Универзитет у Београду

Електротехнички факултет



Радни оквир за отклањање грешака на нивоу продукционог кода

Дипломски рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| проф. др Захарије Радивојевић, ванредни професор | Лука Симић 2019/0368 |

Београд, Август 2023.

Садржај

[Садржај i](#_Toc144219719)

[1. Увод 2](#_Toc144219720)

[2. Преглед литературе 4](#_Toc144219721)

[3. Опис функционалности 7](#_Toc144219722)

[3.1. Архитектура система 7](#_Toc144219723)

[3.2. Проширење развојног окружења 8](#_Toc144219724)

[3.2.1. Генерални начин функционисања 8](#_Toc144219725)

[3.2.2. Структура проширења 9](#_Toc144219726)

[3.3. Апликација за дебаговање 13](#_Toc144219727)

[3.4. Агент 13](#_Toc144219728)

[3.4.1. Комуникација са развијачем 13](#_Toc144219729)

[3.4.2. Тачке бележења 15](#_Toc144219730)

[3.4.3. Израчунавање израза 15](#_Toc144219731)

[4. Манипулација бајткодом 17](#_Toc144219732)

[4.1. Бајткод 17](#_Toc144219733)

[4.2. Објекат кода 17](#_Toc144219734)

[4.3. *CPython* проширење 18](#_Toc144219735)

[4.4. Додавање тачке бележења 18](#_Toc144219736)

[4.4.1. Убацивање испред инструкције 19](#_Toc144219737)

[4.4.2. Замена инструкције скоком 19](#_Toc144219738)

[4.5. Табела линија 20](#_Toc144219739)

[5. Резултати 21](#_Toc144219740)

[5.1. Комплекснији пример манипулације бајткода 21](#_Toc144219741)

[5.2. Перформансе 24](#_Toc144219742)

[6. Закључак 26](#_Toc144219743)

[Литература 28](#_Toc144219744)

[Списак скраћеница 31](#_Toc144219745)

[Списак слика 32](#_Toc144219746)

[Списак табела 33](#_Toc144219747)

1. Увод

Алати за отклањање грешака у софтверу, познатији као дебагери (енгл. *debugger*) су једни од најкориснијих алата у развоју софтвера [44], и данас су обавезан део развојних окружења за различите намене. Они нам дозвољавају да наш програм зауставимо на одговарајућем месту, и одатле долазимо до разних информација о окружењу, од вредности променљивих, стања стека позива функција (енгл. *call stack*) до израчунавања произвољних израза које програмер зада [45]. Насупрот дебагерима, опције за инспекцију понашања програма су такође многе врсте логовања (енгл. *logging*, дословно „бележење”), које нам омогућавају сазнавање тачно онолико информација колико смо ми тражили у самом програмском коду, али не много више од тога. Иако су предности поменутих могућности дебагера над логовањем очигледни, због понеких имплементационих детаља, као и тежине постављања самог дебагера у односу на алате за логовање, алати за логовање могу бити бољи начин да се постигне одређени циљ приликом развоја софтвера.

Један случај, разматран у овом завршном раду, јесу продукциони сервиси. Продукциони сервиси, најчешће у виду веб сервиса и апликација, јесу сервиси пуштени својим клијентима на коришћење, и обично су критични за бизнис компаније која их одржава (компанија зарађује новац на основу рада тих сервиса). За разлику од продукционих сервиса, који се покрећу у окружењу познатом као продукција (енгл. *production*), сервиси такође могу бити покретани у развојном (енгл. *development*) окружењу, ради развоја самог сервиса, тест окружењу, ради покретања одговарајућих тестова над софтвером, као и разним другим окружењима као етапама од развоја до продукције (на пример, *staging* окружења).

Док се дебагери значајно примењују у развојном окружењу, у продукцији нису пронашли велику примену. У оваквом окружењу би заустављање ради дебаговања (какво нам омогућавају стандардни софтверски дебагери) могло бити фатално по стабилност апликације и самим тим нанети новчану штету компанији над чијим се сервисом користи ова техника. Поред тога, дебагери захтевају чување одговарајућих информација о изворном коду ради дебаговања, искључивање одређених оптимизација самог софтвера, као и јако често прекидање рада софтвера како би могли да изврше свој рад, што је за продукционе системе подједнако фатално као и само заустављање, због значајног угрожавања перформанси. Као мањи, али не занемарљив проблем, јесу најразличитије компликације приликом самог процеса за започињање дебаговања, где је потребна додатна мрежна конфигурација (дозвољавање комуникације преко одређених портова ради дебаговања) као и осигуравање (ограничење особа које могу да дебагују сервис) како би се овај процес успешно спровео.

Ипак, без техника дебаговања које нам пружа дебагер, у продукционим сервисима можемо видети све недостатке традиционалне технике дебаговања логовањем: логови нам пружају тачно оне информације које смо ми експлицитно захтевали у програмском коду, и самим тим, уколико желимо другачије информације, или логовање на другачијем месту, потребно је мењање самог програмског кода и још један пролаз тог кода кроз циклус од развојног до продукционог окружења (енгл. *release cycle*). Инжењерима који раде на неком сервису су информације ради дебаговања потребне што пре, како би мало по мало могли да дођу до тачног увида у ситуацију, и зато ради убрзавања целог тог процеса додају што више логовања на сва места која могу да предвиде да ће им бити потребна. То, са друге стране, повећава цену логовања у целој апликацији (што у перформансама и сложености самог кода, што у новцу, уколико се користе софистицирани алати за логовање), нарочито уколико се такав код не уклони након исправљања тражене грешке. Овакви логови су такође познати по томе што могу да испадну неочекивано скупи уколико се користе одређени сервиси у облаку за њихово чување и обраду [37].

Овај проблем је толико изражен, да су данас развијени различити алати који покушавају да га заобиђу. Једна од честих техника се заснива на могућности тачака бележења (енгл. *logpoints*), такође познатих као *tracepoints* [35], *non-breaking breakpoints* или *snapshots*, који не прекидају операцију самог програма а програмеру дозвољавају да без модификације програма забележи информације које су му потребне. Они решавају проблем проласка кроз циклус до продукције (овакве тачке се могу постављати за време покретања програма) као и проблем пречестог логовања (могу се уклонити чим програмер заврши са својим отклањањем грешки). Поред тога, ова техника нам омогућава различите врсте телеметрије у оквиру нашег производа, од откривања мртвог кода, до праћења коришћења одређених функционалности сервиса.

У овом раду разматрају се случајеви употребе једног оваквог система (заједно са постојећим алатима који их имплементирају), као и проблеми и технике у имплементацији оваквог система. На крају, биће имплементиран један овакав систем са одређеним подскупом могућности које постојећи алати садрже, са подршком за верзију 3.10 *Python* програмског језика (*CPython* интерпретер) и проширењем развојног окружења *Visual Studio Code* као графичким корисничким интерфејсом за наш дебагер, како би он могао да буде коришћен директно из развојног окружења.

У поглављу 2, разматрају се тренутна решења поменутих проблема уз навођење различитих аспеката њихове функционалности. У поглављу 3, бавимо се описом функционалности система израђеног у самом раду, са поделом на потпоглавља за различите делове система. У поглављу 4 налази се дискусија о различитим техникама манипулације *CPython* бајткода. У поглављу 5 износимо резултате тестирања перформанси и успешности имплементираног система.

1. Преглед литературе

На тржишту већ постоји неколико производа за намене дебаговања продукционих сервиса, често под ознаком „*developer observability”* [36]. Они се могу поредити по неколико карактеристика, од којих нећемо све разматрати у овом раду:

* **Интеграција са развојним окружењима:** Програмерима је лакше да из познатог окружења у којем већ развијају код такође врше дебаговање. Класични дебагери већ имају своје интеграције са развојним окружењима, па је логичан след тога да и дебагери продукционих сервиса наследе те карактеристике.
* **Могућност временски и количински ограничених тачака логовања:** Као једна од предности дебаговања продукционих сервиса над логовањем наводи се то да превише логовања штети перформансама самог сервиса и уме да повећа операционе трошкове. Дебагери продукционих сервиса, против ових проблема, умеју да дозволе програмерима да својим тачкама бележења доделе временски или количински рок, односно да се тачке бележења саме деактивирају након проласка одређеног времена односно након одређеног броја логовања из тих тачки.
* **Подржани програмски језици, радни оквири и технологије:** У свим дебагерима је неопходно додавати посебну подршку за сваки језик који тим дебагером може да се дебагује, било да су класични или дебагери продукционих сервиса. Оно што је код дебагера продукционих сервиса теже постићи јесте да омогуће очекивано функционисање независно од радних оквира који се користе (због специфичности имплементације), те програмери оваквих дебагера често наведу са којим радним оквирима је њихов алат тестиран да ради.
* **Израчунавање израза за логовање без споредних ефеката:** Како програмерима често не служи много информација да се до неке тачке бележења само дошло, већ хоће и да извиде стање програма у том тренутку, они дебагеру задају различите изразе за израчунавање у том тренутку. Одговорност дебагера продукционих сервиса, која се у класичним дебагерима понекад подразумева а понекад не [42][43], јесте да приликом израчунавања тих израза не дозволе споредне ефекте који могу да наруше стање програма.
* **Модел комуникације са метом дебаговања:** Због тога што се дебагују сервиси на удаљеним рачунарима, потребно је омогућити некакву мрежну комуникацију са њима како би дебаговање могло да се спроведе. Наиван приступ јесте да се на самим удаљеним рачунарима покрећу домаћински програми (сервери) за дебаговање, па да се програмери директно повезују на удаљене рачунаре на којима се налазе програми за дебаговање. Чешћи приступ, који омогућава боље перформансе и мање проблема са мрежним подешавањем, јесте да се удаљени рачунари са софтвером који је потребно дебаговати повезују на централни сервер за дебаговање, преко којег могу да комуницирају са самим програмерима.
* **Могућности профајлирања:** Профајлирање (енгл. *profiling*) јесте процес током ког програмери означавају делове кода чије перформансе желе да мере, и профајлер (алат који служи за профајлирање) им пружи те информације. Дебагери продукционог кода, са својим тачкама бележења, у принципу могу да им лако пруже ове информације (више је ствар корисничког интерфејса како ће те информације бити приказане).
* **Условне тачке бележења:** Као и у класичним дебагерима, где тачке заустављања могу бити условне, тако и у дебагерима продукционог кода тачке бележења могу да врше своје бележење само уколико је услов задат од стране програмера испуњен.
* **Цена:** Неки од ових дебагера су доступни искључиво комерцијално, неки дозвољавају више нивоа функционалности где је најнижи ниво бесплатан, док су неки у потпуности отвореног кода (енгл. *open-source*).

