VILNIAUS UNIVERSITETAS
ŠIAULIŲ AKADEMIJA

INFORMACINIŲ TECHNOLOGIJŲ VALDYMO MAGISTRO STUDIJŲ PROGRAMA

DŽIUGAS MOLIS

Magistro studijų baigiamasis darbas

JavaScript serverio technologinių sprendimų posistemių našumo tyrimas

Darbo vadovas Asist. Dr. Liudvikas Kaklauskas

Šiauliai, 2023

Studijuojančiojo, teikiančio baigiamąjį darbą, GARANTIJA

WARRANTY of Final Thesis

Vardas, pavardė *Name, Surname*	**Džiugas Molis**
Padalinys *Faculty*	**Šiaulių akademija** ***Šiauliai Academy***
Studijų programa *Study Programme*	Informacinių technologijų valdymas Information Technology management
Darbo pavadinimas *Thesis topic*	JavaScript serverio technologinių sprendimų posistemių našumo tyrimas Performance study for JavaScript server technology solutions in runtime environments
Darbo tipas *Thesis type*	**Baigiamasis darbas** ***Final Thesis***

Garantuoju, kad mano baigiamasis darbas yra parengtas sąžiningai ir savarankiškai, kitų asmenų indėlio į parengtą darbą nėra. Jokių neteisėtų mokėjimų už šį darbą niekam nesu mokėjęs.

Šiame darbe tiesiogiai ar netiesiogiai panaudotos kitų šaltinių citatos yra pažymėtos literatūros nuorodose.

I guarantee that my thesis is prepared in good faith and independently, there is no contribution to this work from other individuals. I have not made any illegal payments related to this work.

Quotes from other sources directly or indirectly used in this thesis, are indicated in literature references.

Aš, Džiugas Molis, pateikdamas (-a) šį darbą, patvirtinu (*pažymėti*) ☐

Embargo laikotarpis
Embargo Period

Prašau nustatyti šiam baigiamajam darbui toliau nurodytos trukmės embargo laikotarpį:
I am requesting an embargo of this thesis for the period indicated below:

☐ _____ mėnesių / *months*
(embargo laikotarpis negali viršyti 60 mėn. / *an embargo period shall not exceed 60 months*).

☐ Embargo laikotarpis nereikalingas / *no embargo requested.*

Embargo laikotarpio nustatymo priežastis / *Reason for embargo period*:

Santrauka

Dažnu atveju programuotojai renkasi populiarias technologijas kurti produktus ar spręsti iškilusioms problemos, tačiau ne visada tai yra geriausias pasirinkimas, nes tobulėjant technologijoms atsiranda geresnių sprendimų, tačiau dėl mažesnio populiarumo jos yra nenaudojamos. *JavaScript* yra populiariausia programavimo kalba naudojama internetiniuose puslapiuose ir atsiradus galimybei naudoti *JavaScript* kalbą serverio dalyje, tai palengvino ir pagreitino programinės įrangos kūrimo darbą, nes naudodamas vieną programavimo kalbą specialistas gali atlikti visus darbus serverio ir kliento architektūros programinėje įrangoje neapribodamas savo galimybių vienoje dalyje. Atsiradus pirmajai *JavaScript* serverio vykdymo aplinkai *Node.js* buvo paprasta išsirinkti vykdymo aplinką dėl mažos pasiūlos, tačiau bėgant laikui programuotojai kuria įvairias naujas technologijas – *JavaScript* vykdymo aplinkos ne išimtis. Naujų technologijų tikslas yra pralenkti rinkos lyderius ir pasiūlyti technologiją turinčią geresnes charakteristikas.

Šiame darbe buvo analizuojamos:

- *Node.js* – pati pirmoji *JavaScript* serverio vykdymo aplinka
- *Deno* – *Node.js* kūrėjo antrasis bandymas sukurti geresnę *JavaScript* serverio vykdymo aplinką su ištaisytais *Node.js* dizaino trūkumais ir patobulinta architektūra
- *Bun.js* – vykdymo aplinka, kurios kūrėjai teigia jog tai yra pati efektyviausia ir greičiausia *JavaScript* serverio vykdymo aplinka, kuri integruoja „visus“ reikalingus įrankius.

Darbo metu buvo apžvelgiama *JavaScript* programavimo kalbos galimybių apžvalga. Po galimybių apžvalgos buvo pasirenkamos populiariausios *JavaScript* vykdymo aplinkos ir atliekama minėtų vykdymo aplinkų literatūros analizė. Atlikus literatūros analizę buvo aprašomi kriterijai, įvertinamos technologijos pagal kriterijus, bei atliekamas jų palyginimas. Taip pat buvo atliktas eksperimentinis našumo tyrimas pagal iš anksto suprojektuotą modelį, charakteristikų rezultatų matavimas ir jų palyginimas. Šie tyrimo etapai buvo atlikti norint nustatyti kuri *JavaScript* vykdymo aplinka yra našiausia.

Summary

In many cases, developers choose the most popular technologies to develop products or solve problems, but this is not always the best choice because as technologies improve, better solutions emerge, but due to their lower popularity they are not used. JavaScript is the most popular programming language used in web pages and the availability of JavaScript on the server side has made the work of software development easier and faster because with one programming language a professional can do all the work on the server and client side architecture without being limited to a single part. When the first JavaScript server runtime Node.js appeared, it was easy to choose a runtime because of the small number of available ones but over time developers are creating a variety of new technologies - JavaScript runtimes are no exception. The goal of new technologies is to outperform the market leaders and offer a technology with better performance.

This paper analyses:

- Node.js - the very first JavaScript server runtime
- Deno - Node.js developer's second attempt to build a better JavaScript server runtime with fixed Node.js design flaws and improved architecture
- Bun.js - an runtime environment that the developers claim is the most efficient and fastest JavaScript server runtime environment that integrates "all" the necessary tools.

The work included an overview of the capabilities of the JavaScript programming language. After the overview of the capabilities, most popular runtime environments were selected and a literature analysis of these runtime environments was performed. The literature analysis led to the selection of the runtime environments to be analysed and the description of the criteria, the evaluation of the technologies according to the criteria, and their comparison. An experimental performance study was also carried out using a pre-designed model, measuring the performance results and comparing them. These steps of the study were carried out to determine which JavaScript execution environment has the best performance.

Turinys

Įvadas 1
1. JavaScript galimybių apžvalga 3
1.1. Internetiniai puslapiai 3
1.1.1. Statinis internetinis puslapis 4
1.1.2. Dinaminis internetinis puslapis 4
1.2. Vykdymo aplinkos ir posistemių kūrimas 5
1.2.1. JavaScript programavimo kalbos vykdymo skirtumai naršyklėje ir Node.js serveryje 5
1.3. Mobiliosios programėlės 6
1.4. Darbalaukio programos 7
2. Serverio dalies JavaScript vykdymo aplinkos 8
2.1. Node.js 9
2.1.1. Node.js privalumai 10
2.1.2. Node.js trūkumai 11
2.2. Deno 12
2.2.1. Deno privalumai 13
2.2.2. Deno trūkumai 15
2.3. Bun.js 16
2.3.1. Bun.js privalumai 17
2.3.2. Bun.js trūkumai 19
3. Tyrimo metodologija 19
3.1. Vertinimo kriterijai 20
3.1.1. Technologijos branda 21
3.1.2. Node Package Manager bibliotekų registro palaikymas ir architektūra 22
3.1.3. Optimizuotas modulių saugojimas 22
3.1.4. ECMAScript modulių importavimas kaip technologijos standartas 23
3.1.5. Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript 24
3.1.6. TypeScript palaikymas vykdymo aplinkos kompiliatoriuje 24
3.1.7. Bibliotekų teisių apribojimas kaip saugumo įrankis 25
3.1.8. Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas 25
3.2. Kriterijų įvertinimas 26
3.2.1. Technologijos branda 27
3.2.2. Node Package Manager bibliotekų registro palaikymas ir architektūra 28
3.2.3. Optimizuotas Node Package Manager modulių saugojimas 29
3.2.4. ECMAScript modulių importavimas kaip technologijos standartas 29
3.2.5. Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript 30
3.2.6. TypeScript palaikymas vykdymo aplinkos kompiliatoriuje 31
3.2.7. Bibliotekų teisių apribojimas kaip saugumo įrankis 31

3.2.8. Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas 32
3.3. Vertinimo kriterijų palyginimo rezultatai 33
4. Eksperimentinė tyrimo darbo dalis 34
4.1. Atliekami bandymai 35
4.2. Našumo charakteristikų pasirinkimas 36
4.3. Našumo charakteristikų apskaičiavimas 37
4.4. Naudojama kompiuterinė ir programinė įranga 38
4.5. Bandymai 39
4.5.1. HTTP serverio užklausos atsakymo greitis 39
4.5.2. Tekstinio failo skaitymas 40
4.5.3. SQL duomenų bazės serverio užklausos įvykdymo laikas 41
4.5.4. Skaičiavimams imli užduotis 42
4.5.5. Bibliotekos įdiegimo trukmė 43
5. Bandymų rezultatai 44
5.1. HTTP serverio bandymo rezultatai 44
5.2. Tekstinio failo skaitymo bandymo rezultatai 46
5.3. SQL duomenų bazės serverio užklausos įvykdymo laikas 49
5.4. Skaičiavimams imlios užduoties rezultatai 51
5.5. Bibliotekos įdiegimo trukmės rezultatai 53
5.6. Bandymo rezultatų išvados 54
5.7. Bandymų rezultatų suvestinė lentelėje 55
Rezultatai ir išvados 58
Rekomendacijos 60
Šaltinių sąrašas 61
SĄVOKŲ APIBRĖŽIMAI 62
SANTRAUPOS 63
PRIEDAI 64

Įvadas

Tyrimo objektas ir aktualumas. Interneto ir informacinių technologijų naudojimas yra neatsiejama dienos dalis. Internetą kasdien naudoja milijardai žmonių ir jų kiekvienais metais sparčiai daugėja (1). Tai žmonėms palengvina atlikti įvairius veiksmus, pvz. keistis informacija tarpusavyje, pirkti ar parduoti daiktus, atsiskaityti už paslaugas. Taip pat kiekvienas verslas ar produkto kūrėjai nori turėti savo svetainę, kurioje galėtų skelbti informaciją ar parduoti savo prekes ar paslaugas, taip plėsti savo verslo pasiekiamumą ir papildomai uždirbti. Norint pritaikyti internetą tokiam naudotojų srautui reikalingos vis geresnės kompiuterinės technologijos.

Internetinį puslapį sukurti galima taikant įvairias technologijas, tačiau viena iš pagrindinių programavimo kalbų galinčių atlikti veiksmus vartotojo naršyklėje yra *JavaScript*. *JavaScript* yra naudojama kaip kliento dalies programavimo kalba, ją naudoją 98.8% visų svetainių[1]. Nors devyniasdešimtaisiais ši programavimo kalba buvo naudojama statinius interneto puslapius padarant interaktyviais dėl jos didžiulio populiarumo buvo sukurta daug įvairių įrankių, kurie leidžia nesimokant naujos programavimo kalbos sukurti įvairaus tipo programas. Dėl šios priežasties šiuo metu *JavaScript* programavimo kalba yra naudojama ne tik atlikti veiksmus vartotojo naršyklėje – ją galima naudoti kuriant serverio programą, mobiliąsias programėles ar darbalaukio programas.

Šiame darbe analizuojamas *JavaScript* serverio dalies vykdymo aplinkų našumas, siekiama atskleisti vykdymo aplinkų privalumus ir trūkumus, aiškinamasi, kurią technologiją pasirinkti norint įgyvendinti savo projektą efektyviau.

Darbo metu buvo atliekama vykdymo aplinkas nagrinėjančių mokslinių straipsnių apžvalga. Didelė dalis mokslinių straipsnių lygina populiariausią *JavaScript* serverio vykdymo aplinką *Node.js* su kitas programavimo kalbas naudojančiomis serverio programoms kurti skirtomis technologijomis, tokiomis kaip Java, C# ar PHP. Teigiama jog *Node.js* technologijos našumas yra kur kas didesnis. Straipsnyje „A comperative Analysis of Node.js (Server-Side *JavaScript*)“ buvo atliekamas našumo lyginimas tarp *Node.js* ir Apache-PHP serverių, kurio metu paaiškėjo jog dėl savo įvesties ir išvesties efektyvumo *Node.js* lenkia Apache serverį įvykdant *HTTP* užklausas greičiau (2). Taip pat buvo rasti moksliniai darbai, kurie atskirai nagrinėjo *Node.js* su *Deno* (3) ir *Node.js* su *Bun.js* (4) našumą. Šio magistro darbo metu bus lygiagrečiai lyginamos visų trijų technologijų funkcinės galimybės ir atliekami našumo bandymai.

Darbo tikslas: Įvertinti ir palyginti *JavaScript* serverio dalies vykdymo aplinkų našumą, nustatant, kurios technologijos tarpusavyje lyginant yra pranašesnės.

[1] JavaScript kliento pusės programavimo kalbos naudojimas internetiniuose puslapiuose https://w3techs.com/technologies/details/cp-javascript

Darbo uždaviniai:

1. Išanalizuoti mokslinę literatūrą apie *JavaScript* galimybes ir atrinkti tris populiariausias *JavaScript* vykdymo aplinkas analizei, kurios yra naudojamos serverio programavimui;
2. Remiantis mokslinės literatūros analize, parinkti *JavaScript* vykdymo aplinkos vertinimo kriterijus;
3. Suprojektuoti *JavaScript* vykdymo aplinkų eksperimentinio tyrimo modelį vykdymo našumo analizei pagal atrinktus vertinimo kriterijus;
4. Išanalizuoti ir įvertinti gautus rezultatus.

Darbe naudojama metodika:

- Mokslinės literatūros analizė
- Eksperimentinis tyrimas
- Palyginimo metodas

1. JavaScript galimybių apžvalga

JavaScript yra prototipų pagrindu grįsta, objektiškai orientuota, ne griežtus tipus naudojanti dinaminė kliento pusės programavimo kalba (5). Ji yra kuriama pagal *ECMAScript* (6) programavimo kalbos standartą. *JavaScript* yra naudojama kaip žiniatinklio programavimo (scenarijų rašymo) kalba, kuri leidžia kurti dinaminius puslapius. Ši programavimo kalba leidžia atnaujinti turinį neperkraunant internetinio puslapio, o tai suteikia galimybę rodyti turinio atnaujinimus, interaktyvius žemėlapius, įvairiais animacijas/grafikas. Visas dinaminis turinys, kurį matome internetiniame puslapyje yra atliekamas *JavaScript* pagalba.

Ši programavimo kalba buvo sukurta po interneto (World Wide Web) atsiradimo ir įdiegta tuo metu populiariausioje „Netscape Navigator“ naršyklėje (5). Ji gana greitai buvo pritaikyta naršyklės „Internet Explorer“ naudojimui. Naudojimo paprastumas ir priežastis jog tai buvo vienintelė tuo metu egzistuojanti kliento scenarijų kalba leidžianti turinį keisti dinamiškai lėmė, kad *JavaScript* labai greitai išpopuliarės tarp internetinių puslapių programuotojų (5).

Šios kalbos buvimas pirmąja scenarijų kalba naršyklėse prisidėjo prie įspūdingo jos populiarumo, taip pat visos šiuolaikinės naršyklės, staliniai, nešiojamieji kompiuteriai, planšetės, televizoriai ir telefonai palaiko *JavaScript* programavimo kalbą, o tai leidžia teigti, kad *JavaScript* yra labiausiai naudojama programavimo kalba (5). Kadangi ši kalba taip plačiai naudojama, programuotojai praplėtė jos galimybes ir pritaikė ją, kuriant serverio, darbalaukio, mobiliąsias programas ir taip pat programuojant daiktų interneto įrenginius. Tai leidžia kurti įvairaus tipo programas nesimokant kitų programavimo kalbų, tiesiog pritaikant *JavaScript* programavimo kalbos žinias.

Originaliai *JavaScript* turėjo veikti tik interneto naršyklėje. Šiuo metu tai ir tebėra dažniausia šios programavimo kalbos vykdymo vieta, tačiau 2009 metais išleidus *Node.js* tai išplėtė *JavaScript* panaudojimo vietas. *Node.js* leido ne tik naudotis įprastomis *JavaScript* programavimo kalbos funkcijomis, bet skaityti ir kurti failus, siųsti ir gauti informaciją internetu, priimti ir siųsti *HTTP* užklausas, pasiekti operacinės sistemos funkcijas. *Node.js* technologija leidžia nesudėtingai sukurti *HTTP* serverį ir kurti scenarijų programas kaip alternatyvą terminalo scenarijams (7).

1.1. Internetiniai puslapiai

Iš pradžių internetiniai puslapiai buvo kaip statinių duomenų saugyklą, kuri buvo visiškai neapibrėžta. Žmonės informaciją sukeldavo į interneto puslapius taip iš daugelio atskirų internetinių puslapių sudarydami vieną didelį internetinį puslapį. Tai buvo tinkama naršyti nedidelius informacijos rinkinius, tačiau bėgant laikus šis modelis žlunga, nes technologijos

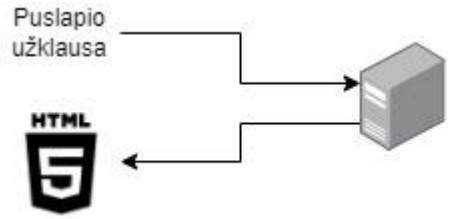
tobulėja ir atsirado galimybė dinamiškai generuoti informaciją.

Kuriant internetinius puslapius, tai galima atlikti ne vienu būdu. Rinkoje vyrauja dviejų tipų internetiniai puslapiai: statiniai ir dinaminiai. Statinių internetinių puslapių pagrindas sudaro *HTML* ir *CSS* technologijos. Dinaminiai puslapiai šalia šių technologijų naudoja ir *JavaScript* programavimo kalbą.

1.1.1. Statinis internetinis puslapis

Statiniai internetiniai puslapiai susideda iš *HTML* žymų bei *CSS* pakopinių stilių. *HTML* aprašo internetinio puslapio elementus, o *CSS* suteikia jiems dydį, spalvą, kitaip sakant pakeičia naršyklės nustatytą pagal nutylėjimą žymės išvaizdą. Šie puslapiai negali būti keičiami kliento, t. y. jų informacija pasikeičia tik tuomet kai puslapis yra įkeliamas iš naujo. Dėl šios priežasties norint naviguoti po interneto puslapius tai užtrunka daug laiko nes kiekvienas puslapis turi būti įkeliamas iš naujo (8).

Šis internetinių puslapių tipas yra puikiai tinkamas pradedantiesiems programuotojams, nes yra nesudėtinga suprasti puslapių struktūrą. Statiniai puslapiai suteikia puikias sąlygas internetinėms paieškos sistemoms (vykdant SEO) lengvai nuskaityti internetinį puslapį, nes jo turinys nesikeičia nuo pat puslapio užkrovimo.



1 pav. Statinio internetinio puslapio duomenų gavimas

1.1.2. Dinaminis internetinis puslapis

Dinaminiai tinklapiai yra kuriami aprašant kliento pusės scenarijus, kai internetinis puslapis neprivalo būti perkraunamas, jo turinys atsinaujina be internetinio puslapio perkrovimo. Šie puslapiai gali būti keičiami atsižvelgiant į naudotojo veiksmus, tokius kaip pelės, klaviatūros paspaudimas ar būsenas tokias kaip vartotojo prisijungimo statusas, pirkinių krepšelio turinys, prekių kiekio likutis sandėlyje (8).

Nors *JavaScript* yra dažniausiai naudojama programavimo kalba šio tipo puslapiams kurti, bet tai galima atlikti ir kitomis programavimo kalbomis. Tokios programavimo kalbos kaip Dart, Python gali būti kompiliuojamos į *JavaScript* programavimo kalbą taip pritaikant jas dinaminių

internetinių puslapių kūrimui.

1.2. Vykdymo aplinkos ir posistemių kūrimas

JavaScript programavimo kalba įgijo gana nemažą rolę ir serverių kūrimo technologijose. Įvairios šios kalbos bibliotekos praplėtė *JavaScript* galimybes vykdyti *JavaScript* kodą ne naršyklėje, o tai įgalino kurti serverio programas naudojant *JavaScript* programavimo kalbą. Šis įvairialypis technologijos pritaikomumas leidžia sukurti pilnavertį produktą pradedant vartotojo sąsaja baigiant serveriu, vien naudojant šią programavimo kalbą. Tai leidžia programuotojams savarankiškai spręsti daugiau problemų, nes sumažėja priklausomybių nuo kitų technologijų išmanymo, išvengiama specifinių technologijų programuotojų samdymo proceso, produkto kūrimas tampa pigesnis bei greitesnis (9).

Pati pirmoji tai įgalinusi technologija – 2009 metais išleista *Node.js* (10). *JavaScript* suteikė galimybę programuotojams, kurie išmano šia programavimo kalbą, kurti praktiškai visų tipų programėles (mobiliąsias, darbalaukio, serverio).

Node.js dar labiau palengvino programuotojų gyvenimą, nes kartu su šia technologija buvo pristatytas *NPM*, kuris tapo didžiausiu pasaulyje programinių bibliotekų registru leidusiu programuotojams dalintis įvairiomis bibliotekomis (11). Galimybė pasinaudoti dažnas problemas sprendžiančiu kodu sumažina programavimo laiką, kurio reiktų rašant kodą nuo nulio – programuotojas gali pasiimti jau sukurtą biblioteką ir ją naudoti.

JavaScript plėtros intensyvumas sužadino naujų serverio kūrimo technologijų atsiradimą ir konkuravimą tarpusavyje. Nuo *Node.js* išleidimo jo konkurentų sąrašo papildė tokios technologijos kaip *Deno, Bun.js, Blueboat, Napa.js, ChakraCore, WasmEdge, JerryScript*. Visos šios technologijos bando būti viena už kitą pranašesnės bei įgyti populiarumą, dėl *Node.js* technologijos išleidimo pirmumo, visos šios technologijos remiasi *Node.js* modulių ir bibliotekų infrastruktūra (11).

1.2.1. JavaScript programavimo kalbos vykdymo skirtumai naršyklėje ir Node.js serveryje

Node.js kaip savo technologijos programavimo kalbą naudoti pasirinko *JavaScript*. Šioje technologijoje naudojama programavimo kalba buvo pritaikyta taip, kad ji palaikytų visas įprastas *JavaScript* kalbos konstrukcijas, tačiau yra skirtumų tarp *JavaScript* naudojimo naršyklėje ir *Node.js*. *Node.js* technologiją galime laikyti kaip papildomą sluoksnį virš *JavaScript*. Tai turi papildomo funkcionalumo, kurio paprasta *JavaScript* kalba naršyklėje neturi (pvz. failų sistemos

pasiekiamumas, operacinės sistemos funkcijų prieiga). Kadangi ši technologija skirta kurti serveriams, išskirtinis dėmesys buvo skiriamas pagerinti kuriamas serverių aplikacijas.

Be papildomo pridėto funkcionalumo *Node.js* kitaip pažvelgė į sistemos modulių pridėjimą. Jeigu lygintume naršyklės *HTML* puslapio veikimą su *Node.js* kai norime įsidiegti papildomą biblioteką į projektą, tai darytume visiškai skirtingais būdais. *HTML* puslapyje panaudotumėme „script“ žymą, kuri leidžia įdėti failą ar biblioteką, pvz. kaip įsidėti Tailwind *CSS* biblioteką:

```
<script src="https://cdn.tailwindcss.com"></script>
```

2 pav. Tailwind bibliotekos importavimo pavyzdys naudojant „script“ žymą

Tačiau taip importuojant failus juos gali matyti visi kas peržiūri *HTML* puslapį ir toks importavimo būdas yra nepatogus ir gali sukelti problemų didėjant projektui. *Node.js* pritaikė kitokį failų ir bibliotekų importavimą, kuris sukelia mažiau problemų nepriklausomai nuo projekto dydžio. Norint tai atlikti reikia panaudoti **require** funkciją:

```
var modulex = require ('module_name');
```

3 pav. Node.js bibliotekos importavimo pavyzdys

Globalus kintamasis interneto naršyklėje vadinamas **window**. Šiame objekto kintamajame galima išsaugoti įvairias reikšmes, kurios bus pasiekiamos programiniame kode iš visų aplikacijos dalių. *Node.js* turi 2 globalius kintamuosius: **global** ir **process**. **global** – **window** alternatyva serverio dalyje. Bet kuri reikšmė išsaugota šiame kintamajame yra pasiekiama visur programiniame kode. **process** – naudojamas saugoti duomenis, kurie yra reikalingi programos vykdymo kontekste, pvz. įvairūs sisteminiai parametrai.

Svarbu naudojant *JavaScript* reikia suprasti kokioje aplinkoje jis bus vykdomas, nes funkcionalumas gali skirtis. Tai gali sukelti programos vykdymo klaidas.

1.3. Mobiliosios programėlės

Dar viena sritis kur *JavaScript* programavimo kalba yra naudojama – mobiliųjų programėlių kūrimas. Šiuo metu populiariausios ir labiausiai prieinamos yra dvi mobiliosios operacinės sistemos: Android ir iOS[2]. *JavaScript* karkasai leidžia programuoti ir kurti mobiliąsias programėles šioms dviem operacinėms sistemoms iš vienos kodo bazės (12).

Tokio tipo technologijos *JavaScript* naudoja kaip aukščiausio lygio programavimo kalbą, o

[2] Mobiliųjų operacinių sistemų rinkos dalis visame pasaulyje https://gs.statcounter.com/os-market-share/mobile/worldwide

žemesniuose lygiuose technologijos naudoja konkrečiai operacinei sistemai pritaikytas bibliotekų kolekcijas, kurios turi iš anksto parašytą kodą kiekvienai operacinei sistemai. Tai yra puikus pasirinkimas išvengiant dvigubos kainos kuriant produktą, dėl didžiulės bendruomenės palaikomo ekosistemos prieinamumo, bendro programavimo kalbos populiarumo. Taip pat *JavaScript* programuotojai, kurie yra reikalingi produktams kurti yra lengviau randami nei specifinės technologijos skirtos *iOS* ir *Android* programėlių kūrimui, nes tokios technologijos kaip *Swift, Objective-C, Java, Kotlin* dėl savo siauresnio panaudojimo turi mažesnį šių technologijų programuotojų bendruomenę.

JavaScript mobiliųjų programėlių kūrimo technologijos gali nepalaikyti pilno siūlomo mobiliųjų operacinių sistemų funkcionalumo spektro. Kuriant šio tipo programėles svarbu įvertinti būsimas mobiliosios programėlės funkcijas ir patikrinti ar būtiną funkcionalumą bus įmanoma įgyvendinti naudojant konkrečią technologiją.

