Financira Evropska unija
NextGenerationEU
REPUBLIKA SLOVENIJA
MINISTRSTVO ZA VISOKO ŠOLSTVO, ZNANOST IN INOVACIJE
ARIS Javna agencija za znanstvenoraziskovalno in inovacijsko dejavnost Republike Slovenije

D6.5
-
Končna različica programske opreme za IaC z omejitvami zmogljivosti, robustnosti in varnosti

15.06.2026

Podatki o dokumentu	
Različica	0.5
Pričakovan datum oddaje	30.6.2026
Datum oddaje	22.6.2026

RRP	6	**Oznaka izročka**	D6.5
Povezani izročki	D6.1,D6.2,D6.3,D6.4	**Dostop**	Javno
Glavni urednik	Daniel Vladušič	**Glavni avtor**	Nikola Sekulovski
Ostali	Nejc Bat, Jure Medvešek		

Ključne besede
Programska oprema, Veliki jezikovni modeli (LLM), IaC

Kazalo

Kazalo Tabel

Kazalo Slik

Povzetek

Izroček D6.5 predstavlja zaključno fazo dela v RRP6 – VeMo-IaC in opisuje končno programsko rešitev za uporabo jezikovnih tehnologij pri generiranju infrastrukturne kode v obliki Ansible nalog. Dokument povezuje rezultate predhodnih izročkov D6.1–D6.4, ki so obravnavali izbor podatkov, začetno učenje in primerjalno vrednotenje modelov, iterativno izboljševanje učnih množic ter sistematično prilagajanje in evalvacijo modela.

V D6.5 je bila odstranjena pomembna omejitev iz prejšnjih razvojnih faz: uporabniku v navodilu ni več treba vnaprej navesti FQCN oziroma polno kvalificiranega imena modula. Za ta namen je bil naučen no-FQCN model, ki iz opisa naloge v naravnem jeziku samodejno generira ustrezno Ansible nalogo. Rezultati kažejo, da je takšen model bistveno primernejši za realno uporabo kot prejšnji model, ki je bil močno odvisen od eksplicitno podanega modula. Dodatno so bili izvedeni eksperimenti RAG za zaznavanje ustreznega FQCN iz uporabniškega navodila, kar predstavlja pomembno razvojno smer za podporo novim modulom in posodobljeni dokumentaciji brez ponovnega učenja modela.

Končna rešitev je integrirana v Steampunk Spotter Visual Studio Code vtičnik, kar omogoča generiranje Ansible nalog neposredno v okolju, v katerem uporabniki pripravljajo IaC kodo. Sistem sprejme navodilo v naravnem jeziku, generira enega ali več kandidatov ter uporabniku ponudi predloge v pregled in sprejem. Orodje Steampunk Spotter je uporabljeno pri končnem pregledu knjižnice, ki jo sestavljajo tako generirane naloge. S tem generirani izhod ni obravnavan kot neposredno zaupanja vreden rezultat, temveč kot splošni rezultat, ki mora biti validiran in potrjen.

Dokument obravnava tudi lokalno oziroma organizacijsko nadzorovano izvajanje modela. Model je bil pretvorjen v format GGUF in kvantiziran, pri čemer je bil format Q4_K_M izbran kot najboljši kompromis med natančnostjo, zakasnitvijo, porabo pomnilnika in možnostjo izvajanja na običajni uporabniški strojni opremi brez namenske GPU pospešitve. Poleg zmogljivosti, dokument sistematično obravnava robustnost in varnost: pripravo podatkov z maskiranjem občutljivih vrednosti, validacijo generiranih nalog s Spotterjem v fazi priprave podatkov, možnost uporabe varnostnih profilov in politik skladnosti ter končno človeško presojo.

Sistem je bil preizkušen v relevantnem okolju, ki odraža dejansko uporabo IaC orodij: Visual Studio Code, Steampunk Spotter vtičnik, lokalno izvajanje modela, generiranje Ansible nalog in pregled rezultatov s strani Ansible strokovnjakov. Na tej podlagi izroček utemeljuje doseganje TRL6 in zaključuje razvojno pot RRP6 s programsko rešitvijo, ki je tehnično validirana, uporabniško integrirana in praktično uporabna.

1 Uvod

1.1 Namen

Izroček D6.5 predstavlja zaključno fazo dela v RRP6 – VeMo-IaC. Namen dokumenta je opisati končno programsko rešitev, ki uporablja jezikovne tehnologije za generiranje infrastrukturne kode v obliki Ansible nalog. Dokument ne opisuje več samo modela ali posameznega eksperimenta, temveč celotno uporabniško izkušnjo: od vnosa uporabniškega navodila v Visual Studio Code, preko generiranja kandidatov za Ansible naloge, do uporabniške potrditve.

D6.5 zato povezuje rezultate predhodnih izročkov D6.1 - D6.4 z zaključno programsko integracijo. V dokumentu so opisani uporabljeni podatki in model, odstranitev zahteve po eksplicitnem FQCN v navodilu (promptu), eksperimenti z RAG za zaznavanje modulov, integracija v Steampunk Spotter vtičnik v Visual Studio Code (VSCode), kvantizacija za lokalno (on-premise) izvajanje ter sistematična obravnava zmogljivosti, robustnosti, varnosti in doseganja TRL6.

V PoVeJMo je RRP6 usmerjen v uporabo velikih jezikovnih modelov za področje Infrastructure-as-Code (IaC). V RRP6 se je razvila napredna aplikacija, ki z uporabo za generiranje kode in dokumentacije prilagojenega velikega generativnega jezikovnega modela pohitri in izboljša kakovost generirane kode za opis računalniške infrastrukture (IaC).

Pri tem mora rešitev temeljiti na računsko učinkovitih modelih, dodatnem domenskem znanju s področja računalniške infrastrukture ter specializirani ukazni učni množici za infrastrukturo kot kodo.

Dosedanji razvoj (razvoj pred izročkom D6.5) je dokumentiran v štirih predhodnih izročkih, pri čemer je vsak obravnaval ločeno fazo cevovoda: izbor podatkov, začetno učenje in primerjalno vrednotenje modelov, iterativno izboljševanje podatkov ter fino prilagajanje modela z evalvacijo. Ta uvod povzema ključne prispevke in spoznanja iz vseh štirih faz ter s tem podaja potreben kontekst za končni izroček, opisan v tem dokumentu.

Izbor podatkov in ocenjevanje metod (D6.1): Postavili smo temelje celotnega RRP6, saj smo opredelili začetni pristop k izboru podatkov, izbiri osnovnega modela in evalvaciji generirane infrastrukturne kode. V tej fazi so bili določeni štirje osnovni kriteriji za izbor učnih podatkov: kakovost kode, priljubljenost oziroma sprejetost v skupnosti, velikost učne množice ter licenčna skladnost. Namen teh kriterijev je bil zagotoviti, da bodo podatki dovolj kakovostni, raznoliki, pravno uporabni in primerni za učenje modela, ki generira Ansible naloge iz naravnega jezika.

Pripravljena sta bila dva začetna vira podatkov. Prvi vir je bila kakovostnejša množica Steampunk, zgrajena iz preverjenih virov na platformi Ansible Galaxy. Obsega približno 37.000 učnih primerov iz 561 zbirk. Pri pripravi te množice so bili podatki varnostno in vsebinsko obdelani: visoko-entropične vrednosti, kot so gesla, API ključi in žetoni, so bile maskirane; celotni playbooki (zbirka večjega števila Ansible nalog za upravljanje nekega vira) so bili razdeljeni na posamezne Ansible naloge; preverjeni sta bili pravilnost YAML sintakse

in ustrezna dolžina nalog; izločene pa so bile tudi naloge, ki so bile prekratke, vsebinsko nerelevantne ali daljše od približno 600 znakov. Poleg tega so bile ohranjene predvsem naloge iz najnovejših različic zbirk, da bi učna množica čim bolje odražala aktualne prakse.

Drugi vir je bila širša množica GitHub, ki je vsebovala približno 41.000 primerov iz javnih repozitorijev. Ta množica je bila pomembna zaradi večjega obsega in širše pokritosti, vendar se je že v začetnih analizah pokazalo, da je njena kakovost bistveno bolj neenakomerno porazdeljena. Pogoste težave so bile uporaba krajših imen modulov namesto polno kvalificiranih imen, slabo opisane ali prekratke naloge ter nekonsistentne prakse pisanja kode. Zato je bila ta množica obravnavana kot potencialno uporaben, vendar zahtevnejši vir, ki potrebuje dodatno čiščenje in preverjanje pred uporabo v učenju.

Vzporedno z izborom podatkov smo v izročku D6.1 obravnavali tudi izbor primernih osnovnih modelov. Primerjani so bili modeli StarCoder2-3b, CodeLlama-7b in CodeGen-350M. Vrednotenje je upoštevalo računsko zahtevnost, kompleksnost modela, predhodno znanje o programiranju in Ansible kodi, rezultate na standardnih testih za generiranje kode, kot so HumanEval, MBPP in MultiPL-E, licenčne pogoje ter varnostne vidike. CodeLlama-7b je izkazal najvišjo natančnost na standardnih testih generiranja kode, medtem ko je StarCoder2-3b predstavljal ugoden kompromis med hitrostjo, velikostjo in kakovostjo rezultatov.

V D6.1 smo uvedli tudi osnovno evalvacijsko logiko, ki je postala izhodišče za nadaljnje izročke. Model mora iz naravnega opisa (navodila) generirati Ansible nalogo, rezultat pa se nato preveri s Steampunk Spotterjem, ki zazna napake, opozorila in skladnost z dobrimi praksami. Na tej podlagi se kakovost izhoda dodatno oceni tudi z jezikovnim modelom, pri čemer se upošteva tako Spotterjev rezultat kot semantična ustreznost generirane naloge.

Za začetno preverjanje je bila pripravljena tudi manjša sintetična evalvacijska množica tridesetih nalog, razdeljenih na lahke, srednje zahtevne in zahtevne primere. S tem je bil Spotter že v prvi fazi postavljen kot osrednji element preverjanja pravilnosti, robustnosti in uporabnosti generirane IaC kode v fazi validacije modela.

Prva programska različica in izbor modela (D6.2): Izvedli smo prvo serijo eksperimentov prilagajanja (fine-tuning) in pripravili prvo programsko različico rešitve VeMo-IaC. Eksperimenti so temeljili na podatkovnih množicah in modelih, opredeljenih v D6.1, njihov namen pa je bil preveriti, kateri osnovni model je najprimernejši za generiranje Ansible nalog iz naravnega jezika ter kakšen vpliv ima kakovost učnih podatkov na končni rezultat.

V tej fazi so bile preizkušene štiri konfiguracije modelov: CodeGen-350M z množico Steampunk, StarCoder2-3B z množico Steampunk, StarCoder2-3B z razširjeno učno množico, ki je poleg Steampunka vključevala tudi GitHub podatke, ter CodeLlama-7B z množico Steampunk. Vsi modeli so bili učeni z nadzorovanim učenjem za sledenje navodilom, pri čemer je bila za učinkovitejše prilagajanje uporabljena metoda LoRA (Low-Rank Adaptation) skupaj s 4-bitno kvantizacijo, ki zmanjšuje pomnilniške zahteve pri učenju.

