
9      return 0;
10 }
11
12 int main() {
13     vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
14     int length = longestIncreasingSubsequence(nums);
15     cout << "The length of the longest increasing subsequence is: " <<
   length << endl;
```

```
    return 0;
}
```

Излаз 2: MistralAI-7B (finetuned)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement this function
    return 0;
}

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    int length = longestIncreasingSubsequence(nums);
    cout << "Length of the Longest Increasing Subsequence: " << length << endl;
    return 0;
}

// Output:
// Length of the Longest Increasing Subsequence: 4

// Explanation:
// The longest increasing
```

Излаз 3: MistralAI-7B (base)

```
async function getUserData(userId) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);
    const data = await response.json();
    return data;
}

async function getPosts(userId) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userId}`);
    const data = await response.json();
    return data;
}

async function getComments(postId) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/comments?
```

Излаз 3: MistralAI-7B (finetuned)

```
async function getUserData(userId) {
    const response = await
fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);
  const userData = await response.json();
  return userData;
}

async function getUserPosts(userId) {
  const response = await
fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userId}`);
  const userPosts = await response.json();
  return userPosts;
}

async function getUserComments(userId) {
  const response = await fetch(`https://jsonplaceholder.ty
```

MistralAI 7B је показао изузетно добре резултате на JavaScript упиту, где други модели имају тенденцију да додају доста непотребних функција. Mistral често користи реалне API позиве (нпр. fetch()), чиме демонстрира блискост са практичним програмским шаблонима.

CodeLlama 7B

Излаз 1: CodeLlama-7B (base)

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (this.balance - amount >= 0) {
            this.balance -= amount;
        }
    }

    public double getBalance() {
        return this.balance;
    }

    public String
```

Излаз 1: CodeLlama-7B (finetuned)

```
// Implement a simple class "BankAccount" with fields: accountNumber, balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > this.balance) {
            System.out.println("Cannot withdraw more than balance");
        } else {
            this.balance -= amount;
        }
    }

    public double getBalance() {
```

Излаз 2: CodeLlama-7B (base)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement
    return 0;
}

int main() {
    vector<int> nums = {10, 22, 9, 33, 21, 50, 41, 60, 80};
    int result = longestIncreasingSubsequence(nums);
    cout << "The length of the longest increasing subsequence is " << result << endl;
    return 0;
}

// Output:
```

```
// The length of the longest increasing subsequence is 5

// Explanation:
//
```

Излаз 2: CodeLlama-7B (finetuned)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement
    return 0;
}

int main() {
    vector<int> nums = {10, 22, 9, 33, 21, 50, 41, 60, 80};
    int result = longestIncreasingSubsequence(nums);
    cout << result << endl;
}

// Complexity:
// O(n log n)
// O(n^2)

// Example:
// Input: nums = [10, 22, 9,
```

Излаз 3: CodeLlama-7B (base)

```
async function getUserData(userId) {
    return new Promise((resolve, reject) => {
    const query = `SELECT * FROM users WHERE id = ${userId}`;
    connection.query(query, (err, result) => {
      if (err) {
        reject(err);
      } else {
        resolve(result[0]);
      }
    });
  });
}

function getUserPosts(userId) {
  return new Promise((resolve, reject) => {
    const query = `SELECT * FROM posts WHERE user_id = ${userId}`;
    connection.query
```

Излаз 3: CodeLlama-7B (finetuned)

```
async function getUserData(userId) {
    const userData = await fetch(`/api/users/${userId}`).then((res) =>
res.json());
       return userData;
}

async function getUserPosts(userId) {
       const userPosts = await
fetch(`/api/posts?userId=${userId}`).then((res) => res.json());
       return userPosts;
}

async function getUserFollowers(userId) {
       const userFollowers = await
fetch(`/api/users/${userId}/followers`).then((res) => res.json
```

CodeLlama 7B је у Java и C++ задацима произвео код који је синтаксно исправан, али понекад мање прецизан у условима и алгоритамској логици. Ипак, после финог подешавања приметно је боље руковање граничним случајевима (нпр. контрола стања биланса у BankAccount класи).

DeepSeek Coder 6.7B

Излаз 1: DeepSeek-6.7B (base)

```
// Implement a simple class "BankAccount" with fields: accountNumber,
balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= this.balance) {
            this.balance -= amount;
        }
    }

    public double getBalance() {
        return this.balance;
    }
}
```

Излаз 1: DeepSeek-6.7B (finetuned)

```
// Implement a simple class "BankAccount" with fields: accountNumber, balance.
// Provide methods: deposit(double amount), withdraw(double amount), and
// getBalance().
// Withdraw should not allow negative balance.