Са тиме у виду, преглед карактеристика постојећих дебагера као и карактеристика дебагера који ће бити имплементиран у овом раду наведен је у табели 2.1. Поједине карактеристике, попут спречавања приступа приватним информацијама које пролазе кроз продукциони сервис програмерима који тај сервис дебагују (из легалних разлога), овде нећемо поредити (нити имплементирати).

Табела 2.1. Преглед карактеристика постојећих дебагера као и дебагера имплементираног у овом раду.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Производ** | **Интеграција** | **Платформе** | **Споредни ефекти** | **Комуникација** | **Профајлирање** | **Условне акције** | **Цена** |
| *Lightrun* [1] | *IntelliJ*, *VS* *Code*, *PyCharm*, *WebStorm* | *Java*, *Python*, *Node.js*, *Docker*, *Kubernetes*, *AWS* *Lambda* | Не | Са *Lightrun* *Cloud* или *on-prem* сервером који прослеђује захтеве одговарајућим машинама | Да | Да | $0, или $1440 годишње [2] |
| *Rookout* [3] | Нема | *JVM*, *.NET*, *Python*, *Ruby*, *Node.js*, *Go*, *Docker*, *AWS Lambda*, *Azure* | ? | Са *Rookout Cloud* или *on-prem* сервером који прослеђује захтеве одговарајућим машинама | Да | Да | $0, $600 или $900 месечно [4] |
| *Fusion* *Reactor* [5] | Нема | *Java* | Није доступно | *FusionReactor Cloud* или локално | Да | Да | $20, $40, $60 или $80 месечно [6] |
| *Azure* *Snapshot Debugger* [7] | *Visual* *Studio* | *.NET*, *Azure* | ? | Преко *Azure* | Не | Да | Бесплатно |
| *Source++* [8] | *IntelliJ* | *Java*, *Python* | ? | *Source++ Cloud*, локално или *on-prem* | Не | Да | Бесплатно или плаћено [9] |
| *Google Cloud Snapshot Debugger* [10] | Нема | *Java*, *Python*, *Node.js*, *GCP* | Не | Преко *GCP* | Не | Да | Бесплатно |
| Овај рад | *VS Code* | *Python* | Не | Локално | Не | Да | 10 ЕСПБ |

Као што можемо видети, велики број постојећих решења нуде своје могућности у облаку преко ког се спроводи дебаговање (како би решили проблеме са повезивањем на мету дебаговања), а нуде и *on-prem* опције где програмери могу да пусте алат на својој инфраструктури, за нешто више цене. Може се приметити да организације које такође имају своје велике платформе у облаку (*Google*, *Microsoft*) праве своје дебагере за конкретно те платформе, и нуде их бесплатно као функционалност самих услуга на платформи. Такође, видимо да су већина језика са којима се интегришу дебагери интерпретирани уместо компајлирани, о чему ће бити речи при крају следећег поглавља.

1. Опис функционалности

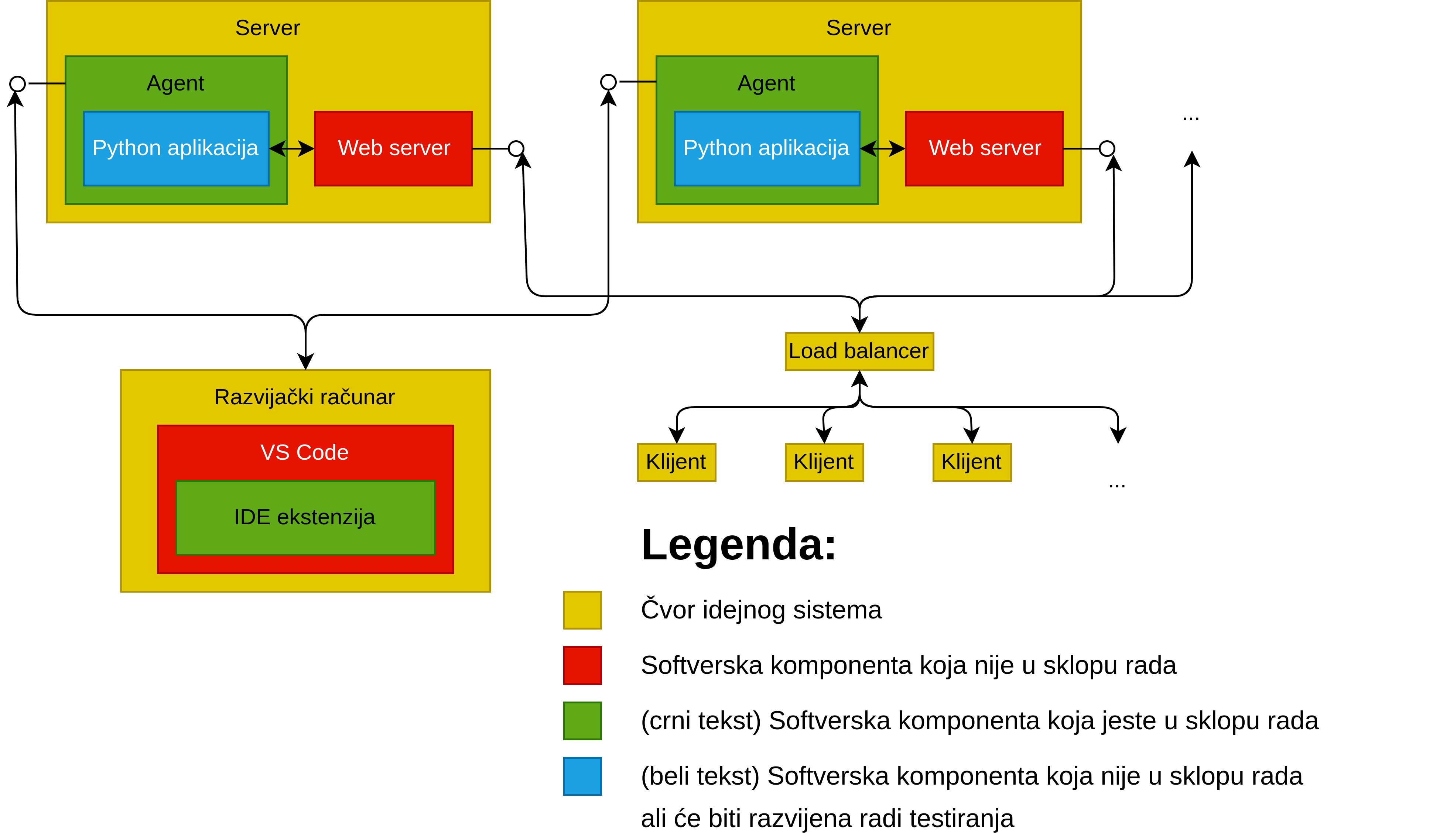
У овом поглављу биће описан начин функционисања целог система једног дебагера продукционих система. На почетку биће објашњена сама архитектура система, а затим ће за сваки део тог система бити објашњено на који начин је он имплементиран у самом раду у наредним одељцима.

* 1. Архитектура система

Архитектура једног дебагера продукционих система нацртана је на дијаграму са слике 3.1.1. Нацртана архитектура јесте наивна архитектура поменута у поглављу 2, на којој се програмер директно повезује на сваки од сервера са апликацијом ради дебаговања. Она ће и бити имплементирана у овом раду, јер архитектура са централним сервером за дебаговање, иако има предности у реалном свету, нема значајних изазова при имплементацији.