Rezultati so potrdili začetna pričakovanja: CodeLlama-7B je dosegel najboljše vrednosti po ključnih metrikah: najnižjo vrednost evalvacijske izgube, najnižjo perpleksnost in najvišjo oceno na testni množici. Dosegel je evalvacijsko izgubo 0,39, perpleksnost 1,49 in rezultat

7,48 na testni množici. CodeGen-350M je bil računsko najlažji in hitrejši, vendar je bil po kakovosti generirane kode bistveno slabši. StarCoder2-3B je predstavljal uporaben kompromis med velikostjo, hitrostjo in kakovostjo, vendar ni dosegel natančnosti modela CodeLlama-7B. Analiza gradientov je dodatno pokazala, da je bilo učenje pri StarCoder2-3B manj stabilno kot pri CodeLlama-7B.

Pomembna ugotovitev poročana v D6.2 je bila, da dodajanje večje količine podatkov ne vodijo nujno v boljši model. Razširitev učne množice StarCoder2-3B z GitHub podatki je rezultat poslabšala, kar je potrdilo, da je kakovost podatkov pomembnejša od njihove količine. Slabše opisane naloge, uporaba krajših imen modulov in nekonsistentne prakse v GitHub podatkih so negativno vplivale na sposobnost modela za natančno sledenje navodilom in generiranje pravilne IaC kode.

Sklep v D6.2 je bil, da nadaljnji razvoj ne sme temeljiti na nekritičnem povečevanju učne množice, temveč na sistematičnem izboljševanju kakovosti podatkov, domenski skladnosti primerov in izbiri modela z dovolj dobro sposobnostjo sledenja navodilom. CodeLlama-7B se je v tej fazi izkazal kot najprimernejša osnova za nadaljnji razvoj, medtem ko so rezultati z GitHub podatki neposredno usmerili naslednje faze v bolj premišljeno čiščenje, bogatenje in strukturiranje podatkovnih množic. Ta ugotovitev je neposredno vodila v podatkovne iteracije D6.3 in kasnejše eksperimente (fine-tuning) ter evalvacije v D6.4.

Iterativno izboljševanje podatkov (D6.3): Fokus razvoja smo premaknili z izbire osnovnega modela na sistematično izboljševanje učnih podatkov. Na podlagi ugotovitev iz D6.2 je bil za nadaljnji razvoj kot osnovni model izbran CodeLlama-7B-Instruct, pri čemer je bila prednost dana natančnosti in sposobnosti sledenja navodilom pred hitrostjo inference.

Prvotni kriteriji za izbor podatkov, določeni v D6.1, so bili v D6.3 ponovno ovrednoteni; Ugotovili smo, da jih je potrebno razširiti. Poleg kakovosti kode, priljubljenosti oziroma sprejetosti v skupnosti, velikosti učne množice in licenčne skladnosti sta bila dodana še dva ključna kriterija: raznolikost nalog in kakovost navodil (promptov). Raznolikost pomeni zadostno število različnih Ansible nalog, modulov in kombinacij parametrov znotraj posameznih modulov. Kakovost navodil pa pomeni jasne, dovolj informativne in nedvoumne opise nalog, ki modelu omogočajo boljše učenje povezave med naravnim opisom, ustreznim modulom, parametri in vrednostmi parametrov. Ta razširitev kriterijev je izhajala iz spoznanja, da za uspešnost modela ni odločilna samo količina podatkov, temveč predvsem njihova širina, globina in opisna kakovost.

V D6.3 so bile zgrajene in ovrednotene zaporedne različice podatkovnih množic, pri čemer je bila vsaka nova različica zasnovana tako, da odpravi pomanjkljivosti prejšnje. Razvoj se je začel z osnovno množico Steampunk, ki je služila kot izhodišče. Sledile so različice z nadvzorčenjem (super sampling), kombinacije s sintetičnimi podatki in podatki iz enotnih testov (unit test), nato pa še različice z obogatenimi opisi, naprednejšim oziroma tipiziranim maskiranjem vrednosti parametrov, ročnim pregledom Ansible strokovnjakov ter izpopolnjenimi GitHub podatki, ki so bili dodatno preverjeni s Spotterjem. Pri maskiranju se vrednosti niso več nadomeščale zgolj z generičnim označevalnikom, temveč z bolj informativnimi oznakami, na primer z oznako za geslo, pot, uporabniško ime ali drugo vrsto

parametra. S tem se je ohranilo več semantičnih informacij, hkrati pa so občutljive vrednosti ostale zakrite.

Končna različica podatkovne množice je združila več pristopov: osnovne podatke Steampunk, sintetične podatke, podatke iz enotnih testov, obogatene opise nalog, ponovno in bolj natančno maskiranje vrednosti, ročni pregled ter prečiščene GitHub podatke. Obsegala je približno 50.000 učnih primerov in pokrivala 803 module. Takšna sestava je omogočila bistveno boljšo pokritost modulov, večjo raznolikost kombinacij parametrov in boljše učenje sledenja navodilom.

Ključni sklep D6.3 je bil jasen: zgolj povečevanje količine učnih podatkov ne izboljša nujno kakovosti modela. Najboljši rezultati nastanejo šele s kombinacijo dovolj velike, dovolj raznolike in kakovostno opisane učne množice. Najboljša različica podatkovne množice je dosegla nič napak in eno opozorilo pri preverjanju z orodjem Spotter. S tem se je pokazalo, da je iterativna, kakovostno vodena gradnja podatkovne množice najpomembnejši dejavnik izboljšanja celotnega razvojnega procesa.

Izboljšano učenje in validacija modela (D6.4): V D6.4 smo dokumentirali eksperimente izvedene na podatkovnih množicah, zgrajenih v D6.3. Kot osnovni model je bil uporabljen CodeLlama-7B-Instruct, saj se je v predhodnih fazah izkazal kot najprimernejša osnova za nadaljnji razvoj z vidika natančnosti in sposobnosti sledenja navodilom.

V primerjavi z D6.2 je bil postopek učenja dodatno izboljšan (točni parametri so razvidni v izročku), hkrati pa je bila odstranjena 4-bitna kvantizacija, saj je bil v tej fazi cilj povečati natančnost modela, tudi za ceno večje porabe računskih virov.

Rezultati so pokazali jasno in postopno izboljševanje kakovosti modela. Poleg izboljšanja avtomatskih metrik so kasnejše različice modela pokazale tudi boljše semantično razumevanje navodil, pravilnejšo izbiro parametrov in zanesljivejše umeščanje več vrednosti parametrov v isto Ansible nalogo.

Evalvacijski okvir je bil v D6.4 dodatno okrepljen. Poleg avtomatskega preverjanja s Steampunk Spotterjem in dodatnega ocenjevanja z jezikovnim modelom je bila vključena tudi človeška presoja Ansible strokovnjakov. Strokovnjaki so potrdili, da kasnejše različice modela bistveno bolje sledijo zahtevnejšim navodilom in da je najboljša različica dovolj kakovostna za interno uporabo. S tem je bil model potrjen ne le na ravni avtomatskih metrik, temveč tudi z vidika praktične uporabnosti pri generiranju IaC kode.

Ključni rezultat D6.4 je bil zato dokaz, da kombinacija kakovostno pripravljene podatkovne množice iz D6.3 in skrbno nastavljene LoRA prilagoditve omogoča razvoj modela, ki na evalvacijski množici dosega skoraj brezhibne rezultate. D6.4 je s tem potrdil tehnično izvedljivost pristopa in pokazal, da lahko specializiran model za generiranje Ansible nalog doseže visoko stopnjo pravilnosti, robustnosti in uporabnosti.

Hkrati pa je D6.4 razkril tudi pomembno omejitev za realno uporabo. Model je bil preizkušen predvsem v režimu, kjer je uporabnik v promptu eksplicitno podal FQCN oziroma polno kvalificirano ime modula, ki ga mora model uporabiti. Takšen režim je uporaben za nadzorovano evalvacijo, vendar ne odraža povsem realnega scenarija uporabe. Končni uporabnik pogosto ne ve, kateri Ansible modul oziroma kateri FQCN je pravilen za določeno nalogo. Zato je ta omejitev neposredno odprla naslednji razvojni problem: sistem mora poleg generiranja same Ansible naloge znati tudi izbrati ali predlagati ustrezen modul na podlagi naravnega opisa uporabnikove zahteve.

1.2 Prehod na končno različico in produktizacijo modela

Razvoj v RRP6 projekta Povejmo je sledil jasnemu zaporedju: od postavitve temeljev, tj. kriterijev za izbor podatkov, izbire osnovnega modela in evalvacijske metodologije, prek iterativnega izboljševanja učnih množic in učnih nastavitev, do validiranega modela, ki je dosegel skoraj brezhibne rezultate na evalvacijskem naboru in je bil hkrati dobro ocenjen s strani Ansible strokovnjakov.

Ključna spoznanja teh faz so bila, da je kakovost podatkov pomembnejša od njihove količine, da je za specializirane domene nujna iterativna gradnja podatkovnih množic s strokovnim pregledom ter da zanesljiva presoja modela zahteva večnivojsko evalvacijo, ki združuje avtomatsko analizo, ocenjevanje z jezikovnim modelom in človeško presojo domenskih strokovnjakov.

Uporabniški problem, odprt po zadnjih raziskovalnih fazah, poročanih v D6.4: generiranje Ansible nalog brez zahteve, da uporabnik vnaprej pozna FQCN. Eksplicitno podajanje FQCN navodilu je primerno za natančno evalvacijo modela, vendar ne odraža v celoti realnega uporabniškega scenarija, saj končni uporabnik pogosto ne ve, kateri Ansible modul mora uporabiti za določeno nalogo.

D6.5 zato predstavlja prehod iz eksperimenta, osredotočenega na model, v končno programsko rešitev. Osredotoča se na odstranitev zahteve, da uporabnik v promptu navede FQCN modula ter na preverjanje, ali je mogoče ustrezen modul zaznati ali predlagati samodejno iz navodila oz. opisa. V ta namen dokument obravnava učenje in evalvacijo modela na realističnejšem uporabniškem scenariju ter RAG eksperimente za avtomatsko zaznavanje ustreznega FQCN iz navodila.

Poleg tega opisujemo integracijo generiranja Ansible nalog v Steampunk Spotter Visual Studio Code vtičnik, s čimer se rezultat približa dejanskemu načinu dela uporabnikov IaC orodij. Rešitev vključuje tudi implementacijo modela preko API-jev, ki so v uporabi v podjetjih in tudi naslavlja vprašanje, kakšna infrastruktura je na voljo pri naročniku (CPU, GPU) - obravnavamo tudi alternativo s kvantizirano CPU izvedbo.

Pomemben del predstavlja pretvorba modela v GGUF format in izbor kvantizacije Q4_K_M kot praktičnega kompromisa med kakovostjo, hitrostjo, velikostjo modela in izvedljivostjo on-premises izvajanja.

Zelo pomembno je poudariti, da uporabljamo Spotter kot validacijski in varnostni sloj po generiranju v fazi evalvacije modela. To pomeni, da generirana Ansible naloga ni obravnavana kot neposreden in zaupanja vreden predlog, ki ga ponudimo temveč zgolj kot predlog, ki ga je treba preveriti z obstoječimi pravili, opozorili in mehanizmi za zaznavanje napak. S tem realno ocenjujemo kvaliteto modela, saj bo pri realni uporabil končni rezultat vedno preverjen tudi s strani orodja Spotter.

