МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ
МЭДЭЭЛЛИЙН ТЕХНОЛОГИ, ЭЛЕКТРОНИКИЙН СУРГУУЛЬ
МЭДЭЭЛЭЛ, КОМПЬЮТЕРИЙН УХААНЫ ТЭНХИМ

Булганы Раднаабазар

Агент суурилсан хиймэл оюун

(AI agents for microservices)

Мэдээллийн технологи (D061304)
Дипломын ажлын тайлан

Улаанбаатар

2025 оны 10 сар

МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ
МЭДЭЭЛЛИЙН ТЕХНОЛОГИ, ЭЛЕКТРОНИКИЙН СУРГУУЛЬ
МЭДЭЭЛЭЛ, КОМПЬЮТЕРИЙН УХААНЫ ТЭНХИМ

Агент суурилсан хиймэл оюун
(AI agents for microservices)

Мэдээллийн технологи (D061304)
Дипломын ажлын тайлан

Удирдагч: _____ Дэд профессор Б.Сувдаа

Хамтран удирдагч: _____

Гүйцэтгэсэн: _____ Б.Раднаабазар (22B1NUM0286)

Улаанбаатар

2025 оны 10 сар

Зохиогчийн баталгаа

Миний бие Булганы Раднаабазар ”Агент суурилсан хиймэл оюун” сэдэвтэй судалгааны ажлыг гүйцэтгэсэн болохыг зарлаж дараах зүйлсийг баталж байна:

- Ажил нь бүхэлдээ эсвэл ихэнхдээ Монгол Улсын Их Сургуулийн зэрэг горилохоор дэвшүүлсэн болно.
- Энэ ажлын аль нэг хэсгийг эсвэл бүхлээр нь ямар нэг их, дээд сургуулийн зэрэг горилохоор оруулж байгаагүй.
- Бусдын хийсэн ажлаас хуулбарлаагүй, ашигласан бол ишлэл, зүүлт хийсэн.
- Ажлыг би өөрөө (хамтарч) хийсэн ба миний хийсэн ажил, үзүүлсэн дэмжлэгийг дипломын ажилд тодорхой тусгасан.
- Ажилд тусалсан бүх эх сурвалжид талархаж байна.

Гарын үсэг: _____

Огноо: _____

ГАРЧИГ

ЗУРГИЙН ЖАГСААЛТ

ХҮСНЭГТИЙН ЖАГСААЛТ

Кодын жагсаалт

УДИРТГАЛ

Энэхүү үйлдвэрийн дадлагын тайлан нь БиДиСЕК ҮЦК компанид Мерчанитийн супер аппликейшнд Mini-App хөгжүүлсэн туршлагыг тайлбарлана. Гол зорилго нь мерчант дээр суурилсан мини аппликейшнээр дамжуулан хэрэглэгчийн бүртгэл, шимтгэл төлөлт, бодит цагийн үнэт цаасны ханш харах, хоёрдогч зах зээлийн арилжаанд оролцох боломжийг бүрдүүлэх явдал байв.

Техникийн хувьд Next.js, React, Node.js, Express, TypeScript, MySQL ашиглаж, SOLID зарчимд суурилсан архитектур бүтээв. Merchant Super App OAuth/refresh token механизм, ҮЦТХТ SOAP API-г амжилттай интеграцчилсан. 300,000 мөр өгөгдөлд binary search ашиглан хурдан хайлт (5–40ms), WebSocket болон cron job ашиглан real-time дата дамжуулах систем хөгжүүлэв.

Бэкенд архитектурыг SOLID зарчмын дагуу зохион байгуулж, Merchant Super App OAuth, ҮЦТХТ retry логик, securities синхронизаци, socket сервер, алдаа мэдэгдэгч сервис зэрэг гол бүрэлдэхүүнүүдийг хэрэгжүүлсэн. Docker ашиглан CI/CD pipeline-тэй VPS дээр суурилуулсан.