Dėl didelio kiekio šios kalbos specialistų atsiranda daug idėjų ir iš jų gimsta įvairių *JavaScript* karkasų tokių kaip: *Apache Cordova, React Native, Mobile Angular UI, jQuery Mobile*. Jie bando tarpusavyje konkuruoti ir būti vienas už kitą pranašesniais. Pigesnė kūrimo kaina, viena kodo bazė leidžia tokioms technologijoms tapti populiariomis ir naudojamomis.

1.4. Darbalaukio programos

Viena iš pirmųjų technologijų leidusi pasitelkti *JavaScript* darbalaukio programoms kurti yra Electron.js. Ji apjungia *HTML5* ir naršyklės funkcijas kartu su *Node.js* vykdymo aplinka, o tai leidžia kurti darbalaukio programas, kurios gali būti paskirstytos per įvairias operacines sistemas, tokias kaip *Windows, MacOS* ir *Linux* (13).

Dėl operacinės sistemos funkcijų *Node.js* aplinkoje gausos, nėra jokių apribojimų kuriant darbalaukio programas. *Electron.js* technologija leidžia panaudoti visas įprastai naršyklėse veikiančių internetinių puslapių technologijas kuriant darbalaukio programėles, t. y. integruoti savo mėgstamas bibliotekas, karkasus ir kitas įvaldytas *JavaScript* technologijas į savo kuriamą programėlę. Tokios technologijos kaip *React, Vue.js, Next.js, TypeScript, Webpack* gali būti prijungtos prie *Electron.js* projekto taip pat lengvai kaip prie bet kurio kito dinaminio internetinio puslapio, taip palengvinant produkto kūrimą (14).

Tai yra puikus pasirinkimas ilgametę patirtį turinčioms IT organizacijoms tiek startuoliams. Ši tarpplatformė technologija buvo pasirinkta žinomų IT organizacijų, kuriant savo produkto darbalaukio programėles *Windows, Linux* ir *MacOS* operacinėms sistemos (14). 1Password, Discord, Dropbox, Figma, GitHub Desktop, Slack, Microsoft Teams, VS Code yra puikūs pavyzdžiai kokio įvairaus pobūdžio darbalaukio programėles galima sukurti naudojant Electron.js.

2. Serverio dalies JavaScript vykdymo aplinkos

Iki *Node.js* atsiradimo įvairių posistemių kūrimas panaudojant *JavaScript* ne naršyklėje buvo tik idėja, kuri galbūt kažkada virs realybe. Ši kalba buvo naudojama išimtinai tik internetiniuose puslapiuose. Internetinių naršyklių kūrėjai įdėjo daug darbo tam, kad *JavaScript* kalbos variklis būtų pagreitintas ir vartotojui suteiktų puikią greitaveiką. Spartos tobulinimas leido panaudoti Google Chrome naršyklės V8 *JavaScript* kalbos variklį kuriant *Node.js* technologiją (15).

Iš anksčiau minėto *JavaScript* vykdymo aplinkų sąrašo analizei buvo pasirinktos šios trys:

1. Node.js – pati pirmoji ir populiariausia JavaScript serverio vykdymo aplinka naudojanti Google V8 *JavaScript* kalbos variklį ir C/C++ programavimo kalbos operacijoms su operacine sistema;
2. Deno – Node.js kūrėjo antrasis bandymas sukurti geresnę JavaScript serverio vykdymo aplinką su ištaisytais Node.js dizaino trūkumais ir patobulinta architektūra, nors ir tebenaudojamas Google V8 *JavaScript* kalbos variklį ir C/C++ programavimo kalbos operacijoms su operacine sistema;
3. Bun.js – vykdymo aplinka, kurios kūrėjai teigia jog tai yra pati efektyviausia ir greičiausia JavaScript serverio vykdymo aplinka, kuri integruoja „visus“ reikalingus įrankius. Ši technologija naudoja kitą *JavaScript* kalbos variklį – *WebKit/Safari JavaScriptCore* ir kitą programavimo kalbą *Zig* operacijoms su operacine sistema.

Šios technologijos buvo pasirinktos pagal populiarumą pagal „2023 metų dažniausiai naudojamus internetinių puslapių karkasai“ statistikos rezultatus[3], taip pat „2023 metų StackOverflow populiariausi internetinių puslapių karkasai ir technologijos“[4] ir *JavaScript* vykdymo aplinkų GitHub repozitorijų žvaigždučių(patiktukų) kiekį. Papildomai buvo pasirinktas repozitorijos žvaigždučių(patiktukų) kiekis, nes tik *Node.js* ir *Deno* buvo populiarios technologijos, kad jos panaudojimo rezultatai matytųsi statistikoje. Trečioji pagal GitHub repozitorijos žvaigždučių(patiktukų) kiekį – *Bun.js*. Šios pasirinktos trys vykdymo aplinkos atlieka tas pačias funkcijas, bet visos jos yra skirtingai suprojektuotos ir įgyvendintos. Šio darbo metu šias technologijas nagrinėsime teoriškai ir praktiškai.

Analizę pradėsime nuo *Node.js* – pačios pirmos vykdymo aplinkos, tolimesniuose skyriuose nagrinėsime *Deno* ir *Bun.js JavaScript* vykdymo aplinkas, kurios buvo sukurtos pralenkti *Node.js*

[3] 2023 metų dažniausiai naudojamus internetinių puslapių karkasai https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

[4] 2023 metų StackOverflow populiariausi internetinių puslapių karkasai ir technologijos https://survey.stackoverflow.co/2023/#section-most-popular-technologies-web-frameworks-and-technologies

technologiją.

2.1. Node.js

2009 metais Ryan Dahl išleido *Node.js* technologija. Jis įžvelgė *JavaScript* programavimo kalbos galimybes tapti serverių kūrimo programavimo kalba (9). Prieš atsirandant *Node.js* technologijai serveryje buvo naudojama kita programavimo kalba, kuri galėdavo atlikti veiksmus su serveriu. Tai reiškė jog norint turėti veikiančią serverio-kliento architektūrą, reikėjo išmanyti bent dvi skirtingas technologijas. *Node.js* leido atsisakyti dalies papildomų technologijų kuriant internetines programas, rašant kliento ir serverio dalies kodą naudojant *JavaScript*.

Node.js technologija yra skirta pagerinti ir pagreitinti produkto kūrimo procesą. Ji veikia asinchroniškai ir pagal įvykiais pagrįstą architektūrą. Šis architektūros tipas leidžia naudoti asinchronines įvesties ir išvesties operacijas, kurių pabaigos nėra laukiama. Vietoje to yra atliekamos kitos operacijos, kurių šiuo metu laukti nereikia ir jas galima apdoroti. Sulaukus blokuojančios operacijos rezultato ji grąžinama į giją tolimesniam vykdymui. Tai reiškia, kad pagrindinė gija nėra blokuojama kol laukiama kvietinių arba funkcijų rezultatų. Ši neblokuojanti architektūra leidžia suvaldyti didelius kiekius lygiagrečių užklausų naudojant vieną giją kitaip negu tradicinių serverių atveju, kai skirtingas užklausas apdoroja skirtingos gijos, pvz. Apache *HTTP* serveris. Tai pat vienos gijos naudojimas leidžia efektyviau išnaudoti resursus (15).

Vienas esminių *Node.js* privalumų – ekosistema. *Node.js* naudojimas suteikia prieigą prie didžiausio pasaulyje *JavaScript* bibliotekų registro turinčio virš 2 milijonų bibliotekų (11). Tai yra vieša atvirojo kodo, skirto *Node.js*, kliento žiniatinklio programų, programėlių mobiliesiems, robotų, maršrutizatorių ir daugybės kitų *JavaScript* bendruomenės poreikių, bibliotekų rinkinys.

Nors bibliotekų kūrėjai yra laisvi pasirinkti savo licencija ir naudojimo sąlygas, *Node.js* yra platinamas naudojant MIT licencija[5]. Ši licencija leidžia naudoti *Node.js* technologiją be mokesčių ar kitų apribojimų tokią kokia ji yra, be kūrėjų suteiktų garantijų.

Programų architektai įvertino šios technologijos privalumus ir naudą panaudodami ją įvairiuose savo projektuose. Technologijos panaudos straipsniuose rašoma, kad ši technologija yra sėkmingai taikoma ir ją naudoja tokie IT gigantai kaip eBay, GoDaddy, Microsoft, Paypal, Uber.

Sritys, kur ši technologija puikiai tinka (15) :

- Programos kurios turi įvesties ir išvesties apribojimus (pvz. failų kūrimo, skaitymo, rašymo programos)
- Srauto perdavimo programos (pvz. transliacijos platforma kaip Twitch ar Netflix)

[5] https://github.com/nodejs/node#license

- Programos, kurios naudoja duomenis gautus realiu laiku (pvz. susirašinėjimo programos)
- Programos kurios naudoja *JSON API* (pvz. programos kurios duomenų apsikeitimo formatą naudoja *JSON*)
- Vieno puslapio programos (angl. Single page application)

Dauguma vartotojo sąsajos programuotojų yra susipažinę su *JavaScript*, todėl jiems bus lengva naudoti *Node.js* serverio dalyje. Tai leis mokytis *Node.js* lengviau ir užtruks mažiau laiko, o tai leis produktui greičiau patekti į rinką, nes yra naudojama viena iš populiariausių programavimo kalbų (15).

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

4 pav. Node.js HTTP serverio pavyzdys

2.1.1. Node.js privalumai

JavaScript programavimo kalba ir *Node.js* technologija suteikia galimybę dirbti su kliento ir serverio dalimis. Nebėra poreikio samdyti papildomus programuotojus serverio kūrimo technologijai, nes viską galima padaryti naudojant *JavaScript* programavimo kalbą, kas leidžia sutaupyti laiko ir pinigų (15).

Node.js buvo pripažintas už išskirtinius rezultatus, nes jis naudoja *Google V8 JavaScript* variklį. Tai leidžia kompiliuoti *JavaScript* kodą į mašininį kodą. Ši vykdymo aplinka pagerina kodo įvykdymo laiką taip suteikiant neblokuojančias įvesties ir išvesties operacijas (15).

Kadangi *Node.js* buvo tokia pirmoji technologija, kuri panaudojo *JavaScript* vykdymo aplinką serverio kūrimui, tai leido susidaryti didelei ekosistemai. Viena iš šios ekosistemos dalių yra bendruomenė, kuri nuolatos prisideda prie infrastruktūros kūrimo ir tobulinimo. Bendruomenė padeda vieni kitiems įvairiuose forumuose dalindamiesi kodo iškarpomis, bei bibliotekų

dalinimosi platformose įkeldami savo sukurtas bibliotekas, kurios išsprendžia pasikartojančias problemas nemokamai.

Dar vienas iš *Node.js* privalumų, jog ši technologija veikia taip, jog dažnai naudojami moduliai programoje yra išsaugomi spartinančiojoje atmintyje. Tai leidžia greičiau grąžinti rezultatą bei sutaupyti resursų, nes nereikia vykdyti kodo, nes užklausa būna analogiška prieš tai buvusiai, kuri išsaugota spartinančiojoje atmintyje.

Node.js puikiai tinka programoms, kurios turi priimti daug užklausų vienu metu. Ateinančios užklausos yra susisteminamos, apdorojamos laiku ir tvarkingai. Taip pat ši technologija gali būti panaudota įvairiems atvejams, kad būtų įmanoma pritaikyti kiekvienam individualiam atvejui. *JSON* formatas gali būti naudojamas grąžinti užklausos duomenis kliento pusei iš serverio. Taip pat tai turi aprašytus *API* sąsajas norint sukurti įvairaus tipo serverius: *HTTP, TCP, DNS*.

2.1.2. Node.js trūkumai

Renkantis *Node.js* technologiją reiktų įvertinti ir jos trūkumus (15). Dažni kodo atnaujinimai sukelia reikšmingus *Node.js API* pakeitimus, dėlto kiekvieną kartą norint atsinaujinti *Node.js* versiją reikia atlikti pakeitimus kode, kada būtų galima naudoti naujausią technologijos versiją. Tai sukelia papildomus sunkumus dirbant prie projekto, nes reikalingi papildomi kodo pakeitimai. Taip pat tai padidina mokymosi kiekį, nes reikia išmokti naudoti naujas *API* funkcijas.

Nors *Node.js* yra įžymus savo *NPM* bibliotekų registrų, tai leidžia bet kam kurti įvairias bibliotekas. Dažniausiai programuotojai jas naudodami nesigilina į kodo kokybę ir kaip pasirinkta biblioteka veikia, kol ji grąžina norimą rezultatą. Tai gali sukelti problemų kai yra naudojama begalė įvairiausių bibliotekų, kurių kokybė yra nežinoma. Problemos gali būti įvairios: saugumo spragos, prastas veikimo greitis, įvairios bibliotekų nereikalingos priklausomybės, reikalingi pastovūs kodo atnaujinimai dėl atsinaujinusių bibliotekų. Naudojant bibliotekas būtinai apsibrėžti kuomet yra pasirenkama naudoti jau paruoštą biblioteką ir kuomet reikia tai kurti pačiam. Kaip pavyzdys kuomet negalima naudoti jau paruoštų bibliotekų, produktas susijęs su sauga kaip slaptažodžių saugykla negali naudoti jau sukurtų bibliotekų, kurios būtų skirtos kriptografiniams veiksmams, nes tai gali kompromituoti klientų duomenis. Norint išvengti šios problemos reikia teisingai planuoti savo projekto naudojamas bibliotekas.

Node.js visiškai neturėjo jokių saugumo mechanizmų, buvo palikta saugumo spraga jog nėra informuojama apie vykdomą privilegijuotą kodą, kuris gali pasiekti apsaugotas sistemos vietas. Tai reiškia jog turint infekuotą kodą, niekas nesužinos ką jis daro. Kaip pavyzdys labai populiarus *NPM* modulis buvo nulaužtas ir infekuotas taip, kad jis pavogtų tavo turimą *Bitcoin* valiutą iš tavo

piniginės jeigu tu ją turėjai[6]. Tai būtų neįmanoma padaryti jeigu *Node.js* turėtų bent sistemos privilegijų valdymą, kuris perspėtų apie norimus pasiekti sistemos *API* prašydamas prieigos.

Nors *Node.js* įvardija tarp savo pliusų jog ji veikia labai efektingai kaip vienos gijos ir įvykiais pagrįsta architektūra, tačiau kai reikia atlikti sunkias skaičiavimo užduotis, kurioms reikalingas procesorius našumas sumažėja. Reikia atsižvelgti ar naudojant šią technologiją bus reikalinga atlikti skaičiavimo užduotis. Šio darbo metu atliksime bandymus, kad būtų galima įvertinti našumo charakteristikas.

2.2. Deno

Dešimtmetį vienintelis būdas kurti serverio programas naudojant *JavaScript* buvo naudoti *Node.js* technologiją, tai pasikeitė kai 2020 metais gegužę buvo išleistas *Deno* (16). Palyginus su *Node.js* gyvavimo amžiumi tai yra visai nauja technologija. Įdomus faktas apie *Deno* yra tas, jog ši technologija yra sukurta to paties kūrėjo, kuris sukūrė *Node.js*. Idėja kurti naują technologiją, kuri būtų skirta nukonkuruoti šiuo metu populiariausią vykdymo aplinką *Node.js* jam kilo ankščiau nei 2020 metai, jis apie tai kalbėjo savo pristatyme 2018 metais, "10 dalykų dėl kurių gailiuosi apie *Node.js*" (17), tuo metu jis jau dirbo prie *Deno* prototipo.

Nors *Deno* buvo lygiai taip pat sukurta panaudojant *Google V8 JavaScript* variklį kaip pagrindą, tačiau daugiau pakeitimų buvo architektūriniuose technologijos sprendimuose. Keletas esminių problemų, kurias jis norėjo išspręsti buvo saugumo spragos *Node.js* platformoje, sudėtinga modulių registro sistema ir plėtinių ne nurodymo galimybė jį importuojant. (18)

Nors *NPM* šiuo metu ir yra didžiausia bibliotekų registro platforma, tačiau ji sukelia daugiau problemų nei iš tikrųjų palengvina programuotojo gyvenimą. Nors šis būdas turėjo pakeisti bibliotekų importavimą kaip internetiniame puslapyje, tačiau Ryan pripažino, kad internetinio puslapio bibliotekų importavimo būdas yra pranašesnis už *Node.js* bibliotekų importavimo būdą. Internetinio puslapio atveju užtenka panaudoti žymą <script> ir biblioteka pus parsiųsta į internetinį puslapį, o *Node.js* atveju ją reikia apsirašyti package.json faile, tuomet reikia ją parsisiųsti lokaliai į programuotojo kompiuterį ir po to ją reikia importuoti į savo norimus failus, visas šis procesas yra kur kas ilgesnis ir sudėtingesnis, nors iš pradžių buvo manoma jog tai palengvins programuotojų darbą.

```
const port = 8080;

const handler = (request: Request): Response => {
  const body = `Your user-agent is:\n\n${
```

[6] The latest npm breach… Or is it? https://blog.logrocket.com/the-latest-npm-breach-or-is-it-a427617a4185/

```
    request.headers.get("user-agent") ?? "Unknown"
  }`;

  return new Response(body, { status: 200 });
};

console.log(`HTTP server running. Access it at: http://localhost:8080/`);
Deno.serve({ port }, handler);
```

5 pav. Deno HTTP serverio pavyzdys

Serverio esminis skirtumas jog *Deno* kaip pagrindinę kalbą renkasi naudoti *TypeScript*, tai yra griežtus tipus turinti programavimo kalba, kuri kompiliavimo metu būna pakeista į *JavaScript*.

2.2.1. Deno privalumai

Šios technologijos sukūrimas yra bandymas ištaisyti ir patobulinti Ryan sukurtos *Node.js* klaidas, taip jog *Deno* taptų šios technologijų srities lyderiu. Trumpai tariant atsižvelgus į šių dienų karščiausias programuotojų temas, pagerinti technologijos saugumą ir tuo pačiu suderinti *TypeScript* programavimo kalbos naudojimą.

Taigi viena iš pagrindinių naujovių yra *TypeScript* (19) kaip pagrindinė šios technologijos programavimo kalba. Aišku, šią programavimo kalbą buvo galima naudoti ir *Node.js*, bet tai reikalavo papildomo konfigūravimo. Reikia įdėti papildomą kompiliavimo žingsnį, kuris sukompiliuotų *TypeScript* kodą į *JavaScript*, kad būtų įmanoma jį naudoti *Node.js* technologijoje. *Deno* veikia truputį kitaip naudojant *TypeScript* ir siūlo iškarto paruoštą sprendimą. *Deno* nereikia konfigūruoti papildomo kompiliavimo žingsnio, nes programa-interpretatorius sugeba pats aptikti ir paversti *TypeScript* kodą į *JavaScript*, šis žingsnis yra optimizuotas pačiame interpretatoriuje, o tai pagreitina kompiliavimo laiką. Apibendrinant, *Deno* palaiko *TypeScript* ir *JavaScript* failus nereikalaujant papildomo išskirtinio konfigūravimo.

Saugumo spragos *Node.js* technologijoje leido permąstyti kaip tai būtų galima patobulinti *Deno* technologijoje (18). Šiai problemai spręsti buvo pasitelktas pavyzdys iš išmaniųjų telefonų, kai norint įsirašyti ir naudoti programėlę su tam tikromis funkcijomis kaip telefono kamera, vietos prieiga ar mikrofono įrašymas reikia suteikti prieigą. *Node.js* technologijoje įdiegus biblioteką ji aklai galėjo atlikti įvairiausius veiksmus nieko apie tai nežinant bibliotekos naudotojui, nes yra aklai pasitikima biblioteka. Pats pavojingiausias pavyzdys yra galimybė pasiekti operacinės sistemos reikšmingus duomenis tokius kaip aplinkos kintamuosius, kuriuose saugomi prisijungimo raktai. Tada šią informaciją persiųsti įsilaužėliui ir ją panaudoti. Ši operacija *Deno* technologijoje būtų įmanoma tik tuomet jeigu bibliotekai būtų suteiktos teisės atlikti šiuos

veiksmus. Jeigu naudojama biblioteka *Deno* technologijoje bando pasiekti jautrią informaciją šis veiksmas yra blokuojamas ir yra perspėjama dėl trūkstamų bibliotekos teisių. Taip pat reikia nurodyti bibliotekos ir kitas teises tokias kaip interneto prieiga. Pavyzdys:

```
error: Uncaught PermissionDenied: read access to "sample1.ts", run again with the
--allow-read flag
    at unwrapResponse ($deno$/ops/dispatch_json.ts:42:11)
    at Object.sendAsync ($deno$/ops/dispatch_json.ts:93:10)
    at async Object.open ($deno$/files.ts:38:15)
    at async Object.readFile ($deno$/read_file.ts:14:16)
    at async file:///Users/testuser/workspace/personal/deno/ts-sample/deno-
cat.ts:35:14
```

6 pav. Deno klaidos žinutė kai bibliotekai trūksta teisių

Node.js bendruomenė ilgai laukė aukščiausio lygio laukimo(await) funkcionalumo programos kode. Ji buvo pridėta *Node.js* 14 versijoje, po to kai *Deno* jau tai turėjo. Aukščiausio lygio rezultato laukimas pagrindiniame projekto metode buvo nepalaikomas. Pagrindiniame metode kur yra paleidžiamas serveris nebuvo galima laukti asinchroninių rezultatų inicializavimo metu, tokių kaip prisijungimas prie duomenų bazės ir serverio paleidimas vienu metu. Tai padaryti yra įmanoma, tačiau tai reikalauja įvairių sprendimo būdų, kurie ne visada veikia užtikrintai. *Deno* tai įgalino ir palengvino programuotojų gyvenimą, paprastai leidžiant laukimo funkcijas pagrindiniame projekto metode.

Naudojant *Node.js* net patys paprasčiausi veiksmai, kaip *HTTP* serverio sukūrimas reikalavo importuoti **http** biblioteką, tačiau *Deno* technologijoje pagrindinės funkcijos yra standartinėje bibliotekoje. *Deno* kūrėjai suprojektavo savo standartinę biblioteką taip, jog *Deno* naudotojai iškarto galėtų naudoti įvairias funkcijas. Standartinių bibliotekų pridėjimo architektūra buvo nukopijuota iš *Go* programavimo kalbos ir jeigu *Deno* dokumentacijoje nerasite standartinės bibliotekos dokumentacijos, yra didelė tikimybė jog nuėję i *Go* programavimo kalbos dokumentacijos puslapį rasite norimos bibliotekos dokumentacijos aprašymą (16).

Deno kūrėjai pergalvojo bibliotekų naudojimą, vėl pritaikė tam tikrus architektūrinius sprendimus *Go* programavimo kalbos *Deno* technologijai. Pagrindinės problemos naudojant *NPM* bibliotekų registro valdymą yra tokios, jog kiekvienas projektas savo naudojamas bibliotekas saugodavo privačiai, taip nesidalindamas tų pačių bibliotekų tarp projektų, nenaudingai išnaudodamas disko vietą, taip pat sukuria daug nereikalingo kodo, kaip package.json failas. *Deno* išsprendė šią problemą leisdamas importuoti norimas bibliotekas tiesiogiai naudojant importavimo nuorodą, labai panašiai kaip naršyklės nurodo nuorodą ir išsaugo biblioteką lokaliai, jog nereiktų siųsti tos pačios bibliotekos antrą kartą. Pavyzdys kaip atrodo bibliotekos importavimas *Deno* technologijoje:

```
import { bgBlue, red, bold, italic } from "https://deno.land/std/fmt/colors.ts";
```

```
if (import.meta.main) {
 console.log(bgBlue(italic(red(bold("Hello world!")))));
}
```

7 pav. Bibliotekos importavimas Deno technologijoje

Deno technologijoje bibliotekos importavimas veikia taip, jog pirmą kartą importavus naują biblioteką kompiliavimo metu ji yra parsiunčiama ir išsaugoma. Kitą kartą kompiliuojant kodą ta pati biblioteka bus iškart naudojama, tai reiškia jog ji yra parsiųsta, išsaugota ir sukompiliuota lokaliai. Taip pat tas pačias naudojamos išorines bibliotekas bus galima naudoti įvairiuose projektuose. Tai reiškia jog bibliotekos nebebus privačiai valdomos kiekvieno projekto node_modules aplanke, taip sumažinant vietos sunaudojimą kompiuteryje.

Bibliotekų valdymo sistema *Deno* technologijoje yra paremta *ES* moduliais[7], o *Node.js* bibliotekų sistema yra paremta *CommonJS*[8]. Taip pat reikia nepamiršti, jog nebereikia aprašyti naudojamų bibliotekų package.json faile.

Dar vienas įdomus šios technologijos privalumas yra galimybė veikti įterptiniuose įrenginiuose (20). Kadangi *Deno* naudoja *Rust* programavimo kalbą bendravimui su operacine sistema tai leidžia naudoti šią technologiją įterptiniams įrenginiams, *ARM* architektūrą palaikančiuose įrenginiuose.

2.2.2. Deno trūkumai

Kaip minėjau *Deno* yra to paties kūrėjo *Node.js* technologijos tęsinys, kuris buvo išleistas po dešimtmečio stengiantis pataisyti prieš 10 metų sukurtos, išleistos ir tapusios *JavaScript* programavimo kalbos serverio kūrimo standartu technologijos rastas spragas. Tai reiškia jog *Deno* turi palengvinti programuotojų darbą tobulinant šios technologijos architektūrinius sprendimus, tačiau kaip ir minėjau *Deno* naudoja tą patį *Google V8 JavaScript* variklį, tai reiškia jog žymaus kodo vykdymo greičio patobulinimo nėra, todėl tikėtis didesnės vykdymo galios būtų neteisinga. Pagal atliktus greitaveikos testavimus *HTTP* užklausų atsakymo greitis pagerėjo, bet kai yra didžiulis kiekis užklausų atsakymo laikas yra didesnis lyginant su *Node.js* technologija.[9]

Taip pat reikia paminėti, kad *Deno* išleista yra tik kelerius metus, todėl šios technologijos stabilumas nėra toks patikimas, nors daug funkcionalumo jau yra išleista, kai kurios standartinės bibliotekos vis dar yra kūrimo fazėje kas gali sukelti nestabilumo (20).

[7] ES Modules https://michaelcurrin.github.io/dev-cheatsheets/cheatsheets/javascript/general/modules/es-modules.html

[8] Node.js documentation about modules https://nodejs.org/api/modules.html

[9] Deno našumo tyrimo duomenys https://deno.com/benchmarks

Nors ši technologija ir turi pakeisti *Node.js*, tačiau dėl didelio kiekio pakeitimų visiškas *JavaScript* bibliotekų palaikymas vis dar nėra galimas. Šių bibliotekų palaikymo galimybė yra numatyta, tačiau įrankiai tam yra vis dar kuriami standartinėje *Deno* bibliotekoje (20).