public class BankAccount {
    private String accountNumber;
    private double balance;
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= this.balance) {
            this.balance -= amount;
        }
    }

    public double getBalance() {
        return this.balance;
    }
}

// Test
public class BankAccountTest {
```

Излаз 2: DeepSeek-6.7B (base)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement this function
    return 0;
}
```

Излаз 2: DeepSeek-6.7B (finetuned)

```
// Implement a function that finds the longest increasing subsequence in a given vector of integers.
// The function should return the length of the longest increasing subsequence.

#include <vector>
using namespace std;

int longestIncreasingSubsequence(const vector<int>& nums) {
    // TODO: implement this function
    return 0;
}

// Test cases

#include <cassert>

int main() {
    assert(longestIncreasingSubsequence({10, 9, 2, 5, 3, 7, 101, 18}) == 4);
    assert(longestIncreasingSubsequence({0, 1, 0, 3, 2, 3}) == 4);
    assert(longestIncreasingSubsequence({7, 7, 7, 7, 7
```

Излаз 3: DeepSeek-6.7B (base)

```
async function getUserData(userId) {
  const response = await fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);
  const userData = await response.json();
  return userData;
}

async function getPostsByUserId(userId) {
  const response = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userId}`);
  const posts = await response.json();
  return posts;
}

async function getCommentsByPostId(postId) {
```

Излаз 3: DeepSeek-6.7B (finetuned)

```
async function getUserData(userId) {
  const user = await fetch(`https://api.github.com/users/${userId}`).then(
    (res) => res.json()
  );

  return user;
}

export default getUserData;
```

DeepSeekCoder 6.7B је показао конзистентне излазе. У свим примерима генерише семантички смислен код и убацује примере тест случајева или коментаре који описују очекивани излаз, што указује на висок ниво контекстуалног разумевања.

Код већих модела, попут 7B варијанти, fine-tuning углавном побољшава стил, али не мења драстично исход, што указује да су базне верзије већ веома компетентне.

Литература

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

[2] Paul, D. G., Zhu, H., & Bayley, I. (2024, July). Benchmarks and metrics for evaluations of code generation: A critical review. In 2024 IEEE International Conference on Artificial Intelligence Testing (AITest) (pp. 87-94). IEEE.

[3] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., ... & Liu, S. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

[4] Eva Thompson, "What is FIM and why does it matter in LLM-based AI", Medium, 2024. , октобар 2025

[5] Introduction - Tree-sitter, октобар 2025

[6] Shi-yan, Tree-sitter Viewer: visualize tree-sitter syntax tree. GitHub Repository , новембар 2025

[7] K4black, CodeBLEU: Pip compatible metric implementation for Linux/macOS/Win. GitHub Repository, октобар 2025

[8] Hugging Face, Fine-tuning a Code LLM on Custom Code on a Single GPU, 2024., септембар 2025

[9] PEFT: Parameter-Efficient Fine-Tuning Library. Hugging Face Documentation., новембар 2025

[10] IBM, What is LoRA (Low-Rank Adaption)?, октобар 2025

[11] GeeksforGeeks, What is QLoRA (Quantized Low-Rank Adapter)?, октобар 2025

Кратка биографија кандидата

Марија Јоловић рођена је 26. фебруара 2003. године у Крагујевцу. Основну школу похађала је у ОШ „Мирко Јовановић“ у Крагујевцу, а средњу у Првој крагујевачкој гимназији, у одељењу ученика са посебним способностима за рачунарство и информатику, као носилац Вукове дипломе 2021. године.

Природно-математички факултет Универзитета у Крагујевцу, смер Информатика-софтверско инжењерство, уписала је 2021. године и положила све предмете предвиђене планом и програмом основних академских студија.

Од октобра 2023. године, изабрана је у звање студента демонстратора. Од марта 2024. године ангажована је на научно-истраживачким пројектима Центра за рачунарско моделовање и оптимизацију Природно-математичког факултета у Крагујевцу.

Дипломски рад је реализован као резултат сарадње Природно-математичког факултета и компаније *High Tech Engineering Center* (HTEC), у оквиру које је аутор током студија била стипендиста компаније.

Од марта 2025. године запослена је у компанији HTEC на позицији инжењера машинског учења.

Универзитет у Крагујевцу
Природно-математички факултет
Институт за математику и информатику

Завршни рад под називом

Развој интелигентног асистента за програмирање

одбрањен је _____.

МЕНТОР:

_____
др Бранко Арсић, доцент,
Природно-математички факултет, Крагујевац

ЧЛАНОВИ КОМИСИЈЕ:

_____
др Бобан Стојановић, редовни професор,
Природно-математички факултет,
Крагујевац

_____
др Марина Свичевић, доцент,
Природно-математички факултет,
Крагујевац

Завршни рад је оцењен оценом _____.