1. УДИРТГАЛ

Сүүлийн жилүүдэд хиймэл оюун ухааны салбар дахь технологийн хурдацтай хөгжил нь програм хангамж хөгжүүлэлтийн арга барилд үндсэндээ өөрчлөлт авчирсан. Тухайлбал, том суурь загваруудын (foundation models) гарч ирэх нь аппликейшн хөгжүүлэлтийн өмнө тулгардаг саад бэрхшээлийг эрс багасгасан бөгөөд энэ нь ХОУ-н инженерчлэл (AI engineering) гэсэн шинэ салбарыг бий болгоход хүргэжээ. Goldman Sachs-ийн судалгаагаар 2025 он гэхэд АНУ-д ХОУ-н хөрөнгө оруулалт 100 тэрбум ам.доллар, дэлхий даяар 200 тэрбум ам.долларт хүрч болзошгүй юм [2].

ХОУ-н инженерчлэл гэдэг нь урьд өмнө бэлэн байгаа суурь загваруудын дээр аппликейшн бүтээх үйл явц юм. Энэхүү чиг хандлагын хамгийн чухал ач холбогдол нь ХОУ-н аппликейшнуудын эрэлт нэмэгдэхийн зэрэгцээ, тэдгээрийг бүтээх саад бэрхшээл багассан явдал юм. Өмнө нь машин сургалт (machine learning) загвар бэлтгэхэд өндөр мэргэжлийн ур чадвар болон асар их өгөгдлийн иж бүрдэл шаардлагатай байсан бол одоо та бэлэн загваруудыг ашиглан аппликейшн хөгжүүлж чадна.

Энэхүү хөгжил нь микросервис архитектур дээр тулгуурласан програм хангамжийн хөгжүүлэлтэд онцгой боломжуудыг нээж өгч байна. Микросервис архитектур нь том нэг системийг жижиг, бие даасан үйлчилгээнүүдэд хуваах замаар уян хатан, өргөжүүлэх боломжтой, найдвартай системүүд бий болгодог. Гэвч эдгээр микросервис хоорондын харилцаа холбоо, өгөгдлийн урсгалыг оновчтой удирдах, хэрэглэгчийн хүсэлтийг олон үйлчилгээнүүдийн хамтын ажиллагаагаар шийдэх нь нарийн төвөгтэй асуудал байсаар ирсэн.

ХОУ агентууд (AI agents) нь энэхүү асуудалд шинэлэг шийдэл санал болгож байна. Агент гэдэг нь өөрийн орчныг мэдрэх, түүн дээр үйлдэл хийх чадвартай систем юм [1]. ХОУ агентууд нь том хэлний загваруудын (Large Language Models) хүчийг ашиглан даалгавруудыг ойлгож, төлөвлөгөө гаргаж, олон алхам бүхий үйл ажиллагааг гүйцэтгэх чадвартай. Эдгээр агентуудыг микросервис архитектурт нэвтрүүлэх нь системийн ухаалаг

орчуулагч, өгөгдөл боловсруулагч, ажлын урсгалын удирдагч зэрэг үүргийг гүйцэтгэх боломжийг олгоно.

Энэхүү судалгааны ажил нь ХОУ агентууд болон микросервис архитектурын уялдааг судалж, практик шийдлүүдийг санал болгохыг зорилго болгож байна. Ялангуяа, суурь загваруудын онол, агентуудын төлөвлөлт болон үйлдэл хийх механизм, мэдлэг нэмэгдүүлэх арга (Retrieval-Augmented Generation), болон эдгээрийг микросервис архитектурт хэрхэн нэгтгэх талаар авч үзэх юм.