S tem se nadaljuje osnovno načelo celotnega RRP6: generiranje kode mora biti povezano z validacijo, saj je pri IaC pravilnost, robustnost in varnost generirane kode neposredno povezana s stabilnostjo računalniške infrastrukture.

V tem končnem poročilu (D6.5) se torej osredotočamo predvsem na to, kako zagotoviti uporabnost našega dela – realističen uporabniški scenarij (odstranitev zahteve po FQCN v navodilu, povezovanje z obstoječimi artefakti samega uporabnika - eksperimentov z RAG).

Osredtočili smo se tudi na tehnično integracijo. Integrirali smo generiranje Ansible nalog v Steampunk Spotter vtičnik za Visual Studio Code (VSCode). Pregledali in preizprašali smo trenutne uporabnike, kakšna infrastruktura (tako programska kot tudi strojna) je na voljo in temu prilagodili dejansko ponudbo modela. Uporabili smo vLLM, llama.cpp ter Ollama. Naredili smo GGUF, kvantizirali model (Q4_K_M) ter omogočili tako CPU kot GPU uporabo.

V nadaljevanju se osredotočimo na zrelost sistema, potem pa opišemo arhitekturo sistema ter nato preidemo na dodatne eksperimente, s katerimi smo razvili sam sistem.

Na tem mestu opišimo zrelost našega sistema. Definicija doseganja TRL 6 je opisana kot: »*Tehnologija je bila predstavljena v ustreznem relevantnem okolju*«. To pomeni, da je tehnologija predstavljena kot prototipni sistem v okolju, ki odraža realne pogoje delovanja. Na stopnji TRL 6 je prototip po zmogljivosti, zanesljivosti in obliki podoben načrtovanemu operativnemu sistemu.

Doseganje TRL6 dokazujemo z integriranim prototipom, ki deluje v relevantnem razvojnem okolju: uporabnik uporablja Visual Studio Code in vtičnik Steampunk Spotter, navodilo vnese v naravnem jeziku neposredno v urejevalniku, sistem prek API-ja pokliče prilagojeni model, generira enega ali več kandidatov Ansible nalog in rezultat vrne uporabniku v pregled. Rešitev je bila preizkušena z Ansible strokovnjaki in v okolju, primerljivem z dejansko prakso priprave IaC kode. S tem naše delo dokazuje uporabnost modela in delovanje celotne programske verige, kar ustreza zahtevi TRL6 po predstavitvi tehnologije v relevantnem okolju.

Razvijalci in IaC eksperti uporabljajo sistem – saj ne razvijajo zgolj orodja Steampunk Spotter temveč delujejo tudi v sklopu komercialnih pogodb in projektov, kjer razvijajo IaC za določene naročnike. S tem je potrjujemo sposobnost demonstratorja, da v okolju, primerljivem z dejansko razvojno prakso uporabnikov IaC, generira uporabne Ansible naloge.

2 Arhitektura sistema

Končna rešitev, rezultat RRP6, je bila validirana v relevantnem okolju, ki vključuje uporabo Visual Studio Code urejevalnika, orodja Steampunk Spotter, modela, ki je bil nameščen lokalno (kot sledi iz uporabniških zahtev) ter generiranja Ansible nalog (taskov) glede na navodila v naravnem jeziku (prompt).

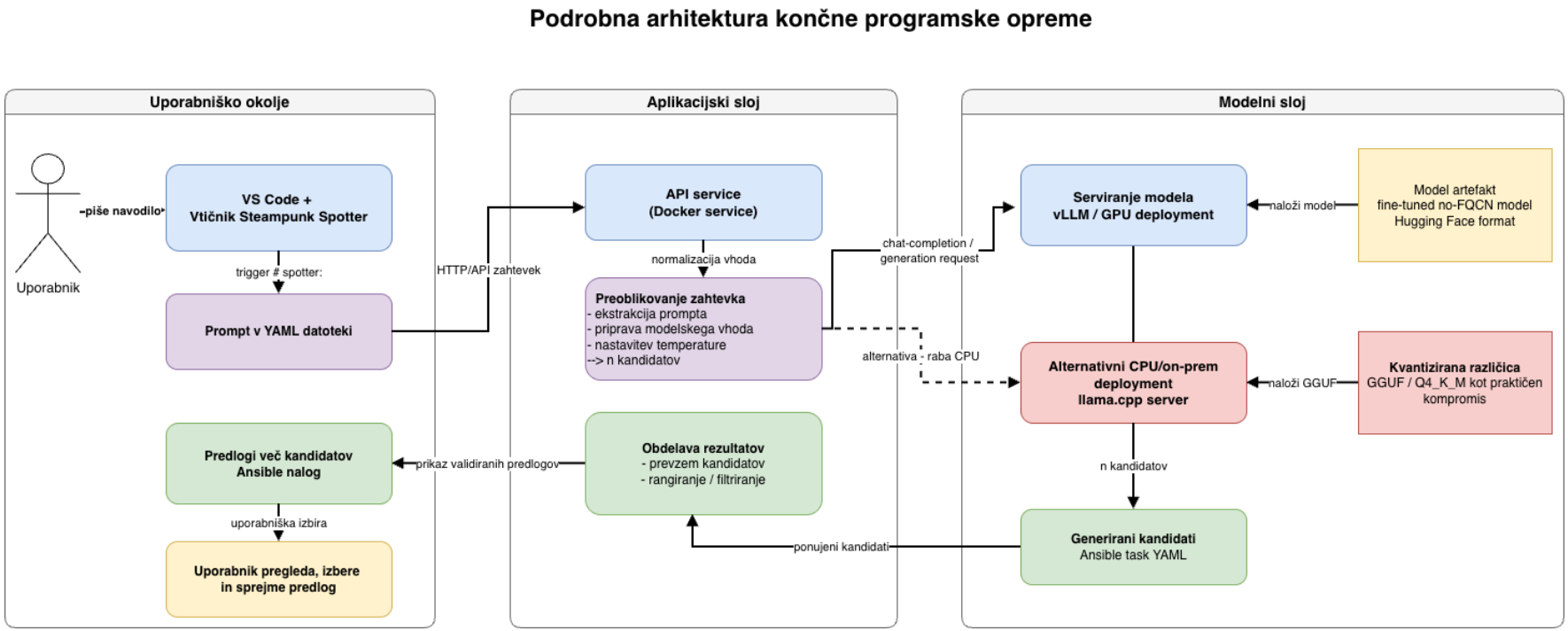
Slika 1 prikazuje končno arhitekturo celotne programske opreme, oz. delovanje produkta, ki je vpet v obstoječa orodja in omejen z uporabniški zahtevami. Možni so različni načini postavitve modela, odvisni od strojnih zmogljivosti ter obstoječega programskega okolja.

Končna programska rešitev je sestavljena iz več povezanih komponent (opisanih v Tabela 1 in prikazanih v Slika 1): uporabniškega vmesnika v Visual Studio Code (vtičnik), programskega vmesnika (API) za sprejem in obdelavo zahtevkov ter samega modela (generiranje Ansible nalog) Uporabnik v VSCode vnese navodilo v naravnem jeziku, sistem generira enega ali več kandidatov Ansible nalog, ki ponudijo uporabniku v izbiro in sprejem.

Tabela 1: Posamezne komponente rešitve in njihova vloga.

Komponenta	Vloga	Tehnologija / izvedba	Prispevek k D6.5
Vtičnik VS Code	Uporabniški vmesnik za vnos prompta in prikaz predlogov.	Steampunk Spotter extension	Končna uporabniška integracija.
API service	Sprejem zahtev, normalizacija vhoda, priprava navodila in klic modela.	HTTP API / Docker service	Povezovalni sloj med odjemalcem in modelom.
Serviranje modela	Izvajanje fine-tuned modela in generiranje kandidatnih nalog.	vLLM za GPU režim, llama.cpp za CPU režim	Generiranje Ansible taskov.
Model	Fine-tuned no-FQCN model.	Hugging Face format / GGUF	Jedro jezikovne tehnologije.
Kvantizirana različica	CPU/on-premise izvedba za okolja brez GPU.	GGUF Q4_K_M	Zmanjšana poraba pomnilnika in praktična lokalna raba.
Uporabnik	Pregleda, izbira in potrdi predlog.	VS Code workflow	Human-in-the-loop kontrola.

Zelo pomembno je poudariti, da smo pri učenju v fazi evalvacije vsak generirani izhod preverili preko orodja Spotter in tako ocenjevali kvaliteto modela. S tem smo posredno povečali robustnost izbranih rešitev in omogočili uporabniku, da izbira glede na celostno in varnostno kvaliteto rešitve. Uporaba orodja Spotter nikakor ne more zagotoviti 100% pravilnosti izbranega kandidata, zagotovi pa, da so izbrani kandidati tehnično pravilni.



Slika 1: Podrobna arhitektura končne programske opreme

3 Podatki in model

Kot je bilo pojasnjeno v uvodu ter podrobneje opisano v izročkih D6.3 in D6.4, postopki zbiranja podatkov, obdelave podatkov in prilagajanja modela niso osrednji predmet tega dokumenta. V tej fazi smo izvedli eno dodatno učno iteracijo, pri čemer smo uporabili enako podatkovno osnovo kot v zadnjem eksperimentu D6.3, torej kombinacijo Steampunk množice, obogatenih opisov, sintetičnih podatkov, podatkov iz enotnih testov, ponovnega maskiranja in izpopolnjenih GitHub podatkov.

Ključna sprememba v tej iteraciji je bila, da smo iz vsakega navodila odstranili FQCN (polno kvalificirano ime modula). Posledično se model ni več mogel opreti na eksplicitno podano ime modula, temveč se je moral naučiti povezave med opisom naloge v naravnem jeziku in ustrezno izbiro modula. S tem smo se približali realnejšemu uporabniškemu scenariju, v katerem uporabnik pogosto ne ve, kateri Ansible modul oziroma FQCN mora uporabiti.

Primer razlike med prejšnjim in novim načinom podajanja navodil je naslednji:

Prej:

```
Create a directory at {{ mailhog_install_dir }}, set owner and group to root, and assign permissions of 493. Use the module: ansible.builtin.file.
```

Zdaj:

```
Create a directory at {{ mailhog_install_dir }}, set owner and group to root, and assign permissions of 493.
```

Poleg spremembe navodil smo prilagodili tudi nekatere učne parametre. Povečali smo velikost paketov (batch size), akumulacijo gradientov (gradient accumulation) in število epoh (number of epochs). Število epoh smo povečali z 2 na 3. Namen teh sprememb je bil omogočiti stabilnejše učenje v zahtevnejšem režimu, kjer model nima več neposredno podanega modula in mora del naloge rešiti sam.

Za evalvacijo smo pripravili novo evalvacijsko množico, sestavljeno iz primerov iz evalvacijskih množic, uporabljenih v D6.3 in D6.4, sintetično ustvarjenih primerov ter primerov, ki so jih ročno pripravili naši Ansible strokovnjaki. Skupaj je evalvacijska množica vsebovala 42 primerov.

Razlog za vključitev človeško napisanih navodil je bil, da imajo Ansible strokovnjaki drugačen slog pisanja navodil kot sintetično generirani primeri. Človeško napisana navodila so praviloma krajša, bolj neposredna in vsebujejo manj eksplicitnih informacij.