Prie minėtų *Deno* technologijos pliusų rašiau jog yra naudojamas *TypeScript* programavimo kalbos kompiliatorius pačioje *Deno* technologijoje. Tai reiškia, kad kompiliuojant projektą prisideda papildomas žingsnis, *TypeScript* programavimo kodo vertimas į *JavaScript*, kuris prailgina kompiliavimo laiką. Kiekvieną kartą norint atlikti kodo pakeitimus ir paleidžiant serverį bus ilgiau kompiliuojamas kodas, kas atitinkamai nuo projekto dydžio prailgins kompiliavimo laiką. Šios problemos dydis gali būti sumažinamas kompiliavimo metu naudojant nekeistų ir prieš tai sukompiliuotų failų duomenis, kompiliuojant tik naujai pakeistus failus, kurių nėra išsaugota atmintyje. *Deno* kūrėjai žada paspartinti šį žingsnį *TypeScript* kompiliatorių perrašant su Rust programavimo kalba (20).

2.3. Bun.js

Paskutinė nagrinėjama *JavaScript* vykdymo aplinka yra viena iš naujausių bandymų tapti *JavaScript* vykdymo aplinkų lydere. *Bun.js* yra sukurtas Jarred Sumner[10] su tikslu jog tai bus pati greičiausia *JavaScript* vykdymo aplinka, kuri „galės viską“ ir bus pakaitalas dabartiniam srities lyderiui *Node.js*[11]. Jos tikslas yra žinomas *Node.js* architektūrines problemas patobulinti, su tikslu turėti geriau veikiančią *JavaScript* vykdymo aplinką. Pirmoji stabili versija buvo išleista Rugsėjo 8, 2023 metais, tuo metu tai buvo pirmoji stabili versija paruošta naudojimui, tačiau ji veikė tik *MacOS* ir *Linux* aplinkose. *Node.js* ir *Deno* technologijose kaip pagrindas yra pasirenkama *Google V8 JavaScript* variklis tačiau *Bun.js* pasirinko kitą kelią, viską sukurti nuo nulio, kaip pagrindą pasirenkant *WebKit/Safari JavaScriptCore* variklį, tai sumažina paleidimo laiką ir atminties naudojimą (21).

Įvairios *Bun.js* bibliotekos yra kuriamos naudojant C ir Zig programavimo kalbas. Ypatingai yra stengiamasi išvengti visų *Node.js* ar *NPM* bibliotekų priklausomybių, kuo mažiau naudojant *JavaScript* kodo *Bun.js* technologijos bibliotekose. Šie sprendimai buvo priimti norint turėti kuo trumpesnį kodo vykdymo laiką. *JavaScript* programavimo kalbos *API* tokių kaip interneto, įvesties ir išvesties operacijų perrašymas žemesnio lygio programavimo kalba suteikia geresnį našumą. Šis architektūrinis sprendimas yra didžiulis įsipareigojimas, nes norint tai atlikti reikia įdėti daug

[10] Jarred Sumner LinkedIn profilis https://www.linkedin.com/in/jarred-sumner-a8772425/

[11] What Is Bun.js and Why Is the JavaScript Community Excited About It? https://www.makeuseof.com/what-is-bunjs-why-the-javascript-community-excited/

darbo[12].

```
Bun.serve({
    fetch(req) {
      return new Response("Bun!");
    },
  });
```

8 pav. Bun.js HTTP serverio pavyzdys

2.3.1. Bun.js privalumai

Ši technologija kaip savo didžiausią pliusą pristato, jog tai nėra vien *JavaScript* programavimo kalbos vykdymo aplinka, tai yra kur kas daugiau. Technologija yra suprojektuota taip jog visi įrankiai skirti programavimui, tokie kaip bibliotekų tvarkyklė, aprašytų scenarijų ir testų leidimas, įvairių bibliotekų apjungimas ir kiti įrankiai jau būtų įdiegti ir pritaikyti naudojimui pačiame *Bun.js*. Tai leidžia sumažinti darbo kiekį programuotojams ir pagreitinti projekto kūrimą, taip sumažinant pinigines išlaidas.

Taip pat vienas iš pagrindinių dalykų apie kurį daug teigia šios technologijos kūrėjai yra jos greitis. Jų internetiniame puslapyje jie teigia jog *Bun.js* procesai veikia 4x kartus greičiau nei tie patys *Node.js* procesai (21), o magistro darbe buvo apskaičiuota jog *Bun.js* bandymų metu veikė 1.88 karto greičiau nei *Node.js* (22). Tai reiškia jog kūrimo, testavimo, diegimo procesai truktų trumpiau, visa tai lemia mažesnes pinigines išlaidas ir trumpesnį laiką norint paleisti produktą į rinką.

Deno vienas iš pliusų buvo numatytas *TypeScript* failų palaikymas, *Bun.js* taip pat palaiko *TypeScript* failus, bet ir .tsx tipo failus, kurie yra tie patys *TypeScript* failai, bet rašomi JSX sintakse, kurie yra skirti aprašyti React bibliotekos komponentus. *Bun.js* lygiai taip pat kaip *Deno* šių tipo failus paverčia į paprastą *JavaScript* kompiliavimo metu (21).

Bun.js ne tik turi naujų *ECMAScript* modulių palaikymą, bet ir palaiko kitą bibliotekų importavimo standartą *CommonJS*, tai leidžia naudoti senas ir naujas *JavaScript* bibliotekas, nes milijonai *NPM* bibliotekų vis dar naudoja senąjį *CommonJS* modulių importavimo būdą (21).

JavaScriptCore variklis yra naudojamas *Bun.js* technologijoje. Šis variklis yra kuriamas Apple ir jis yra naudojamas Safari naršyklės, todėl visos *WEB API* funkcijos kaip fetch, WebSocket veikia tiesiogiai kaip pačioje Safari naršyklėje, kas užtikrina aukštą kodo kokybę ir veikimą (22).

Be greitesnio procesų įvykdymo greičio yra žadamas labai lengvas migravimo procesas nuo

[12] Explore Bun.js: The all-in-one JavaScript runtime https://www.infoworld.com/article/3688330/explore-bunjs-the-all-in-one-javascript-runtime.html

Node.js į *Bun.js* technologiją, todėl daugelis *Node.js API* funkcijų bus palaikomos *Bun.js* technologijos, kas leis nekeičiant kodo paleisti savo projektą naudojant kitą vykdymo aplinką (21).

Vienas iš daugiausia dėmesio sulaukusių šios technologijos galimybių programuotojų tarpę yra „hot reload“ funkcija, kuri nepraranda būsenos. Tai reiškia jog programavimo metu norint pamatyti padarytus kodo pakeitimus nebus privaloma perkrauti projekto ir per naujo atlikti veiksmus savo kuriamoje sistemoje norint patekti į prieš tai buvusią projekto būseną. *Bun.js* naudoja WebSocket technologija, kuri leidžia informuoti apie pakitusią kodo vietą ir ją pakeisti naujai sukompiliuota, taip taupant laiką kai nereikia atkurti būsenos ir tęsti programavimo darbus[13]. Pats puikiausias pavyzdys šiuo atveju jeigu žemiau pavaizduotas kodas būtų pakeistas ir išsaugotas, kai serveris yra paleistas lokaliai programavimo režimu, **count** kintamojo reikšmė išliktų.

Kodo pavyzdys:

```
// make TypeScript happy
declare global {
  var count: number;
}

globalThis.count ??= 0;
console.log(`Reloaded ${globalThis.count} times`);
globalThis.count++;

// prevent `bun run` from exiting
setInterval(function () {}, 1000000);
```

9 pav. Bun.js „hot reload“ kodo pavyzdys

Rezultatų langas:

```
bun --hot index.ts
Reloaded 1 times
Reloaded 2 times
Reloaded 3 times
```

10 pav. Bun.js „hot reload“ kodo pavyzdžio rezultatas

Po dešimtmečio nuo *Node.js* sukūrimo buvo sukurta *Deno* technologija skirtą pakeisti pirmąją *JavaScript* vykdymo aplinką *Node.js*, tai darydami jie ypatingai pabrėžė technologijos saugumo spragas ir kaip tai turėtų būti išspręsta su *Deno* vykdymo aplinka, tačiau *Bun.js* sekė *Deno* technologijos ypatumus ir tai pritaikė savo kuriamoje technologijoje. *Bun.js* taip pat reikalauja aprašyti reikalingas teises savo naudojamoms bibliotekoms, tačiau pačios

[13] Bun.js „hot“ rėžimas https://bun.sh/docs/runtime/hot#hot-mode

populiariausios bibliotekos yra iš anksto patvirtintos. Tai reiškia, kad programuotojas neturės įdėti papildomo darbo aprašant kiekvienos naudojamos bibliotekos teises, nes pačios populiariausios bibliotekos yra jau patikrintos ir jų teisės aprašytos *Bun.js* komandos. Taip stengiamasi sutaupyti laiko, bei kiek įmanoma apsaugoti nuo galimų grėsmių. Jeigu yra būtinybė apriboti ar pridėti kokias tai teises bibliotekai, tai galima atlikti.

2.3.2. Bun.js trūkumai

Vienas iš *Bun.js* trūkumų, kuris yra keliantis nerimą – šios technologijos naujumas. Šiuo metu yra visiškai neaiški šio projekto ateitis ir kaip technologija įsilies į rinką. Minėjau jog *Deno* irgi yra apynauja technologija, nors ji gyvuoja apie 3 metus, tačiau vis dar dauguma žmonių renkasi *Node.js* vietoj *Deno,* nes tai tiesiog yra plačiau naudojamas ir žinomas įrankis. Tuo tarpu *Bun.js* pirmoji stabili versija šiuo metu nėra nei pusės metų senumo, todėl apie šios technologijos galimybės kalbėti yra anksti, galima tik atlikti analizę ką ši technologija gali ir kuo yra pranašesnė šiuo metu.

Technologijos naujumas ne tik sukelia stabilumo problemų, tačiau šiuo metu stabili *Bun.js* versija tėra išleista *MacOS* ir *Linux* operacinėse sistemose, todėl norint dirbti su ja reikia turėti kompiuterį, kuris veikia naudojant šias operacines sistemas. *Windows* operacinės sistemos *Bun.js* versija yra bandomoje stadijoje su daug limituotų funkcijų. *Bun.js* veikimas *Windows* operacinėje sistemoje yra numatytas ateityje, tačiau jos dar reikia palaukti.

Dėl savo technologijos naujumo kai kurios numatytosios funkcijos gali būti neegzistuojančios *Bun.js* bibliotekose. *Node.js API* funkcionalumas yra vienas iš jų, todėl norint įvykdyti *Bun.js* kūrėjų siūlomą vykdymo aplinkos pakeitimą iš *Node.js* į *Bun.js* šiuo metu gali būti jog bus neįmanomas. Vienintelis būdas tai sužinoti yra išbandyti *Node.js* migraciją į *Bun.js* arba pasitikrinti *Bun.js* dokumentacijos puslapyje, kuriame yra aprašytas *Node.js* modulių palaikomumas[14]. Didelė tikimybė jog nedideli projektai, kurie naudoja mažą kiekį *Node.js API* funkcijų nesusidurs su migracijos sunkumais, tačiau didesni projektai, kurie naudoja didelį kiekį *Node.js* modulių neturės galimybės pilnai atlikti migracijos. Tai bus įmanoma atlikti tik tuomet kai bus įdiegtas trūkstamas funkcionalumas arba įgyvendinta trūkstama kodo dalis nuo nulio.

3. Tyrimo metodologija

Norint nustatyti našiausią *JavaScript* vykdymo aplinką, bus atliekamas tyrimas šiais etapas:

[14] Bun.js vykdymo aplinkos Node.js API sąsajų palaikymas https://bun.sh/docs/runtime/nodejs-apis

***JavaScript* vykdymo aplinkų apžvalga.** Šiame tyrimo etape buvo susipažįstama su *JavaScript* vykdymo aplinkų technologijomis, išaiškintos vykdymo aplinkų galimybės, privalumai ir trūkumai. Pagal analizuotą literatūrą buvo pasirinkti kriterijai ir atliekama jų lyginamoji analizė, norint nustatyti technologiją, kuri atitinka daugiausia išsikeltų kriterijų.

Eksperimentinis tyrimas našumui nustatyti. Šiame etape buvo aprašomi atliekami bandymai pagal literatūroje išvardintus pačios pirmosios *JavaScript* vykdymo aplinkos *Node.js* technologijos panaudojimo išvardytas teigiamas, neigiamas savybes ir dažnai atliekamus veiksmus. Taip pat aprašyta kaip bus apskaičiuojamos našumo charakteristikos, bandymams naudojama kompiuterinė ir programinė įranga, aptariami bandymų rezultatai, pagal gautus rezultatus nustatoma našiausia vykdymo aplinka.

3.1. Vertinimo kriterijai

Serverio-kliento architektūroje nėra aprašytų serverio kūrimo standartų. Šios architektūros dalys tarpusavyje bendrauja naudodamos įvairius protokolus *HTTP, FTP, SMTP* (23). Kai yra naudojami standartiniai protokolai klientas neturi galimybės suprasti kokia technologija slepiasi už naudojamo serverio dalies ar kliento dalies aplikacijos, todėl renkantis kriterijus, jie bus pasirenkami pagal *JavaScript* serverio kūrimo vykdymo aplinkų technologijų rastą literatūrą ir praėjusiame skyriuje aprašytus kiekvienos technologijos privalumus ir trūkumus.

Straipsnyje „Choosing the right JavaScript runtime: an in-depth comparison of Node.js and Bun“ (3) yra aprašomi *Node.js* ir *Bun.js* privalumai ir lyginamas šių technologijų našumas. Pagal šio straipsnio išreikštus technologijų privalumus buvo pasirinkti šie kriterijai:

- **Technologijos branda** – straipsnyje yra pabrėžiama šių dviejų technologijų skirtumai, kai *Node.js* laikoma patikrinta technologija, o *Bun.js* savo vertę dar tik bando įrodyti, todėl tai svarbu palyginti visų lyginamų technologijų kontekste.
- **Node Package Manager bibliotekų registro palaikymas ir architektūra** – pažymima jog tai yra didžiausias pasaulyje bibliotekų registras, kuris leidžia programuotojams įgyvendinti norimus sprendimus greičiau su egzistuojančių bibliotekų pagalba.
- **ECMAScript modulių importavimas kaip technologijos standartas** – naujausių standartų laikymasis palengvina programuotojų darbą, nes jiems nereikia atlikti papildomo konfigūravimo darbų, pvz. norint palaikyti kitokius importavimo būdus nei *ECMAScript* standartas.
- **Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript** – nėra reikalingi išoriniai įrankiai tokie kaip *Webpack*, kad galėtume atlikti įvairius

procesus darbo metu, pvz. diegimui skirtas projekto kompiliavimo procesas.

- **TypeScript palaikymas vykdymo aplinkos kompiliatoriuje** – integruotas *TypeScript* palaikymas sumažina papildomą laiką skirtą konfigūravimui, bei naudojant šią technologiją kodas yra paprasčiau suprantamas ir lengviau perrašomas, nes turi griežtus tipus.

Daugelyje skaitytų straipsnių apie JavaScript vykdymo aplinkas buvo išreikštas *NPM* registro naudojimas ir jo svarba. Darbe „Comparison of JavaScript package managers“ (24) buvo lyginamos *NPM, Yarn* ir *PNMP* paketų tvarkykles. Šiame darbe buvo atlikta apklausa ir aptariami darbo rezultatai. Aptarimo metu buvo išskirta besidubliuojančių bibliotekų problema ir kaip *PNPM* paketų tvarkyklė tai sprendžia. Pagal šį darbą buvo išskirtas šis kriterijus:

- **Optimizuotas modulių saugojimas** – svarbu palyginti ar vykdymo aplinkos sprendžia besidubliuojančių bibliotekų problemą, efektyviau išnaudodamos savo resursus.

Darbe „Deno – A new Node.js?“ (4) buvo lyginamos *Node.js* ir *Deno* technologijos, atliekama apklausa. Vienas iš išskirtų *Deno* pliusų yra teisių prieigos valdymas. Apklausos metu buvo išsiaiškinta, kad iš 24 respondentų net 16 rinktųsi *Deno* vien dėl šios funkcijos. Pagal tai buvo pasirinktas šis kriterijus:

- Bibliotekų teisių apribojimas kaip saugumo įrankis

Viena iš *Bun.js* kūrėjų išskiriamų naujovių jų technologijoje yra būdas kaip pagreitinti programavimo darbus. Tai yra galimybė matyti kodo pakeitimus neprarandant būsenos[15]. Pagal išskirtą šios funkcijos naujumą palyginsime kitas vykdymo aplinkas ar iš tiesų šios technologijos neturi šios funkcijos pagal šį kriterijų:

- Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas

3.1.1. Technologijos branda

Dažnu atveju atsiradus naujai technologijai sunku nuspėti ar ta technologija pasiteisins toje srityje, ilgai bus palaikoma ir naudojama. Taip pat reikia išsiaiškinti ar sritys kur ją galima taikyti yra geriau veikiančios su nauja technologija. Turi praeiti laiko, jog iš tiesų būtų įrodytas technologijos pranašumas, palaikymas, trūkumai, aprašytos ir naudojamos geriausios praktikos. Galime teigti jog technologijos, kurios yra tobulinamos ir atnaujinamos daugiau nei dešimtmetį yra tvirtai užimančios lyderio vaidmenį tam tikroje srityje. (9)

[15] Dokumentacija kaip veikia kodo pakeitimo funkcija realiu laiku https://bun.sh/docs/runtime/hot

Taip pat prie technologijos ekosistemos egzistavimo prisideda ir patys programuotojai. Jie kuria įrankius, kurie palengvina kūrimo darbus, bei prisideda prie iškilusių problemų sprendimo. Kuo daugiau žmonių naudoja tam tikra technologiją ar įrankį, tuo daugiau žmonių gali susidurti su panašiomis problemomis ir jų iškilusios problemos ir sprendimai yra pasidalinami internete, kur kiti programuotojai gali jais pasinaudoti.

Šis kriterijus bus matuojamas pagal parsisiuntimų ir išleistų versijų kiekį. Technologija turinti daugiausia parsisiuntimų ir išleistų versijų – atitinka kriterijų, antrą vietą užimanti pagal parsisiuntimų ir išleistų versijų kiekį bus dalinai atitinkanti kriterijų, o turinti mažiausiai – neatitinka kriterijaus.

3.1.2. Node Package Manager bibliotekų registro palaikymas ir architektūra

NPM bibliotekų registras yra didžiausias bibliotekų registras pasaulyje (24). *NPM* bibliotekų registro naudojimas *JavaScript* ekosistemoje yra neišvengiamas. Node Package Manager registras buvo išleistas 2010 metais ir tai yra kaip dalis *Node.js* technologijos. 2014 metais dėl didelio šio registro populiarumo buvo sukurta atskira įmonė, kuri valdo šį bibliotekų registro projektą. Ši kompanija siūlo nemokamą bibliotekų talpyklą, taip pat ir privačias kodo talpyklas skirtas verslui. Galimybė įdiegti *NPM* bibliotekas projektuose kuriuose naudoja *JavaScript* yra vienas iš svarbių kriterijų.

Taip pat naudojant *NPM* bibliotekų registrą yra reikalinga specifinė projekto architektūra. Projektas, kuris bibliotekų tvarkymui naudoja *NPM* bibliotekų registrą turi turėti „package.json“ failą kuriame yra aprašomas projekto informacija, scenarijų komandos, naudojamų bibliotekų sąrašas bei nustatomos jų versijos. Šis failas yra skirtas tam, kad kai pirmą kartą yra paleidžiamas projektas visos priklausomybės būtų sudiegtos norint paleisti projektą. Taip pat po visų reikiamų bibliotekų įrašymo yra automatiškai sugeneruojamas „package-lock.json“ failas, kuris yra skirtas išsaugoti visas priklausomybes, kurios buvo sugeneruotos į vieną medį, kad vėlesni bibliotekų diegimai būtų identiški, taip leidžiant kitiems programuotojams lengviau pasiruošti darbui prie projekto, bei sekti naudojamų bibliotekų versijų pasikeitimus.

Technologija bus laikoma atitinkanti kriterijų, kai technologijos projektas turės package.json failą, skirtą apsirašyti visoms bibliotekų priklausomybėms ir palaikys *NPM* bibliotekas. Dalinai atitiks kriterijų jeigu palaikys bent vieną iš reikalavimų, o neatitiks jeigu nepalaiko jokių reikalavimų.

3.1.3. Optimizuotas modulių saugojimas

Viena iš didžiausių problemų naudojant *NPM* yra neefektyvus kaupiamosios atminties išnaudojimas, dėl node_modules aplanko (24). Tam buvo sukurta net kitokia *NPM* versija pavadinimu *PNPM*. Šios problemos esmė yra ta, jog kiekviena projekto biblioteka yra parsiunčiama atskirai į projektą. Tai reiškia jeigu kompiuteryje yra keli projektai ir jie naudoja tas pačias bibliotekas, tai tos pačios bibliotekos bus parsiunčiamos keletą kartų. Taip susidaro failų dublikatai, kurie užimą daugiau vietos. Visa tai yra išsprendžiama turint globalų aplanką, kuriame yra išsaugomos bibliotekos, kurios yra bendrai naudojamos kelių projektų, taip užtikrinant jog kompiuteryje nėra tų pačių bibliotekų dublikatų.

Šis kriterijus bus laikomas atitikusiu jeigu technologija turės globalią bibliotekų saugyklą iš kurios bus pernaudojamos keliuose projektuose. Kriterijus bus laikomas neatitikusiu jeigu nebus pernaudojamos parsiųstos bibliotekos.

3.1.4. ECMAScript modulių importavimas kaip technologijos standartas

Yra keletas būdų *JavaScript* programavimo kalboje kaip supakuoti savo programinį kodą ir jį importuoti į norimas projekto vietas. Du paplitę būdai *JavaScript* programavimo kalboje yra *CommonJS* ir *ECMAScript* moduliai. (3)

Naudojant *CommonJS* kiekvienas failas yra traktuojamas kaip atskiras modulis[16]. Pavyzdys kaip importuoti kodą naudojant *CommonJS* modulių sistemą:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

11 pav. Bibliotekos importavimas naudojant CommonJS modulių sistemą

ECMAScript yra standartas scenarijų kalboms, kuria pagrįsta *JavaScript*. Kompanija „Ecma International“[17] yra atsakinga už *ECMAScript* standartizavimą (6). Šios kompanijos tikslas yra aprašyti *JavaScript* kalbos funkcionalumą, kuris vėliau bus įdiegtas į vykdymo aplinkas naršyklėse. Ši kompanija patį pirmą standartą aprašė dar 1997 metais ir tai daro iki šiol. Nors nuo 1997 metų buvo leidžiami *ECMAScript* standartai, pats pirmas standartas, kuris buvo naudotas *JavaScript* atnaujinimui buvo penktasis(5^{th}) *ECMAScript* standarto leidimas, daugiau nei už dešimtmečio 2009 metais, tačiau tik 2015 metais pirmą kartą buvo įgyvendintas numatytas *JavaScript* modulių palaikymas (24). Pavyzdys kaip importuoti kodą naudojant *ECMAScript* modulių sistemą:

```
// app.mjs
```

[16] Node.js dokumentacija apie CommonJS modulius https://nodejs.org/api/modules.html#modules-commonjs-modules

[17] ECMA standartizacijos svetainė https://www.ecma-international.org/

```
import { addTwo } from './addTwo.mjs';

// Prints: 6
console.log(addTwo(4));
```

12 pav. Bibliotekos importavimas naudojant ECMAScript modulių sistemą

Šie du importavimo būdai atsirado tik dėlto, jog *Node.js* atsiradimo metu nebuvo standarto kaip importuoti *JavaScript* kodą. *ECMAScript JavaScript* kodo importavimo standartas atsirado kur kas vėliau.

Technologija atitiks kriterijų kai numatytą bibliotekų importavimo standartą naudoja *ECMAScript*. Dalinai atitiks kriterijų jeigu palaiko *ECMAScript* bibliotekų importavimo standartą. Neatitiks kriterijaus jeigu nepalaiko *ECMAScript* bibliotekų importavimo standarto.

3.1.5. Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript

Kiekvienas papildomas konfigūravimo žingsnis norint išleisti savo kuriamą produktą užima laiko, bei sukelia nenumatytų problemų, todėl naujos technologijos pagrindinį dėmesį skiria supaprastinti įvairius programavimo darbus (3). Šie programavimo darbai *JavaScript* vykdymo aplinkose gali būti iškarto atlikti pačios vykdymo aplinkos. Tokie darbai kaip naujo projekto sukūrimas, bibliotekų diegimas, paruošta naudoti testavimo biblioteka, *TypeScript* palaikymas, programavimo kodo formatavimo taisyklės, kodo rašymo taisyklių aprašymas ir tikrinimas ir kita. Šios funkcijos gali ne tik paspartinti produkto išleidimą, tačiau ir palengvinti programuotojų darbą, nes nereikės investuoti papildomo laiko konfigūruojant projektą ir dažniausiai atliekami žingsniai produkto kūrimo metu jau bus įgyvendinti.

Kriterijus bus laikomas atitikusiu jeigu technologija turės šias komandas:

- Paleisti *JavaScript* ar *TypeScript* failą
- Įdiegti biblioteką
- Paleisti testavimo biblioteką
- Tuščio naujo projekto kūrimas
- Naujo projekto sukūrimas pagal aprašytą šabloną

Jeigu bent vienos aukščiau aprašytos komandas technologija nepalaiko ji laikoma dalinai atitikusi kriterijų, o jei technologija nepalaiko nei vienos komandos ji laikoma neatitikusi kriterijaus.

3.1.6. TypeScript palaikymas vykdymo aplinkos kompiliatoriuje

TypeScript yra griežtus tipus turinti programavimo kalba, kuri yra pagrįsta *JavaScript*.

Programuotojai, kurie dirba su *JavaScript* dažnai susiduria su programinio kodo klaidomis, kai objektas neturi parametro, tai nutinka dėlto, kad *JavaScript* neturi griežtų tipų, tai reiškia jog įvairūs objektai gali būti kuriami be taisyklių, todėl didėjant projektui tai gali sukelti daug problemų, kai tiesiog neįmanoma žinoti kokias reikšmes objektas turės. Taip pat *TypeScript* gali būti lengviau suprantamas programuotojams, kurie prieš tai dirbo su C# ar Java programavimo kalbomis, nes sintaksė yra panaši. *TypeScript* gali pagerinti įvairius projektus, kur yra naudojamas *JavaScript*. Tai padės išvengti dažnai pasitaikančių klaidų, palengvins kodo perrašymo darbus ar kitiems programuotojams bus tiesiog lengviau suprasti kodą, kai žinos iš kokių parametrų tas objektas susideda. (19)

TypeScript palaikymas pačios *JavaScript* vykdymo aplinkos, kai yra nereikalingas konfigūracijos žingsnio pridėjimas verčiant *TypeScript* programinį kodą į *JavaScript* programinį kodą, sumažins konfigūravimo darbo laiką, taip dar labiau pagreitinant darbus.