Приказан је систем у ком се апликације на серверима (једна апликација је дистрибуирана на више сервера) извршавају у оквиру агената за дебаговање. На сваком од сервера покренут је веб сервер који сервира садржај апликације, и тај садржај се сервира клијентима преко распоређивача оптерећења (енгл. *load balancer*). Свака апликација покренута је у оквиру једног агента за дебаговање, који програмерима дозвољава конекције ка себи ради дебаговања. На програмерским рачунарима покренуто је неко развојно окружење, у овом случају *Visual Studio Code* (верзија 1.79.2, конкретно *Code - OSS* варијанта), које програмеру пружа графички интерфејс за интераговање са агентом за дебаговање кроз одговарајуће проширење (енгл. *extension*) развојног окружења.

Као што је објашњено на слици 3.1.1, делови архитектуре развијени за потребе овог завршног рада јесу апликација у програмском језику *Python* над којом се врши дебаговање, проширење за развојно окружење *Visual Studio Code* које пружа графички интерфејс програмеру апликације за дебаговање, као и агент за дебаговање у оквиру ког се покреће сама апликација. Ови делови објашњени су у наредним одељцима.

Слика 3.1.1. Архитектура једног дебагера продукционих система.

* 1. Проширење развојног окружења

Као графички интерфејс за дебагер развијен у овом завршном раду, користи се развојно окружење *Visual Studio Code*, са додатим проширењем за сам интерфејс. *Visual Studio Code* [12], такође познат као *VS Code*, је развојно окружење засновано на *Electron* [11] платформи за развој десктоп апликација као веб апликација које се покрећу у *Chromium* претраживачу. Са тим у виду, окружење је развијено у *TypeScript* језику [16], који је заснован на *JavaScript* са могућностима означавања типова променљивих, и који се компајлира у *JavaScript* ради покретања унутар претраживача. У овом језику се, стога, развијају и проширења за ово развојно окружење.

* + 1. Генерални начин функционисања

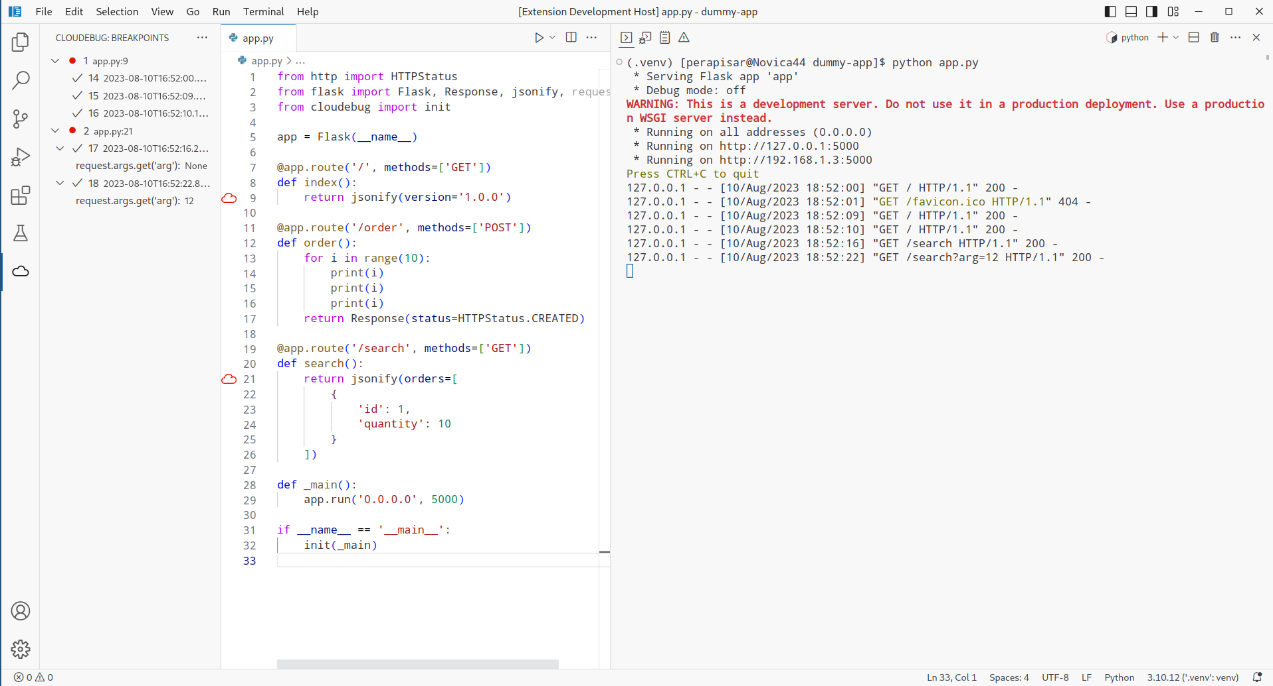
Од функционалности које се програмеру пружају кроз кориснички интерфејс за дебаговање, ту су:

* **Тачке бележења:** Основна операција нашег дебагера јесте да логује одговарајуће информације када програм који се дебагује дође до одређеног места (без заустављања самог сервиса), и то врши на местима задатим од стране програмера тог сервиса. Како би то било могуће, кориснички интерфејс дебагера мора да кориснику омогући да ове тачке бележења поставља. Постављање, као и код корисничког интерфејса класичних дебагера, се очекује да буде на нивоу линије.
* **Преглед логова:** Чим се дође до неке од тачки бележења у току извршавања кода сервиса, логују се одговарајуће информације. Ово значи да наш кориснички интерфејс дебагера мора да програмерима нуди опцију кроз коју могу за сваку тачку бележења виде кад је све достигнута и које су биле вредности одређених израза у тренутку достизања.
* **Израчунавање израза:** Програмерима сама информација да се до неке тачке бележења дошло не значи толико, нарочито када на тим тачкама не могу да зауставе програм и извиде стање. Зато је потребно понудити им неки начин извиђања стања програма. Једно решење за овај проблем јесте логовати цело стање програма у том тренутку, како би програмер касније могао да, помоћу неког алата за извиђање стања, провери шта се тада десило и шта могу бити узроци грешке коју отклања, али стање програма може садржати доста информација и његово логовање би одузело доста времена (а и простора) датом продукционом сервису. Као компромис између не логовања ничега и логовања свега, програмеру се даје могућност да зада конкретне изразе који ће бити израчунати у току извршавања, и њихове вредности ће му бити доступне у логовима дебагера.
* **Условне тачке бележења:** Програмер често жели да неке информације логује само под одређеним условом, како би имао кроз мање логова да гледа и како би мање оптеретио сервис у оквиру кога се те информације логују. У овом случају, он може задати одређени услов који се израчунава приликом достизања тачке бележења, и логовање се једино дешава уколико је тај услов испуњен.
* **Спречавање неауторизованог приступа:** Природно, један овакав систем дебаговања је мета за нападаче који желе да извуку информације из сервиса који се дебагује или му на неки начин нашкоде. У овом завршном раду имплементиран је (релативно наиван, али довољно сигуран) систем где дебагер може тражити лозинку од програмера који се на њега повезују и одбити им повезивање у случају да та лозинка није добра.
  + 1. Структура проширења

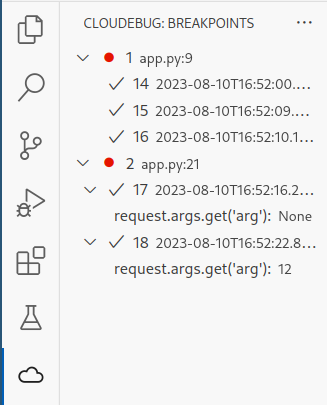
Структура једног проширења за *VS Code* јесте структура стандардног *Node.js* [13] пакета, у томе што садржи одговарајући *package.json* фајл [14] који служи као манифест и садржи метаподатке о пакету, додатне пакете од којих тај пакет зависи и остаје информације потребне како би се тај пакет објавио на регистру пакета попут *npm* [15]. За потребе *VS Code* проширења, у овом фајлу се додатно наводе информације о тачкама проширења самог развојног окружења [15], где програмер који то проширење пише може да наведе које делове окружења жели да прошири и на који начин. На овом месту се не задају функционалности самог проширења, већ само ствара костур проширења са менијима, командама и погледима које ће он додати. Овај пакет из своје главне датотеке мора да извози две функције: једну која се позива приликом активације самог проширења, а другу приликом деактивације. Приликом активације могуће је свим тачкама проширења дефинисаним у манифесту проширења доделити функционалност у виду кода који се позива приликом интеракције са тим тачкама проширења.