Primer razlike med človeško napisanim in sintetičnim navodilom:

Človeško napisano navodilo:

```
Remove folder /opt/spotter
```

Sintetično navodilo:

```
Set the ownership of the /etc/nginx/conf.d/ directory to nginx user and group,
ensure it exists as a directory, and set its permissions to 0750.
```

Rezultati (Tabela 2) jasno pokažejo vpliv odstranitve FQCN iz navodil. Kot pričakovano, model iz D6.4, testiran v režimu, kjer je FQCN še vedno eksplicitno podan v navodilu, doseže najboljše rezultate: 0 napak, indeks resnosti (severity score[1]) 2 in GPT oceno (GPT score) 8,05. To potrjuje visoko kakovost prejšnje iteracije modela, kadar je uporabljena v pogojih, za katere je bila naučena.

Ko pa istemu modelu iz D6.4 odstranimo FQCN iz navodila, se njegova zmogljivost močno poslabša. Število napak se poveča z 0 na 13, indeks resnosti naraste z 2 na 132, GPT ocena pa pade na 6,0. To potrjuje, da je bil model iz D6.4 močno odvisen od eksplicitno podanega modula in da se njegova kakovost ne znižuje postopno, temveč izrazito, ko ta informacija ni več na voljo. Tak model zato ni primeren za realen uporabniški scenarij, v katerem uporabnik v prosti obliki zapiše navodilo v naravnem jeziku in ne poda FQCN.

Novo naučeni no-FQCN model se obnaša bistveno drugače; test v mešanem režimu, kjer je FQCN še vedno prisoten v navodilu, ustvari samo 2 napaki, doseže indeks resnosti 21 in GPT oceno 7,74. To kaže, da model ni izgubil sposobnosti uporabiti ime modula, kadar je to slučajno podano. Pomembnejši pa je rezultat v režimu, za katerega je bil model dejansko namenjen, torej brez FQCN v navodilu. V tem primeru model ustvari 3 napake, 1 opozorilo, indeks resnosti 31 in GPT oceno 7,59. Ti rezultati so slabši od zgornje meje, ki jo predstavlja model iz D6.4 v idealnem režimu s podanim FQCN, vendar so več kot štirikrat boljši od rezultatov modela D6.4 v enakih realističnih pogojih brez FQCN (indeks resnosti 31 vs. 132).

Na podlagi teh rezultatov smo za končno programsko rešitev izbrali konfiguracijo no-FQCN modela brez FQCN v navodilu (zadnja vrstica v Tabela 2). Ta izbira pomeni zavestno odpoved absolutni natančnosti v primerjavi z idealnim režimom iz D6.4, vendar omogoča bistveno boljšo uporabniško izkušnjo in robustnejše delovanje v realnem scenariju, kjer uporabnik ne navede modula. Preostala razlika med to konfiguracijo in zgornjo mejo iz D6.4 je bila razlog za dodatno raziskavo pristopa RAG za eksplicitno zaznavanje modulov, ki je predstavljena v naslednjem poglavju.

Tabela 2: Primerjava rezultatov modela iz D6.4 in novo naučenega no-FQCN modela. Oba modela sta bila testirana v dveh režimih: z eksplicitno podanim FQCN v navodilu in brez FQCN v navodilu.

	Napake	**Opozorila**	**Indeks resnosti**	**GPT ocena**
Model iz D6.4 (FQCN model) + FQCN v navodilu	0	2	2	8.05
Model iz D6.4 (FQCN model) + BREZ FQCN v navodilu	13	2	132	6
NO-FQCN model + FQCN v navodilu	2	1	21	7.74
NO-FQCN model + BREZ FQCN v navodilu	3	1	31	7.59

[1] Metrike so opisane v izročkih D6.3, D6.4 in prejšnjih izročkih.

4 Eksperimenti RAG za zaznavanje FQCN

Kot je pokazalo prejšnje poglavje, no-FQCN model ostane razmeroma robusten tudi takrat, ko uporabnik v navodilu ne poda imena modula, vendar še vedno ostaja razlika do zgornje meje, ki jo predstavlja model iz D6.4 v režimu z eksplicitno podanim FQCN. Naravno vprašanje je ali je mogoče to razliko zmanjšati tako, da sistem ime modula ponovno doda samodejno, vendar brez zahteve, da bi moral uporabnik poznati ali vpisati pravilen FQCN.

Pristop RAG (Retrieval-Augmented Generation) omogoča prav takšen način delovanja. Na podlagi navodila v naravnem jeziku lahko iskalna komponenta poskusi določiti najverjetnejši FQCN in ga nato samodejno vključi v navodilo pred generiranjem Ansible naloge.

V tem poglavju ocenjujemo več konfiguracij RAG [1] in jih primerjamo s prilagojenim no-FQCN modelom. Namen eksperimentov je bil ugotoviti, kateri pristop je najbolj sposoben napovedati pravilen FQCN neposredno iz uporabniškega navodila. Evalvacijska množica je bila enaka kot v prejšnjem poglavju in je vsebovala 42 primerov.

Pri eksperimentih smo zaradi razmeroma majhnega števila modulov uporabili FAISS[2] z osnovnim iskanjem. Učna množica je vsebovala približno 800 modulov, zato kompleksnejši iskalni pristopi v tej fazi niso bili nujni – zadošča relativno enostaven pristop z osnovnim vektorskim iskanjem: FAISS knjižnica za učinkovito iskanje podobnih vektorskih predstavitev. Vsak vnos v bazi znanja smo najprej z modeli vgrajevanja (embedding model) pretvorili v t.i. embedding, nato pa ga indeksirali v FAISS. Ob uporabniškem navodilu sistem preko modela vgrajevanja sam izračuna t.i. embedding navodila in s pomočjo FAISS poišče najbližje vnose, iz katerih izpelje najverjetnejše kandidate za FQCN.

Preizkusili smo različne kombinacije treh skupin komponent: modele vgrajevanja (embedding model), modele za ponovno razvrščanje (re-ranking) oziroma križne kodirnike (crossencoder) in različne oblike baz znanja.

Pri pristopu RAG smo uporabili naslednje modele:

- Modeli vgrajevanja: Qwen3-Embedding-0.6B [2], multilingual-e5-large-instruct, bge-base-en-v1.5
- Modeli za re-ranking/crossencoder: ms-marco-MiniLM-L6-v2, ms-marco-TinyBERT-L2-v2
- Baze znanja (knowledge base):
    - Summary: Povzetki dokumentacije posameznih modulov
    - examples_5, examples_10, examples_30: Zbirke 5, 10 ali 30 YAML primerov nalog za posamezen modul
    - yaml_task: Prečiščena množica unikatnih YAML nalog, kjer ima lahko isti FQCN več vnosov

Spremljali smo tri metrike:

- Top-1 Rate: Delež primerov, pri katerih je bil pravilen FQCN izbran kot prvi rezultat

[2] https://faiss.ai Facebook AI Similarity Search

- Top-10 Rate: Delež primerov, pri katerih se je pravilen FQCN pojavil med prvimi desetimi predlogi
- Top-1 After Reranking: Delež primerov, pri katerih je bil po ponovnem razvrščanju pravilen FQCN izbran kot prvi rezultat

4.1 Baze znanja

Za eksperimente RAG smo pripravili več oblik baz znanja. Namen teh različic je bil preveriti, ali model lažje prepozna ustrezen modul iz opisne dokumentacije, iz konkretnih primerov uporabe ali iz večjega števila posameznih Ansible nalog v YAML zapisu.

4.1.1 Povzetki dokumentacije modulov

Pri prvi obliki baze znanja smo za vsak FQCN uporabili dokumentacijo modula, ki vključuje opise parametrov, primere uporabe in dodatna pojasnila. To dokumentacijo smo posredovali modelu GPT-4o-mini, ki je za vsak modul pripravil razumljiv opis. Namen takšnega pristopa je bil ustvariti semantično bogatejši opis posameznega modula, ki bi ga embedding model lažje povezal z uporabniškim navodilom.

Primer povzetka za modul ansible.builtin.file:

```
# Description:

The `ansible.builtin.file` module is designed to manage the attributes of files, directories, and symlinks, including their targets. This module can also be used to remove files, symlinks, or directories. It shares many options with other modules, including `ansible.builtin.copy`, `ansible.builtin.template`, and `ansible.builtin.assemble`. For Windows targets, the `ansible.windows.win_file` module should be used instead.

**Key Functionalities:**

- Change ownership, group, and permissions of files and directories.
- Create symbolic and hard links.
- Touch files to update access and modification times.
- Recursively set file attributes on directory contents.
- Remove files or directories safely.
- Update modification and access times based on specific values.
- Manage SELinux context settings.
- Control operations on file attributes using atomic operations.

**Available Options:**

- `src`: Path of the file to link to (used with `state=link` and `state=hard`).
- `mode`: Permissions of the resulting filesystem object.
- `path`: Required path to the file being managed (aliases: `dest`, `name`).
- `force`: Force the creation of links under certain conditions.
- `group`: Name of the group that should own the filesystem object.
- `owner`: Name of the user that should own the filesystem object.
- `state`: Desired state of the file (e.g., `absent`, `directory`, `file`, `hard`, `link`, `touch`).
- `follow`: Indicates whether filesystem links should be followed.
- `serole`: SELinux role part of the filesystem object context.
```

```
- `setype`: SELinux type part of the filesystem object context.
- `seuser`: SELinux user part of the filesystem object context.
- `recurse`: Recursively set specified file attributes on directory contents.
- `selevel`: SELinux level part of the filesystem object context.
- `attributes`: Attributes the resulting filesystem object should have.
- `access_time`: Time to set the file's access time.
- `unsafe_writes`: Control the use of atomic operations to prevent data
corruption.
- `modification_time`: Time to set the file's modification time.
- `access_time_format`: Format used for access_time.
- `modification_time_format`: Format used for modification_time.
```

4.1.2 Zbirke primerov nalog

Pri drugi obliki baze znanja smo uporabili učno množico in za vsak FQCN izbrali 5, 10 ali 30 YAML primerov. Te primere smo združili v en dokument za posamezen modul. Namen tega pristopa je bil preveriti, ali lahko iskalni sistem pravilneje zazna FQCN na podlagi konkretnih primerov uporabe modula in ne zgolj na podlagi opisne dokumentacije.

````
# YAML task examples

## Example 1

```yaml

- name: task name
  ansible.builtin.file:
    name: '{{nginx_conf_dir}}/conf.d/{{ item }}'
    state: absent
  with_items: '{{ config_files.stdout_lines | default([]) }}'
  when: item[:-5] not in nginx_configs.keys()
  notify:
  - reload nginx

```

## Example 2

```yaml

- name: task name
  ansible.builtin.file:
    path: '{{nginx_conf_dir}}/sites-enabled/default'
    state: absent
  notify:
  - reload nginx

```

## Example 3

```yaml

- name: task name
````
````

```
  ansible.builtin.file:
    path: '{{nginx_conf_dir}}/conf.d/default.conf'
    state: absent
  when: '''default'' not in nginx_configs.keys()

    '
  notify:
  - reload nginx

```

Example 4

```yaml

- name: task name
  ansible.builtin.file:
    state: link
    src: '{{ nginx_conf_dir }}/sites-available/{{ item.key }}.conf'
    dest: '{{ nginx_conf_dir }}/sites-enabled/{{ item.key }}.conf'
  with_dict: '{{ nginx_sites }}'
  when: item.key not in nginx_remove_sites
  notify:
  - reload nginx