1.1 Судалгааны зорилго

Энэхүү судалгааны ажлын гол зорилго нь дараах асуултуудад хариулах явдал юм:

1. ХОУ-н инженерчлэлийн үндсэн онол болон суурь загваруудын хөгжлийн түүхийг тайлбарлах
2. ХОУ агентуудын архитектур, төлөвлөлтийн механизм, багажуудын ашиглалтыг судлах
3. Микросервис архитектурын давуу болон сул талуудыг тодорхойлох
4. ХОУ агентуудыг микросервис архитектурт нэвтрүүлэхэд тулгарах асуудлуудыг тодорхойлох
5. ХОУ агентуудыг микросервис архитектурт нэгтгэсэн загвар санал болгох
6. Санал болгож буй загварыг практик жишээгээр батлан харуулах

2. ОНОЛЫН ХЭСЭГ

2.1 Хиймэл оюун ухааны хөгжлийн түүх

2.1.1 ХОУ-н гурван давалгаа

Хиймэл оюун ухаан нь өөрийн хөгжлийн явцад гурван чухал давалгааг туулсан. Эдгээр давалгаа бүр нь өмнөх давалгааны хязгаарлалтыг даван туулах замаар бий болсон.

Нэгдүгээр давалгаа: Таамаглах загварууд

Анхны давалгаа нь уламжлалт машин сургалтад тулгуурласан бөгөөд нарийн тодорхойлсон даалгавруудад таамаглал хийх чадварт чиглэгдсэн. Эдгээр загварууд нь тодорхой domain-д зориулагдсан бөгөөд тухайн domain-ийн мэдлэг нь сургалтын өгөгдөлд агуулагдсан байдаг.

Гэвч эдгээр загварууд хатуу байсан. Шинэ domain-д дасан зохицох нь төвөгтэй, ихэвчлэн эхнээс нь эхлэх шаардлагатай болдог. Энэ нь өргөжүүлэх боломжгүй, хөгжлийг удаашруулах шалтгаан болсон.

Хоёрдугаар давалгаа: Үүсгэх загварууд

Генератив ХОУ (Generative AI) нь гүн сургалтаар дэмжигдэн чухал эргэлт болсон. Нэг domain-д хязгаарлагдахын оронд эдгээр загварууд өргөн хүрээний, олон янзын өгөгдлийн иж бүрдэл дээр сургагдаж, янз бүрийн контекстэд ерөнхийлөх чадвартай болсон. Тэд текст, зураг, бүү□□эс видео үүсгэж чаддаг болсон.

Гэвч энэ давалгаа нь өөрийн гэсэн сорилтуудтай ирсэн:

- **Цаг хугацаанд бэхлэгдсэн**: Загварууд нь шинэ, динамик мэдээллийг өөртөө нэгтгэж чадахгүй
- **Дасан зохицох хүндрэл**: Finetuning хийх боломжтой боловч үнэтэй, алдаа гаргах магадлал өндөр

- **Domain-ийн мэдлэгийн дутагдал**: Олон нийтийн өгөгдөл дээр сургагдсан тул тодорхой domain-ийн мэдлэгт хүрч чадахгүй

Гуравдугаар давалгаа: Агентын ХОУ

Salesforce-ийн CEO Marc Benioff сая хэлэхдээ, бид LLM-ийн хийж чадах зүйлийн дээд хязгаарт хүрчихсэн байна гэжээ. Google-ийн Gemini загвар нь том хэмжээний өгөгдөл дээр сургагдсан хэдий ч дотооддоо хүлээлтээ хангаж чадахгүй байна гэж мэдээлэгдсэн. OpenAI-ийн Orion загварт ч төст асуудал байна.

Ирээдүй нь бие даасан агентууд - бодож, дасан зохицож, бие даан үйлдэл хийх чадвартай системүүдэд оршино. Агентууд нь шинэ зүйл авчирч байна: динамик, контекст дээр суурилсан workflow. Тогтмол зам биш, агентын системүүд дараагийн алхмуудыг тухайн нөхцөл байдалд үндэслэн шийддэг.

2.1.2 ХОУ инженерчлэл гэж юу вэ

ХОУ-н инженерчлэл гэдэг нь урьд өмнө бэлтгэгдсэн суурь загваруудын дээр аппликейшн хөгжүүлэх үйл явц юм. Энэ нь уламжлалт машин сургалтын инженерчлэл (ML engineering) эсвэл MLOps-оос ялгаатай байдаг. Хэрэв уламжлалт MS инженерчлэл нь загвар хөгжүүлэхэд чиглэсэн бол, ХОУ инженерчлэл нь одоо байгаа загваруудыг ашиглахад илүү анхаарал хандуулдаг [1].