Technologija atitiks šį kriterijų jeigu kompiliatorius automatiškai palaiko *TypeScript* programinio kodą be papildomos konfigūracijos (3). Technologija neatitinkanti šio kriterijaus bus laikoma, kuriai reikia atlikti papildomus veiksmus norint palaikyti *TypeScript*.

3.1.7. Bibliotekų teisių apribojimas kaip saugumo įrankis

Norint saugiai naudotis bibliotekų registrų, reikia jas apriboti nuo nereikalingų prieigų prie tam tikrų sistemos dalių, tokių kaip operacinės sistemos failai, funkcijos, interneto prieiga. Kaip ir minėjau skyriuje kur buvo aprašyti *Node.js* trūkumai vienas iš jų buvo galimybė bibliotekoms atlikti, bet kokias funkcijas, kaip pavyzdys bibliotekai yra suteikiama interneto prieiga. Ši spraga leidžia kenksmingam programiniam kodui atlikti visus galimus veiksmus niekam nežinant. Vienintelis būdas tokiu atveju yra skaityti bibliotekos kodą ir pačiam užtikrinti naudojamos bibliotekos saugumą. Minimali apsauga gali būti atlikti kai biblioteka gauna prieiga prie tam tikrų privilegijų tik tuomet kai jos yra suteikiamos tai bibliotekai. Taip yra uždedamas dar vienas papildomas apsaugos lygis norint apsisaugoti nuo nenorimų prieigų sistemoje.

Technologija atitiks šį kriterijų kai įdiegtoms bibliotekoms reikės aprašyti teises norint naudoti jų funkcijas. Technologija bus laikoma neatitikusi kriterijaus jei neribos įdiegtų bibliotekų teisių prieigos.

3.1.8. Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas

Labai paplitęs įrankis programuojant internetinius puslapius yra „hot reload“ funkcija. Tai funkcija, kai kiekvienas failo pakeitimas, kai projekto serveris yra paleistas lokaliai programavimo

režimu iškarto yra iš naujo sukompiliuojamas, taip visi kodo pakeitimai yra įvykdomi iškarto, programuotojas sutaupo laiko, nes nereikia perkrauti serverio ir kompiliuoti failų rankiniu būdu. Kiekviena technologija turi šio funkcionalumo atitikmenį, Java turi JRebel[18], *JavaScript* turi *NPM* biblioteką hot-reload[19]. Iš esmės šios technologijos stebi failų pasikeitimus, po pakeitimų juos sukompiliuoja ir pakeičia, tačiau naudojant šias technologijos yra keli būdai kaip tai atlikti. Vienas iš būdų kai padarius pakeitimus serveris persikrauna, taip prarasdamas būseną arba kitas būdas kai yra pakeičiamas kodas be serverio persikrovimo taip išsaugant būseną. Kodo pakeitimas nauju išsaugant būseną yra kur kas efektyvesnė ir laiką taupanti funkcija, nes programuotojas nėra priverstas vėl atkurti sistemos būsenos, taip matant visus norimus kodo pakeitimus iškarto, todėl šių dienų technologijose tai yra vienas iš funkcijų, kurios padeda taupyti kūrimo laiką[20].

Technologija bus laikoma atitinkanti šį kriterijų kai palaiko įrankį, kai kodo pakeitimai bus matomi iškarto po failo išsaugojimo ir programos būsena yra paliekama tokia pati(išsaugoma), dalinai atitinka kriterijų kai galima sukonfigūruoti jog būtų matomi kodo pakeitimai iškarto po failo išsaugojimo ir programos būsena nėra išsaugoma, neatitiks kriterijaus kai technologijoje negalima sukonfigūruoti jog būtų matomi kodo pakeitimai iškarto po failo išsaugojimo.

3.2. Kriterijų įvertinimas

Viso vertinimo metu bus remiamasi kiekvienos technologijos dokumentacijos internetiniu puslapiu. Šios technologijos tarpusavyje bus lyginamos pagal pasirinktus kriterijus. Kiekvieno kriterijaus poskyrio paskutinėje pastraipoje yra aprašyta kokiomis sąlygomis bus vertinami kriterijai.

Kriterijų vertinamo reikšmės:

- \+ – pilnai atitinkantis kriterijų
- **X** – neatitinka kriterijaus
- ! – dalinai atitinkantis kriterijų

Kiekvienos *JavaScript* vykdymo aplinkos versijos kriterijų lyginimo metu:

- Node.js - v20.8.0
- Deno - v1.37.2
- Bun.js - v1.0.6

[18] Java programavimo kalbos pagalbinė programinė įranga skirta pagreitinti programavimo procesą https://www.jrebel.com/products/jrebel

[19] NPM biblioteka skirta automatiškai sukompiliuoti pakeistus failus ir juos pakeisti naujais https://www.npmjs.com/package/hot-reload

[20] Bun.js dokumentacijos puslapis apie automatizuotą pakeistų failų kompiliavimą https://bun.sh/docs/runtime/hot#hot-mode

3.2.1. Technologijos branda

Node.js yra *JavaScript* vykdymo aplinkų lyderis išleistas 2009 metais. Tai buvo pirmoji atsiradusi technologija, kuri galėjo vykdyti *JavaScript* programinį kodą ne naršyklėje. Kadangi *JavaScript* yra viena iš labiausiai naudojamų programavimo kalbų internetiniuose puslapiuose, nes ji buvo pirmoji panaudota internetiniams puslapiams, taip pat ir *Node.js* tapo populiariausia *JavaScript* vykdymo aplinka. Tai leido tapti lyderiu, kuris diktavo taisykles ir architektūrinius trūkumus *JavaScript* vykdymo aplinkų technologijoje. Pritaikyti programėlių kūrimą ne tik kuriant serverius, bet ir mobiliąsias ar darbalaukio programas, sukuriant didžiausią *JavaScript* bibliotekų registrą *NPM* ir kiti mažesni pasiekimai. *Node.js* išleista daugiau nei prieš 14 metų. Šios technologijos parsisiuntimų kiekis pradėtas skaičiuoti tik 2014 metais, rašymo metu ši technologija buvo parsisiųsta iš oficialaus puslapio net 1475,137,881 kartų ir buvo išleistos 364 versijos[21]. (+)

Deno kaip *Node.js* išsišakojimas buvo išleista *Node.js* kūrėjo su tikslu ištaisyti visas klaidas ir netobulumus, kurie buvo atrasti *Node.js* naudojimo metu, taip patobulinant ir sukuriant geresnę technologiją, kuri galėtų pakeisti rinkos lyderį *Node.js*. Palyginus su *Node.js* tai yra ganėtinai nauja technologija, nes ji buvo išleista 2018 metais, todėl yra funkcijų, kurių vis dar trūksta, jog būtų galima atlikti pilną migracija nuo *Node.js* į *Deno* technologiją, įvairių standartinių bibliotekų trūkumas ir pačios technologijos netobulumas, nes ji nėra pakankamai paplitusi, ištestuota ir išbandyta technologija, kas leistų tapti šios srities lyderiu. *Deno* išleista prieš 4 metus. Pagal 2022 metais išleistą straipsnį *Deno* buvo parsisiųstas 4.1 milijono kartų ir buvo išleistos 247 versijos[22]. (!)

Bun.js yra visiškai kitaip projektuota *JavaScript* vykdymo aplinka, kuri turėtų atlikti tas pačias funkcijas ir pakeisti *Node.js*, tačiau su visiškai kitomis naudojamomis technologijomis. Naudojamas kitas *JavaScript* variklis, mažai žinoma programavimo kalba *Zig*, kuri skirta vykdyti funkcijas su operacine sistema. Ši technologija pasirinko kitokį požiūrį, labiau fokusuodama viską kurti nuo nulio, bandyti išgauti kuo geresnes našumo charakteristikas visose programavimo darbų srityse, bei paruošti kiek įmanomą labiau išbaigtą įrankį visiems produkto kūrimo atvejams nuo pradinių programavimo darbų iki pabaigos. Ši technologija darbo rašymo metu tik prieš mėnesį, t. y. 2023 metų rugsėjį išleido pirmąją stabilią versiją v1.0.0, dėlto ši technologija yra labai jauna. Nors ji jau ir išleista, tačiau ne visos *Node.js API* funkcijos yra įgyvendintos, kas neleidžia visiškai pakeisti vykdymo aplinkos technologijos jei yra naudojamos trūkstamos funkcijos. Norint didinti šios technologijos populiarumą taip pat reikia palaikyti *Windows* operacinę sistemą, nes šiuo metu

[21] *Node.js* parsisiuntimų statistika https://nodejs.org/metrics/
[22] *Deno* parsisiuntimų statistika https://deno.com/blog/changes

Bun.js yra oficialiai išleista tik *Linux* ir *MacOS* operacinėse sistemose. Taip pat dėl jos naujumo visiškai yra neaiški šios technologijos ateitis, patikimumas, trūkumai ir geriausios praktikos, todėl skambūs šios technologijos kūrėjų pareiškimai šiuo metu nėra išbandyti realiomis sąlygomis. Jei *Bun.js* technologijos kūrėjai toliau vystys šį projektą, pildys funkcionalumą, taisys klaidas ir pridės bendruomenės norimas funkcijas, ši technologija turi galimybę tapti rinkos lyderiu. *Bun.js* išleista prieš mažiau nei metus. Pagal NPM Bun.js puslapio duomenis, ši technologija buvo parsisiųsta 2,282,964 kartų ir 14 versijų, skaičiuojant nuo oficialiosios 1.0.0 versijos išleidimo[23]. (X)

3.2.2. Node Package Manager bibliotekų registro palaikymas ir architektūra

Node.js yra pirmoji vykdymo aplinkų technologija ir Node package manager bibliotekų registras buvo dalis *Node.js* technologijos. Ši technologija buvo pirmoji pritaikiusi *NPM* bibliotekų registro veikimą savo technologijai. *Node.js* naudoja package.json failą ir palaiko NPM bibliotekas, nes ši technologija tai sukūrė. (+)

Node.js technologija užvaldė *JavaScript* vykdymo aplinkų rinką, todėl visos kitos vykdymo aplinkos norinčios pakeisti *Node.js* turi palaikyti *NPM* bibliotekų registrą, nes tai yra vienas iš esminių funkcijų. *Deno* technologijoje *NPM* bibliotekos gali veikti, bet reikia nepamiršti jog reikės atlikti papildomus kodo pakeitimus, nes išorinės bibliotekos yra importuojamos kitaip nei *Node.js* technologijoje. Bibliotekų importavimo skirtumai ir kitokia bibliotekų valdymo architektūra lėmė, kad *Deno* neturi package.json failų ir kitos *NPM* registrui būdingos architektūros. Dėl nenaudojamos NPM bibliotekų registro architektūros *Deno* turi net savo bibliotekų registrą, kuris vadinasi „Deno land“[24]. Šie pokyčiai pasak *Node.js* ir *Deno* technologijų kūrėjo yra architektūrinis patobulinimas, kuris padės tapti *Deno* technologijai patogesnei kai yra dirbama su bibliotekomis. *Deno* dalinai atitinka kriterijų, nes nenaudoja package.json failo, bet palaiko NPM bibliotekas. (!)

Bun.js kūrėjai yra užsibrėžę tikslą pakeisti *Node.js*, todėl *NPM* bibliotekų registro palaikymas yra viena iš esminių funkcijų. *Bun.js* sukūrė savo komandinės eilutės sąsają, kuri yra skirta bibliotekų valdymui, palaiko *NPM* komandas ir projekto architektūrą. Tai yra atskiras įrankis, kuris veiks jau esamuose *Node.js* projektuose. Pagrindinis šio įrankio teigiamas privalumas yra trumpesnis įvykdymo greitis įvairiose operacijose, kai yra naudojama ši komandinės eilutės sąsaja. Tokie veiksmai kaip: bibliotekos pridėjimas, bibliotekų įdiegimas pirmą kartą, bibliotekų ištrynimas yra atliekami kur kas greičiau. *Bun.js* naudoja package.json failą ir palaiko NPM bibliotekas. (+)

[23] *Bun.js* parsisiuntimų statistika https://npm-stat.com/charts.html?package=bun
[24] *Deno* bibliotekų registras https://deno.land/x

3.2.3. Optimizuotas Node Package Manager modulių saugojimas

Node.js kaip *NPM* bibliotekų registro kūrėjas buvo tokios architektūros pradininkas, todėl geriausios praktikos bibliotekų registro naudojimui dar nebuvo atrastos. Visi šio bibliotekų registro trūkumai išryškėjo bėgant laikui, todėl bibliotekų saugojimas projekte buvo įgyvendintas pačiu paprasčiausiu būdu tuo metu, kuriuo yra išsaugomos kiekvieno projekto bibliotekos kiekviename projekte atskirai, jų nepernaudojant. *Node.js* neatitiko šio kriterijaus, nes kiekvienas projektas atskirai saugo savo bibliotekas. (X)

Kadangi *Deno* technologija buvo kurta to paties žmogaus kaip *Node.js*, jo tikslas kuriant *Deno* buvo ištaisyti didžiausias klaidas, kurios išryškėjo bėgant laikui *Node.js* technologijoje. Nesaugoti bibliotekų globaliai, kurios yra naudojamos keliuose projektuose yra viena iš *Node.js* technologijos trūkumų, todėl į tai buvo atsižvelgta ir ištaisyta, todėl taip bus sutaupoma kompiuterio atminties. *Deno* atitinka kriterijų, nes saugo bibliotekas globaliai ir jas gali naudoti keliuose projektuose.(+)

Bun.js kaip ir minėjau sukūrė savo bibliotekų registro valdymo komandinės eilutės sąsajos įrankį, kuris fokusuojasi į greitį, todėl norint turėti geriau veikiančią bibliotekų registro valdymo įrankį, vienas iš dalykų yra optimizuotas bibliotekų saugojimas, kurį *Bun.js* technologija turi. Turint globalią vietą kur yra saugomos bibliotekos galima nebesiųsti jos dar kartą ir taip sutaupyti laiko bei resursų. *Bun.js* atitinka kriterijų, nes saugo bibliotekas globaliai ir jas gali pernaudoti keliuose projektuose.(+)

3.2.4. ECMAScript modulių importavimas kaip technologijos standartas

Node.js sukurdami *NPM* turėjo išspręsti bibliotekų importavimo problemą kaip tai bus atliekama, nes *Node.js* išleidimo metu *JavaScript* bibliotekų modulių standartas nebuvo aprašytas. Tam buvo pasirinktas *CommonJS* standartas, kuris nėra palaikomas naršyklių. Bėgant laikui buvo aprašytas *ECMAScript* standartas kaip turėtų būti importuojamos bibliotekos *JavaScript* programavimo kalboje. Bibliotekų importavimo standartas *CommonJS* taip ir išliko *Node.js* standartu, nors laikui bėgant buvo pridėtas *ECMAScript* standarto palaikymas, bet kaip technologijos standartas yra pasirinktas *CommonJS*. *Node.js* dalinai atitinka šį kriterijų, nes *ECMAScript* importavimo standartas nėra numatytas, tai yra tik palaikoma. (!)

Deno technologija buvo išleista jau po *ECMAScript* bibliotekų importavimo standarto atsiradimo, tai buvo laikomasi *ECMAScript* standarto kaip technologijos standarto, todėl ši technologija atitinka kriterijų. (+)

Bun.js technologijoje, dėl aiškiai apibrėžto *ECMAScript* bibliotekų importavimo standarto

jis ir buvo pasirinktas kaip numatytas, tačiau yra palaikomas ir *CommonJS*. (+)

3.2.5. Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript

Node.js pati pirmoji *JavaScript* vykdymo aplinka neturėjo pavyzdžių, gerųjų praktikų, ko reiktų tokio tipo technologijai. Visa infrastruktūra aplink *Node.js* technologiją tobulėjo palaipsniui, todėl būtinybė įvairioms funkcijoms atsirado bėgant laikui. Šios funkcijos laikui bėgant buvo sukurtos *Node.js* technologijoje, bet didžioji dauguma funkcionalumo turi būti sukonfigūruojama paties programuotojo, taip atliekant pasikartojančius veiksmus kiekviename projekte.

Sąrašas įrankių kurie yra sukurti *Node.js* technologijoje[25]:

- npm init (naujo projekto kūrimas)
- npm install (įdiegti biblioteką)
- npm build (sukompiliuoti projektą)

Node.js pasižymi tuo, jog net paprastas *HTTP* serveris nėra sukuriamas be išorinių bibliotekų pagalbos, todėl sukonfigūruoti norimas funkcijas yra įmanoma tik tai užims laiko. Ši technologija dalinai atitinka kriterijų, nes trūksta funkcionalumo kai galima paleisti testavimo biblioteką. (!)

Deno technologija kaip ir minėjau yra geresnė *Node.js* versija su ištaisytomis *Node.js* architektūrinėmis klaidomis, todėl turi gan nemažą sąrašą įrankių skirtų darbui su *JavaScript*.

Sąrašas įrankių kurie yra sukurti *Deno* technologijoje[26]:

- deno init (naujo projekto kūrimas)
- deno bench (*JavaScript* kodo vykdymo greičio lyginimui)
- deno compile (programinio kodo kompiliavimui)
- deno install (įdiegti biblioteką)
- deno jupyter (galimybė naudoti Jupyter branduolį per *Deno API* sąsają)
- deno info (bibliotekos priklausomybių sąrašo sudarymas)
- deno doc (sukuria JSDoc komentarą norimam failui)
- deno fmt (failų formatavimo įrankis)
- deno lint (kodo klaidų žymėjimo įrankis *JavaScript* ir *TypeScript* programavimo kalboms)
- deno task (galima apsirašyti užduotis, kurios vykdys aprašytas komandas)
- deno test (testavimo biblioteka)

[25] NPM dokumentacijos puslapis apie integruotus įrankius https://docs.npmjs.com/cli/v6/commands

[26] Deno dokumentacijos puslapis apie integruotus įrankius https://docs.deno.com/runtime/manual/tools#built-in-tooling

- deno vendor (galimybė parsiųsti norimo failo bibliotekas į specifinį folderį)

Deno turi didesnį nei kriterijuje aprašytą kiekį reikalingų įrankių, todėl ši technologija atitinka kriterijų. (+)

Bun.js kūrėjai apibūdinant technologiją rašo jog tai yra „viskas viename“ įrankių rinkinys skirtas *JavaScript* ir *TypeScript* programoms ir visa tai yra vienoje „bun“ komandoje.

Sąrašas įrankių kurie yra sukurti *Bun.js* technologijoje[27]:

- bun run (paleisti *JavaScript* ar *TypeScript* failą)
- bun install (įdiegti biblioteką)
- bun test (testavimo biblioteka)
- bun init (tuščio naujo projekto kūrimas)
- bun create (naujo projekto sukūrimas pagal aprašytą šabloną)
- bun x (automatiškai įrašo bibliotekas ir jas paleidžia)
- bun package (projekto sukompiliavimas ir paruošimas diegimui)

Bun.js taip pat turi didesnį nei kriterijuje aprašytą kiekį reikalingų įrankių, todėl ši technologija atitinka kriterijų. (+)

3.2.6. TypeScript palaikymas vykdymo aplinkos kompiliatoriuje

Node.js technologijos vienintelė palaikoma kalba yra *JavaScript*. Tuo metu kai buvo išleista *Node.js* technologija *TypeScript* programavimo kalba nebuvo sukurta. Ji atsirado gerokai vėliau 2012 metais, jog padėtų spręsti *JavaScript* programavimo kalbos problemas[28]. Galimybė naudoti *TypeScript* kartu su *Node.js* yra, norint tai atlikti reikia pridėti *NPM TypeScript* biblioteką[29] ir ją sukonfigūruoti, tačiau pats *Node.js* kompiliatorius neturi tokio palaikymo. (**X**)

Deno kaip geresnė *Node.js* versija pridėjo *TypeScript* failų palaikymą kompiliatoriuje, todėl nereikia atlikti papildomų veiksmų norint naudoti *TypeScript* programavimo kalbą. (+)

Bun.js taip pat nepraleido progos palaikyti *TypeScript* be papildomos konfigūracijos. (+)

3.2.7. Bibliotekų teisių apribojimas kaip saugumo įrankis

Node.js dažnai yra minima kaip prastą saugumą turinti technologiją. Kiekviena technologija turi savo saugumo spragų, o viena iš žinomų *Node.js* saugumo spragų yra *NPM* bibliotekų

[27] Bun.js technologijos GIT puslapis https://github.com/oven-sh/bun

[28] *TypeScript* kūrėjų paaiškinimas kodėl buvo sukurtas TypeScript https://www.typescriptlang.org/why-create-typescript

[29] *TypeScript NPM* biblioteka https://www.npmjs.com/package/typescript

naudojimas. *NPM* bibliotekos *Node.js* technologijoje gauna visas teises, tai reiškia, kad jos gali atlikti net OS lygio veiksmus be jokių teisių suteikimo veiksmų. Jeigu *NPM* bibliotekos yra naudojamos neatsargiai, tai gali sukelti saugumo problemų, kaip ir jau anksčiau minėtame straipsnyje, kai buvo vagiami sistemos kintamųjų reikšmės, taip gaunant prieigą prie įvairių sistemų. *Node.js* 21.2.0 versijoje yra pridėtas teisių valdymo mechanizmas, apribojantis prieigą prie konkrečių išteklių vykdymo metu[30]. Ši funkcija yra nestabili(kūrimo stadijoje), todėl šiuo metu *Node.js* neturi paruoštos bibliotekų teisių apribojimo galimybės. (X)

Deno norėdamas ištaisyti bibliotekų saugumo spragas įdiegė teisių apribojimus. Tokie veiksmai kaip: interneto prieiga, failų sistemos prieiga, sistemos kintamieji, procesų kūrimas yra pagal numatytus nustatymus iškarto apriboti, norint atlikti šiuos veiksmus reiks specifiškai suteikti prieigą savo norimam veiksmui atlikti. (+)

Bun.js taip pat pritaikė teisių apribojimo funkciją, tačiau patikrintoms bibliotekoms tos teisės jau yra apibrėžtos ir to daryti patiems nebereikės. Tai reiškia, kad nežinomos bibliotekos bus apribotos ir programuotojas bus pats atsakingas kokias teises reikia suteikti tai bibliotekai. (+)

3.2.8. Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas

Išleidus *Node.js* 2009 metais tokios sąvokos kaip „hot reload“ dar nebuvo, todėl norint matyti kodo pakeitimus *Node.js* serverį reikėdavo iš naujo paleisti, taip sulėtinant programavimo procesą. Šiais laikais dažniausiai kiekviena populiari technologija turi įrankį skirtą stebėti failų pasikeitimus, juos sukompiliuoti ir pakeisti be būtinybės perkrauti serverį norint pamatyti savo pakeitimus, tačiau *Node.js* neturi tokio funkcionalumo. Norint tai turėti reikia įdiegti ir sukonfigūruoti papildomą *NPM* biblioteką, bet ir po pakeitimų automatiškai perkraus serverį ir neišsaugos būsenos, taip įrodant jog *Node.js* tiesiog nepalaiko šio funkcionalumo, bet jis gali būti sukonfigūruotas. (!)

Node.js kūrėjo tarp pabrėžtų šios technologijos problemų neišskyrė „hot reload“ funkcionalumo trūkumo, todėl tikėtis šios funkcijos tiesiogiai sukurtos *Deno* technologijoje nereikėjo tikėtis. Analogiškai kaip ir *Node.js* taip pat ir *Deno* norint turėti šią funkciją reikia susikonfigūruoti biblioteką, kuri stebėtų failų pakeitimus, juos sukompiliuotų ir pakeistų neperkraunant serverio, tačiau išsaugoti būsenos negali, nes po pakeitimų yra perkraunamas serveris. (!)

Kadangi *Bun.js* nori būti įrankis su visais būtiniausiais įrankiais, tai vienas iš jų yra „hot reload“ funkcionalumas, kadangi šiais laikais dauguma programuotojų neįsivaizduoja projektų

[30] *Node.js* teisių apribojimo mechanizmas https://nodejs.org/api/permissions.html#process-based-permissions

prie kurių dirba be greito programinio kodo pakeitimų matymo. Taip pat *Bun.js* žengė dar vieną papildomą žingsnį norint sutaupyti laiką. Ši technologija geba išsaugoti serverio būseną, taip papildomai sutaupant laiko atkartojant serverio būseną, pvz. prisijungimas prie sistemos, formos užpildymas ir kita. Tai leis dar greičiau programuotojams rašyti ir testuoji programinį kodą, taupant laiką kuris būtų skirtas pasikartojantiems veiksmams, kurie gali būti šitaip automatizuoti. (+)

3.3. Vertinimo kriterijų palyginimo rezultatai

Žemiau esančioje lentelėje galite matyti kiekvienos technologijos kriterijų įvertinimus. Apibendrinti kiekvienos technologijos rezultatai:

- Node.js – 2 +, 3 !, 3 X
- Deno – 5 +, 3 !
- Bun.js – 7 +, 1 X

1 lentelė. JavaScript serverio vykdymo aplinkų kriterijų palyginimo lentelė

Kriterijus	Node.js	Deno	Bun.js
Technologijos branda	+	!	X
Node Package Manager bibliotekų registro palaikymas ir architektūra	+	!	+
Optimizuotas Node Package Manager modulių saugojimas	X	+	+
ECMAScript modulių importavimas kaip technologijos standartas	!	+	+
Technologijoje integruoti įrankiai palengvinantys darbą su JavaScript	!	+	+
TypeScript palaikymas vykdymo aplinkos kompiliatoriuje	X	+	+
Bibliotekų teisių apribojimas kaip saugumo įrankis	X	+	+
Keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas	!	!	+

Pagal šiuos kriterijus ir jų vertinimus galime teigti, jog analizuojamos *JavaScript* serverio vykdymo aplinkos kuo jos yra naujesnės tuo labiau turi daugiau papildomų funkcijų. Daugiausia kriterijų atitiko naujausia technologija *Bun.js*, jos vienintelis ir didžiausias trūkumas, jog tai yra nesenai išleista technologija, kuriai reikia laiko ir trūkstamų funkcijų įgyvendinimo jog būtų galima laikyti ja išbandyta ir subrendusia technologija. Antrąja vietą pagal atitinkamus kriterijus užėmė *Deno* technologija. Ši technologija taip pat yra mažiau populiari dėl jos naujumo, nepalaiko *NPM* architektūros ir neturi vienos iš naujausių funkcijų, gyvų kodo pakeitimų programavimo metu kai yra išsaugoma programos būsena. Paskutinė vieta pagal atitinkamus kriterijus atiteko

Node.js technologijai. Ši technologija yra labiausiai populiari *JavaScript* serverio vykdymo aplinka, bet ji buvo išleista pati pirmoji, todėl daug funkcijų, kurias siūlo naujai išleistos vykdymo aplinkos ši technologija tiesiog neturi arba atitinka nepilnai, kai yra reikalingas papildomas konfigūravimas norint naudoti trūkstamą funkcionalumą, kai tuo tarpu *Deno* ir *Bun.js* kūrėjai šias funkcijas iškarto įdiegė pačioje technologijoje.