```

Example 5

```yaml

- name: task name
  ansible.builtin.file:
    path: /etc/supervisor/conf.d
    state: directory

```
```

### 4.1.3 Unikatne YAML naloge

Pri tretji obliki baze znanja nismo združevali več nalog v en dokument. Namesto tega smo prefiltrirali učno množico tako, da so v njej ostale samo unikatne naloge. Posledično je imel lahko vsak FQCN več ločenih vnosov v bazi znanja. Moduli, ki so bili v učni množici pogosteje zastopani, so imeli več primerov, manj pogosti moduli pa manj vnosov.

Primer vnosa za ansible.builtin.file:

```
- name: task name
  ansible.builtin.file:
    name: '{{nginx_conf_dir}}/conf.d/{{ item }}'
    state: absent
  with_items: '{{ config_files.stdout_lines | default([]) }}'
  when: item[:-5] not in nginx_configs.keys()
  notify:
  - reload nginx
```
```

```
- name: task name
  ansible.builtin.file:
    name: '{{nginx_conf_dir}}/conf.d/{{ item }}'
    state: absent
  with_items: '{{ config_files.stdout_lines | default([]) }}'
  when: item[:-5] not in nginx_configs.keys()
  notify:
  - reload nginx
```

```
- name: task name
  ansible.builtin.file:
    path: '{{nginx_conf_dir}}/conf.d/default.conf'
    state: absent
  when: '''default'' not in nginx_configs.keys()

    '
  notify:
  - reload nginx
```

4.2 Rezultati RAG eksperimentiranja

Rezultati evalvacije so prikazani v

Tabela 3. Tabela primerja prilagojeni no-FQCN model in različne konfiguracije RAG. Oznaka summary se nanaša na bazo znanja s povzetki dokumentacije modulov, examples_5, examples_10 in examples_30 na baze znanja z združenimi 5, 10 ali 30 YAML primeri za posamezen modul; yaml_task pa na bazo znanja z ločenimi unikatnimi YAML nalogami. Tabela je urejena padajoče po metriki Top-1 After Reranking, razen prve vrstice, ki prikazuje rezultat prilagojenega modela in služi kot referenčna vrednost.

Rezultati kažejo, da je prilagojeni no-FQCN model dosegel najboljši rezultat med vsemi pristopi, in sicer 85,71 % pravilno napovedanih FQCN pri prvi izbiri. Najboljša konfiguracija RAG je uporabila yaml_task kot obliko vgrajenih podatkov, Qwen3-Embedding-0.6B kot embedding model in ms-marco-MiniLM-L6-v2 kot cross-encoder. Ta konfiguracija je po ponovnem razvrščanju dosegla 78,57 % pravilnih napovedi pri prvi izbiri, kar je približno 7 odstotnih točk manj od prilagojenega modela.

Tabela 3: Rezultati eksperimentov RAG za zaznavanje FQCN. summary označuje povzetke dokumentacije modulov, examples_5, examples_10 in examples_30 označujejo baze znanja z združenimi 5, 10 ali 30 YAML primeri za posamezen modul, yaml_task pa bazo znanja z ločenimi unikatnimi YAML nalogami. Tabela je urejena padajoče po metriki Top-1 After Reranking, pri čemer prva vrstica prikazuje referenčni rezultat prilagojenega no-FQCN modela

Embedded Data	Embedding Model	Cross Encoder	Top-1 Rate	Top-10 Rate	Top-1 After Reranking
D6.5 model	/	/	0.8571	/	/
yaml_task	Qwen3-Embedding-0.6B	ms-marco-MiniLM-L6-v2	0.6190	0.9762	**0.7857**
yaml_task	Qwen3-Embedding-0.6B	ms-marco-TinyBERT-L2-v2	0.6190	0.9762	0.7381
yaml_task	bge-base-en-v1.5	ms-marco-MiniLM-L6-v2	0.6429	0.8810	0.7143
yaml_task	multilingual-e5-large-instruct	ms-marco-TinyBERT-L2-v2	0.6429	0.9286	0.6905
yaml_task	bge-base-en-v1.5	ms-marco-TinyBERT-L2-v2	0.6429	0.8810	0.6905
yaml_task	multilingual-e5-large-instruct	ms-marco-MiniLM-L6-v2	0.6429	0.9286	0.6429
examples_30	Qwen3-Embedding-0.6B	ms-marco-MiniLM-L6-v2	0.6429	**1.0000**	0.4762
examples_30	Qwen3-Embedding-0.6B	ms-marco-TinyBERT-L2-v2	0.6429	**1.0000**	0.4762
examples_10	Qwen3-Embedding-0.6B	ms-marco-MiniLM-L6-v2	0.4762	0.9048	0.4762
examples_10	Qwen3-Embedding-0.6B	ms-marco-TinyBERT-L2-v2	0.4762	0.9048	0.4762
summary	bge-base-en-v1.5	ms-marco-MiniLM-L6-v2	0.4048	0.7619	0.4524
summary	Qwen3-Embedding-0.6B	ms-marco-MiniLM-L6-v2	0.5000	0.6905	0.4286
summary	bge-base-en-v1.5	ms-marco-TinyBERT-L2-v2	0.4048	0.7619	0.4048
examples_5	Qwen3-Embedding-0.6B	ms-marco-MiniLM-L6-v2	0.3810	0.8810	0.3810
summary	Qwen3-Embedding-0.6B	ms-marco-TinyBERT-L2-v2	0.5000	0.6905	0.3810
summary	multilingual-e5-large-instruct	ms-marco-MiniLM-L6-v2	0.3810	0.6667	0.3810
examples_10	multilingual-e5-large-instruct	ms-marco-TinyBERT-L2-v2	0.2857	0.5952	0.3810
examples_30	multilingual-e5-large-instruct	ms-marco-TinyBERT-L2-v2	0.2857	0.5952	0.3810

examples_5	Qwen3-Embedding-0.6B	ms-marco-TinyBERT-L2-v2	0.3810	0.8810	0.3571
examples_5	multilingual-e5-large-instruct	ms-marco-TinyBERT-L2-v2	0.3095	0.6667	0.3571
examples_10	multilingual-e5-large-instruct	ms-marco-MiniLM-L6-v2	0.2857	0.5952	0.3571
examples_30	multilingual-e5-large-instruct	ms-marco-MiniLM-L6-v2	0.2857	0.5952	0.3571
examples_5	bge-base-en-v1.5	ms-marco-TinyBERT-L2-v2	0.2857	0.5952	0.3333
examples_5	bge-base-en-v1.5	ms-marco-MiniLM-L6-v2	0.2857	0.5952	0.3095
examples_10	bge-base-en-v1.5	ms-marco-MiniLM-L6-v2	0.2619	0.5476	0.3095
examples_10	bge-base-en-v1.5	ms-marco-TinyBERT-L2-v2	0.2619	0.5476	0.3095
examples_30	bge-base-en-v1.5	ms-marco-MiniLM-L6-v2	0.2619	0.5476	0.3095
examples_30	bge-base-en-v1.5	ms-marco-TinyBERT-L2-v2	0.2619	0.5476	0.3095
summary	multilingual-e5-large-instruct	ms-marco-TinyBERT-L2-v2	0.3810	0.6667	0.2857
examples_5	multilingual-e5-large-instruct	ms-marco-MiniLM-L6-v2	0.3095	0.6667	0.2857

Naš sklep je, da prednost modela no-FQCN izhaja iz dejstva, da je model dobro osvojil vzorce, prilagojene nalogi, v učni fazi, zato lahko sklepa na podlagi celotnega konteksta navodila. Model (in njegove zmogljivosti) ni odvisen od ločenega iskalnega koraka, ki lahko vnese šum ali spregleda relevantne informacije. Takšen pristop zahteva manj zmogljivosti pri uporabniku in je enostavnejši za postavitev, saj ne zahteva dodatne programske opreme (RAG cevovoda, vektorske baze in dodatnega koraka ponovnega razvrščanja).

Pristop RAG je, kljub slabšim rezultatom, pomembna razvojna smer. Z razvojem novih različic Ansible modulov, se s takšnim pristopom izognemo posodobitvi modela. Stanje pri uporabnikih je namreč zelo različno – nekaj jih takoj preide na najnovejšo različico, večina pa ostaja na starejših.

V primerih novih različic, lahko cevovod RAG omogoči dostop do ažurne dokumentacije brez ponovnega učenja modela. Delo, predstavljeno v tem poglavju zato predstavlja naš pristop k uporabi RAG, ki ga bomo skozi eksperimente izboljšali. Podobno kot pri gradnji modela, smo se osredotočili na podatke (npr. yaml_task) in ugotovili, da ima izbira modela vgrajevanja večji vpliv na rezultat kot izbira križnega kodirnika. Te ugotovitve bodo usmerjale prihodnje iteracije sistema.

5 Vtičnik

Za končno uporabniško integracijo smo uporabili obstoječi vtičnik **Steampunk Spotter Visual Studio Code extension**. Vtičnik je bil že razvit za orodje Steampunk Spotter in je v prejšnji različici vseboval funkcionalnost suggest, ki je delovala kot pametni iskalni mehanizem. Na podlagi uporabniškega navodila je sistem iz podatkovne baze poiskal najprimernejšo Ansible nalogo, pri čemer je šlo za iskanje na podlagi vnaprej ročno pripravljenega seznama/indeksa za omejeno število Ansible nalog.

Ker je bila odjemalska stran v Visual Studio Code že razvita, smo obstoječo funkcionalnost nadgradili z uporabo prilagojenega modela. Namesto enostavnega priklica obstoječe naloge iz baze sistem zdaj generira Ansible naloge z uporabo prilagojenega modela, opisanega v prejšnjih poglavjih. V ta namen smo pripravili programski vmesnik API, preko katerega vtičnik pošlje uporabniško navodilo modelu, model pa vrne enega ali več kandidatov Ansible nalog.

Arhitektura API dela rešitve je sestavljena iz dveh Docker storitev. Prva storitev sprejema zahtevke iz odjemalca (vtičnika Visual Studio Code), jih normalizira, ustrezno strukturira in posreduje modelu. Druga storitev izvaja t.i. serviranje modela. Za izvajanje modela v GPU režimu smo uporabili knjižnico vLLM, ki je uveljavljena knjižnica za serviranje modelov in inferenco. Ta storitev prejme zahtevek iz prve storitve, generira Ansible nalogo in rezultat vrne nazaj prvi storitvi, ta pa ga nato posreduje nazaj vtičniku v Visual Studio Code. V testni postavitvi je bil model nameščen na dveh grafičnih karticah RTX 2080 Ti.

Funkcionalnost za pomoč pri pisanju playbookov je integrirana neposredno v Steampunk Spotter Visual Studio Code extension in uporabnikom omogoča generiranje Ansible nalog neposredno v urejevalniku kode. Uporabnik funkcionalnost sproži tako, da v YAML datoteko vnese sprožilni zapis (trigger) `#spotter:`, ki mu sledi opis želene avtomatizacijske naloge v naravnem jeziku. Po prejemu vnosa sistem generira in prikaže ustrezno Ansible nalogo kot predlog kode neposredno v urejevalniku.