Хүчирхэг суурь загваруудын олдоц болон хүртээмж нь дараах гурван хүчин зүйлийг бүрдүүлж, ХОУ инженерчлэл хурдан өсч буй салбар болоход хүргэсэн:

1. **Өндөр эрэлт**: Компаниуд ХОУ-г өрсөлдөх давуу тал болгон үзэж байна. FactSet-ийн судалгаагаар 2023 оны хоёрдугаар улиралд S&P 500 компаниудын гуравны нэг нь өөрсдийн орлогын тайланд ХОУ-г дурдсан нь өмнөх оноос гурав дахин их юм.
2. **Бага саад бэрхшээл**: Өмнө нь ХОУ систем бүтээхэд өндөр мэргэжлийн ур чадвар, их хэмжээний өгөгдөл, тооцооллын нөөц шаардлагатай байсан бол одоо API дуудлага хийх замаар чадварлаг загварууд ашиглах боломжтой болсон.

3. **Том боломж**: ХОУ технологи нь үйлдвэрлэлийн өсөлт, автоматжуулалт, шинэ бүтээгдэхүүнүүдийг бий болгох асар том эдийн засгийн боломжуудыг санал болгож байна.

ХОУ-н инженерчлэлийн багажуудын популяр болох хурд нь өмнө нь байгаагүй түвшинд хүрсэн. Хоёрхан жилийн дотор дөрвөн нээлттэй эх код бүхий ХОУ инженерчлэлийн багаж (AutoGPT, Stable Diffusion WebUI, LangChain, Ollama) нь GitHub дээр Bitcoin-оос ч илүү од цуглуулжээ.

2.1.3 Суурь загваруудын хөгжил

Хэл загваруудаас том хэлний загвар, суурь загвар руу хөгжих үйл явц нь хэдэн арван жилийн технологийн дэвшлийн үр дүн юм. Энэхүү хэсэгт гол түлхүүр цэгүүдийг авч үзэх болно.

Хэл загваруудын үндэс

Хэл загвар (language model) гэдэг нь нэг буюу хэд хэдэн хэлний тухай статистик мэдээллийг кодлодог загвар юм. Энэхүү мэдээлэл нь өгөгдсөн контекстэд тодорхой үг гарах магадлалыг илэрхийлдэг. Жишээлбэл, ”Миний дуртай өнгө бол __” гэсэн контекст өгөхөд Монгол хэлийг кодолсон хэл загвар нь ”машин”-аас илүүтэйгээр ”цэнхэр” гэсэн үгийг таамаглах ёстой.

Анхны текстийг токен болгон хуваах үйл явцыг токенжуулалт (tokenization) гэнэ. GPT-4-ийн хувьд дунджаар нэг токен нь үгийн ойролцоогоор 75%-ийн уртад тохирно. Тиймээс 100 токен нь ойролцоогоор 75 үг юм.

Хэл загваруудын хоёр үндсэн төрөл байдаг:

- **Masked language models (Далдлагдсан хэл загварууд)**: Өгүүлбэр доторх далдлагдсан үгсийг таамаглах замаар сурдаг. BERT нь энэ төрлийн алдартай жишээ юм.

- **Autoregressive language models (Авторегрессив хэл загварууд)**: Өмнөх токенуудад үндэслэн дараагийн токенийг таамаглах замаар сурдаг. GPT гэр бүл нь энэ төрлийн загвар юм.

Өөрийгөө удирдсан сургалт

Хэл загваруудын хамгийн чухал давуу тал нь өөрийгөө удирдсан сургалт (self-supervision) ашиглах чадвар юм. Өөрийгөө удирдсан сургалт нь удирдлагатай сургалт (supervised learning)-аас ялгаатай байдаг. Удирдлагатай сургалт нь тэмдэглэгдсэн өгөгдөл (labeled data) шаарддаг бөгөөд энэ нь үнэтэй, цаг хугацаа их зарцуулдаг.