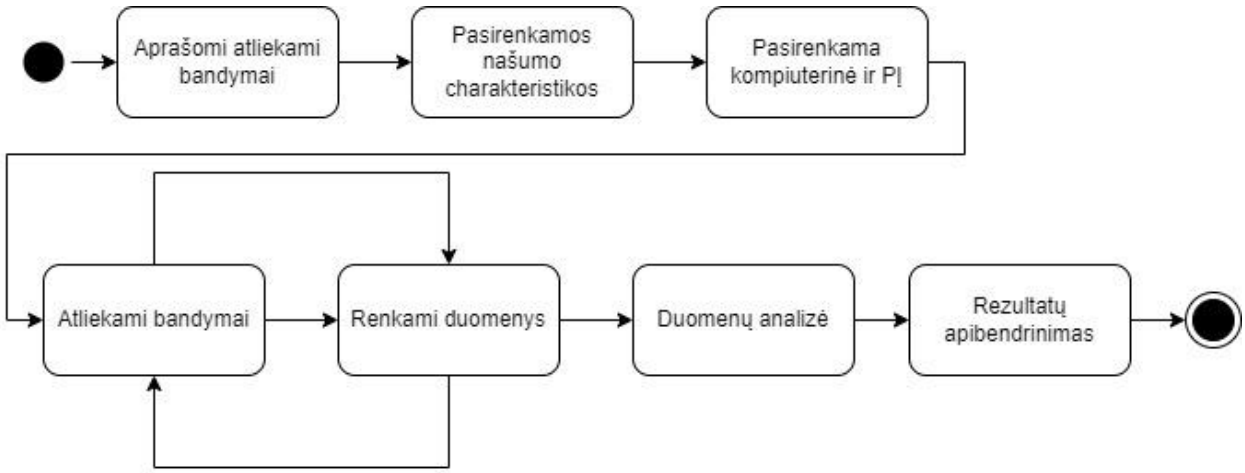
4. Eksperimentinė tyrimo darbo dalis

Keliuose šio darbo skyriuose buvo apžvelgiami kiekvienos technologijos privalumai ir trūkumai, aprašyti vertinimo kriterijai ir pagal juos lygintos technologijos, tačiau tai buvo daroma teoriniame lygmenyje. Šiame skyriuje bus atliekamas realus scenarijų testavimas, kurių metu bus išmatuojamos charakteristikos, atliekama lyginamoji analizė. Lyginamosios analizės metu bus lyginamos kiekvienos technologijos charakteristikos, kurios technologija išmatuotos charakteristikų rezultatai yra geresni, bei aptariami jų rezultatai.

Eksperimento vykdymo eiga bus vykdoma šiais etapais:

1. Atliekamų bandymo aprašymas – aprašomas pilnas sąrašas atliekamų bandymų ir priežastys kodėl jie buvo pasirinkti.
2. Našumo charakteristikų pasirinkimas – detaliai aprašoma kaip bus matuojamas našumas ir kokios našumo charakteristikos yra pasirenkamos.
3. Kompiuterinės ir programinės įrangos pasirinkimas – detaliai aprašoma koks bus naudojamas kompiuteris bandymams atlikti, kokia programinė įranga bus naudojama išmatuoti našumo charakteristikas, kokios *JavaScript* vykdymo aplinkų versijos bus naudojamos bandymų metu.
4. Bandymų atlikimas – kiekvienas aprašytas bandymas atliekamas atskirai jam sudarytu atveju.
5. Duomenų rinkimas – gauti bandymo rezultatai yra išsaugomi ir suvedami.
6. Duomenų analizė – pagal gautus duomenis yra sudaromi grafikai ir analizuojami duomenys.
7. Rezultatų apibendrinimas – visi bandymų rezultatų duomenys yra suvesti į lentelę, apskaičiuojami geriausi rezultatai kiekvieno matavimo metu, nustatoma našiausia technologija.

Žemiau paveikslėlyje yra pateikiama eksperimento vykdymo eigos schema.



pav. 13 Eksperimento vykdymo eigos schema

4.1. Atliekami bandymai

Atliekami bandymai yra pasirinkti pagal literatūroje išvardintus pačios pirmosios *JavaScript* vykdymo aplinkos *Node.js* technologijos panaudojimo išvardytas teigiamas, neigiamas savybes ir dažnai atliekamus veiksmus(15).

JavaScript vykdymo aplinkų aprašymo metu *Node.js* buvo minėta, jog ši technologija pasižymi neblokuojančiomis įvesties ir išvesties operacijų veikimu, daugiau nei viena užklausa gali būti apdorojama vienu metu. Visos gautos užklausos yra apdorojamos ir įvykdomos tvarkingu eiliškumu. Kitos konkuruojančios technologijos dėl savo kitokios architektūros ir veikimo sunkiau susidoroja su daug užklausų vienu metu lyginant su *Node.js* (15). Įvesties ir išvesties veiksmai yra tokie kaip *HTTP* užklausų vykdymas, failų skaitymas ar rašymas į diską, bendravimas su duomenų baze. Šie veiksmai buvo pasirinkti kaip keletas bandymo atvejų:

- *HTTP* serverio užklausos atsakymo greitis
- Tekstinio failo skaitymas
- *SQL* duomenų bazės serverio užklausos įvykdymo laikas

Kita *Node.js* technologijos pusė, dėl savo neblokuojančio veikimo ir vienos gijos architektūros ši technologija prastai veikia atliekant intensyvius skaičiavimus kai yra naudojamas procesorius (15). Su šiuo trūkumu yra susiduriama ir kituose vykdymo aplinkose, todėl reikia nustatyti, kuri technologija našiausiai vykdo užduotis, kurios intensyviai naudoja procesoriaus resursus. Bus atliekama skaičiavimams imli užduotis – fibonačio skaičiaus skaičiavimas.

Tai pat viena iš *Node.js* technologijos esminių dalių *NPM* bibliotekų registras. Labai dažnas veiksmas atliekamas naudojant *NPM* yra bibliotekų įdiegimas. Norint pridėti jau įgyvendintą funkcionalumą, kuris yra bibliotekoje ar pasiruošti projektą darbui reikia įsidiegti trūkstamas

bibliotekas, nes kitu atveju projektas nesusikompiliuos. Dažnu atveju projektas naudoja dešimtis bibliotekų dėl ko įdiegimo trukmė kartais užima nemažą dalį laiko, o laikas kurį praleidžiame laukdami yra iššvaistytas laikas, kuris neatneša jokios vertės arba nuostolį įmonei, nes bus atliekamas mažesnis kiekis darbų. Greitesnis bibliotekų diegimo laikas taip pat gali sutrumpinti projekto paruošimo laiką diegimui į pasirinktą aplinką. Visais atvejais trumpesnis bibliotekos įdiegimo laikas reikš sutaupytus pinigus, todėl bus lyginamas tos pačios bibliotekos įdiegimo greitis skirtinguose vykdymo aplinkose.

Galutinis sąrašas atliekamų bandymų:

- *HTTP* serverio užklausos atsakymo greitis
- Tekstinio failo skaitymas
- *SQL* duomenų bazės serverio užklausos įvykdymo laikas
- Skaičiavimas imli užduotis
- Bibliotekos įdiegimo trukmė

4.2. Našumo charakteristikų pasirinkimas

Programinė įranga ir įvairios sistemos yra kuriamos, jog būtų galima palengvinti, pagreitinti, padidinti pelną įvairių verslo atliekamų funkcijų. Kuo sėkmingiau yra išvystomas produktas, tuo daugiau naudos tai gali atnešti verslui. Programinės įrangos užsakovai yra suinteresuoti jog programinė įranga, kuri yra kuriama jiems vykdys visas jų apibrėžtas funkcijas ir tai darys tinkamai, tačiau norint užtikrinti programinės įrangos kokybę reikia sekti programinės įrangos kokybės standartą (25). Šis standartas susideda iš apibrėžtų programinės įrangos charakteristikų, kurios vertinant apibudina programinės įrangos produkto savybes. ISO/IEC 25010 kokybės standartas išskiria šias charakteristikas: naudojamumas, palaikomumas, našumas/efektyvumas, funkcionalumo tinkamumas, patikimumas, saugumas, suderinamumas, perkeliamumas (25).

Viena iš charakteristikų yra efektyvumas/našumas, tai kas bus tiriama šioje eksperimentinėje dalyje. Ši charakteristika parodo našumą, palyginti su naudojamų išteklių kiekiu nustatytomis sąlygomis. Šią charakteristiką sudaro šios dalinės charakteristikos:

- Laiko charakteristika – produkto ar sistemos reakcijos laikas, per kurį sistema reaguoja į įvykį. Pagrindiniai laiko elgsenos rodikliai yra atsakymo laikas, apdorojimo laikas ir pralaidumo rodiklis.
- Išteklių panaudojimas – produkto ar sistemos naudojamų išteklių kiekio ir tipų, kai jie atlieka savo funkcijas, atitikimo reikalavimams laipsnis. Tai reiškia, kad reikia veiksmingai valdyti išteklius, įskaitant procesorių, atmintį, disko vietą, tinklo pralaidumą ir kitus, siekiant užtikrinti, kad jie tenkintų poreikius ir būtų kuo mažiau

švaistomi arba nepakankamai naudojami. Pagrindiniai rodikliai: procesoriaus, atminties ir disko vietos panaudojimas.

- Talpa – produkto ar sistemos parametrų maksimalių ribų atitikimo reikalavimams laipsnis. Pagal tai nustatoma ar sistema gali suvaldyti didėjančias apkrovas, daugiau duomenų rinkinių neviršijant nustatytų ribų. Įvertinus pajėgumą lengviau pastebėti galimus apribojimus ir garantuoja, kad sistema gali būti plečiama ir efektyviai atitikti augančią paklausą.

Remdamiesi šiuo standartu ir naudodami šią charakteristiką ištirsime kiek kiekvienas bandymas užtrunka laiko, sunaudoja išteklių ir kaip reaguoja į didesnį kiekį vykstančių užklausų tuo pačiu metu. Kiekvienai technologijai *Node.js, Deno* ir *Bun.js* bus sukurti tokie patys scenarijai, bus išmatuotas laikas kiek užtrunka scenarijus ir jeigu scenarijus apima skaičiavimo veiksmus bus išmatuojami sunaudoti ištekliai, bei kaip keičiasi reikalingo laiko trukmė kai yra skirtingi kiekiai lygiagrečiai ateinančių užklausų.

4.3. Našumo charakteristikų apskaičiavimas

Kiekvienai bandomai technologijai bus naudojamos specializuotos programos įvertinti programos veikimo charakteristikas. Bandymų tikslas yra apskaičiuoti atminties sąnaudos, atsakymo laiko trukmę ir vykdymo laiką.

HTTP užklausų našumo įvertinimui bus naudojamas „Bombardier“ (26) lyginamosios analizės įrankis. Naudodamas šį įrankį nustatysime kaip našiai veikia *Node.js*, *Deno* ir *Bun.js* *HTTP* serveriai. Atliekant bandymus su *HTTP* užklausomis bus atliekami keli scenarijai, kurių metu bus keičiamas užklausų kiekis, lygiagretūs prisijungimai, jog būtų galima stipriai apkrauti serverius. Galime įvertinti serverių našumą pagal atsako laiką ir pralaidumą. nustatant jiems šias apkrovos aplinkybes. Laiko sąnaudų kiekis, pvz. atsako laikas (mediana, 95-oji procentilė, vidurkis ir maksimumas) ir kiek laiko užtruko atlikti norimą kiekį užklausų - visus šiuos duomenis pateikė „Bombardier“.

Išmatuoti didžiausią programos atminties naudojimą jų vykdymo metu buvo naudota GNU operacinės sistemos komandinės eilutės įrankis „time“[31]. Naudodami šį įranki sužinosime kaip efektyviai yra naudojama atmintis didelių užklausų apkrovų metu.

Išmatuoti atskirą programinio failo scenarijų buvo pasirinktas „Hyperfine“[32] komandinės eilutės įrankis skirtas našumo įvertinimui. Šio įrankio tikslas apskaičiuoti kuo tikslesnį vykdymo

[31] Time bibliotekos puslapis https://www.gnu.org/software/time/
[32] Hyperfine bibliotekos GIT puslapis https://github.com/sharkdp/hyperfine

laiką. Šis įrankis veikia taip, jog norimas programinis kodas yra paleidžiamas 10 kartų, kurių metu yra gaunami tikslūs statistiniai duomenys apie vykdymo trukmę: vykdymo trukmės vidurkį, ilgiausią ir trumpiausią vykdymo trukmę. Naudodami šį įrankį galėsime palyginti skirtingų vykdymo aplinkų vykdymo laikus atliekant sudėtingus skaičiavimus.

Yra būtina pasirinkti tinkamus įrankius tokiems scenarijams kur greitis ir tikslumas yra būtinas. „Bombardier“ įrankis yra parašytas naudojant „Go“ programavimo kalbą ir naudoja „fasthttp“ biblioteką, kuri užtikrina didelę greitaveiką (26). „Hyperfine“ yra parašytas naudojant „Rust“ programavimo kalbą, kuri taip pat veikia labai našiai. Naudojant šiuos įrankius bus galima surinkti visus charakteristikų duomenis, juos atvaizduoti ir palyginti.

4.4. Naudojama kompiuterinė ir programinė įranga

Norint atlikti kuo tikslesnius bandymus reikalinga specifinė bandymų aplinka, todėl visų bandymų metu bus naudojama ta pati kompiuterinė ir programinė įranga. Tai leis atlikti patikimus bandymus tarp *Node.js*, *Deno* ir *Bun.js* vykdymo aplinkų. Naudojamo kompiuterio specifikacijos:

- Procesorius: Intel i7-1065G7 1.30GHz
- Atmintis: 8GB
- Operacinė sistema: Ubuntu 22.04.2

Visi bandymai bus atliekami Ubuntu operacinės sistemos komandinėje eilutėje. Bus įdiegtos programos norint įvertinti našumo charakteristikas:

- Bombardier – v1.2.6
- Hyperfine – v1.12.0
- GNU time – v1.9.0

JavaScript vykdymo aplinkų naudojamos versijos:

- Node.js – v18.18.2
- Deno – v1.37.2
- Bun.js – v1.0.6

Apsibrėžus bandymų atlikimo konfigūraciją bus lengviau ir tiksliau įvertinti vykdymo aplinkas, gauti naudingos informacijos kaip našiai veikia šios technologijos. Bandymų rezultatai padės nustatyti našumo charakteristikas.

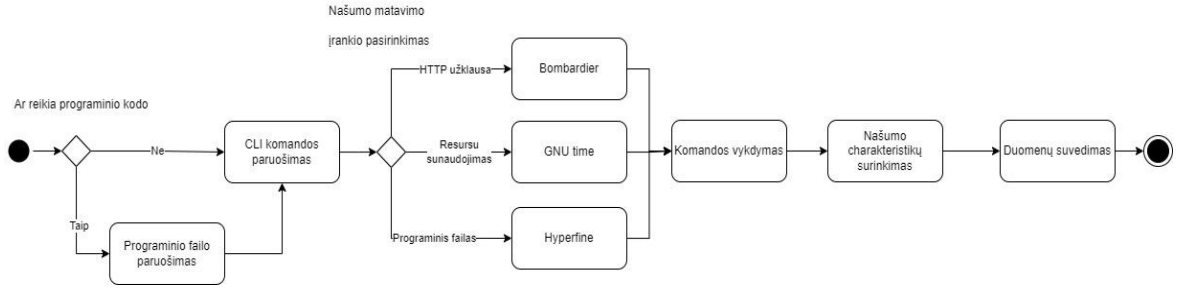
4.5. Bandymai

Nors visos bandomos technologijos yra *JavaScript* vykdymo aplinkos ir jos palaiko šią programavimo kalbą tačiau jos turi skirtingas savo technologijų *API* sąsajas, todėl bandymo scenarijus, kuris naudoja technologijos *API* sąsają bus aprašytas atskirame faile kiekvienai technologijai.

Bandymo atlikimo eiga:

1. Nustatoma ar reikalingas programinis kodas bandymui atlikti
    a. Aprašomas programinis kodas bandomai technologijai (jeigu tai yra reikalinga norint atlikti bandymą)
2. Paruošiama komandinės eilutės komanda
3. Pasirenkamas našumo matavimo įrankis
    a. HTTP užklausai Bombardier
    b. Maksimaliam resursų sunaudojimo apskaičiavimui GNU time
    c. Programinio failo paleidimo našumo charakteristikų apskaičiavimui
4. Vykdomas programinis kodas naudojant komandinės eilutės įrankius
5. Bandymo metu gautų našumo charakteristikų duomenų surinkimas
6. Gautų našumo charakteristikų rezultatų išsaugojimas ir suvedimas į lentelę

Žemiau yra pateikiama bandymo atlikimo eigos schema.



pav. 14 Bandymo atlikimo eigos schema

4.5.1. HTTP serverio užklausos atsakymo greitis

Kliento serverio architektūroje sistemų bendravimui tarpusavyje naudojami įvairūs protokolai: *HTTP*[33], *FTP*, *SMTP* (23). Naudojant *HTTP* protokolą galima keistis failų, pvz. vaizdai, tekstas tarp kliento ir serverio. *HTTP* veikia naudodamas užklausos ir atsakymo protokolą. Naudodamiesi *Node.js*, *Deno* ir *Bun.js* technologijomis vienas iš pagrindinių atvejų yra *HTTP*

[33] HTTP protokolo aprašymas https://www.ietf.org/rfc/rfc2616.txt

serverių kūrimas. Vienas iš būdų nustatyti šių vykdymo aplinkų našumą yra išbandyti šių technologijų *HTTP* serverių efektyvumą įvairiomis sąlygomis. Kaip keičiasi serverių efektyvumas didėjant užklausų kiekiui ir daugėjant lygiagrečių užklausų kiekiui.

Šiam bandymui atlikti bus atlikti trys scenarijai. Kiekviename scenarijuje bus atliekamas milijonas *HTTP* užklausų, tačiau kiekvieno scenarijaus metu bus didinamas lygiagrečių užklausų kiekis. Bus pradedama su 10, po to 100 ir 500 lygiagrečių užklausų. Visa tai atlikti padės anksčiau aprašytas įrankis *HTTP* našumo testavimui „Bombardier“. Norint apskaičiuoti bandymo metu sunaudotą didžiausią atminties kiekį bus pasitelkta anksčiau aprašytas GNU operacinės įrankis „time“. Šie įrankiai padės nustatyti *HTTP* užklausos atsakymo trukmes, visą laiką atlikti scenarijui ir didžiausią sunaudotą atminties kiekį.

Nustatyti didžiausią sunaudotą atminties kiekį vykdymo aplinkose bus naudojamos komandos:

```
/usr/bin/time node node-http.js
time deno run –allow-net deno-http.js
time bun bun-http.js
```

15 pav. Didžiausios sunaudotos atminties kiekio komandos kiekvienai technologijai

Išmatuoti kiekvieną *HTTP* užklausų scenarijų bus naudojamos šios „Bombardier“ komandos:

```
Bombardier -c 10 -n 1000000 -l http://localhost:3000
Bombardier -c 100 -n 1000000 -l http://localhost:3000
Bombardier -c 500 -n 1000000 -l http://localhost:3000
```

16 pav. Komandos skirtos išmatuoti kiekviena iš HTTP užklausų scenarijų su Bombardier

4.5.2. Tekstinio failo skaitymas

Failo skaitymas yra įvesties ir išvesties operacija. Aprašant *Node.js* technologiją ir jos privalumus buvo paminėta jog su įvesties ir išvesties operacijomis *Node.js* puikiai susitvarko. *Node.js, Deno* ir *Bun.js* technologijos skirtingai įgyvendinta failo skaitymą, todėl kiekvieno iš jų išpildymas yra skirtingas. Kadangi *Deno* ir *Bun.js* bando būti geresnė technologija nei *Node.js* todėl šis bandymas yra puikus būdas išmatuoti, kuri technologija geriausiai susitvarkys su didžiulio tekstinio failo skaitymo užduotimi.

Šio bandymo metu bus atliekamas 6.31Mb dydžio, 128457 eilučių ir 6617121 simbolių tekstinio failo skaitymas. *HTTP* serveris nebus naudojamas šiam bandymui atlikti, bus fokusuojamasi tik į failo skaitymo vykdymą, todėl šis bandymas ir jo našumas bus išmatuojamas naudojant „Hyperfine“ komandinės eilutės našumo matavimo įrankį. Apskaičiuoti didžiausią sunaudojamą atminties kiekį bus pasitelkta anksčiau aprašytas GNU operacinės įrankis „time“.

Kaip atrodo tekstinio failo maža dalis:

The Project Gutenberg EBook of The Adventures of Sherlock Holmes
by Sir Arthur Conan Doyle
(#15 in our series by Sir Arthur Conan Doyle)

Copyright laws are changing all over the world. Be sure to check the
copyright laws for your country before downloading or redistributing
this or any other Project Gutenberg eBook.

Komandinės eilutės komanda kuri bus naudojama atlikti šį bandymą:

```
hyperfine 'node node-read.js' 'deno run --allow-read deno-read.js' 'bun bun-read.js'
```

17 pav. Komandinės eilutės komanda skirta išmatuoti tekstinio failo skaitymo bandymo rezultatus

4.5.3. SQL duomenų bazės serverio užklausos įvykdymo laikas

Kliento serverio architektūroje serverio viena iš funkcijų yra keistis ar saugoti duomenis, tačiau dažniausiai serveris tėra tarpinė programinė įranga. Trejų pakopų kliento serverio sistemos architektūra susideda iš kliento kompiuterio, duomenų bazės serverio ir pačio aplikacijos serverio (23). Šio tipo architektūrą galima plėsti iki N lygių, nes pats aplikacijos serveris gali bendrauti ne tik su viena duomenų baze, bet tai gali būti N duomenų bazių serverių ar kitų aplikacijų serverių. Tai leidžia vienam serveriui būti atsakingam už daugelį klientų, bei turėti galimybę komunikuoti su neribotu kiekiu kitų serverių, nes aplikacijos serveris yra kaip tarpininkas, kuris gali turėti atskirą logiką ir kiti naudojami serveriai gali būti saugomi nuo išorinės prieigos. Šioje architektūroje vienas iš dažniausiai atliekamų veiksmų yra duomenų grąžinimas klientui, todėl bus atliekamas bandymas kai *JavaScript* vykdymo aplinka kreipiasi į duomenų bazės serverį ir matuojama per kiek laiko yra grąžinami duomenis iš duomenų bazės serverio.

Buvo pasirinkta *MySQL* duomenų bazė, ji yra antra populiariausia naudojama duomenų bazė pagal 2022 metų statistiką[34]. Pasirinkta ši duomenų bazė, nes ji yra nemokama atviro kodo duomenų bazė. Norint naudotis Oracle duomenų baze reikia sumokėti licencijos mokestį.

Bus sukurtas *MySQL* duomenų bazės serveris bei atliekama SELECT užklausa, kurios metu bus grąžinami lentelėje esantys duomenys. Bus sukurta lentelė **authors**, kuri susidės iš šių

[34] Pačios populiariausios duomenų bazės 2022 https://statisticsanddata.org/data/most-popular-databases-2006-2022/

stulpelių:

2 lentelė. MySQL duomenų bazėje esanti authors lentelės stulpelių sąrašas

Lauko pavadinimas	Tipas
id	int
first_name	varchar(50)
last_name	varchar(50)
Email	varchar(100)
birthdate	date
added	timestamp

Ši lentelė bus užpildyta 100 įrašų, kurie buvo sugeneruoti iš netikrų duomenų. Šis bandymas imituos serverio kreipimąsi į duomenų bazės serverį norint gauti *SQL* užklausos duomenis, šiuo atveju duomenis apie autorius. *SQL* duomenų bazės serveris veiks tame pačiame kompiuteryje.

HTTP serveris nebus naudojamas šiam bandymui atlikti, bus fokusuojamasi tik į *SQL* užklausos vykdymą ir rezultato grąžinimą iš serverio. Bendravimui su MySQL duomenų bazės serveriu bus naudojama „mysql2" *NPM* biblioteka[35]. Našumas bus išmatuojamas naudojant „Hyperfine" komandinės eilutės našumo matavimo įrankį. Apskaičiuoti didžiausią sunaudojamą atminties kiekį bus pasitelkta anksčiau aprašytas GNU operacinės įrankis „time".

```
hyperfine 'node node/index.js' 'deno run --allow-net --allow-read deno/main.ts'
'bun bun/index.ts'
```

18 pav. Komandinės eilutės komanda skirta išmatuoti SQL duomenų bazės serverio užklausos bandymo rezultatus

4.5.4. Skaičiavimams imli užduotis

Fibonačio skaičių skaičiavimas yra programavimo problema, kurią yra mokomasi išspręsti mokant matematikos ar informatikos. Fibonačio skaičių seka (Fn) - tai nuo nulio ir vieneto prasidedanti skaičių seka, sudaryta iš kurioje kiekvienas paskesnis skaičius yra lygus dviejų ankstesnių skaičių sumai. Fibonačio skaičių seką galima apibrėžti:

$$F_{n+1} = F_n + F_{n-1}$$

Fibonačio skaičių sekos skaičiavimą įgyvendinome rekursyviniu algoritmu. Fibonačio skaičių seka bandant apskaičiuoti rekursiškai tai atitinka laiko sudėtingumą, kuris auga eksponentiškai(O(2n)). Norint išbandyti *JavaScript* vykdymo aplinkų efektyvumą vykdant

[35] Bendravimui su MySQL serveriu skirta biblioteka https://www.npmjs.com/package/mysql2

sunkias skaičiavimo užduotis bus įgyvendintas šis algoritmas rekursyviniu algoritmu. Bandymo atveju bus ieškomas 35-asis Fibonači skaičių sekos narys. Bandymo metu nebus naudojami jokie spartinančios atminties metodai, tai leis užtikrinti sąžiningus bandymo rezultatus.