Uporabnik lahko predlog sprejme ali pa pregleda druge ponujene kandidate z uporabo navigacijskih kontrol, ki jih omogoča urejevalnik. S tem se funkcionalnost ne obnaša kot ločeno zunanje orodje, temveč kot del običajnega razvojnega toka v Visual Studio Code. Uporabnik ostane v isti YAML datoteki, v kateri pripravlja playbook, generirani predlog pa se vstavi neposredno na mesto kazalca.

Primer uporabe:

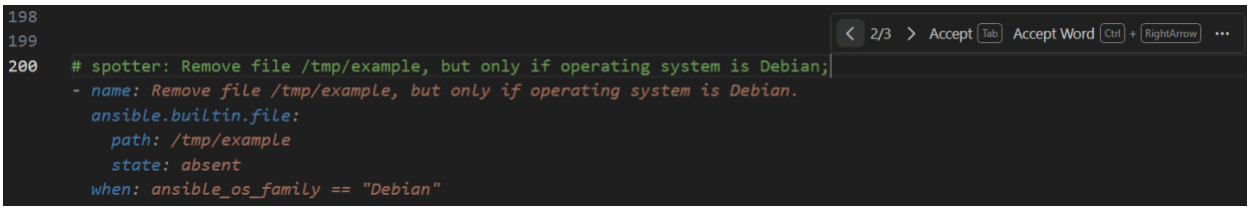
```
# spotter: Remove file /tmp/example, but only if operating system is Debian;
```

Za takšno navodilo sistem generira strukturirano Ansible nalogo, ki vključuje ustrezen modul, potrebne parametre in pogoje. Sistem hkrati pripravi več kandidatnih predlogov, kar uporabniku omogoča izbiro rešitve, ki najbolje ustreza njegovemu primeru uporabe. Kandidati se lahko razlikujejo v strukturi naloge, uporabljenih parametrih ali v nekaterih

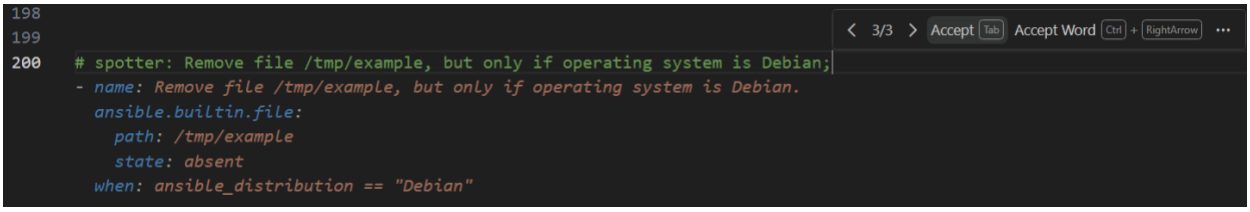
primerih tudi v izbranem FQCN. To je pomembno predvsem zato, ker uporabnik pri realni uporabi pogosto ne pozna pravilnega modula vnaprej, zato mu več kandidatov ponudi tudi implicitno pomoč pri izbiri ustreznega modula.

Slika 2, Slika 3 in Slika 4 prikazujejo tri generirane Ansible naloge, med katerimi uporabnik preklaplja, Slika 5 pa prikazuje Ansible nalogo, ki jo je uporabnik izbral.

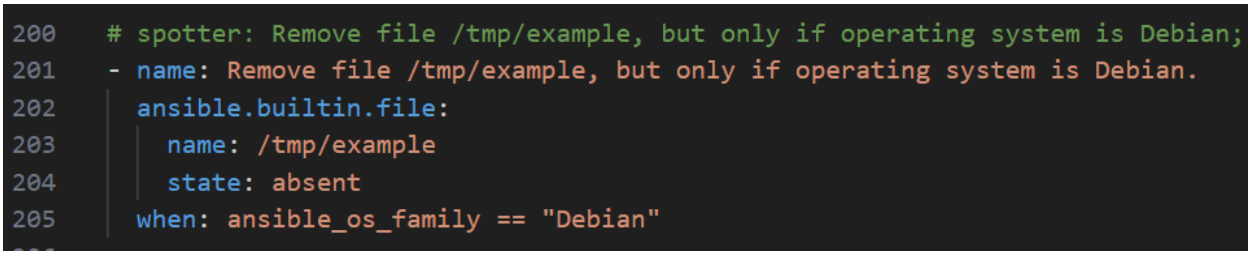
Slika 2:Prikaz delovanja vtičnika v VS Code – prvi kandidat



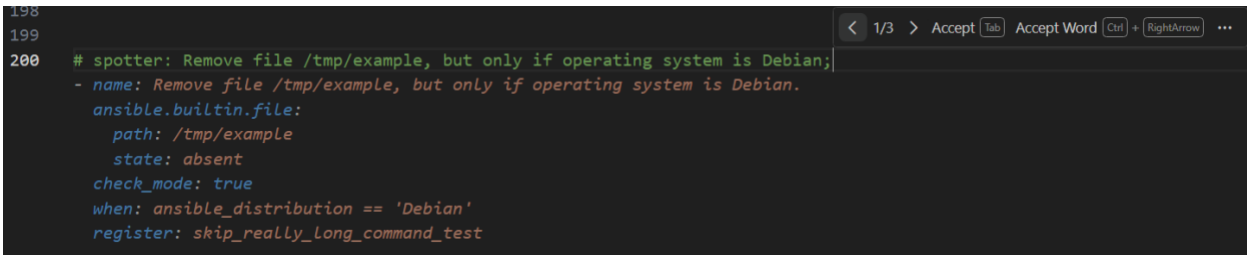
Slika 3: Prikaz delovanja vtičnika v VS Code – drugi kandidat



Slika 4: Prikaz delovanja vtičnika v VS Code – tretjii kandidat



Slika 5: Prikaz delovanja vtičnika v VS Code – izbrana Ansible naloga s strani uporabnika



Integracija v VS Code je pomembna za produktizacijo rezultata RRP6. Samega modela ne preizkušamo več kot izolirano raziskovalno komponento, temveč kot del uporabniškega orodja, ki ga razvijalci IaC kode lahko uporabljajo v svojem običajnem delovnem okolju. Izdelava dodatkov za uporabo vtičnika zato predstavlja prehod od modelnega eksperimenta k uporabniško dostopni programski funkcionalnosti, ki povezuje navodilo v naravnem jeziku, generiranje Ansible nalogo in končno uporabniško presojo.

6 Testiranje z eksperti – evalvacija sistema

Človeško evalvacijo so izvedli naši Ansible strokovnjaki. Za razliko od testiranj, opisanih v D6.4, ki so bila osredotočena predvsem na izhod modela oz. generiranje Ansible nalog, je bila v tej fazi ocenjena celotna programska rešitev, kot jo uporablja končni uporabnik v Visual Studio Code. To pomeni, da evalvacija ni zajemala samo kakovosti generiranih Ansible nalog, temveč tudi delovanje širšega sistema: vtičnika, programskega vmesnika API, odzivnosti sistema in celotne interakcije v urejevalniku.

Oba evalvatorja sta izrazila visoko stopnjo zadovoljstva z delovanjem vtičnika. Kot najpomembnejša izboljšava uporabniške izkušnje je bila izpostavljena odstranitev zahteve, da mora uporabnik v navodilu podati FQCN. V realni uporabi uporabnik pogosto ne ve, kateri Ansible modul je najprimernejši za določeno nalogo. Če bi moral uporabnik vnaprej navesti polno kvalificirano ime modula, bi to predstavljalo pomembno oviro pri uporabi sistema. Odstranitev te zahteve zato bistveno približa rešitev dejanskemu načinu dela uporabnikov IaC orodij.

Dodatna prednost sistema je generiranje več kandidatnih Ansible nalog za isto navodilo. To smo omogočili z nastavitvijo temperature vzorčenja na 0,9, kar poveča raznolikost generiranih predlogov. Uporabnik lahko zato pregleda več alternativ in izbere tisto, ki je najprimernejša za njegov primer uporabe. V nekaterih primerih predlogi vključujejo tudi različne FQCN, kar uporabniku posredno pomaga pri izbiri najprimernejšega modula za podano nalogo.

Evalvatorji so kot pomembno prednost izpostavili tudi tesno integracijo z urejevalnikom. Uporabnik lahko navodilo zapiše neposredno v YAML datoteko, ki jo že ureja, generirana Ansible naloga pa se vstavi neposredno na mesto kazalca. S tem ni potreben prehod v zunanjo dokumentacijo, spletni vmesnik ali ločeno orodje. Uporabnik ostane v svojem običajnem razvojnem toku, kar poveča praktično uporabnost sistema in zmanjša trenje pri uporabi.

Človeško testiranje je zato potrdilo, da je rešitev uporabna kot celovit pomočnik za pripravo Ansible nalog v razvojnem okolju. Pomembno je, da so evalvatorji presojali sistem v kontekstu dejanske uporabe: vnos navodila v naravnem jeziku, generiranje več kandidatov, prikaz predlogov v Visual Studio Code in uporabniško izbiro ustreznega rezultata. Takšna evalvacija bolje odraža realne pogoje uporabe kot izolirano preverjanje izhodov modela.

7 Kvantizacija

Da bi zadostili zahtevam uporabnikov po lokalnem izvajanju, smo obravnavali tudi možnost postavitve modela na računalnikih z omejenimi računskimi viri, na primer na prenosnikih ali osebnih računalnikih brez namenske GPU strojne opreme. Zaradi velikosti modela smo preverili kvantizacijo kot pristop za zmanjšanje pomnilniškega odtisa/povečanje hitrosti in omogočanje inference samo z uporabo CPU, brez večjega poslabšanja kakovosti izhoda.

Za kvantizacijo modela in CPU izvajanje smo uporabili orodja knjižnice llama.cpp [3]. Postopek kvantizacije je bil sestavljen iz naslednjih korakov:

- Uteži modela in konfiguracijske datoteke, shranjene v formatu Hugging Face (safetensor datoteke z utežmi, konfiguracija tokenizerja, Jinja predloge (templates) in pripadajoči metapodatki), smo najprej pretvorili v eno samo datoteko GGUF v polni natančnosti, tj. F32. Za pretvorbo smo uporabili pretvorbeno orodje iz knjižnice llama.cpp.
- Nato smo izdelali matriko pomembnosti. To smo izvedli tako, da smo skozi model v polni natančnosti poslali 500 reprezentativnih navodil iz učne množice. Ta kalibracijski korak oceni občutljivost posameznih uteži in omogoči, da postopek kvantizacije ohrani večjo natančnost pri utežeh, ki imajo večji vpliv na izhod modela, medtem ko pri manj kritičnih delih dopušča večjo kompresijsko napako.
- Datoteko GGUF v polni natančnosti smo nato kvantizirali z orodjem llama-quantize. Izhodni tenzor smo ohranili v izvorni, višji natančnosti, saj je odgovoren za izračun končnih logitov nad besediščem in je zato posebej občutljiv na šum, ki ga uvede kvantizacija. Povečanje velikosti datoteke zaradi ohranitve tega tenzorja je relativno majhno v primerjavi z morebitnim poslabšanjem kakovosti izhoda.
- Ovrednotili smo štiri kvantizacijske formate: IQ3_M, IQ4_NL, Q4_K_M in Q5_K_M. Ti formati vključujejo tako standardne metode kvantizacije po blokih, označene z Q, kot metode kvantizacije z upoštevanjem pomembnosti, označene z IQ, ki so namenjene boljšemu ohranjanju kakovosti modela pri nižjih bitnih širinah.