Өөрийгөө удирдсан сургалтад шошго нь оролтын өгөгдлөөс дүгнэгддэг. Өөрөөр хэлбэл, хэл загваруудыг номууд, блог нийтлэл, өгүүллүүд, Reddit-ийн сэтгэгдэл зэрэг текст дараалал ашиглан ямар нэг шошгогүйгээр сургаж болно. Энэ нь асар их сургалтын өгөгдөл бүрдүүлэх боломжийг олгож, хэл загваруудыг Том Хэлний Загвар (LLM) болтол өргөжүүлэх боломжтой болгосон.

Том хэлний загвараас суурь загвар руу

2017 онд Transformer архитектур гарч ирснээр хэл загваруудын чадавхи үсрэнгүй нэмэгдсэн. Attention механизм нь загваруудад урт хугацааны хамаарлыг илүү сайн ойлгох боломжийг олгосон. Энэ нь GPT, BERT, T5 зэрэг алдартай загваруудын үндэс болсон.

Том хэлний загварууд (Large Language Models, LLMs) нь хэл загваруудын томорсон хувилбар бөгөөд тэрбум тооны параметр агуулдаг. Параметр гэдэг нь сургалтын явцад загварын сурч авдаг утга юм. Жишээлбэл, GPT-3 нь 175 тэрбум параметртэй, харин GPT-4 нь үүнээс ч илүү параметртэй гэж үздэг.

Суурь загваруд (foundation models) нь LLM-ээс цааш өргөжсөн ойлголт юм. Эдгээр нь зөвхөн текст биш, зураг, аудио, видео зэрэг олон төрлийн өгөгдөл боловсруулж чаддаг том мульти модаль загварууд юм. Суурь загваруудын гол онцлог нь даалгаврын тодорхой (task-specific) загвараас ерөнхий зориулалтын (general-purpose) загвар руу шилжсэн явдал юм.

2.1.4 Суурь загваруудын сургалт

Суурь загваруудыг бэлтгэх нь хоёр үндсэн үе шаттай:

Урьдчилан сургалт (Pre-training)

Урьдчилан сургалт нь өөрийгөө удирдсан сургалт ашиглан их хэмжээний өгөгдөл дээр загварыг сургах үйл явц юм. Энэ үе шатанд загвар нь хэл, ертөнцийн мэдлэг, энгийн дүрэм зүйлсийг суралцдаг. Гэвч энэхүү үе шатны загвар нь хэрэглэгчдийн хүсэлтэд нийцсэн хариулт өгөхөд сайн биш байдаг, учир нь зөвхөн өгүүлбэр үргэлжлүүлэх (completion) дээр сургагдсан байдаг.

Дараах сургалт (Post-training)

Урьдчилан сургасан загварыг хэрэглэгчдийн хүсэлтэд тохируулахын тулд дараах сургалт хийдэг. Энэ нь хоёр үе шаттай:

1. **Удирдлагатай нарийвчлал (Supervised Finetuning, SFT)**: Өндөр чанартай зааварчилгаа өгөгдөл (instruction data) дээр загварыг нарийвчлан сургаж, үргэлжлүүлэх биш харин ярианы горимд оновчтой болгоно.

2. **Хүний сонголтод тохируулах (Preference Finetuning)**: Загварыг хүний сонголттой нийцсэн хариулт өгөхийн тулд цаашид нарийвчлан сургана. Үүнд RLHF (Reinforcement Learning from Human Feedback), DPO (Direct Preference Optimization), RLAIF (Reinforcement Learning from AI Feedback) зэрэг аргууд ордог.