Šis bandymas ir jo našumas bus išmatuojamas naudojant „Hyperfine“ komandinės eilutės našumo matavimo įrankį. Šis skaičiavimas bus atliekamas 10 kartų ir bus išmatuojamas vidutinis laikas kiek laiko užtruko atlikti skaičiavimus. Šitoks matavimo būdas užtikrina tikslius rezultatus. Apskaičiuoti didžiausią sunaudojamą atminties kiekį bus pasitelkta anksčiau aprašytas GNU operacinės įrankis „time“.

```
hyperfine 'node fib-apskaiciavimas.js' ' 'deno run fib-apskaiciavimas.js' ' 'bun
fib-apskaiciavimas.js' '
```

19 pav. Komandinės eilutės komanda skirta išmatuoti skaičiavimams imlaus bandymo rezultatus

Rekursinė funkcija skirta apskaičiuoti fibonačio skaičių seką:

```
function fib(n) {
    if (n <= 0) return 0;
    if (n <= 1) return 1;
    if (n <= 2) return 2;
    return fib(n - 1) + fib(n - 2);
}
```

20 pav. Fibonačio skaičių sekos skaičiavimo JavaScript programinis kodas

4.5.5. Bibliotekos įdiegimo trukmė

Node Package Manager bibliotekų registro palaikymas ir architektūra buvo pasirinktas kaip vienas iš kriterijų, nes daugelis projektų naudoja papildomas bibliotekas, kurios yra reikalingos jų projektui, pvz. *TypeScript* kalbos ar minėta hot-reload biblioteka. Bandymo metu bus matuojama kaip kiekviena technologija greitai prideda biblioteką. *Node.js*, *Deno* ir *Bun.js* naudoja skirtingus įrankius darbui su bibliotekomis, todėl šio bandymo metu turėtume sužinoti, kurios technologijos įrankis veikia našiausiai.

Šiam bandymui atlikti buvo pasirinkta Express[36] *NPM* registro biblioteka. Tai yra viena iš labiausiai naudojamų karkasų, kurio tikslas yra suteikti patikimas *HTTP* serverių priemones, kurios būtų puikus sprendimas vieno puslapio programoms, svetainėms ar viešoms *API* sąsajoms.

Šis bandymas bus matuojamas su *GNU* operacinės sistemos komandinės eilutės įrankiu „time“. Šis įrankis padės nustatyti trukmė kiek laiko truko įvykdyti bibliotekos įrašymo komandą.

[36] https://www.npmjs.com/package/express

```
/usr/bin/time npm install express
/usr/bin/time deno install --allow-net --allow-read 'npm:express@4.18.2'
/usr/bin/time bun install express
```

21 pav. Komandinės eilutės komanda skirta išmatuoti bibliotekos įdiegimo trukmę

5. Bandymų rezultatai

Šiame skyriuje bus aptariami *Node.js*, *Deno* ir *Bun.js* technologijų bandymų rezultatai. Rezultatai buvo išmatuoti ankščiau minėtomis „Bombardier“, „Hyperfine“ ir „time“ komandinės eilutės įrankiais. Naudojant šiuos įrankius mums pavyko išmatuoti tiriamų technologijų svarbiausias našumo charakteristikas, kurios aprašytos ISO/IEC 25010 programinės įrangos įvertinimo kokybės standarte: atminties sunaudojimą, atsakymo laiką ir visą vykdymo laiką.

Bus atvaizduojami kiekvieno bandymo rezultatai, bei aptariami įvykdymo laikai, atminties sunaudojimas ir visas vykdymo laikas. Bus nustatyta, kuri technologija yra našiausia atlikto bandymo metu ir aptariami *JavaScript* serverio vykdymo aplinkų apribojimai.

5.1. HTTP serverio bandymo rezultatai

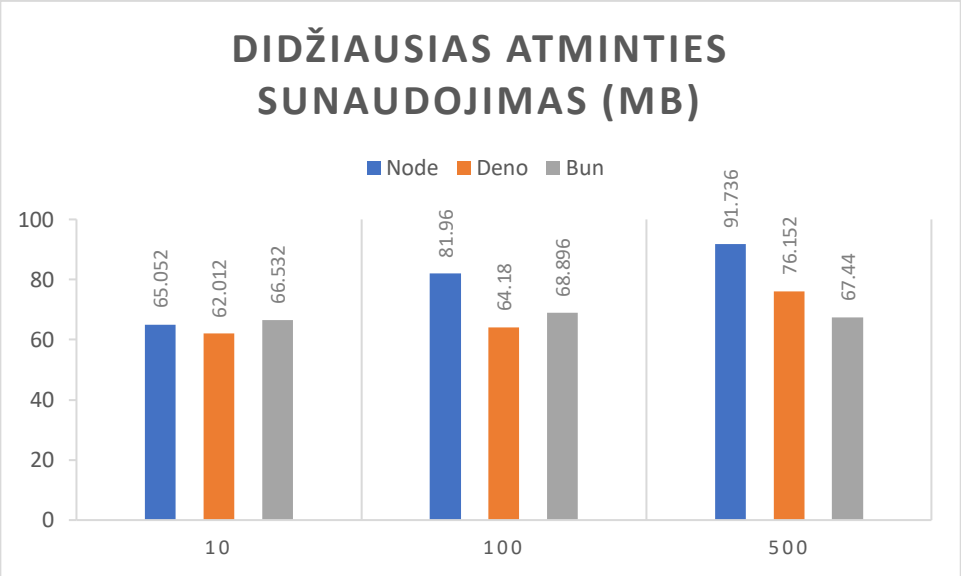


diagrama 1 Didžiausias atminties sunaudojimas

Pirmojo bandymo metu bandėme trejų vykdymo aplinkų *HTTP* serverio galimybes. Vienas iš atliktų matavimų didžiausias spartinančios atminties sunaudojimas. Šis rodiklis yra svarbus serverio našumo indikatorius kaip efektyviai yra naudojamasi spartinančia atmintimi. *Node.js* kiekvieno bandymo metu daugėjant lygiagrečiai vykstančių užklausų kiekiui šios technologijos sunaudojamos spartinančios atminties kiekis didėjo daugiausiai. Antroje vietoje pagal didžiausią

sunaudojamą spartinančios atminties kiekį – *Deno*. Ši technologija gan efektyviausiai susitvarkė su 10 ir 100 lygiagrečių užklausų kiekiu užimdama pirmą vietą pagal mažiausią sunaudotą spartinančiosios atminties kiekį, kai sunaudojamos spartinančios atminties kiekis paaugo tik nuo 62MB~ iki 64MB~, tačiau kai lygiagrečių užklausų kiekis padidėjo iki 100, sunaudojamos spartinančios atminties kiekis padidėjo net iki 76MB~, bet tai buvo vis tiek mažiau nei *Node.js* 91MB~. *Bun.js* sunaudojamos spartinančiosios atminties kiekis visais atvejais kito mažiausiai lyginant su kitomis išbandytomis technologijomis. *Bun.js* geriausiai pasirodė kai buvo bandomas didžiausias kiekis lygiagrečių užklausų. Atlikę ši bandymą ir palyginę pagal sunaudojamą didžiausią atminties kiekį galime teigti, jog *Bun.js* efektyviausiai išnaudoja spartinančią atmintį iš visų trejų lygintų technologijų kas gali padėti sumažinti išlaidas bei pagerinti aplikacijos greitaveiką.

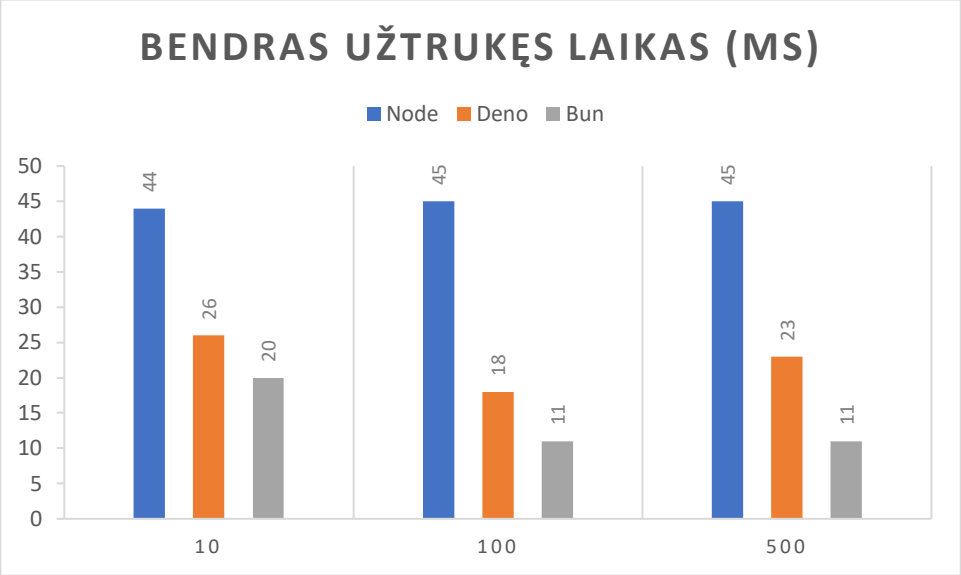


diagrama 2 Užtrukęs bandymo laikas

Apžvelgiant kitus *HTTP* serverio bandymo rezultatus pasirinkome bendrą bandymo laiką. Suskaičiavę kiekvienos technologijos bendrus bandymo laikus galime teigti jog ilgiausiai užtruko *Node.js* – 134 sekundes. Antroje vietoje *Deno* technologija užtrukusi 67 sekundes. Geriausiai šiame bandyme pagal užtrukusį laiką pasirodė *Bun.js*. Šiai technologijai reikėjo tik 42 sekundžių. Šis bandymas parodo jog *Bun.js HTTP* serveris veikia greičiau už *Node.js* ir *Deno* technologijas. Už *Node.js HTTP* serverį veikia 3,19 kartų greičiau, o už *Deno* 1,6 kartų greičiau. Šitie rezultatai parodo, kuri technologija geriausiai apdoroja didelį kiekį lygiagrečių užklausų naudojant HTTP serverį, kuris tik grąžina „Hello <technologijos pavadinimas>!” tekstą.

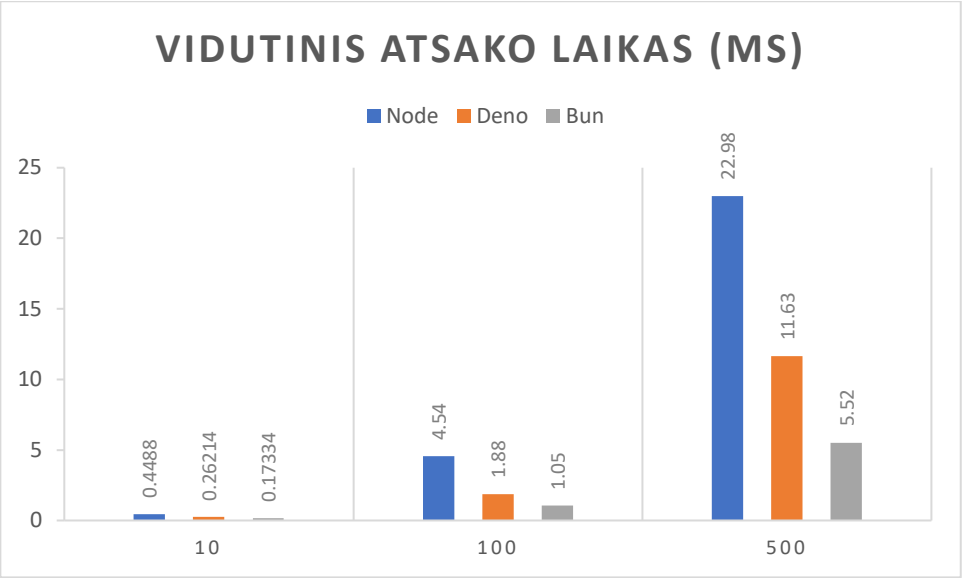


diagrama 3 Vidutinis HTTP užklausos vykdymo laikas

Kaip greit yra įvykdoma užklausa atsako vidutinis užklausos vykdymo laiko matavimas. Vaizduojamoje antroje diagramoje greičiausiai bandymai buvo atlikti *Bun.js* technologijos, analogiškai ir šiame matavime pirmąją vietą pagal trumpiausia užklausos vykdymo laiką visuose trejuose scenarijuose užima *Bun.js*, antrąja vietą užima *Deno*, o paskutinė vieta atitenka *Node.js*, ši technologija užtruko daugiausia laiko.

Apibendrinant šio bandymo rezultatus, prie nedidelio kiekio lygiagrečių užklausų visos lyginamos technologijos rodė panašius našumo charakteristikas, tačiau padidinus lygiagrečių užklausų kiekį buvo galima aiškiau matyti, kurios technologijos veikia našiau. *Node.js* pasirodė prasčiausiai iš visų trejų lyginamų technologijų. Visų matavimų metu *Node.js* rodė prasčiausius rodiklius, išskyrus sunaudojamas atminties kiekis buvo šiek tiek mažesnis nei *Bun.js* kai buvo atliekama 10 lygiagrečių užklausų. *Deno* technologija taip pat lenkė pagal sunaudojamos atminties kiekį *Bun.js* technologiją 10 ir 100 lygiagrečių užklausų metu, tačiau kai buvo padidintas lygiagrečių užklausų kiekis iki 500 *Bun.js* sunaudojo mažiausiai atminties. Jeigu lyginsime bendrą bandymų laiką ir vidutinį *HTTP* užklausos vykdymo laiką, visais atvejais yra užimamos vienodos vietos. Didžiausius laikus užtrunka *Node.js*, po to yra *Deno*, o mažiausiai laiko užtrunka *Bun.js* Pagal šį bandymą galime teigti, jog *Bun.js* turi geresnį *HTTP* užklausų pralaidumą ir efektyvumą lygiagrečių užklausų metu, lyginant su *Deno* ar *Node.js*. *Bun.js* gali greitai ir efektyviai naudojant atmintį valdyti didelius kiekius užklausų, todėl ši technologija yra puikus pasirinkimas programoms, kurioms reikalingas greitas užklausų apdorojimas ir mažesni išlaidų kaštai.

5.2. Tekstinio failo skaitymo bandymo rezultatai

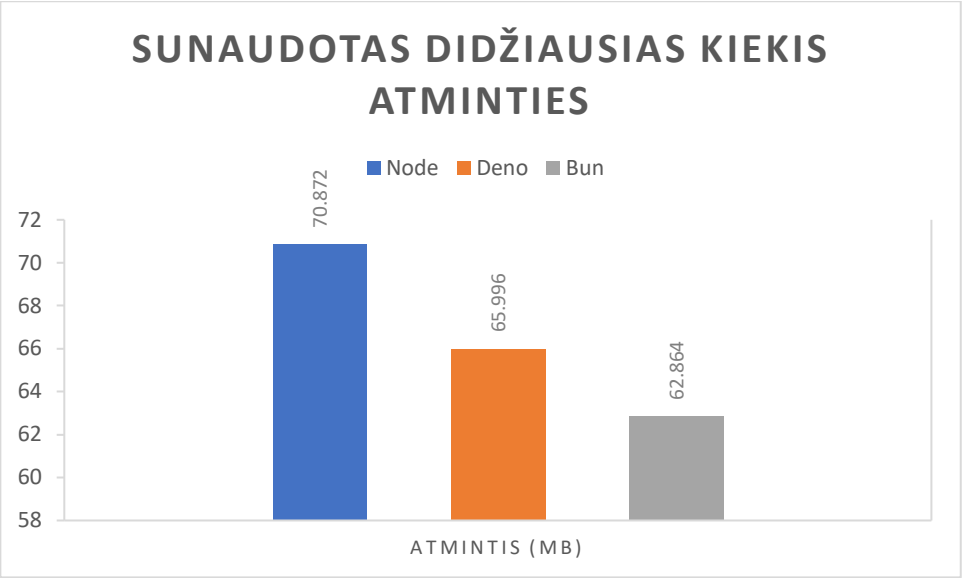


diagrama 4 Sunaudotas didžiausias atminties kiekis

Kitas iš atliktų bandymų buvo tekstinio failo skaitymas. Šios trys technologijos turi individualiai kurtus failo skaitymo metodus, todėl šio bandymo metu bus galima nustatyti, kurios technologijos failo skaitymo metodas yra įgyvendintas našiausiai. Vienas iš matavimų buvo spartinančiosios atminties sunaudojimas. Šio matavimo metu daugiausia atminties sunaudojo *Node.js*, antroje vietoje *Deno*, o mažiausiai atminties sunaudojo *Bun.js*.

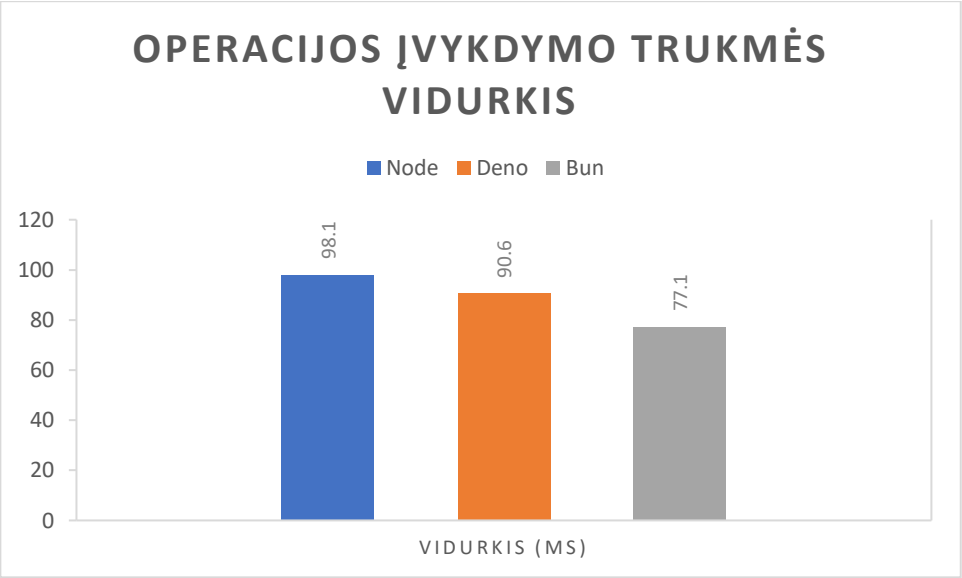


diagrama 5 Tekstinio failo skaitymo operacijos įvykdymo trukmės vidurkis

Kadangi šiai operacijai buvo naudojamas įrankis „Hyperfine”, norint gauti kuo tikslesnius matavimus programinio kodas buvo vykdomas 10 kartų. Jų metu buvo apskaičiuotas operacijos įvykdymo vidurkis. Šis vidurkis parodo analogiškus rezultatus lyginant su sunaudota atmintimi. Greičiausiai operacija buvo įvykdyta *Bun.js*, antroji vieta atiteko *Deno*, o paskutinė vieta *Node.js*.

Reikia pabrėžti jog lyginant *Deno* su *Node.js* operacijos vykdymo vidurkis buvo mažesnis tik 7,5ms, kai tuo tarpu lyginant *Bun.js* su *Node.js* operacijos vykdymo vidurkis buvo mažesnis 21ms, todėl yra matomas akivaizdus *Bun.js* technologijos pranašumas failo skaitymo operacijoje.

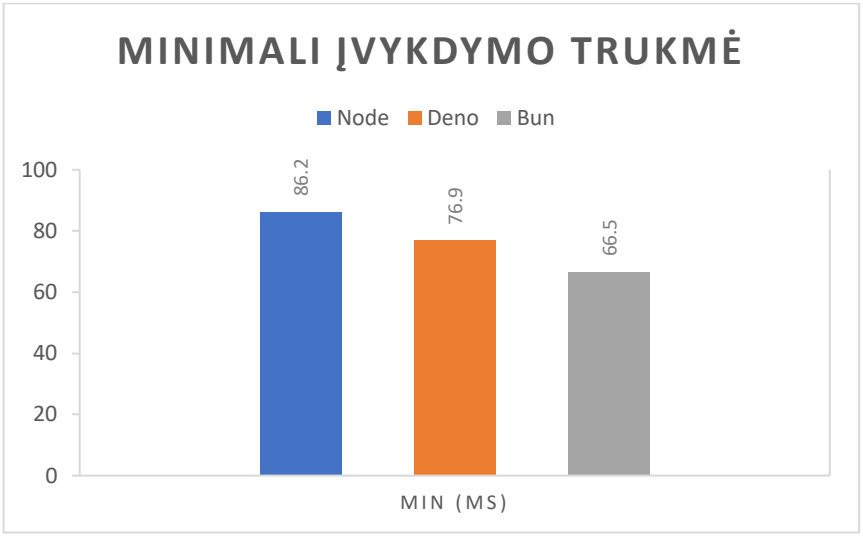


diagrama 6 Tekstinio failo skaitymo minimali vykdymo trukmė

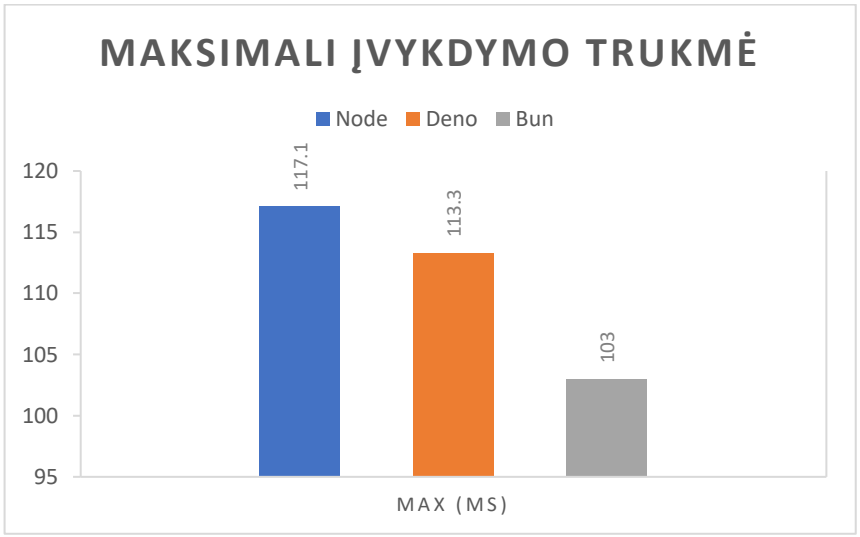


diagrama 7 Tekstinio failo skaitymo maksimali įvykdymo trukmė

Šio bandymo metu taip pat buvo apskaičiuotos minimalios ir maksimalios operacijos įvykdymo trukmės reikšmės. Šios reikšmės parodo, jog neįvyko jokių anomalijų vykdant programinį kodą, nes kaip ir operacijos vykdymo vidurkyje šios technologijos yra pasiskirsčiusios taip pat. Greičiausiai operacija yra įvykdymo *Bun.js* technologijos, antra vieta – *Deno*, o lėčiausiai operacija buvo įvykdyta *Node.js* technologijos.

5.3. SQL duomenų bazės serverio užklausos įvykdymo laikas

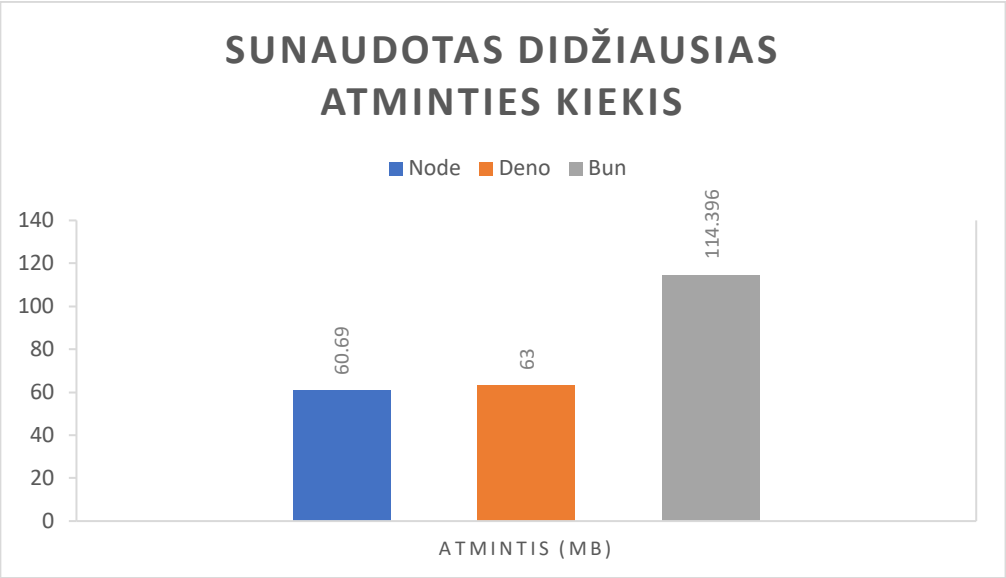


diagrama 8 SQL užklausos operacijos įvykdymo sunaudotas didžiausias atminties kiekis

Nors prieš tai buvusiuose bandymuose *Bun.js* buvo lyderis pagal našumą, tačiau kai yra dirbama su *MySQL* duomenų baze, ši technologija sunaudoja daugiausiai atminties. Antroje vietoje pagal sunaudojamą atmintį yra *Deno*, o optimaliausiai veikia *Node.js*. *Node.js* su *Deno* technologija rodė panašius rezultatus, o *Bun.js* sunaudojo beveik du kartus daugiau atminties. Tai yra pirmasis matavimas kur *Node.js* pirmauja pagal gautus rezultatus.

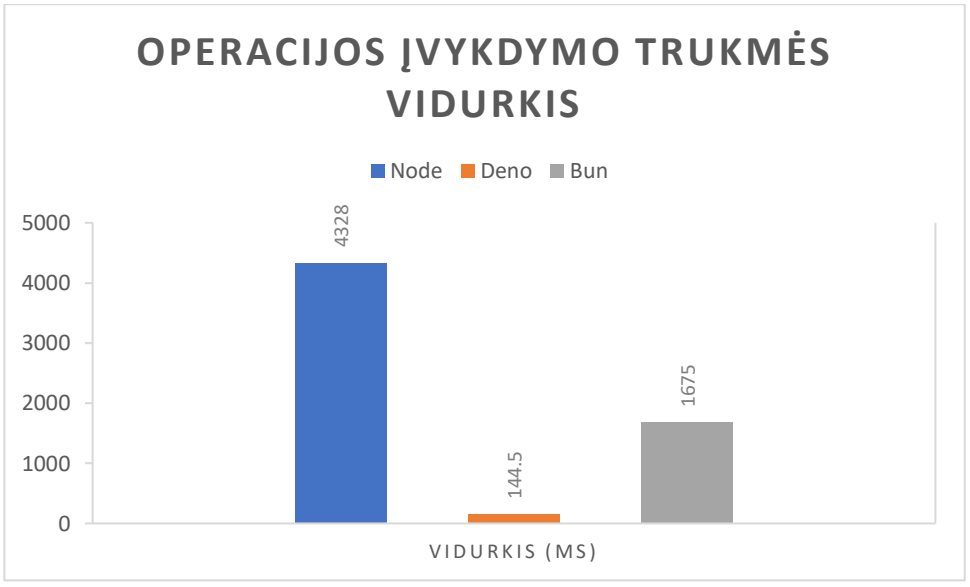


diagrama 9 SQL užklausos operacijos įvykdymo trukmės vidurkis

Nors *Node.js* vykdant *MySQL* serverio užklausą sunaudojo mažiausiai atminties tačiau operacijos įvykdymo trukmės vidurkis buvo didžiausias iš visų trejų lyginamų technologijų. Šio

bandymo rezultatai taip pat parodė nors *Bun.js* ir sunaudojo daugiau atminties, bet tai buvo vidutiniškai įvykdoma lėčiau nei naudojant *Deno* technologiją. *Deno* ypatingai greitai įvykdydavo bandymo operaciją užtrukdamas vidutiniškai tik 144,5ms, kai tuo tarpu *Bun.js* 1675ms, o *Node.js* – 4328ms.