Од тачака проширења, проширење развијано за потребе овог завршног рада додаје следеће:

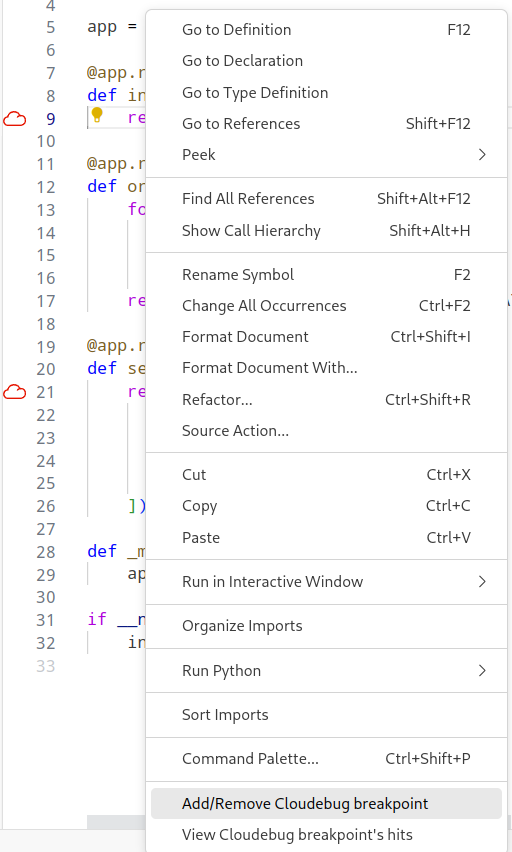
* **Леви панел:** Главни панел за интеракцију са нашим дебагером. Кроз њега се врши повезивање на сервер за дебаговање, а затим се унутар њега прикажу све тачке бележења као и све забелешке направљене на тим тачкама, заједно са вредностима израза које је програмер тражио да се израчунају у тим тачкама. На слици 3.2.2.1 приказан је изглед целог окружења са левим панелом и постављених пар тачака бележења, а на слици 3.2.2.2 приказан је и леви панел одвојено. Одавде је тачке бележења такође могуће уклонити.
* **Индикатори на линијама:** Програмеру је на неки начин потребно скренути пажњу на то на којим линијама се налазе тачке бележења. На слици 3.2.2.1 се, између осталог, могу видети црвени облачићи као индикатори да се на тим линијама налазе тачке бележења.
* **Команде за постављање тачака бележења:** На самим линијама унутар уређивача кода могуће је поставити и уклонити тачке бележења, као и погледати за одређену тачку бележења који су сви логови направљени. Приликом додавања тачке бележења, од корисника се тражи да унесе услов под којим ће логовање да се деси, и изразе који ће да буду израчунати у том тренутку. На слици 3.2.2.3 приказан је мени за додавање односно уклањање тачке бележења, док је на слици 3.2.2.4 приказан процес додавања једне тачке бележења.
* **Команда за повезивање:** Повезивање са дебагером могуће је покренути из горњег дела левог панела. Приликом повезивања, од корисника се тражи адреса дебагера (подразумевана вредност овог поља је оптимална за испробавање дебагера) и лозинка за повезивање на дебагер. На слици 3.2.2.5 приказан је процес повезивања на сервер.



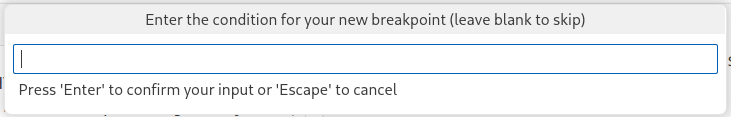
Слика 3.2.2.1. Изглед развојног окружења *Visual Studio Code* са проширењем за дебаговање продукционог кода.

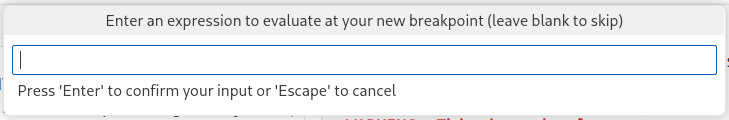


Слика 3.2.2.2. Леви панел проширења *Visual Studio Code* за дебаговање продукционог кода.

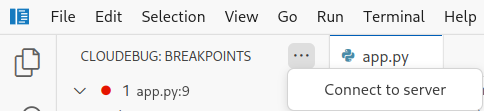


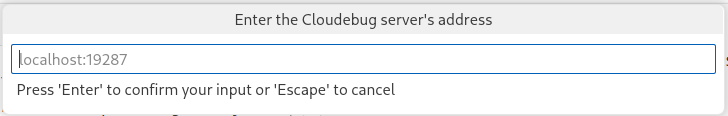
Слика 3.2.2.3. Приказ менија за додавање/уклањање тачке бележења у проширењу за *Visual Studio Code*.

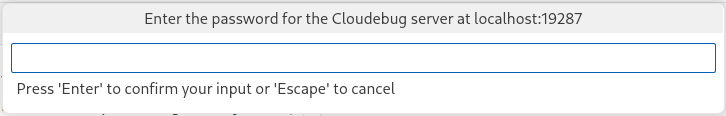




Слика 3.2.2.4. Додавање тачке бележења у проширењу за *Visual Studio Code*. Унос услова бележења (горе) и израза за израчунавање у току бележења (доле).







Слика 3.2.2.5. Процедура повезивања на сервер за дебаговање продукционог сервиса кроз проширење за *Visual Studio Code*. Бирање опције за повезивање (горе), уношење адресе сервера (средина) и уношење лозинке за повезивање (доле).

* 1. Апликација за дебаговање

За потребе овог рада развијена је једноставна апликација за дебаговање над којом се тестирао дебагер. Апликација је развијена у радном оквиру *Flask*, како би што боље симулирала неки продукциони веб сервис који сервира захтеве клијената.

Направљена је са три приступне тачке: *GET /*, *POST /order* и *GET /search*. *GET /* је једноставна приступна тачка направљена тако да преко ње могу да се тестирају разлике у перформансама апликације са једном тачком бележења у односу на апликацију без тачака бележења и апликацију без дебаговања. *POST /order* је приступна тачка чији је код написан тако да тестира неке граничне случајеве манипулације бајткода.

Приликом покретања апликације, она позива функцију за иницијализацију дебагера, а затим покреће *Flask* веб сервер.

* 1. Агент

Срж целог система дебаговања продукционих апликација јесте агент, код који се покреће уз саму апликацију и омогућава програмеру да је дебагује споља. Ова компонента има два задужења:

* да на команду програмера постави тачку бележења са задатим условом и изразима за израчунавање, и
* да логује информације о достизању тачке бележења и израчунатим изразима и те информације јави програмеру (било одмах, било накнадно).

Из овога видимо да се агент суштински може поделити на два дела, део који служи за комуникацију са програмером и део који служи за руковање тачкама бележења. Надаље их у овом раду разматрамо у одвојеним одељцима.

* + 1. Комуникација са програмером

Како комуникација са програмером има потенцијал да буде у реалном времену, односно да програмера занимају информације о белешкама из тачака бележења чим се догоде, као протокол за комуникацију са програмером у овом раду изабран је *WebSocket* [17], као протокол који омогућава комуникацију у реалном времену и подржан је у свим модерним претраживачима (укључујући и *Chromium*, на ком је *Visual Studio Code* заснован).

На почетку извршавања сервиса који се дебагује, неопходно је да код сервиса позове функцију за иницијализацију окружења дебагера. Затим, дебагер покреће *WebSocket* сервер и отвара везу са базом података у којој чува информације о дебаговању, која је у овом раду направљена као *SQLite* [18] база података због своје брзине и једноставности. Преко *WebSocket* везе се подаци размењују у *JSON* [19] формату. Програмер дебагеру може да зада следеће команде:

* **Додавање тачке бележења:** За задату линију задатог фајла, дебагер додаје тачку бележења са опционално задатим условом и опционално задатим изразима за израчунавање. Програмеру се враћају информације о додатој тачки, укључујући њен идентификатор.
* **Уклањање тачке бележења:** За задат идентификатор, дебагер уклања тачку бележења са тим идентификатором.
* **Дохватање тачака бележења:** Приликом успостављања нове везе, програмер не зна које су све тачке бележења у дебагеру, па овом командом може тражити да му их пошаље.
* **Дохватање логова за тачку бележења:** Приликом успостављања нове везе, програмер не зна који су се све догађаји десили за сваку тачку бележења, па овом командом може да за задату тачку бележења сазна информације о томе која све достизања су логована, као и вредности задатих израза у тренутку логовања.

Поред овога, дебагер самоиницијативно шаље свим програмерима поруке у тренутку када се деси логовање, са информацијама као у претходно наведеној команди.

Дебагер свој комуникациони посао врши паралелно са *Python* програмом који дебагује. У *Python* је доступно више механизама који обезбеђују паралелизам. Овде искоришћен приступ коришћењем *threading* библиотеке [20] покреће одвојену нит са *WebSocket* сервером у којој се врши комуникација са програмером кад је то потребно. Ипак, нити из *threading* библиотеке не могу у потпуности да врше свој посао у паралели — због механизма познатог као *Global Interpreter Lock* [23] у оквиру *CPython* интерпретера за *Python* код (који је најчешће коришћени *Python* интерпретер, референтна имплементација, и на кога се фокусирамо у овом раду), само једна нит може да извршава *Python* код. Други модел паралелизације доступан јесте паралелизација помоћу *asyncio* модула [21], који употребљава *event loop* механизам [41] за асинхроно обављање појединих послова, сличан оном присутном у *JavaScript* програмском језику. Ипак, овај механизам свој посао обавља у једној нити, тако да би дебаговани програм морао да буде написан тако да сарађује са дебагером на дељењу *event loop*, те овај модел није био добра опција. Трећи модел паралелизације јесте помоћу *multiprocessing* модула [22], који покреће више *Python* интерпретера како би вршио свој посао. Овај модел омогућава стварну паралелност у извршавању *Python* кода, али је дебагеру неопходно да се извршава у оквиру интерпретера који дебагује како би у време извршавања могао да прави измене, тако да се и од овог приступа одустало.

* + 1. Тачке бележења

Главна функционалност дебагера продукционих система јесте могућност да поставља тачке бележења. Постављање ових тачки даје дебагеру још доста могућности: он може на њима израчунавати изразе, мерити перформансе, бележити аналитику и тако даље.