Evalvacija je bila izvedena na evalvacijski množici, opisani v Poglavju 3. Pred evalvacijo smo množico dodatno obogatili s strukturiranim poljem, ki je vsebovalo FQCN referenčne naloge in pripadajoče pare parametrov in vrednosti. Te informacije so bile neposredno izluščene iz referenčnih nalog in so omogočile sistematično primerjavo med izhodi modela in pričakovanimi rezultati.

Vsaka kvantizirana različica modela je bila ovrednotena zaporedno. Za posamezen model smo zagnali namensko instanco strežnika za inferenco, nato pa smo vsa navodila iz evalvacijske množice poslali kot ločene API zahtevke v formatu `chat-completion`, skladnem z OpenAI. Parametri generiranja so bili nastavljeni tako, da je model za vsak zahtevek ustvaril en izhod, pri čemer je bila temperatura nastavljena na 0,9. S tem smo omogočili zmerno raznolikost generiranja, hkrati pa ohranili smiselno variabilnost izhodov.

Za vsak zahtevek je bila določena časovna omejitev 30 sekund. Vsak zahtevek, ki je presegel to omejitev, je bil zabeležen kot neuspešen poskus. Po zaključku vseh navodil za

posamezen model smo instanco strežnika ustavili in nato inicializirali novo instanco za naslednji model.

Vse strežniške instance so bile konfigurirane z 8 nitmi, kar je učinkovito omejilo uporabo CPU na približno 800 %. Med evalvacijo smo za vsak model neprekinjeno spremljali in beležili porabo sistemskih virov, vključno z uporabo CPU in pomnilnika.

Evalvacijski okvir zajema štiri vidike: natančnost, kakovost izhoda, zakasnitev inference in porabo virov. Primarna metrika natančnosti je stopnja ujemanja FQCN, ki predstavlja delež evalvacijskih primerov, pri katerih se polno kvalificirano ime modula v generirani nalogi natančno ujema s polno kvalificiranim imenom modula v referenčni nalogi. Ta metrika meri splošno natančnost izbire modula.

Poleg tega smo beležili tudi število veljavnih odgovorov, torej število primerov, pri katerih je model vrnil rezultat znotraj določene časovne omejitve. S tem smo upoštevali tudi razpoložljivost modela v omejenih pogojih inference.

Kakovost izhoda smo ocenili s povprečnim odstotkom ujemanja parametrov. Ta metrika je bila izračunana izključno na primerih, kjer je bil FQCN pravilno napovedan. Meri stopnjo prekrivanja med napovedanimi in pričakovanimi pari parametrov in vrednosti ter tako ocenjuje sposobnost modela, da po izbiri pravilnega modula reproducira ustrezno strukturo naloge.

Zakasnitev inference smo opisali s povprečnim, minimalnim, maksimalnim in medianim časom odziva čez vse evalvacijske zahtevke. Te metrike omogočajo vpogled v tipično, najboljšo in najslabšo odzivnost posamezne kvantizirane različice. Porabo virov smo spremljali z najvišjo uporabo CPU in najvišjo porabo pomnilnika. Uporaba CPU je izražena v odstotkih, pri čemer vrednosti nad 100 % odražajo uporabo več procesorskih jeder. Poraba pomnilnika je izražena v gigabajtih in predstavlja pomnilniški odtis modela med inferenco.

Po evalvaciji vseh različic modela smo rezultate združili v primerjalni povzetek, kar kaže Tabela 4.

Tabela 4: Evalvacija štirih kvantizacijskih formatov GGUF. Tabela primerja veljavnost odgovorov, ujemanje FQCN, zakasnitev, ujemanje parametrov ter uporabo CPU in pomnilnika. Poudarjene vrednosti označujejo najboljši rezultat v posamezni kategoriji.

Metrika	**IQ3_M**	**IQ4_NL**	**Q4_K_M**	**Q5_K_M**
Veljavni odgovori	25	**42**	**42**	39
Ujemanja FQCN	21	36	**39**	32
Povprečna zakasnitev (s)	14,734	**9,841**	10,342	10,800
Minimalna zakasnitev (s)	7,724	**3,837**	4,115	5,689
Maksimalna zakasnitev (s)	20,842	**18,548**	24,196	22,147

Metrika	**IQ3_M**	**IQ4_NL**	**Q4_K_M**	**Q5_K_M**
Mediana zakasnitve (s)	15,468	**9,183**	9,381	10,361
Povprečno ujemanje parametrov (%)	**78,52**	73,78	71,04	72,36
Stopnja ujemanja FQCN (%)	84,00	85,71	**92,86**	82,05
Najvišja uporaba CPU (%)	748,8	759,6	750,7	751,5
Najvišja poraba pomnilnika (GB)	**6,36**	9,86	9,56	7,35
Minimalna poraba pomnilnika (GB)	**4,72**	8,09	7,81	5,62

Med ovrednotenimi konfiguracijami je Q4_K_M pokazal najugodnejše ravnovesje med metrikami. Dosegel je najvišjo natančnost ujemanja FQCN, in sicer 92,86 % oziroma 39 od 42 primerov. Hkrati je ohranil sprejemljivo zakasnitev, z mediano približno 9,4 sekunde na zahtevek, ter najvišjo porabo pomnilnika približno 9,6 GB.

IQ4_NL je dosegel najhitrejši povprečni čas inference, približno 9,8 sekunde, vendar z nekoliko nižjo natančnostjo. Q5_K_M je kompromis med porabo pomnilnika in zmogljivostjo. IQ3_M je imel najmanjši pomnilniški odtis, približno 6,4 GB pri največji porabi, vendar za ceno občutno počasnejše inference, približno 14,7 sekunde v povprečju, in slabše zanesljivosti. V časovni omejitvi 30 sekund je vrnil veljavne odgovore le za 25 od 42 evalvacijskih primerov.

Na podlagi teh rezultatov smo za lokalno (on-premise) izvajanje izbrali kvantizacijski format Q4_K_M. Ta format ohranja konkurenčno kakovost izhoda v primerjavi z modelom v polni natančnosti, hkrati pa omogoča inferenco na običajni uporabniški strojni opremi brez namenske GPU pospešitve.

8 Zmogljivost sistema

Zmogljivost končnega sistema smo obravnavali na dveh ravneh: na ravni samega modela in na ravni celotne uporabniške izkušnje (upoštevanje uporabniške poti - workflowa), od vnosa poziva v razvojnem okolju do prikaza generiranega predloga uporabniku.

Takšna razdelitev je pomembna, ker zgolj merjenje hitrosti inference modela ne opiše celotne uporabniške izkušnje. V končni programski rešitvi na zaznano hitrost vplivajo tudi prenos zahteve iz vtičnika v VSCode, obdelava zahteve, generiranje enega ali več kandidatov ter prikaz rezultatov.

Končna rešitev je zasnovana kot pomočnik (assistant) za pripravo Ansible nalog v razvojnem okolju VSCode. Uporabnik vnese navodilo (opis naloge v naravnem jeziku), vtičnik pa to zahtevo pošlje modelu, ki generira enega ali več kandidatov Ansible nalog.

Z vidika zmogljivosti je zato smiselno ločiti naslednje korake uporabniške poti, ki jih kaže Tabela 5:

Tabela 5: Vplivi korakov na zakasnitev, ki jo občuti uporabnik.

Korak	Opis	Vpliv na zmogljivost
Vnos uporabniškega navodila	Uporabnik v VS Code zapiše navodilo v naravnem jeziku - opis želene Ansible naloge	Brez pomembnega sistemskega vpliva
Pošiljanje zahteve	Preko API pošljemo navodilo, parametre generiranja	Odvisno od lokalne ali omrežne postavitve
Generiranje naloge	S pomočjo modela generiramo enega ali več kandidatov Ansible naloge	Glavni vir zakasnitve
Prikaz uporabniku	Kandidati se vrnejo v VS Code in prikažejo kot predlogi	Majhen vpliv, odvisen od števila kandidatov

Kot vidimo na Slika 1, sta dva režima postavitve v lokalnem programskem in strojnem okolju – CPU ter GPU režim.

V GPU režimu je sistem namenjen hitrejšemu generiranju in bolj odzivni uporabi v razvojnem okolju. V tej postavitvi se model izvaja na namenski grafični strojni opremi, ki je primerna za interno uporabo; Včasih je na voljo tudi okolje kjer je na voljo centralizirana infrastruktura za izvajanje modela. Prednost GPU režima je nižja latenca in večja zmogljivost pri generiranju več kandidatov, slabost pa je večja odvisnost od namenske infrastrukture in posledično manjša primernost za lokalno uporabo pri uporabnikih brez GPU strojne opreme.

CPU oziroma kvantizirani režim naslavlja drugačno omejitev: možnost uporabe modela v okoljih z omejenimi računskimi viri. V tem režimu se model pretvori v kvantizirano obliko, kar zmanjša velikost modela, porabo pomnilnika in odvisnost od namenske GPU infrastrukture.

Kvantizacija zato neposredno podpira zahtevo, da je rešitev uporabna tudi v okoljih, kjer uporabnik ne želi ali ne sme pošiljati infrastrukturne kode morebitnim zunanjim storitvam, čeprav sam nima ustrezne strojne opreme.

Cena takšnega pristopa je višja latenca pri generiranju, vendar je ta sprejemljiva za takšen način uporabe, kjer uporabnik pričakuje predlog kode med pisanjem in pričakuje določeno zakasnitev, glede na manjko namenske opreme (tj., GPU).

Pomemben vidik zmogljivosti je tudi število generiranih kandidatov. Generiranje več Ansible nalog izboljša uporabnost sistema, ker uporabniku omogoči izbiro med različnimi ustreznimi rešitvami in v nekaterih primerih tudi med različnimi moduli. Vendar pa večje število kandidatov poveča čas generiranja, zato je število kandidatov konfiguracijski kompromis med kakovostjo uporabniške izkušnje in odzivnostjo sistema. Za interaktivno uporabo v urejevalniku je na voljo tudi možnost omejitve števila kandidatov na manjše število, ki še vedno omogoča izbiro, vendar ne povzroči prevelike zakasnitve, kar pride v poštev tam, kjer ni na voljo zmogljiva strojna oprema in je večje število razvijalcev.

Paralelnost oz. sočasnost odziva sistema je omejena predvsem z izbranim načinom delovanja/inference modela oz. razpoložljivo strojno opremo. GPU režim praviloma bolje podpira več zaporednih ali sočasnih zahtev, medtem ko je CPU režim primernejši za individualno uporabo oziroma manjše število uporabnikov. Ker je končna rešitev v tem izročku obravnavana kot funkcionalnost v razvojnem okolju, cilj ni serviranje večjemu številu uporabnikov, temveč zanesljivo lokalno generiranje predlogov z ustrezno stopnjo nadzora nad podatki.

Na tem mestu spet poudarimo, da se celota vedno preveri orodjem Spotter. Tega ne štejemo v čas zakasnitve, saj se opravi takrat, ko uporabnik sam želi preveriti svojo kodo. Spotter zato deluje kot posredni kontrolni sloj po generiranju kode. S tem optimiziramo uporabnost, robustnost in varnost kode, ki vsebuje tudi generirano kodo. Pri oceni zmogljivosti je zato treba Spotter obravnavati kot zavesten del arhitekture, čeprav ni naveden eksplicitno pri tem vtičniku.