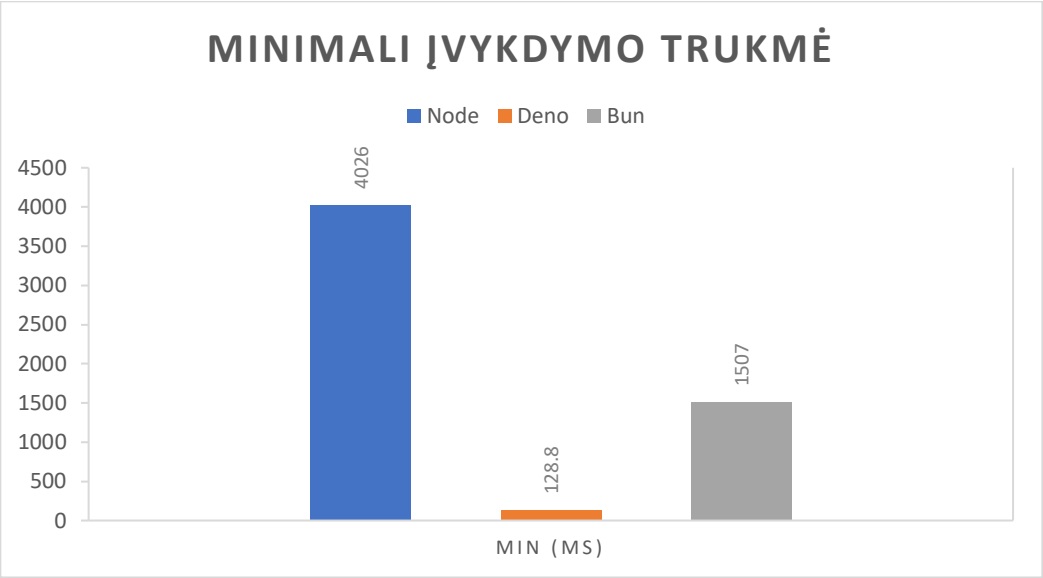


diagrama 10 SQL užklausos operacijos minimali įvykdymo trukmė

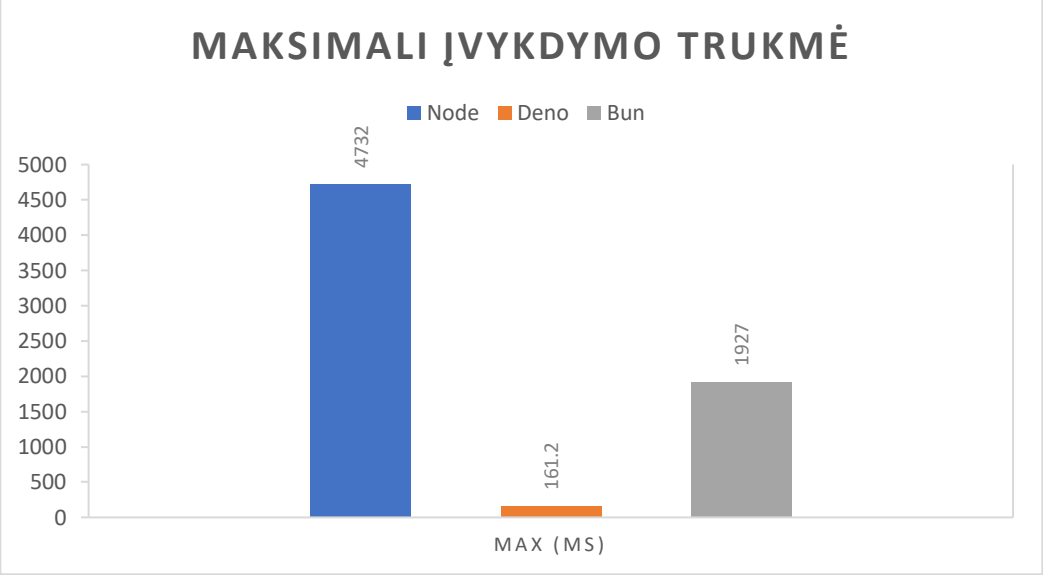


diagrama 11 SQL užklausos operacijos maksimali įvykdymo trukmė

Išmatavus minimalią ir maksimalią įvykdymo trukmes rezultatai išliko analogiški vidutinei operacijos įvykdymo trukmei, jokių didelių anomalijų nepastebėta. Šio bandymo aiškus nugalėtojas yra *Deno*. Ši technologija operaciją įvykdė 11,95 karto greičiau nei *Bun.js*, o lyginant su *Node.js* – 29,35 karto greičiau.

Nors *Node.js* sunaudojo mažiausiai atminties, bet pagal operacijos įvykdymo trukmę ši technologija veikė prasčiausiai. *Bun.js* sunaudojo daugiausia atminties, bet lyginant operacijos

įvykdymo trukmę *Deno* buvo daug kartų greitesnė technologija. Reikia nepamiršti jog vykdant šiuos bandymus bendravimui su *MySQL* serveriu buvo naudojama *NPM* biblioteka[37], tai galėjo pabloginti arba pagerinti technologijų bandymų rezultatus, todėl naudojant kitą šio tipo biblioteką galima tikėtis skirtingų rezultatų. Šiuo atveju geriausiai pasirodė *Deno* technologija.

5.4. Skaičiavimams imlios užduoties rezultatai

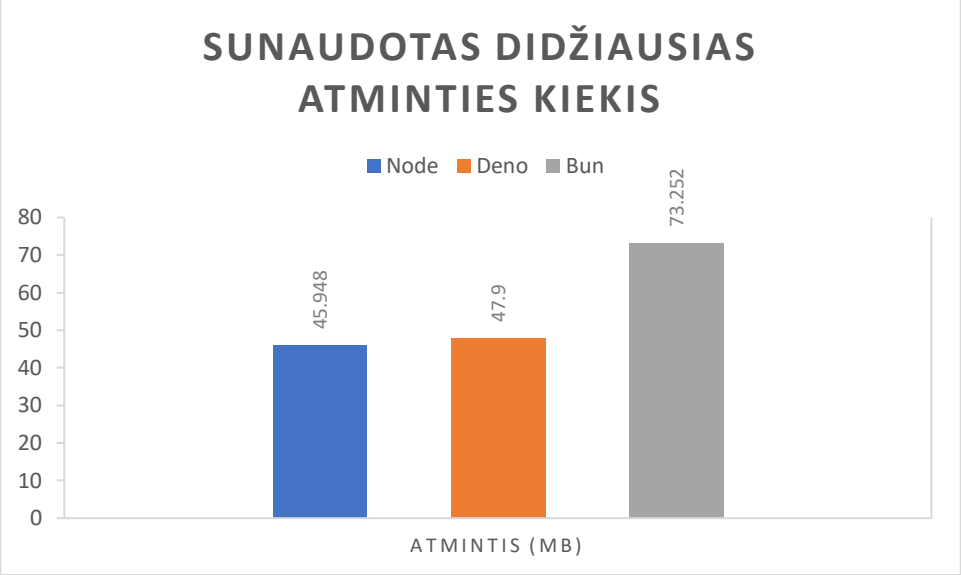


diagrama 12 Skaičiavimams imlios užduoties operacijos sunaudotas didžiausias atminties kiekis

Fibonači skaičiaus apskaičiavimo bandymo metu technologijos pasiskirstė tose pačiuose vietose kaip ir prieš tai buvusio bandymo metu pagal sunaudotą didžiausią atminties kiekį. *Node.js* sunaudojo mažiausiai atminties, antroje vietoje liko *Deno*, o paskutinę vieta užėmė *Bun.js*. Rezultatai pasiskirstė panašiai, kur *Node.js* ir *Deno* sunaudotas atminties kiekis skyrėsi nežymiai, o *Bun.js* sunaudotas atminties kiekis buvo ženkliai didesnis.

[37] Bendravimui su MySQL serveriu skirta biblioteka https://www.npmjs.com/package/mysql2

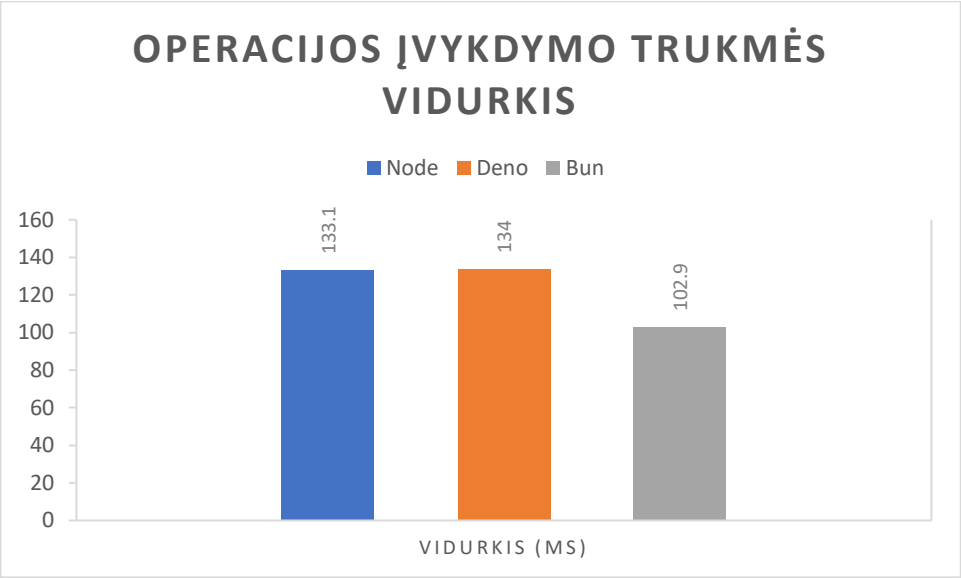


diagrama 13 Skaičiavimams imlios užduoties operacijos įvykdymo trukmės vidurkis

Šis bandymas parodė visiškai kitokius rezultatus, nes šiuo atveju *Bun.js* išmatuotas operacijos įvykdymo trukmės vidurkis yra mažiausias. *Node.js* užėmė antrąją vietą, o paskutinė vieta atiteko *Deno*, tačiau *Node.js* ir *Deno* vidurkio rezultatas tesiskyrė tik 0,9ms.

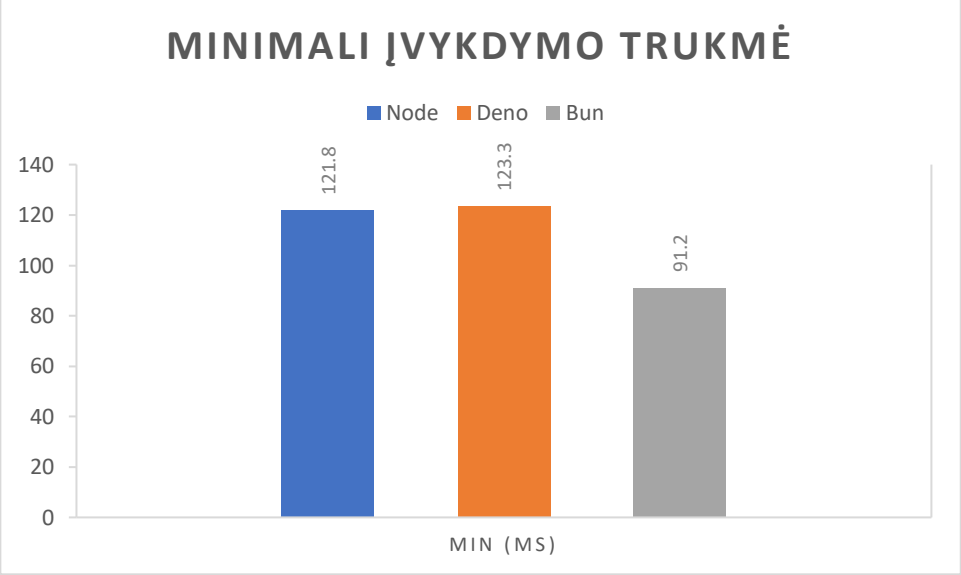


diagrama 14 Skaičiavimams imlios užduoties operacijos minimali įvykdymo trukmė

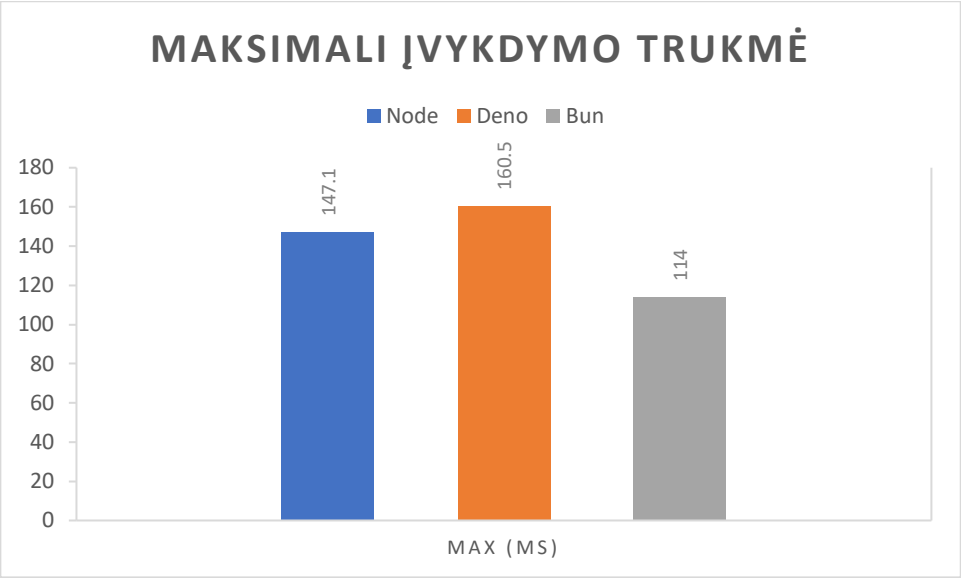


diagrama 15 Skaičiavimams imlios užduoties operacijos maksimali įvykdymo trukmė

Išmatuotos minimalios ir maksimalios operacijos įvykdymo trukmės neparodė jokių anomalijų, visos technologijos užėmė tas pačias vietas. Kitaip negu *SQL* užklausos įvykdymo bandyme, kai yra atliekami intensyvus skaičiavimai, kurie išnaudoja daug CPU resursų *Bun.js* nors ir išnaudojo daugiausia atminties, tačiau vidutinė, minimali, maksimali įvykdymo trukmė buvo trumpiausia iš visų technologijų.

5.5. Bibliotekos įdiegimo trukmės rezultatai

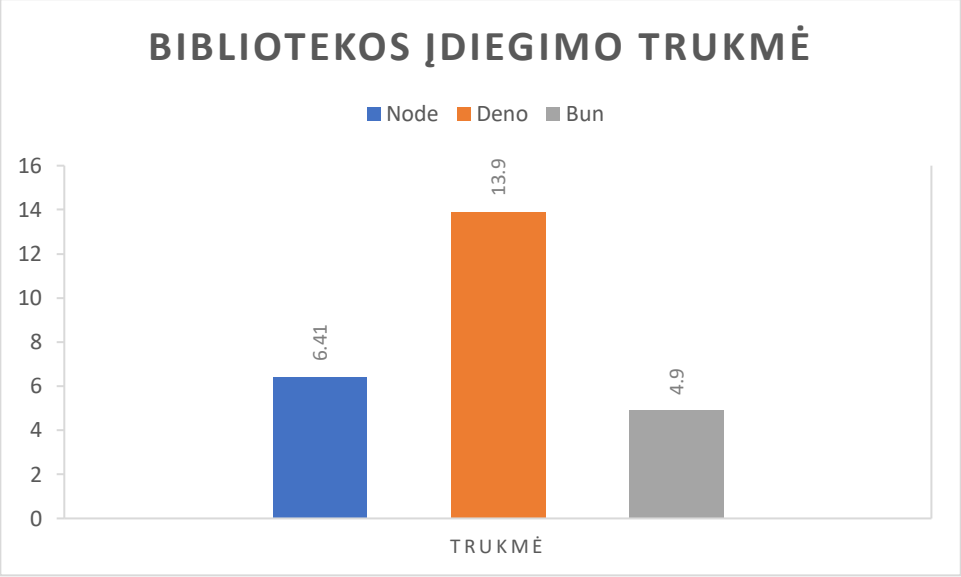


diagrama 16 Bibliotekos įdiegimo trukmė

Šis bandymas parodė jog greičiausiai biblioteką iš *NPM* bibliotekų registro įdiegia *Bun.js*, antroje vietoje liko *Node.js*, paskutinę vieta atiteko *Deno*. 25 kartus greitesnis bibliotekų įrašymas

yra minimas *Bun.js* internetiniame puslapyje, tačiau šiuo atveju lyginant su *Node.js* bibliotekų įrašymo įrankiu **npm** tik 1,30 karto greičiau.

5.6. Bandymo rezultatų išvados

Buvo atliekamas realus scenarijų testavimas, kurių metu buvo išmatuojamas scenarijaus atlikimo laikas, sunaudotas atminties kiekis bei aptariami *Node.js, Deno* ir *Bun.js* technologijų rezultatai. Šių bandymų rezultatai atskleidė technologijų galimybes šių bandymų atvejais, todėl į juos reiktų atsižvelgti renkantis *JavaScript* vykdymo aplinką.

Beveik visų bandymų metu pagal vykdymo trukmę aiškiai išsiskyrė *Bun.js* technologija. Ši technologija lenkdavo kitas technologijas tas pačias operacijas įvykdydama greičiau. Vienintelio *SQL* užklausos bandymo metu *Deno* buvo greičiau veikianti technologija. Reikia pabrėžti, jog *Bun.js* naudojo daugiau atminties už kitas technologijas *SQL* užklausos ir skaičiavimams imlios užduoties bandymų metu.

Deno daugumoje atliktų bandymų užėmė antrąją vietą. *Deno* kaip geriausia technologija buvo *SQL* užklausos bandyme įspūdingai aplenkusi kitas technologijas, bei kituose bandymuose nežymiai atsilikdami nuo kitų technologijų.

Node.js kaip ir buvo galima manyti pasirodė prasčiausiai iš visų lyginamų technologijų. Kelių bandymų metu matuojant sunaudotą didžiausią atminties kiekį ši technologija užėmė pirmąją vietą sunaudojant mažiausiai atminties, tačiau tų bandymų vykdymo trukmės matavime užėmė paskutinę vietą.

Rezultatai apskaičiavus grafikų kiekvienos technologijos užimtas pirmąsias vietas: *Node.js* – 2, *Deno* – 5, *Bun.js* - 18. Našumo lyginime akivaizdžiai pirmauja *Bun.js* technologija, antrąją vietą užima *Deno*, o paskutinėje yra *Node.js*.

Remiantis atliktais bandymų rezultatais, galima teigti, kad *Bun.js* užtikrina geresnį našumą nei *Node.js* ir *Deno*, kalbant apie serverio *HTTP* užklausų atlikimą, failo skaitymo, skaičiavimams imlios užduoties, bibliotekos įrašymo atvejais. *Bun.js* veikia geriau dėl patobulintos architektūros, efektyviausių operacinės sistemos metodų pasirinkimo, *JavaScriptCore* variklio ir Zig programavimo kalbos naudojimo. Antra pagal našumą liko *Deno* technologija. Ši technologija stipriai išsiskyrė *MySQL* serverio užklausos įvykdymo bandyme parodydama dešimtimis kartų geresnius rezultatus už kitas technologijas. Prasčiausiai pasirodė *Node.js* technologija, bet buvo atvejų kelių bandymų metu sunaudotas didžiausias atminties kiekis buvo mažiausias lyginant su kitomis technologijomis. Tokių rezultatų buvo galima tikėtis, nes tai yra pirmoji *JavaScript* vykdymo aplinka po kurios atsiradimo ir buvo išskirti metodai kaip patobulinti procesus ir architektūrą.

Reikia pabrėžti jog *Deno* ir *Node.js* naudoja tą pati Google Chrome naršyklės V8 *JavaScript* kalbos variklį ir C/C++ programavimo kalbas darbui su operacine sistema. Pagal našumo tyrimo rezultatus galime teigti, jog *Deno* technologija veikia efektyviau lyginant našumo charakteristikas remiantis atliktais bandymais nors ir naudoja tas pačias technologijas, kuriant vykdymo aplinkas, tai reiškia jog antruoju bandymu tam pačiam kūrėjui pavyko sukurti geresnę šio tipo technologiją.

5.7. Bandymų rezultatų suvestinė lentelėje

Pagal žemiau pateiktą lentelę buvo apskaičiuota, jog *Bun.js* daugiausia kartų bandymų metu parodė geriausius rezultatus, antroji vieta atiteko *Deno*, o paskutinė vieta – *Node.js*.

lentelė 3 Atliktų bandymų rezultatų suvestinė

Bandymas	Node	Deno	Bun	Geriausias rezultatas	Node (skirtumas nuo geriausio rezultato)	Deno (skirtumas nuo geriausio rezultato)	Bun (skirtumas nuo geriausio rezultato)
HTTP užklausa(10 lygiagrečių prisijungimų) maksimalus atsako laikas	24,10	24,27	20,03	20,03	-4,07	-4,24	GREIČIAUSIA
HTTP užklausa(100 lygiagrečių prisijungimų) maksimalus atsako laikas	101,25	52,38	51,85	51,85	-49,40	-0,53	GREIČIAUSIA
HTTP užklausa(500 lygiagrečių prisijungimų) maksimalus atsako laikas	396,36	1190,00	223,29	223,29	-173,07	-966,71	GREIČIAUSIA
HTTP užklausa(10 lygiagrečių prisijungimų) vidutinis atsako laikas	0,44	0,26	0,17	0,17	-0,27	-0,09	GREIČIAUSIA
HTTP užklausa(100 lygiagrečių prisijungimų) vidutinis atsako laikas	4,54	1,88	1,05	1,05	-3,49	-0,83	GREIČIAUSIA

HTTP užklausa(500 lygiagrečių prisijungimų) vidutinis atsako laikas	22,98	11,63	6,31	6,31	-16,67	-5,32	GREIČIAUSIA
HTTP užklausa(10 lygiagrečių prisijungimų) bendras atsako laikas	44,00	26,00	20,00	20,00	-24,00	-6,00	GREIČIAUSIA
HTTP užklausa(100 lygiagrečių prisijungimų) bendras atsako laikas	45,00	18,00	11,00	11,00	-34,00	-7,00	GREIČIAUSIA
HTTP užklausa(500 lygiagrečių prisijungimų) bendras atsako laikas	45,00	23,00	11,00	11,00	-34,00	-12,00	GREIČIAUSIA
HTTP užklausa(10 lygiagrečių prisijungimų) RAM sunaudojimas	65,05	62,01	66,53	62,01	-3,04	GREIČIAUSIA	-4,52
HTTP užklausa(100 lygiagrečių prisijungimų) RAM sunaudojimas	81,96	64,18	68,90	64,18	-17,78	GREIČIAUSIA	-4,72
HTTP užklausa(500 lygiagrečių prisijungimų) RAM sunaudojimas	91,74	76,15	67,44	67,44	-24,30	-8,71	GREIČIAUSIA
Tekstinio failo skaitymas minimalus atsako laikas	86,20	76,90	66,50	66,50	-19,70	-10,40	GREIČIAUSIA
Tekstinio failo skaitymas maksimalus atsako laikas	117,10	113,30	103,00	103,00	-14,10	-10,30	GREIČIAUSIA
Tekstinio failo skaitymas	98,10	90,60	77,10	77,10	-21,00	-13,50	GREIČIAUSIA

vidutinis atsako laikas							
Tekstinio failo skaitymas RAM sunaudojimas (MB)	70,87	66,00	62,86	62,86	-8,01	-3,13	GREIČIAUSIA
SQL užklausa minimalus atsako laikas	4026,00	128,80	1507,00	128,80	-3897,20	GREIČIAUSIA	-1378,20
SQL užklausa maksimalus atsako laikas	4732,00	161,20	1927,00	161,20	-4570,80	GREIČIAUSIA	-1765,80
SQL užklausa Vidutinis atsako laikas	4328,00	144,50	1675,00	144,50	-4183,50	GREIČIAUSIA	-1530,50
SQL užklausa RAM sunaudojimas (MB)	60,69	63,00	114,40	60,69	GREIČIAUSIA	-2,31	-53,71
Skaičiavimams imli užduotis minimalus atsako laikas	121,80	123,30	91,20	91,20	-30,60	-32,10	GREIČIAUSIA
Skaičiavimams imli užduotis maksimalus atsako laikas	147,10	160,50	114,00	114,00	-33,10	-46,50	GREIČIAUSIA
Skaičiavimams imli užduotis vidutinis atsako laikas	133,10	134,00	102,90	102,90	-30,20	-31,10	GREIČIAUSIA
Skaičiavimams imli užduotis RAM sunaudojimas (MB)	45,95	47,90	73,25	45,95	GREIČIAUSIA	-1,95	-27,30
NPM Bibliotekos įrašymas operacijos trukmė	6,41	13,90	4,90	4,90	-1,51	-9,00	GREIČIAUSIA

Rezultatai ir išvados

1. Literatūros apie JavaScript platformas, naudojamas serverio programavimui, analizės metu buvo pasirinktos šios vykdymo aplinkos: Node.js, Deno ir Bun.js. Taip pat ši programavimo kalba yra naudojama ne tik internetinių puslapių kūrimui, bet ir mobiliosioms programėlėms, darbalaukio ir serverio programoms. Aprašyta kuo skiriasi kai *JavaScript* vykdomas naršyklėje ir pačioje vykdymo aplinkoje.
2. Išanalizavus mokslinius straipsnius apie JavaScript vykdymo aplinkas, buvo parinkti vertinimo kriterijai: technologijos branda, Node Package Manager bibliotekų registro palaikymas ir architektūra, optimizuotas Node Package Manager modulių saugojimas, ECMAScript modulių importavimas kaip technologijos standartas, technologijoje integruoti įrankiai palengvinantys darbą su JavaScript, TypeScript palaikymas vykdymo aplinkos kompiliatoriuje, bibliotekų teisių apribojimas kaip saugumo įrankis, keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymas. Buvo atliktas kriterijų įvertinimas ir palyginimas. Kriterijų analizėje daugiausia kriterijų atitiko *Bun.js* technologija. Šios technologijos vienintelis trūkumas buvo jos branda, nes tai yra visiškai nesenai išleista technologija. Antroje vietoje pagal kriterijų įvertinimus liko *Deno* technologija. *Deno* nėra pilnai subrendusi technologija, taip pat nors ir palaiko *NPM* registro bibliotekas, bet nepalaiko projekto architektūros ir neturi keičiamo kodo vykdymo realiu laiku išsaugant būseną palaikymo. Prasčiausiai kriterijų lyginime pasirodė *Node.js*. Ši technologija kriterijų lyginime labiausiai atitiko kriterijų kaip brandžiausia technologija ir kuri pilnai palaiko *NPM* bibliotekų registrą ir jo architektūrą. Kitų kriterijų ši technologija neatitiko. Bendri kriterijai, kuriuos atitiko *Deno* ir *Bun.js*:
    - Optimizuotas modulių saugojimas
    - *ECMAScript* modulių importavimas kaip technologijos standartas
    - Technologijoje integruoti įrankiai palengvinantys darbą su *JavaScript*
    - *TypeScript* palaikymas vykdymo aplinkos kompiliatoriuje
    - Bibliotekų teisių apribojimas kaip saugumo įrankis

3. Pagal suprojektuotą eksperimento tyrimo modelį buvo atliktas eksperimentinis tyrimas, kurio metu buvo sudaryti identiški bandymo scenarijai kiekvienai technologijai, jie buvo vykdomi ir išmatuojamos jų našumo charakteristikos. Buvo atlikti 5 skirtingo tipo bandymai:
    - *HTTP* serverio užklausos atsakymo greitis

- Tekstinio failo skaitymas
- *SQL* duomenų bazės serverio užklausos įvykdymo laikas
- Skaičiavimams imli užduotis
- Bibliotekos įdiegimo trukmė

Nebuvo vienos technologijos, kuri būtų geriausia visų bandymų metu, tačiau daugiausia bandymų nugalėtoja tapo *Bun.js*. – keturiuose iš penkių bandymų ši technologija veikė našiausiai. *SQL* duomenų bazės serverio užklausos vykdymo bandyme buvo užimta antra vieta. Šio bandymo metu greičiausiai veikė *Deno*, aplenkdama *Bun.js* atlikdama šį bandymą 11,95 karto greičiau. *Deno* daugelyje bandymu užimdavo antrąją vietą operacijos įvykdymo greičio matavimuose. Lėčiausiai bandymuose pasirodė *Node.js*, nei vieno bandymo metu operacija nebuvo įvykdyta greičiausiai. Taip pat įrodyta, kad naujesnės vykdymo aplinkos veikia našiau nei pati pirmoji *Node.js*.