Први приступ овом проблему може бити на исти начин на који му класични дебагери приступају. Први покушај може бити коришћењем *pdb* библиотеке за дебаговање [24], али се убрзо долази до закључка да је он преспор за употребу у продукционим сервисима, јер значајно угрожава њихове перформансе. Уместо тога може се пробати коришћење уграђене функције *sys.settrace* [25] намењене за дебаговање, која нам дозвољава да хватамо различите догађаје унутар *CPython* интерпретера у сваком тренутку извршавања, али су претходна истраживања на ову тему [26] показала да ни ово не даје задовољавајуће перформансе, због интерне имплементације овог механизма у *CPython*.

Метод којем најчешће прибегавају дебагери продукционог кода је манипулација *CPython* бајткода. Овај метод погодан је за све дебагере који дебагују језике који се интерпретирају од стране неког интерпретера, попут *Python* (*CPython*), *JavaScript* (*V8*), *Java* (*JVM*) или C# (*CLR*). Он нам омогућава да на неком месту у измењеном бајткоду кажемо да треба да се скочи на наш код за бележење, и тако направимо тачку бележења. У *CPython* није могуће мењати бајткод програма који се извршава из *Python* кода, па је зато за ове потребе потребно направити *C*/*C++* проширење за *CPython* [27] како би се ове измене извршиле.

Поједини алати наведени у другом поглављу, попут *Lightrun* и (сада угашеног) *Sidekick* [28], прибегли су коришћењу *Google Cloud Debugger* [10] кода за манипулацију бајткодом. У овом раду се, такође, осврћемо на њихову имплементацију на неколико места. У наредном поглављу детаљније се залази у интерно функционисање *CPython* и методе манипулације бајткода.

* + 1. Израчунавање израза

Једном кад имамо имплементиране тачке бележења, ми у тим тачкама можемо извршавати код за различите сврхе. Поред самог бележења, од поменутих функционалности нашег дебагера овде су релевантне условне тачке бележења, израчунавање израза и споредни ефекти током тог израчунавања.

Условне тачке бележења и израчунавање израза се, заправо, имплементирају на исти начин. Како би се проверило да ли је неки услов испуњен, израз за тај услов пре тога мора бити израчунат. Једина разлика јесте у месту израчунавања: израз за услов мора бити први израчунат, па уколико је он испуњен може да се пређе на израчунавање осталих израза и њихово бележење.

Израчунавање израза врши се помоћу *eval()* функције стандардне библиотеке [38]. Њој је као контекст могуће проследити локалне и глобалне променљиве функције унутар које се израчунава израз, па ову могућност можемо да користимо како бисмо јој проследили локалне и глобалне променљиве функције из које се дошло до тачке бележења.

Поред тога, неопходно је да обезбедимо да израчунавање задатих израза буде без споредних ефеката, како не бисмо имали могућност да нарушимо стабилност наше апликације самим постављањем тачке бележења. Провера споредних ефеката, у овом завршном раду, заснована је на начину провере споредних ефеката који се користи у *Google Cloud Debugger*. Он се састоји из постављања функције за праћење тока програма преко горепоменуте *sys.settrace* функције и реаговању на догађаје који се десе на следећи начин:

* **Позив функције:** Приликом позива функције, проверава се да позвана функција не користи неко од забрањених назива који имплицирају споредне ефекте, попут *\_\_setitem\_\_* (додела вредности кључу у речнику, елементу низа и тако даље).
* **Позив уграђене функције:** Како се из *Python* интерпретера могу звати и функције које нису имплементиране у *Python*, ми немамо начин да проверимо на који начин су те функције имплементиране. Зато се овде ограничава које уграђене функције могу да се позову на скуп унапред познатих уграђених функција које не изазивају споредне ефекте (математичке функције, функције које раде над низовима без њиховог мењања, и тако даље).
* **Извршавање линије:** Приликом извршавања једне линије *Python* кода, морамо да проверимо да се не извршавају неке инструкције које умеју да изазову споредне ефекте (на пример, *PRINT\_EXPR*, која штампа израз на стандардни излаз). У овом процесу проверавамо које инструкције су обухваћене линијом која се извршава, и за сваку проверавамо да ли може или не може да изазове споредне ефекте. На овом месту смо такође, уместо праћења извршавања линија, могли да пратимо извршавање појединачних инструкција, али би то изазвало значајнији пад перформанси.

Уколико наиђемо на одступање од правила изнад, из функције за праћење тока програма можемо да бацимо грешку коју ће ухватити функција за израчунавање израза и ту грешку пријавити програмеру током бележења.

1. Манипулација бајткодом

У овом поглављу објашњено је интерно функционисање *CPython* интерпретера до нивоа потребног за разумевање метода манипулације бајткодом коришћених у овом раду, а затим су објашњене и саме методе.

* 1. Бајткод

*CPython* интерпретер [29] за *Python* програмски језик је референтна имплементација једног интерпретера за тај језик у програмском језику *C*. Када се користи термин „*Python* интерпретер", обично се мисли на *CPython*, иако постоје и други *Python* интерпретери [30]. У овом раду се фокусирамо на верзију 3.10 овог интерпретера, направљеног да интерпретира ту исту верзију програмског језика.

*CPython* интерпретер [33] функционише тако што прво преведе *Python* програмски код у бајткод [31], а затим интерпретира тај бајткод. Ово, поред смислене поделе имплементације, му дозвољава да не врши превођење за изворни код који је већ преведен, уколико га је сачувао у свом кешу [32], као и дистрибуцију унапред преведеног *Python* кода. Овај бајткод, иако јесте документован, је *CPython* имплементациони детаљ и може се значајно мењати између верзија интерпретера.

У *CPython* 3.10 бајткоду свака инструкција је величина два бајта: први бајт представља операциони код, а други операнд (у том смислу, *CPython* интерпретер је једноадресна машина). Док је 256 операционих кодова довољно за функционисање интерпретера, 256 могућих вредности није. Овај проблем решен је увођењем *EXTENDED\_ARG* инструкције, која дозвољава проширење операнда (по конвенцији могу се поређати три ове инструкције заредом, али интерпретер то не проверава). Приликом интерпретације бајткода се такође користи стек.

* 1. Објекат кода

Једна јединица превођења *Python* бајткода назива се објекат кода (енгл. *code object*). Један објекат кода представља контејнер за *Python* код, и може представљати функцију, модул (уколико се код пише ван функција), методу и тако даље. Један објекат кода са собом такође садржи величину стека потребног за евалуацију (она је фиксна и израчуната у току превођења), називе који се користе у коду (локалних променљивих, глобалних променљивих, поља, модула, и тако даље), константе, информације о мапирању бајткода на линије изворног кода и одређене информације о окружењу преведеног блока кода.

Објекти кода (*PyCodeObject*) су један тип *PyObject* структуре, што значи да им се може приступати из *Python* кода. Функције, на пример, имају своје поље *\_\_code\_\_* које приступа њиховом објекту кода, и одатле се може читати бајткод те функције. Такође, у стандардној библиотеци постоји *dis* модул [31] који се може користити као помоћ при дисасемблирању. Ипак, ова поља објеката кода није могуће изменити из *Python* кода, те је неопходно написати *CPython* проширење у језицима *C* или *C++* како би се то омогућило.

* 1. *CPython* проширење

Проширења за *CPython* интерпретер пишу се у програмским језицима *C* или *C++* а затим динамички повезују са интерпретером у време извршавања. Једно проширење мора да садржи *PyInit\_naziv\_modula* функцију коју интерпретер позива при иницијализацији проширења (приликом увожења модула који проширење извози), и која као повратну вредност има *PyModuleObject* објекат који представља модул са функцијама које ово проширење извози.

Проширење развијено у овом раду извози две функције: једну за додавање, а другу за уклањање тачке бележења. Обе функције примају као аргументе објекат кода над којим се врши операција, линију на којој је потребно додати или уклонити тачку бележења, као и њен идентификатор (одређивање о ком објекту кода за дати назив фајла и линију се ради се врши из *Python* кода који позива функцију проширења).

* 1. Додавање тачке бележења

Како би се додала једна тачка бележења потребно је изменити бајткод тако да на месту где почиње линија за коју је тражено додавање тачке бележења стоји позив функције а затим се прелази на оригиналне инструкције. Позив функције може се извршити на следећи начин:

1. ***LOAD\_GLOBAL*:** Помоћу ове инструкције на стек учитавамо објекат модула из којег желимо да позовемо функцију. Аргумент јесте индекс у *co\_names* пољу објекта кода (који представља *Python* торку) са називом глобалне променљиве у којој је сачуван модул. То значи да при додавању тачке бележења морамо изменити ово поље, и такође додати одговарајући модул као глобалну променљиву.
2. ***LOAD\_METHOD*:** Помоћу ове инструкције са стека скидамо модул, из њега дохватамо једну методу, и ту методу стављамо на стек (заједно са *NULL*, који означава да нисмо учитали методу објекта). Аргумент је такође индекс у *co\_names* пољу, па и за ово морамо мењати то поље.
3. ***LOAD\_CONST*:** Како би позвана функција знала о којој тачки бележења се ради, треба јој проследити идентификатор те тачке. Ова инструкција нам омогућава да тај идентификатор учитамо на стек ради прослеђивања тој функцији. Аргумент је индекс константе у *co\_consts* пољу објекта (који, опет, представља *Python* торку), па на сличан начин као *co\_names* мењамо и ово поље како бисмо тај идентификатор убацили.
4. ***CALL\_METHOD*:** Ова инструкција позива методу са стека и скида њен аргумент и претходно додати *NULL*. Аргумент је број позиционих аргумената (у овом случају, 1). На стек се после извршења методе ставља њена повратна вредност.
5. ***POP\_TOP*:** Како нам повратна вредност претходно извршене методе не значи ништа, овом инструкцијом је скидамо са стека. Аргумент се игнорише.