Končna ocena zmogljivosti je, da je sistem primeren za lokalno uporabo pri pripravi infrastrukturne kode.

9 Robustnost

Robustnost končnega sistema ne temelji samo na natančnosti prilagojenega (fine-tuned) modela, temveč na celotni verigi: Izbiri učnih podatkov, načinu učenja, ocenjevanju, validaciji tudi z uporabo orodja Spotter in človeški evalvaciji. Pri uporabi samega sistema gre pri generiranju za več kandidatnih rešitev, končni uporabniški presoji v razvojnem okolju in predvsem naknadni validaciji s Spotterjem. Skupek kode, bodisi generirane, bodisi napisane s strani človeka, ni obravnavan kot neposredno zaupanja vreden rezultat, temveč kot kandidat, ki mora biti pred uporabo preverjen.

Spotter v tej arhitekturi ni uporabljen kot avtomatski kontrolni sloj po generiranju pri uporabi – zgolj pri ocenjevanju kvalitete modela. Dejstvo je, da bi uporaba orodja Spotter za vsako generirano Ansible nalogo prinesla preveliko latenco med navodilom in rezultatom, zato smo se zavestno omejili samo na preverjanje kvalitete modela.

Robustnost je dodatno izboljšana z generiranjem več kandidatov. Če prvi predlog ni ustrezen, lahko uporabnik izbere alternativno rešitev, ki uporablja drug modul ali drugačno strukturo naloge. Tak pristop je posebej uporaben po odstranitvi zahteve, da uporabnik vnaprej pozna FQCN modula, saj sistem prevzame del odločitve o izbiri ustreznega Ansible modula. Hkrati pa to poveča potrebo po validaciji (kar se kasneje tudi opravi), ker več kandidatov pomeni tudi več možnih uporabniških izbir.

Na ravni uporabniške izkušnje robustnost vključuje tudi obravnavo neuspelih zahtev. Če model ne uspe z generacijo kode iz uporabniškega poziva, sistem ne blokira urejevalnika, temveč zahtevo prekine in uporabniku vrniti razumljivo informacijo.

Končna robustnost sistema je zato rezultat kombinacije treh ravni: model generira domensko ustrezne predloge, uporabnik pa v razvojnem okolju sprejme ustrezen rezultat. Na koncu, ko je uporabnik s svojo Ansible skripto (playbook, sestavljen iz več Ansible nalog) zadovoljen, Spotter preveri pravilnost in označi odstopanja ter predlaga izboljšave. Takšna zasnova ustreza realni uporabi IaC orodij.

10 Varnost / omejitve

Ker gre pri IaC (Infrastructure-as-Code) za kodo, ki lahko neposredno vpliva na konfiguracijo sistemov, omrežij, storitev in dostopov, generirani izhod ne sme biti privzeto obravnavan kot varen – tudi če ga pregleduje človek. Končna rešitev zato varnost obravnava večplastno: na ravni učnih podatkov in na ravni človeške potrditve pred uporabo. Spet poudarjamo, da se Ansible playbook, ki ga je napisal bodisi človek, bodisi vsebuje dele generiranih nalog, vedno preveri z orodjem Spotter.

Prvi varnostni ukrep je bil uveden že pri pripravi podatkov. Učni podatki so bili pred uporabo očiščeni in maskirani, zlasti tam, kjer so se pojavljale visoko-entropične vrednosti, kot so gesla, ključi, žetoni ali druge občutljive vrednosti. Kasnejše iteracije podatkov so dodatno izboljšale maskiranje z bolj tipiziranimi nadomestnimi vrednostmi. S tem se zmanjša tveganje, da bi model reproduciral občutljive podatke iz učne množice ali se učil neprimernih vzorcev ravnanja s skrivnostmi.

Drugi ukrep je vztrajanje na lokalnem/on-premise izvajanju oz. uporabi modela. Ker uporabniki pri IaC pogosto delajo z občutljivimi informacijami o infrastrukturi, je pomembno, da se pozivi in generirana koda ne pošiljajo zunanjim storitvam. Lokalno oziroma organizacijsko nadzorovano izvajanje modela omogoča večji nadzor nad podatkovnim tokom, dostopom, hrambo in obdelavo navodil (prompt). Pri tem mora biti jasno določeno tudi, ali se prompti in odgovori beležijo, kdo ima dostop do dnevnikov, ali se hranijo, ...

Tretji ukrep je preverjanje celotne kode s Spotterjem. Osnovno preverjanje omogoča zaznavanje tehničnih napak, opozoril in slabih praks. Pri varnostno občutljivih scenarijih je treba uporabiti varnostni oziroma širši profil preverjanja, ki validacijo razširi na varnostne in skladnostne vidike. Tako se lahko zaznajo potencialno tvegane konfiguracije, problematični vzorci avtomatizacije, ranljive ali zastarele odvisnosti in odstopanja od organizacijskih pravil.

Četrti ukrep je možnost uporabe politik skladnosti. V organizacijskem okolju zgolj splošno preverjanje ni vedno dovolj, ker imajo podjetja lastna pravila glede pravic dostopa, upravljanja skrivnosti, dovoljenih modulov, konfiguracijskih vzorcev in sprememb na infrastrukturi. Zato je treba Spotterjevo validacijo razumeti tudi kot mehanizem za preverjanje skladnosti generirane IaC kode z internimi pravili.

Kljub tem ukrepom sistem ne zagotavlja absolutne varnosti generirane kode. Jezikovni model lahko generira rešitev, ki je formalno veljavna, vendar operativno neprimerna za konkretno okolje. Zato mora končna rešitev jasno ohraniti princip človeške potrditve: generirani predlog je pomoč pri pisanju kode in ne pa samodejno odobrena produkcijska sprememba infrastrukture.

11 Zaključek

Ta izroček zaključuje delo v RRP6 in predstavlja končno iteracijo sistema VeMo-IaC. Na podlagi štirih predhodnih izročkov, ki so zaporedoma obravnavali izbor podatkov, začetno učenje modelov, iterativno izboljševanje podatkov in sistematično prilagajanje modela, D6.5 združuje te rezultate v celovito programsko rešitev za generiranje Ansible nalog iz navodil v naravnem jeziku.

V tej fazi so bili doseženi štirje ključni prispevki. Prvič, iz uporabniških navodil smo odstranili zahtevo po eksplicitni navedbi FQCN in naučili namenski model za vhodna navodila brez FQCN. Takšna konfiguracija ne doseže zgornje meje, ki jo je model iz D6.4 dosegal v idealnih pogojih, kjer je bil FQCN eksplicitno podan v navodilu. Vendar pa je v realističnem scenariju, kjer FQCN ni podan, več kot štirikrat boljša od istega modela iz D6.4. To je pomembno zato, ker prav ta režim najbolje odraža dejansko uporabo sistema: uporabnik običajno opiše, kaj želi narediti, ne ve pa nujno, kateri Ansible modul oziroma FQCN mora uporabiti. Ta kompromis je bil sprejet zavestno, v prid uporabnosti sistema.

Drugič, izvedli smo sistematično preverjanje pristopa RAG kot alternativne poti za zaznavanje FQCN. Preizkusili smo tri vgrajevalne (embedding) modele, dva križna kodirnika (cross-encoder) in več oblik baz znanja. Najboljša konfiguracija RAG je dosegla 78,57 % natančnost pri prvi izbiri po ponovnem razvrščanju, kar je približno sedem odstotnih točk manj od prilagojenega no-FQCN modela, ki je dosegel 85,71 %. Zato smo kot primarni mehanizem ohranili prilagojeni model. Kljub temu RAG ostaja pomembna razvojna smer, predvsem za primere, ko bo treba podpreti nove Ansible module ali nove različice modulov brez takojšnjega ponovnega učenja modela.

Tretjič, model smo integrirali v Steampunk Spotter Visual Studio Code extension. S tem je generiranje Ansible nalog postalo dostopno neposredno v razvojnem okolju, v katerem uporabniki že pripravljajo IaC kodo. Funkcionalnost se sproži z zapisom `# spotter:` neposredno v YAML datoteki, nato pa sistem na podlagi navodila v naravnem jeziku pripravi enega ali več kandidatov Ansible nalog. Človeško testiranje z Ansible strokovnjaki je potrdilo, da je takšna integracija uporabna in produktivna, saj uporabniku omogoča delo brez prehoda v zunanja orodja ali dokumentacijo.

Četrtič, model smo kvantizirali z uporabo knjižnice `llama.cpp` in ovrednotili štiri formate GGUF. Kot najprimernejši format je bil izbran `Q4_K_M`, saj je dosegel najboljše ravnovesje med natančnostjo, zakasnitvijo in porabo pomnilnika. Dosegel je 92,86 % stopnjo ujemanja FQCN, sprejemljivo zakasnitev in pomnilniški odtis, ki omogoča lokalno oziroma on-premises izvajanje na običajni uporabniški strojni opremi brez namenske GPU pospešitve.

Skupaj ti prispevki izpolnjujejo cilje, zastavljene za RRP6. Končni sistem omogoča natančno in zanesljivo generiranje Ansible nalog, deluje lokalno oziroma on-premise in s tem varuje občutljive infrastrukturne podatke, hkrati pa je izvedljiv na strojni opremi, ki je praviloma že prisotna v uporabniških okoljih. Kombinacija kakovostno pripravljene učne množice, ciljno prilagojenega modela, integracije v urejevalnik kode in učinkovite kvantizacije je pripeljala do rešitve, ki ni zgolj tehnično validirana, temveč tudi praktično uporabna.

Ocenjujemo, da razvita rešitev dosega cilj TRL6, saj je bila tehnologija predstavljena in preizkušena v relevantnem okolju. Sistem je bil uporabljen v okolju, ki odraža dejansko razvojno prakso uporabnikov IaC orodij: v Visual Studio Code, z uporabo Steampunk Spotterja, pri generiranju Ansible nalog na podlagi navodil v naravnem jeziku in z vključeno človeško presojo. S tem je demonstrator po funkcionalnosti, obliki in načinu uporabe primerljiv z načrtovanim operativnim sistemom.

Nadaljnje delo ostaja odprto v več smereh. Pristop RAG, čeprav v tej fazi ni bil izbran kot primarni mehanizem, predstavlja najbolj naravno pot za širitev podpore na nove module in nove različice modulov v ekosistemu Ansible. Nadaljnje izboljšave predstavitve baz znanja, izbire embedding modelov in postopkov ponovnega razvrščanja bi lahko zmanjšale razliko do prilagojenega modela. Evalvacijsko metodologijo bi bilo smiselno dodatno okrepiti z večjim deležem človeško napisanih navodil, saj ta najbolje odražajo realno uporabo, vendar trenutno predstavljajo najmanjši del evalvacijske množice. Nazadnje bo nadaljnje spremljanje uporabe vtičnika v dejanskih razvojnih okoljih omogočilo zaznavanje napak in robnih primerov, ki jih je težko odkriti v nadzorovani evalvaciji, ter bo usmerjalo naslednji cikel zbiranja podatkov in prilagajanja modela.

12 Literatura

[1] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[2] Y. Zhang *et al.*, "Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models," *ArXiv*, vol. abs/2506.05176, 2025, [Online]. Available: https://api.semanticscholar.org/CorpusID:279243736

[3] ggml-org, "llama.cpp," https://github.com/ggml-org/llama.cpp.