4. Atlikta bandymų rezultatų analizė. Remiantis bandymų rezultatais nustatyta jog, *Bun.js* pasirodė geriau nei *Deno* vidutiniškai 1,74 karto, o už *Node.js* 2,15 karto vykdant *HTTP* užklausas, skaitant tekstinį failą, atliekant skaičiavimo užduotis ar programavimo metu įdiegiant projektui reikalingas bibliotekas. Programuotojams ir sistemų architektams ši informacija gali padėti pasirenkant tinkamiausią *JavaScript* vykdymo aplinką savo sprendžiamai problemai.

Reikia nepamiršti, kad visi atlikti bandymai neapima visų įmanomų situacijų ir sudėtingų projektų architektūrų. Šie bandymo rezultatai gali skirtis, nes buvo atliekami esant konkrečioms sąlygoms ir kiti faktoriai gali turėti įtakos rezultatams, todėl iš tikrųjų norint įvertinti savo pasirinktą vykdymo aplinką reikia realiai išbandyti norimus scenarijus. Šiuo tyrimu buvo pasirinktos tam tikros *JavaScript* vykdymo aplinkos, todėl gali būti, kad yra kitų vykdymo aplinkų, kurios kai kuriais atvejais gali veikti našiau. Tyrimas gali būti išplėstas lyginant daugiau technologijų, bei atliekant įvairesnių bandymų, kurie apimtų daugiau realių situacijų.

Šio tyrimo metu buvo labiau susipažinta su *JavaScript* programavimo kalba ir jos pritaikymu, taip pat vykdymo aplinkomis. Buvo aptarti kiekvienos technologijos bandymų metu gauti rezultatai ir nustatyta, kad *Bun.js* tyrimo ir bandymų metu pasirodė našiausiai.

Rekomendacijos

Renkantis vykdymo aplinką reikia nepamiršti, kad našumas yra tik vienas iš reikalavimų. Savybės kaip saugumas, stabilumas, patikimumas yra keletas iš kokybiškos programinės įrangos savybių, kurios yra ne mažiau svarbios renkantis vykdymo aplinką.

Našiausiai bandymų metu veikė *Bun.js* technologija. Taip pat turi daug papildomų funkcijų lyginant su Node.js. Ši technologija buvo našiausia atliekant įvairias įvesties ir išvesties, skaičiavimo operacijose, bet ir naudojant pačios technologijos įrankius kaip bibliotekos pridėjimas prie projekto. Jeigu kuriamo produkto reikalavimuose yra nurodyta, kad produktas turi būti kiek įmanoma našesnis, tuomet šią technologiją būtina išbandyti norimiems sprendimams, tačiau reikia apsvarstyti jog dėl šios technologijos naujumo gali kilti problemų ateityje, tokių kaip *Bun.js* projekto nutraukimas ar nebe palaikymas.

Antroji vieta atiteko *Deno* technologijai. Nors ši technologija yra ne tokia naši kaip *Bun.js*, tačiau ji yra ilgiau naudojama ir neatsilieka savo siūlomomis funkcijomis, o kai kuriuose srityse net lenkia *Bun.js*. *Deno* technologija puikiai tinka ieškant našios ir turinčios daugelį naujausių funkcijų, kurias gali turėti *JavaScript* vykdymo aplinka technologijos. Reikia nepamiršti, jog ši technologija yra naudojama ne vienerius metus ir pastoviai yra atnaujinama, todėl tai kelia didesnį pasitikėjimą, kad *Deno* projektas nebus sustabdytas greitu metu.

Paskutinė vieta atiteko pačiai pirmajai ir populiariausiai vykdymo aplinkai *Node.js*. Jeigu vienintelis produkto reikalavimas kūrimo metu naudoti *JavaScript* programavimo kalbą, tuomet tai yra puikus pasirinkimas, nes tai yra pati pirmoji ir populiariausia *JavaScript* vykdymo aplinka, kuri yra patikrinta ir išbandyta technologija, šiek tiek atsiliekanti funkcionalumu nuo savo konkurentų, tačiau trūkstamos funkcijos yra pridedamos bėgant laikui.

Svarbu paminėti, kad atlikti bandymai neapima visų situacijų, kurios gali įvykti naudojant *JavaScript* vykdymo aplinką, todėl prieš pasirenkant būtina išbandyti sukuriant minimalios vertės norimą produktą. Visi bandymai buvo atlikti aukščiau minėtoje konkrečioje sistemoje, todėl skirtingos sistemos atveju bandymų rezultatai gali skirtis. *Node.js*, *Deno* ir *Bun.js* buvo lyginamos konkrečios šių vykdymo aplinkų versijos, todėl našumas gali kisti po šių technologijų atnaujinimų, nes laikui bėgant jos bus tobulinamos. Taip pat reikia pabrėžti, kad visi bandymai buvo gana paprasti ir atlikti kontroliuojamoje aplinkoje. Realiais atvejais vykdomos programos yra kur kas sudėtingesnės, kurios gali turėti visai kitokius našumo savybes, todėl šių bandymų rezultatai gali netiksliai atvaizduoti realų sudėtingų kodo vykdymo scenarijų rezultatą.

Visi bandymų programinis kodas ir rezultatai yra patalpinti GitHub kodo saugykloje[38].

[38] Bandymų kodo saugykla https://github.com/DziugasMolis/js_runtime_env_efficiency

Šaltinių sąrašas

1. Hannah Ritchie, Edouard Mathieu, Max Roser and Esteban Ortiz-Ospina. Internet usage in the world. [Tinkle] 2023 m. [Cituota: 2023 m. 10 18 d.] https://ourworldindata.org/internet.

2. Chhetri, Nimesh. *A Comparative Analysis of Node.js (Server-Side JavaScript).* s.l. : St. Cloud State University, 2016.

3. Kniazev I., Fitiskin A. *Choosing the right JavaScript runtime: an in-depth comparison of Node.js and Bun.* 2023.

4. *Deno – A new Node.js?* Gautam, Sampurna. 2021 m.

5. Flanagan, David. *JavaScript: The Definitive Guide, 7th Edition.* s.l. : O'Reilly Media, Inc., 2020.

6. Official documentation of ECMAScript. [Tinkle] [Cituota: 2023 m. 10 13 d.] https://tc39.es/ecma262/.

7. Rauch, Guillermo. *Smashing Node.js: JavaScript Everywhere.* 2012.

8. *Static and Dynamic Semantics of the Web.* Christopher Fry, Mike Plusch. 2003 m.

9. Brown, Ethan. *Web Development with Node and Express Leveraging the JavaScript Stack.* s.l. : O'Reilly Media, 2019.

10. Teixeira, Pedro. *Professional Node.js: Building Javascript Based Scalable Software 1st Edition.* 2012.

11. Introduction to the NPM package manager. [Tinkle] [Cituota: 2023 m. 10 13 d.] https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/.

12. Eisenman, Bonnie. *Learning React Native Building Native Mobile Apps with JavaScript.* 2017.

13. Sheiko, Dmitry. *Cross-platform Desktop Application Development: Electron, Node, NW.js, and React.* 2017.

14. Official documentation of Electron. [Tinkle] [Cituota: 2023 m. 10 13 d.] https://www.electronjs.org/docs/latest/.

15. Uzayr, Sufyan bin. *Conquering JavaScript: Node.js.* s.l. : Taylor & Francis Ltd, 2023.

16. Official documentation of Deno. [Tinkle] [Cituota: 2023 m. 10 13 d.] https://docs.deno.com/runtime/manual.

17. *10 Things I Regret About Node.js.* Dahl, Ryan. s.l. : JSConf EU.

18. Doglio, Fernando. *Introducing Deno A First Look at the Newest JavaScript Runtime.* 2020.

19. Cherny, Boris. *Programming TypeScript.* 2019.

20. Santos, Alexandre Portela dos. *Deno Web Development.* 2021.

21. Official documentation of Bun.js. [Tinkle] [Cituota: 2023 m. 10 13 d.] https://bun.sh/docs.

22. Ahmod, Md Feroj. *JavaScript runtime performance analysis: Node and Bun.* 2023.

23. *Client-Server Model.* Oluwatosin, Haroon Shakirat. s.l. : IOSR Journal of Computer Engineering, 2014 m.

24. Jacobs, Alexander. *Comparison of JavaScript package managers.* 2019.

25. ISO 25000 Standards. [Tinkle] [Cituota: 2023 m. 10 17 d.] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

26. Official documentation of Bombardier. [Tinkle] [Cituota: 2023 m. 10 18 d.] https://pkg.go.dev/github.com/codesenberg/bombardier.

SĄVOKŲ APIBRĖŽIMAI

Kalbos variklis (angl. **engine**) – programa, kurios pagrindinė užduotis yra skaityti ir vykdyti kodą.

Vykdymo aplinka (angl. **runtime**) – aplinka, kurioje yra vykdoma programavimo kalba. Tai programa, kuri praplečia kalbos variklį ir suteikia papildomo funkcionalumo. Vykdymo aplinka yra naudojama kurti programinę įrangą pasirinkta programavimo kalba.

Kliento-serverio architektūra (angl. **client-server architecture**) – tai sistema, atliekanti kliento ir serverio funkcijas, kad būtų galima dalytis informacija. Ji leidžia daugeliui naudotojų tuo pačiu metu naudotis ta pačia duomenų baze vienu metu.

Bitcoin – tai skaitmeninis žetonas, kurį galima iškeisti elektroniniu būdu. Jis neturi fizinio pavidalo. Bitkoinus kuria ir stebi ne kokia nors institucija ar organizacija, bet kompiuterių tinklas, naudojantis sudėtingas matematines formules.

Go – taip pat vadinama Golang, statiškai rašoma, atviro kodo programavimo kalba sukurta Google.

Rust – tai nepaprastai greita, taupanti atmintį statiškai kompiliuojama programavimo kalba.

CommonJS – standartas skirtas struktūrizuoti ir naudoti JavaScript programinį kodą.

Google V8 – V8 yra nemokamas atvirojo kodo JavaScript ir WebAssembly variklis, sukurtas Chromium Project, skirtas Chromium ir Google Chrome žiniatinklio naršyklėms.

Windows – yra grafinė operacinė sistema, kurią sukūrė ir išleido "Microsoft".

Linux – atviro kodo Unix-tipo operacinių sistemų, naudojančių Linux branduolį, šeima.

MacOS – operacinė sistema, skirta „Apple“ bendrovės gaminamiems kompiuteriams, vadinamiems „Macintosh“.

WebKit/Safari JavaScriptCore – yra integruotas "WebKit" "JavaScript" variklis. Šiuo metu jis įgyvendina ECMAScript standartą.

SANTRAUPOS

NPM (trumpinys nuo angl. **Node Package Manager**) – „JavaScript“ programavimo kalbos paketų tvarkyklė.

PNPM (trumpinys nuo angl. **Performant Node Package Manager**) – alternatyvi NPM paketų tvarkyklė, kuri saugo visas bibliotekas centralizuotai, tai leidžia pernaudoti tas pačias bibliotekas keliuose projektuose.

API (trumpinys nuo angl. **application programming interface**) – sąsaja, kad programuotojas per kitą programą galėtų pasiekti jos funkcionalumą ar apsikeistų su ja duomenimis.

HTML (trumpinys nuo angl. **HyperText Markup Language**) – Hiperteksto ženklinimo kalba, dažniausiai vartojama tinklalapiams rašyti.

CSS (trumpinys nuo angl. **Cascading Style Sheets**) – Pakopinių stilių aprašo kalba, skirta HTML kalba rašomų dokumentų stiliams aprašyti.

HTTP (trumpinys nuo angl**. HyperText Transfer Protocol**) – Hipertekstų persiuntimo protokolas žiniatinklio duomenims (ištekliams) persiųsti. Apibrėžia HTTP serverio ir kliento programos (dažniausiai naršyklės) sąveiką. Protokolo pavadinimu prasideda juo persiunčiamų interneto išteklių universalieji adresai, pavyzdžiui, http://pvz.lt/pavyzdžiai/.

TCP (trumpinys nuo angl. **Transmission Control Protocol**) – standartas, apibrėžiantis, kaip sukurti ir palaikyti tinklo ryšį, kuriuo naudodamosi programos gali keistis duomenimis.

DNS (trumpinys nuo angl. **Domain Name System**) – pavadinimų duomenų bazė, kurioje yra interneto domenų vardai, verčiami į interneto protokolo (IP) adresus.

JSON (trumpinys nuo angl. **JavaScript Object Notation**) – lengvas duomenų saugojimo ir perdavimo formatas.

SQL (trumpinys nuo angl. **Structured Query Language**) – standartinė kalba, skirta prieigai prie duomenų bazių ir darbui su jomis.

MySQL (trumpinys nuo angl. **My Structured Query Language**) – viena iš reliacinių duomenų bazių valdymo sistemų, palaikanti daugelį naudotojų, dirbanti SQL kalbos pagrindu.

ES (trumpinys nuo angl. **ECMAScript**) – yra scenarijų kalbų, įskaitant "JavaScript", "JScript" ir "ActionScript", standartas.

FTP (trumpinys nuo angl. **File transfer protocol**) – būdas atsisiųsti, įkelti ir perkelti failus iš vienos vietos į kitą internete ir tarp kompiuterių sistemų.

SMTP (trumpinys nuo angl. **Simple Mail Transfer Protocol**) – standartas el.pašto laiškų perdavimui internete. Naudojamas elektroniniams laiškams pristatyti į gavėjo el.pašto dėžutę.

ARM (trumpinys nuo angl. **ARM architecture family**) – ARM yra RISC instrukcijų rinkinio architektūrų šeima kompiuterių procesoriams.

PRIEDAI

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 10 -n 1000000 -l http://localho
st:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 10 connection(s)
 1000000 / 1000000 [=============================================================================================================] 100.00% 22266/s 44s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     22345.11    4368.97   35371.74
  Latency      444.88us   196.32us    24.10ms
  Latency Distribution
     50%   394.00us
     75%   528.00us
     90%   740.00us
     95%     0.90ms
     99%     1.37ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     4.62MB/s
```

pav. 22 Node.js bandymo rezultatai su 10 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 100 -n 1000000 -l http://localh
ost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 100 connection(s)
 1000000 / 1000000 [============================================================================================================] 100.00% 21973/s 45s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     22032.06    4167.16   35214.01
  Latency        4.54ms     1.01ms   101.25ms
  Latency Distribution
     50%     4.36ms
     75%     5.23ms
     90%     6.20ms
     95%     7.08ms
     99%     9.49ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     4.56MB/s
```

pav. 23 Node.js bandymo rezultatai su 100 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 500 -n 1000000 -l http://localh
ost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 500 connection(s)
 1000000 / 1000000 [============================================================================================================] 100.00% 21740/s 45s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     21746.85    3195.66   30875.15
  Latency       22.98ms     7.88ms   398.36ms
  Latency Distribution
     50%    22.39ms
     75%    24.44ms
     90%    26.62ms
     95%    28.39ms
     99%    33.95ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     4.50MB/s
```

pav. 24 Node.js bandymo rezultatai su 500 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 10 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 10 connection(s)
 1000000 / 1000000 [============================================================================================================] 100.00% 37476/s 26s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     37730.62    3668.40   50117.00
  Latency      262.14us    81.77us    24.27ms
  Latency Distribution
     50%   237.00us
     75%   323.00us
     90%   418.00us
     95%   489.00us
     99%   695.00us
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     7.63MB/s
```

pav. 25 Deno bandymo rezultatai su 10 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 100 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 100 connection(s)
 1000000 / 1000000 [============================================================================================================] 100.00% 52984/s 18s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     53100.29    8874.47  100013.58
  Latency        1.88ms   502.03us    52.38ms
  Latency Distribution
     50%     1.71ms
     75%     2.16ms
     90%     2.75ms
     95%     3.17ms
     99%     4.04ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:    10.73MB/s
```

pav. 26 Deno bandymo rezultatai su 100 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 500 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 500 connection(s)
 1000000 / 1000000 [=============================================================================================================================================] 100.00% 42906/s 23s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     43108.31    5741.94   67927.89
  Latency       11.63ms    15.87ms      1.19s
  Latency Distribution
     50%    11.33ms
     75%    12.52ms
     90%    13.58ms
     95%    14.54ms
     99%    16.02ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     8.69MB/s
```

pav. 27 Deno bandymo rezultatai su 500 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 10 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 10 connection(s)
 1000000 / 1000000 [=============================================================================================================================================] 100.00% 48296/s 20s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     48704.37   18926.31   92273.58
  Latency      202.71us   244.95us    12.42ms
  Latency Distribution
     50%   148.00us
     75%   228.00us
     90%   372.00us
     95%   491.00us
     99%     1.13ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:     8.87MB/s
```

pav. 28 Bun.js bandymo rezultatai su 10 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 100 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 100 connection(s)
 1000000 / 1000000 [=============================================================================================================================================] 100.00% 90532/s 11s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     91694.58   15163.55  124502.44
  Latency       1.09ms     0.88ms    77.80ms
  Latency Distribution
     50%     0.85ms
     75%     1.10ms
     90%     1.53ms
     95%     2.05ms
     99%     6.49ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:    16.69MB/s
```

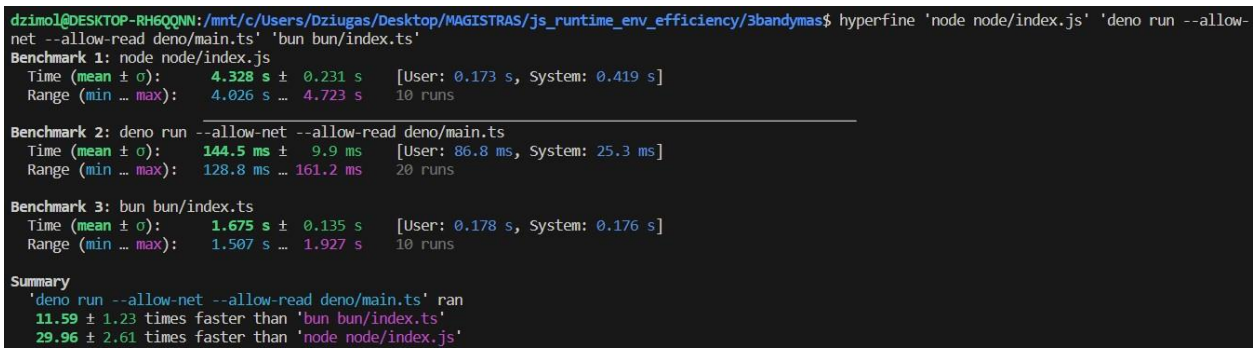
pav. 29 Bun.js bandymo rezultatai su 100 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency$ ./bombardier-linux-amd64 -c 500 -n 1000000 -l http://localhost:3000
Bombarding http://localhost:3000 with 1000000 request(s) using 500 connection(s)
 1000000 / 1000000 [=============================================================================================================================================] 100.00% 90375/s 11s
Done!
Statistics        Avg      Stdev        Max
  Reqs/sec     90802.93   11329.07  114631.87
  Latency       5.52ms     4.53ms   215.08ms
  Latency Distribution
     50%     5.11ms
     75%     5.65ms
     90%     6.55ms
     95%     7.86ms
     99%    11.28ms
  HTTP codes:
    1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:    16.50MB/s
```

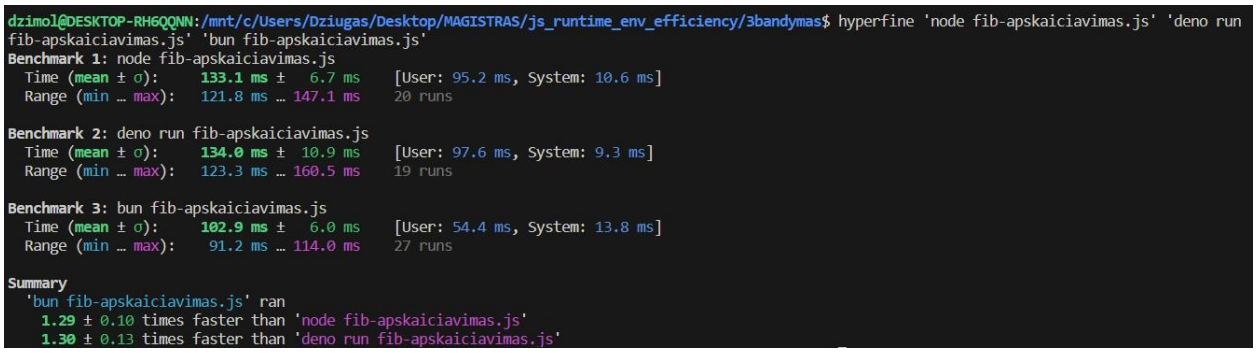
pav. 30 Bun.js bandymo rezultatai su 500 lygiagrečių prisijungimų

```
dzimol@DESKTOP-RH6QQNN:/mnt/c/Users/Dziugas/Desktop/MAGISTRAS/js_runtime_env_efficiency/2bandymas$ hyperfine 'node node-read.js' 'deno run --allow-r
ead deno-read.js' 'bun bun-read.js'
Benchmark 1: node node-read.js
  Time (mean ± σ):      98.1 ms ±   8.2 ms    [User: 34.3 ms, System: 17.8 ms]
  Range (min … max):    86.2 ms … 117.1 ms    30 runs

Benchmark 2: deno run --allow-read deno-read.js
  Time (mean ± σ):      90.6 ms ±   9.0 ms    [User: 29.8 ms, System: 18.5 ms]
  Range (min … max):    76.9 ms … 113.3 ms    33 runs

Benchmark 3: bun bun-read.js
  Time (mean ± σ):      77.1 ms ±  10.8 ms    [User: 10.1 ms, System: 17.0 ms]
  Range (min … max):    66.5 ms … 103.0 ms    34 runs

Summary
  'bun bun-read.js' ran
    1.17 ± 0.20 times faster than 'deno run --allow-read deno-read.js'
    1.27 ± 0.21 times faster than 'node node-read.js'
```

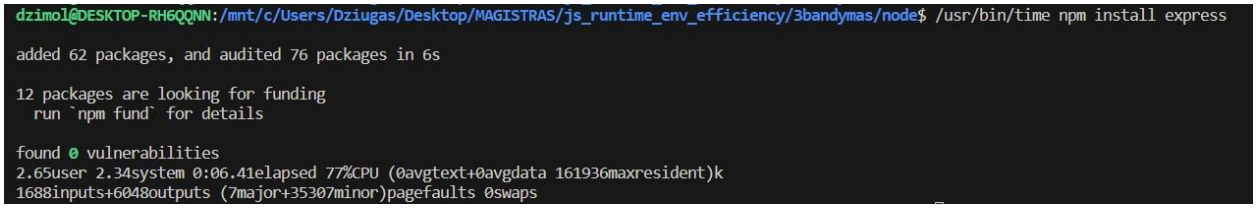
pav. 31 Tekstinio failo skaitymo bandymo rezultatai



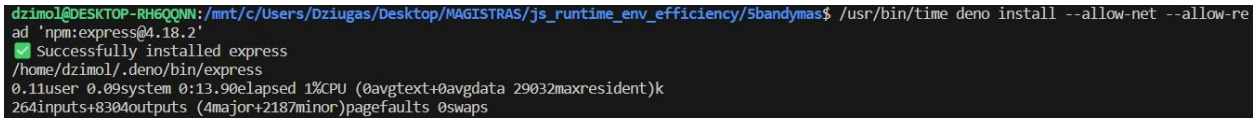
pav. 32 SQL duomenų bazės serverio užklausos įvykdymo bandymo rezultatai



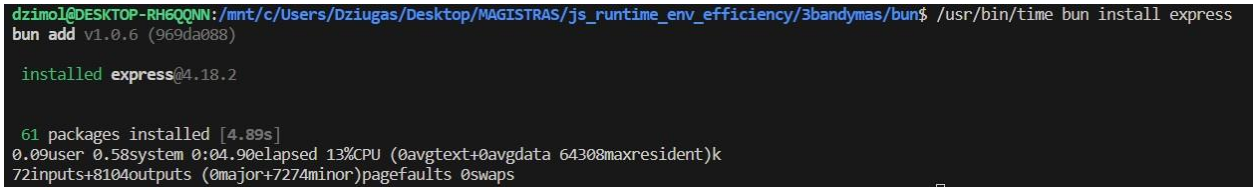
pav. 33 Skaičiavimams imlios užduoties bandymo rezultatai



pav. 34 NPM bibliotekos įdiegimo Node.js vykdymo aplinkoje bandymo rezultatai



pav. 35 NPM bibliotekos įdiegimo Deno vykdymo aplinkoje bandymo rezultatai



pav. 36 NPM bibliotekos įdiegimo Bun.js vykdymo aplinkoje bandymo rezultatai