Ова секвенца може се скратити и на три инструкције (*LOAD\_CONST*, *CALL\_FUNCTION*, *POP\_TOP*), као што се то ради у *Google Cloud Debugger*, уколико се сама метода дода као константа, и уколико се не шаље идентификатор тачке бележења као аргумент методе већ се на неки други начин одреди о којој се тачци бележења ради.

Проблем настаје када ова секвенца инструкција треба да се убаци у сам програм. Постоји више начина за убацивање ове секвенце, и ниједан од њих не ради у свакој ситуацији, али могу постићи довољно добар успех.

* + 1. Убацивање испред инструкције

Овом методом, када се пронађе прва инструкција на линији за коју је тражено додавање тачке бележења, испред ње се додају инструкције за позивање одговарајуће функције. На овај начин, интерпретер прво изврши позив функције приликом доласка до те линије, а затим настави на регуларне инструкције. Очигледан проблем у овој методи јесу скокови — скокови који су релативно скакали уназад а налазе се иза додатих инструкција, скокови који су релативно скакали унапред а налазе се испред додатих инструкција, као и апсолутни скокови унутар тог објекта кода, морају да се измене. Приликом измене, неки од скокова могу променити своје аргументе тако да више није довољна једна инструкција за изражавање скока, већ је неопходно додати и *EXTENDED\_ARG* инструкцију.

Ипак, колико год комплексно звучало померање скокова, то је ипак решив проблем. Много већи проблем овој методи стварају инструкције *YIELD\_FROM* и *YIELD\_VALUE*, које се користе за имплементацију генераторских функција [34]. Како генераторске функције након позивања *yield* морају да при следећем позиву наставе од места где су претходно изашле, било каква промена локација инструкција у функцији између два позива може да поремети нормалан рад програма у који се додаје тачка бележења. У *Google Cloud Debugger*, делимично решење за ово јесте да се примени метода објашњена у следећем одељку.

* + 1. Замена инструкције скоком

Друга могућа метода убацивања тачке бележења јесте да се инструкције на месту линије на којој треба убацити тачку замене са скоком на неки други део кода, који може бити додат на крај постојећег. Из новододатог дела кода се затим врши позив функције, а онда се изврше оригиналне инструкције и врати назад на место са ког је скочено. На овај начин постижемо то да се локације остатка инструкција не мењају, те је могуће у генераторским функцијама поставити тачке бележења без да *YIELD\_FROM* и *YIELD\_VALUE* инструкције праве проблем.

Ипак, ова метода у општем случају ради горе него претходна. Први проблем лежи у томе што није могуће заменити све инструкције. *YIELD\_FROM* и *YIELD\_VALUE* инструкције нам и даље праве проблем овде, јер уколико бисмо њих морали да заменимо, њихова локација би се променила и дошли бисмо до истог проблема као у првој методи. *Google Cloud Debugger* приликом примене ове методе такође одбија да замени било какве инструкције релативног скока, јер поједине инструкције могу да скачу унапред. Ипак, у овом раду смо експериментално утврдили да се уз одговарајући број *EXTENDED\_ARG* инструкција скокови унапред могу изменити да скачу уназад.

Други проблем лежи у томе што скок на новододати код може да чини више од једне инструкције, уколико се прескаче више од 256 инструкција. То значи да заменом две или више инструкција скоком можемо проузроковати да се нађе нека друга инструкција скока која скаче у сегмент кода између убачених инструкција, па ова метода у том случају не може да се примени.

У овом раду је овај метод само теоретски истражен, али његова имплементација није спроведена, из разлога што се генераторске функције свакако много ређе користе него обичне. Уколико буде покушано додавање тачке бележења у оквиру генераторске функције, проширење за манипулацију бајткодом ће пријавити одговарајућу грешку.

* 1. Табела линија

Један мали, али неизоставни део манипулације бајткода јесте ажурирање табеле линија, која садржи информације о пресликавању бајткода на линије изворног кода. Ова табела, поред тога што пружа нама могућност да корисникову команду која захтева да се на одређену линију одређене датотеке изворног кода дода тачка бележења пресликамо у конкретно место у бајткоду које је неопходно изменити како би се та тачка бележења поставила, такође омогућава кориснику да види на којој се тачно линији десио неки изузетак у коду. Уколико не мењамо ову табелу приликом додавања тачке бележења, она ће остати неажурна и корисник ће добијати нетачне информације о томе на којој линији се десио изузетак, па је из тог разлога изузетно важна.

Формат табеле линија мењао се између верзија 3.9 и 3.10, као и између верзија 3.10 и 3.11 *CPython* интепретера. У верзији 3.10, ова табела се чува у *co\_linetable* пољу објекта кода и садржи парове од по два бајта који представљају својеврсне команде. Интерпретирање ове табеле се врши тако што почнемо од помераја у бајткоду који је 0, линији изворног кода која се чита из *co\_firstlineno* поља објекта кода, и читамо два по два бајта команди из табеле линија. Први бајт команде нам говори за колико бајтова треба да се померимо у бајткоду, а други члан за колико линија. Након сваке команде, ми знамо да се број линије који смо добили пресликава у померај у бајткоду који смо добили, и на тај начин се извршава пресликавање између та два. Специјално, линијски померај са означеном вредношћу -128 означава да на том месту стоји бајткод који се не пресликава ни у једну линију.

Приликом манипулације бајткода методом убацивања, наш задатак је релативно једноставан: при сваком убацивању је потребно да пронађемо команду уз линију где се налази бајткод који мењамо, и у њој повећамо померај у бајткоду за онолико бајтова колико смо додали на тој линији. При манипулацији методом замене инструкције долази до проблема, јер је неопходно да на крају целог кода асоцирамо одређени део бајткода са одређеном претходном линијом. Иако су негативни помераји у бајткоду дозвољени у командама, овај проблем није лак, те се унутар *Google Cloud Debugger* приликом примене ове методе табела линија не мења уопште. То значи да, уколико се деси изузетак у извршавању инструкција које смо заменили приликом додавања тачке бележења, интерпретер неће знати на којој линији се тај изузетак десио. Ипак, ово обично не прави проблем јер се ова метода не примењује често, а кад се примењује онда се не дешава често да инструкције које замењује могу да изазову изузетке.

1. Резултати

У овом поглављу дати су примери на којима развијени дебагер успешно додаје тачке бележења без обзира на њихову комплексност, и осврнули смо се на перформансе поменуте у уводном поглављу, доказујући да је имплементирани дебагер ипак у том погледу бољи од класичних.

* 1. Комплекснији пример манипулације бајткода

У овом примеру је приказан начин манипулације бајткода на примеру *POST /order* приступне тачке у нашој апликацији за тестирање. Исечак њеног изворног кода приказан је у табели 5.1.1. Када се ова функција дисасемблира уграђеним *dis* модулом, добије се излаз чији је исечак приказан у табели 5.1.2. У табелама 5.1.3 и 5.1.4 можемо видети шта се догађа када се у тај код додају тачке бележења.

Дисасемблирани код је приказан тако да се у крајњој левој колони приказује број линије на којој се налази инструкција, у следећој се индикатором >> показује да ли се на то место скаче из неке друге инструкције, затим је приказан померај дате инструкције у односу на почетак објекта кода (у бајтовима, с тим што је свака инструкција два бајта), симболички назив инструкције, аргумент инструкције и у загради поред аргумента значење тог аргумента (у контексту инструкције). Ради читљивости, новододате инструкције означене су **подебљаним текстом**, док су измењене означене **подебљаним и подвученим текстом**.

Између табела 5.1.2 и 5.1.3 можемо видети неколико разлика. На линији 50 уметнута је тачка бележења са идентификатором 1 која се састоји од поменутих 5 инструкција, померајући све инструкције испод себе. Као резултат тога, инструкција *POP\_JUMP\_IF\_FALSE* (која се користи за скакање на *else* грану) је добила једну *EXTENDED\_ARG* инструкцију изнад себе, јер се сада тачка на коју скаче налази ниже, и више није могуће њену дужину забележити у једном бајту. Инструкције *FOR\_ITER* и *JUMP\_ABSOLUTE* (за итерирање по *for* петљи) су исто тако морале да помере своје адресе скока. Приметимо како су инструкције *POP\_JUMP\_IF\_FALSE* и *JUMP\_ABSOLUTE* инструкције апсолутног скока, док је инструкција *FOR\_ITER* инструкција релативног скока*.* У табели 5.1.4 исто тако можемо видети како је након додавања 5 тачки бележења инструкција *FOR\_ITER* морала да добије своју *EXTENDED\_ARG* инструкцију.

Табела 5.1.1. Изворни код функције која обрађује захтеве на *POST /order* приступној тачки.

|  |
| --- |
| @app.route('/order', methods=['POST'])  def order():  if int(request.args.get('a', 123)) % 2 == 0:  for i in range(10):  print('A')  print('A')  # ... исечак избачен ...  print('A')  print('A')  else:  print('B')  print('C')  return Response(status=HTTPStatus.CREATED) |

Табела 5.1.2. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки.

|  |
| --- |
| 20 0 LOAD\_GLOBAL 0 (int)  2 LOAD\_GLOBAL 1 (request)  4 LOAD\_ATTR 2 (args)  6 LOAD\_METHOD 3 (get)  8 LOAD\_CONST 1 ('a')  10 LOAD\_CONST 2 (123)  12 CALL\_METHOD 2  14 CALL\_FUNCTION 1  16 LOAD\_CONST 3 (2)  18 BINARY\_MODULO  20 LOAD\_CONST 4 (0)  22 COMPARE\_OP 2 (==)  24 **POP\_JUMP\_IF\_FALSE 253 (to 506)**  21 26 LOAD\_GLOBAL 4 (range)  28 LOAD\_CONST 5 (10)  30 CALL\_FUNCTION 1  32 GET\_ITER  >> 34 **FOR\_ITER 234 (to 504)**  36 STORE\_FAST 0 (i)  22 38 LOAD\_GLOBAL 5 (print)  40 LOAD\_CONST 6 ('A')  42 CALL\_FUNCTION 1  44 POP\_TOP  ... исечен део кода ...  79 494 LOAD\_GLOBAL 5 (print)  496 LOAD\_CONST 6 ('A')  498 CALL\_FUNCTION 1  500 POP\_TOP  502 **JUMP\_ABSOLUTE 17 (to 34)**  21 >> 504 JUMP\_FORWARD 4 (to 514)  81 >> 506 LOAD\_GLOBAL 5 (print)  508 LOAD\_CONST 7 ('B')  510 CALL\_FUNCTION 1  512 POP\_TOP  82 >> 514 LOAD\_GLOBAL 5 (print)  516 LOAD\_CONST 8 ('C')  518 CALL\_FUNCTION 1  520 POP\_TOP  83 522 LOAD\_GLOBAL 6 (Response)  524 LOAD\_GLOBAL 7 (HTTPStatus)  526 LOAD\_ATTR 8 (CREATED)  528 LOAD\_CONST 9 (('status',))  530 CALL\_FUNCTION\_KW 1  532 RETURN\_VALUE |

Табела 5.1.3. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки након додавања једне тачке бележења.

|  |
| --- |
| ... исечен део кода ...  24 **EXTENDED\_ARG 1**  26 **POP\_JUMP\_IF\_FALSE 259 (to 518)**  21 28 LOAD\_GLOBAL 4 (range)  30 LOAD\_CONST 5 (10)  32 CALL\_FUNCTION 1  34 GET\_ITER  >> 36 **FOR\_ITER 239 (to 516)**  38 STORE\_FAST 0 (i)  ... исечен део кода ...  50 **264 LOAD\_GLOBAL 9 (cloudebug)**  **266 LOAD\_METHOD 10 (breakpoint\_callback)**  **268 LOAD\_CONST 10 (1)**  **270 CALL\_METHOD 1**  **272 POP\_TOP**  274 LOAD\_GLOBAL 5 (print)  276 LOAD\_CONST 6 ('A')  278 CALL\_FUNCTION 1  280 POP\_TOP  ... исечен део кода ...  514 **JUMP\_ABSOLUTE 18 (to 36)** |

Табела 5.1.4. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки након додавања пет тачки бележења.

|  |
| --- |
| ... исечен део кода ...  24 EXTENDED\_ARG 1  26 **POP\_JUMP\_IF\_FALSE 280 (to 560)**  21 28 LOAD\_GLOBAL 4 (range)  30 LOAD\_CONST 5 (10)  32 CALL\_FUNCTION 1  34 GET\_ITER  >> 36 **EXTENDED\_ARG 1**  38 **FOR\_ITER 259 (to 558)**  40 STORE\_FAST 0 (i)  ... исечен део кода ... |

* 1. Перформансе

Како смо на почетку рада поменули неадекватност осталих дебагера за ове потребе, ове претпоставке је потребно тестирати, као и евалуирати перформансе нашег имплементираног система.

У табели 5.2.1 изложени су резултати тестирања у виду просечног броја захтева по секунди који је наша апликација могла да опслужи. Веб сервер коришћен за сервирање захтева јесте развојни *Flask* сервер, а тестирано је на две крајње тачке: *POST /order* (чији се изворни код може наћи у одељку изнад), и *GET /*, која садржи једну линију која враћа JSON одговор кориснику.

Тестирање је извршено коришћењем алата *wrk* [39], покренутим са 8 нити, 400 истовремених конекција и 30 секунди тестирања. Окружење у којем је тестирано јесте *Arch Linux* оперативни систем без покренутог графичког окружења и минималним бројем сервиса који могу да интерферирају у процесу тестирања, осим за последња два теста где је за потребе тестирања покренуто графичко окружење и у њему *Visual Studio Code*. Процесор машине на којој је вршено тестирање јесте *AMD Ryzen 7 3700U*, са 4 физичка и 8 логичких језгара.

Табела 5.2.1. Резултати тестирања перформанси изражени у просечном броју опслужених захтева у секунди.

|  |  |  |
| --- | --- | --- |
| **Тест** | ***GET /*** | ***POST /order*** |
| Без дебаговања | 989.45 | 86.03 |
| Наш дебагер | 977.34 | 88.68 |
| *pdb* дебагер | 821.20 | 83.15 |
| Наш дебагер са једном тачком бележења | 489.13 | 53.57 |
| Наш дебагер са једном тачком бележења и једним изразом | 462.23 | 51.30 |
| Наш дебагер са једном тачком бележења и једном конекцијом | 504.37 | 52.41 |
| Наш дебагер са пет тачки бележења | - | 16.91 |
| *VS Code* дебагер (са графичким окружењем) | 190.87 | 36.54 |
| Наш дебагер (са графичким окружењем) | 503.26 | 46.47 |

Неопходно је поменути како је у другом и трећем тесту постојала једна тачка бележења, односно један *breakpoint*, у крајњој тачки која се није позивала. Седми тест нема резултате за *GET /* крајњу тачку, јер нема довољно линија за постављање пет тачака бележења.

Из овога потврђујемо да класични дебагери нису погодни за употребу у продукционим сервисима због својих лоших перформанси. Ипак, уколико се место које дебагујемо јако често позива у неком сервису, можемо видети да наш дебагер може да преполови перформансе тог дела кода, па је потребно додатно дорађивање у том погледу.

1. Закључак

Фокус овог рада јесу тзв. *developer observability* алати који су се појавили у последњих пар година, који се фокусирају на балансирање између могућности коришћења техника логовања у продукционом коду и корисности дебагера као алата за отклањање грешака, тиме што покушавају да дебагере доведу у продукционе сервисе са посебним освртом на њихове перформансе. Они се углавном заснивају на техникама манипулације бајткода за одговарајуће окружење у којем се продукциони код покреће. Постоји неколико оваквих алата већ развијених, попут *Lightrun*, *Rookout*, *Google Cloud Debugger* и осталих, са разликама у могућностима које нам пружају, отворености кода, и цени по којој пружају своје услуге.

У оквиру овог рада истражили смо могућност отклањања грешака на нивоу продукционог кода алатима који су налик дебагерима које користимо у нашим развојним окружењима. Написано је C++ проширење за *CPython* интерпретер који интерпретира верзију 3.10 *Python* програмског језика ради манипулације бајткода, око ње је написан одговарајући код за комуникацију са програмером, и са програмерове стране развијено је проширење за окружење *Visual Studio Code* која служи као графички кориснички интерфејс за дебагер који се покреће у оквиру продукционог сервиса. Овај систем доступан је јавно као софтвер отвореног кода [40].

Оно што није урађено у овом раду јесте друга метода манипулације бајткодом која ради на одређеним случајевима генераторских функција, и нису истражене нове методе манипулације бајткодом које би потенцијално решиле граничне случајеве у досадашњим методама. Перформансе, иако показане као поприлично задовољавајуће, се такође могу побољшати. Из мерења изнад можемо видети како једна тачка бележења уме да успори код у којем се налази и за 50%, те иако генерално не успорава остатак кода може бити проблем за често коришћене функционалности. Проширење развијено за потребе манипулације бајткодом тренутно врши само ту функцију, али је могуће цео пакет имплементирати изнова у коду проширења, што би сигурно на неки начин побољшало перформансе. На уска грла у перформансама није обраћена много велика пажња, али се претпоставља да успорења у односу на код без тачки бележења углавном долазе из самог кода за комуникацију, те је могуће имплементирати и алтернативне, једноставније, методе комуникације (на пример, *WebSocket* сервер у одвојеном процесу, који са постојећим комуницира преко *Unix* прикључница).

Даља истраживања могу бити на тему разрешавања досадашњих недоумица у манипулацији бајткодом унутар генераторских функција (као алтернативе методе замене инструкција која не ради у свим случајевима), побољшања перформанси целог система, истраживање додатних функционалности имплементираних у постојећим алатима за ове сврхе (сакривање поверљивих информација, боља комуникациона архитектура, профајлирање и аналитичке тачке бележења, и тако даље), као и смишљање систематичнијег начина тестирања ових дебагера.

Литература

1. *Home - Lightrun* [Online]. Available: <https://lightrun.com/> (07.08.2023.)
2. *Pricing - Lightrun* [Online]. Available: <https://lightrun.com/pricing/> (07.08.2023.)
3. *Cloud-Native* *Debugging* *Tools | Rookout* [Online]. Available: <https://www.rookout.com/> (07.08.2023.)
4. *Pricing | Rookout* [Online]. Available: <https://www.rookout.com/pricing/> (07.08.2023.)
5. *FusionReactor Java APM* [Online]. Available: <https://www.fusion-reactor.com/> (07.08.2023.)
6. *FusionReactor APM Pricing, Monthly, Annual and on-demand* [Online]. Available: <https://www.fusion-reactor.com/apm-pricing/> (07.08.2023.)
7. *Debug live ASP.NET Azure apps* [Online]. Available: <https://learn.microsoft.com/en-us/visualstudio/debugger/debug-live-azure-applications> (07.08.2023.)
8. *Continuous feedback platform for developers | Source++* [Online]. Available: <https://sourceplus.plus/> (07.08.2023.)
9. *Plan & Pricing | Source++* [Online]. Available: <https://sourceplus.plus/pricing> (07.08.2023.)
10. *Python Cloud Debugger* [Online]. Available: <https://github.com/GoogleCloudPlatform/cloud-debug-python> (07.08.2023.)
11. *Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron* [Online]. Available: <https://www.electronjs.org/> (07.08.2023.)
12. *Visual Studio Code - Code editing. Redefined.* [Online]. Available: <https://code.visualstudio.com/> (07.08.2023.)
13. *Node.js* [Online]. Available: <https://nodejs.org/> (07.08.2023.)
14. *package.json | npm Docs* [Online]. Available: <https://docs.npmjs.com/files/package.json/> (07.08.2023.)
15. *Contribution Points | Visual Studio Code Extension API* [Online]. Available: <https://code.visualstudio.com/api/references/contribution-points> (07.08.2023.)
16. *TypeScript: JavaScript With Syntax For Types.* [Online]. Available: <https://www.typescriptlang.org/> (07.08.2023.)
17. I. Fette *et al.*, „The WebSocket Protocol," *RFC 6455*, Децембар 2011. (<https://www.rfc-editor.org/rfc/rfc6455.html>, 08.08.2023.)
18. *SQLite Home Page* [Online]. Available: <https://sqlite.org/> (08.08.2023.)
19. *JSON* [Online]. Available: <https://www.json.org/> (08.08.2023.)
20. *threading — Thread-based parallelism* [Online]. Available: <https://docs.python.org/3/library/threading.html> (08.08.2023.)
21. *asyncio — Asynchronous I/O* [Online]. Available: <https://docs.python.org/3/library/asyncio.html> (08.08.2023.)
22. *multiprocessing — Process-based parallelism* [Online]. Available: <https://docs.python.org/3/library/multiprocessing.html> (08.08.2023.)
23. *Glossary — global interpreter lock* [Online]. Available: <https://docs.python.org/3/glossary.html#term-global-interpreter-lock> (08.08.2023.)
24. *pdb — The Python Debugger* [Online]. Available: <https://docs.python.org/3/library/pdb.html> (08.08.2023.)
25. *sys — System-specific parameters and functions* [Online]. Available: <https://docs.python.org/3/library/sys.html#sys.settrace> (08.08.2023.)
26. L. Haimovitch, „Building a Non-Breaking Breakpoint for Python Debugging," *Opensource.com*, Август 2019. (<https://opensource.com/article/19/8/debug-python>, 08.08.2023.)
27. *Extending Python with C or C++* [Online]. Available: <https://docs.python.org/3/extending/extending.html> (08.08.2023.)
28. *runsidekick/sidekick: Sidekick is no longer in service* [Online]. Available: <https://github.com/runsidekick/sidekick> (08.08.2023.)
29. *python/cpython: The Python programming language* [Online]. Available: <https://github.com/python/cpython> (09.08.2023.)
30. *Alternative Python Implementations* [Online]. Available: <https://www.python.org/download/alternatives/> (09.08.2023.)
31. *dis — Disassembler for Python bytecode* [Online]. Available: <https://docs.python.org/3.10/library/dis.html> (09.08.2023.)
32. B. Warsaw, „PEP 3147 - PYC Repository Directories," *peps.python.org*, Децембар 2009. (<https://peps.python.org/pep-3147/>, 09.08.2023.)
33. *The Bytecode Interpreter* [Online]. Available: <https://devguide.python.org/internals/interpreter/> (09.08.2023.)
34. *Generators - Python Wiki* [Online]. Available: <https://wiki.python.org/moin/Generators> (09.08.2023.)
35. *Tracepoints (Debugging with GDB)* [Online]. Available: <https://sourceware.org/gdb/onlinedocs/gdb/Tracepoints.html> (10.08.2023.)
36. S. Almog, „How observability is redefining the roles of developers," *The Overflow*, Јул 2022. (<https://stackoverflow.blog/2022/07/18/how-observability-is-redefining-the-roles-of-developers/>, 10.08.2023.)
37. Y. Cui, „Save money on Serverless: common costly mistakes and how to avoid them," *Lumigo*, Август 2023. (<https://lumigo.io/blog/save-money-on-serverless-common-costly-mistakes-and-how-to-avoid-them/>, 10.08.2023.)
38. *Built-in Functions — eval()* [Online]. Available: <https://docs.python.org/3/library/functions.html#eval> (17.08.2023.)
39. W. Glozer „wg/wrk: Modern HTTP benchmarking tool," *GitHub.com*, Март 2012. (<https://github.com/wg/wrk>, 19.08.2023.)
40. Л. Симић, „KockaAdmiralac/Cloudebug: Debugging in the cloud - my B.Sc. graduation work," *GitHub.com*, Август 2023. (<https://github.com/KockaAdmiralac/Cloudebug>, 19.08.2023.)
41. Roberts, P., „What the heck is the event loop anyway?" *JSConf EU*, Октобар 2014. (<https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html>, 19.08.2023.)
42. *Conditions (Debugging with GDB)* [Online]. Available: <https://sourceware.org/gdb/onlinedocs/gdb/Conditions.html> (29.08.2023)
43. *Expressions in the Visual Studio debugger* [Online]. Available: <https://learn.microsoft.com/en-us/visualstudio/debugger/expressions-in-the-debugger> (29.08.2023)
44. Aggarwal, S.K., Kumar, M.S., „Debuggers for Programming Languages," *The Compiler Design Handbook*, pp. 295-327, 2002.
45. Rosenberg, J.B., *How debuggers work*, Wiley, 1996.

Списак скраћеница

|  |  |
| --- | --- |
| JSON | *JavaScript Object Notation* |

Списак слика

[Слика 3.1.1. Архитектура једног дебагера продукционих система. Легенда: жуто (светло сиво): чвор идејног система црвено (тамно сиво): софтверска компонента која није у склопу завршног рада зелено (средње сиво са црним текстом): софтверска компонента која јесте у склопу завршног рада плаво (средње сиво са белим текстом): софтверска компонента која није у склопу завршног рада али ће бити развијена ради тестирања 8](#_Toc144219754)

[Слика 3.2.2.1. Изглед развојног окружења *Visual Studio Code* са проширењем за дебаговање продукционог кода. 10](#_Toc144219755)

[Слика 3.2.2.2. Леви панел проширења *Visual Studio Code* за дебаговање продукционог кода. 11](#_Toc144219756)

[Слика 3.2.2.3. Приказ менија за додавање/уклањање тачке бележења у проширењу за *Visual Studio Code*. 12](#_Toc144219757)

[Слика 3.2.2.4. Додавање тачке бележења у проширењу за *Visual Studio Code*. Унос услова бележења (горе) и израза за израчунавање у току бележења (доле). 12](#_Toc144219758)

[Слика 3.2.2.5. Процедура повезивања на сервер за дебаговање продукционог сервиса кроз проширење за *Visual Studio Code*. Бирање опције за повезивање (горе), уношење адресе сервера (средина) и уношење лозинке за повезивање (доле). 13](#_Toc144219759)

Списак табела

[Табела 2.1. Преглед карактеристика постојећих дебагера као и дебагера имплементираног у овом раду. 5](#_Toc144219766)

[Табела 5.1.1. Изворни код функције која обрађује захтеве на *POST /order* приступној тачки. 22](#_Toc144219767)

[Табела 5.1.2. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки. 22](#_Toc144219768)

[Табела 5.1.3. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки након додавања једне тачке бележења. 23](#_Toc144219769)

[Табела 5.1.4. Дисасемблирани код функције која обрађује захтеве на *POST /order* приступној тачки након додавања пет тачки бележења. 24](#_Toc144219770)

[Табела 5.2.1. Резултати тестирања перформанси изражени у просечном броју опслужених захтева у секунди. 24](#_Toc144219771)