2.1.5 Загваруудын үр дүнг хэмжих

Суурь загваруудыг үнэлэх нь эрсдэлийг бууруулах, боломжуудыг илрүүлэх талаас чухал ач холбогдолтой. Үнэлгээ нь загвар сонгох, дэвшлийг хэмжих, аппликейшн ашиглалтад бэлэн эсэхийг тодорхойлох, үйлдвэрлэлд асуудал болон боломжуудыг илрүүлэх зэрэгт шаардлагатай.

Загваруудын чадавхи сайжирч байгаа нь тодорхой харагдаж байна. Жишээлбэл, 2024 оны Llama 3-8B загвар нь 2023 оны Llama 2-70B загвараас ч илүү сайн үр дүнг MMLU

benchmark дээр харуулжээ. Энэ нь зөвхөн загварын хэмжээ биш, сургалтын аргууд болон өгөгдлийн чанар хамгийн чухал болохыг харуулж байна.

Үнэлгээний гол асуудлууд:

- **Нээлттэй төгсгөлтэй гаралт**: Суурь загварууд нь нээлттэй төгсгөлтэй хариулт үүсгэдэг тул үнэлэхэд хэцүү.
- **Тогтворгүй байдал**: Загвар нь ижил эсвэл бага зэрэг өөр prompt-д маш өөр хариулт өгч болно.
- **Hallucination (Төөрөгдөл)**: Загвар нь баримт дээр үндэслээгүй хариулт өгч болно.

2.2 Prompt инженерчлэл

Prompt инженерчлэл гэдэг нь загвараас хүссэн үр дүнг гаргуулахын тулд зааврыг бичих үйл явц юм. Энэ нь загварын жинг өөрчлөхгүйгээр түүний зан үйлийг удирдах хамгийн хялбар бөгөөд түгээмэл загвар дасан зохицох арга юм.

2.2.1 Prompt бичих шилдэг арга барил

OpenAI-ийн санал болгосон дараах стратегиудыг дагах нь илүү сайн үр дүн өгдөг:

1. **Тодорхой зааварчилгаа өгөх**: Юу хийлгэхээ хоёрдмол утгагүй байдлаар тайлбарлах хэрэгтэй.
2. **Persona өгөх**: Загвараас тодорхой дүрд тоглуулж болно. Жишээ нь: ”Та туршлагатай программист. Кодыг шалгаад алдаа зааж өг.”
3. **Жишээ өгөх**: Жишээ нь хариултын формат, стилийн талаар хоёрдмол утгыг багасгадаг. Энэ нь few-shot learning гэгддэг.
4. **Хангалттай контекст өгөх**: Контекст нь hallucination-ийг багасгадаг. Хэрэв загвар шаардлагатай мэдээллээр хангагдаагүй бол өөрийн дотоод мэдлэгтээ найдах бөгөөд энэ нь найдваргүй байж болно.

5. **Нарийн төвөгтэй даалгаврууды хялбар дэд даалгаврууд хувааж өгөх**: Энэ нь chain-of-thought prompting гэгддэг.

2.2.2 Sampling стратегиуд

Загвар нь гаралтаа sampling гэж нэрлэгддэг процессоор бүтээдэг. Sampling нь ХОУ-н гаралтыг магадлалтай (probabilistic) болгодог. Дараах параметрууд нь sampling-д нөлөөлдөг:

- **Temperature**: Температур өндөр байх тусам загвар илүү бүтээлч, гэнэтийн хариулт өгдөг. Температур бага байх тусам илүү таамагладаг, баттай хариулт өгнө.
- **Top-k**: Хамгийн магадлалтай k ширхэг токеноос сонгодог.
- **Top-p (nucleus sampling)**: Нийлбэр магадлал нь p-д хүрэх хамгийн бага токенуудын багцаас сонгодог.

2.2.3 Хамгаалалтын prompt инженерчлэл

Аппликейшн олон нийтэд ашиглагдах болмогц гурван төрлийн довтолгооноос хамгаалах шаардлагатай:

- **Prompt extraction**: Аппликейшнийг хуулбарлах эсвэл хэрэглэх зорилгоор system prompt-ыг задруулах оролдлого.
- **Jailbreaking болон prompt injection**: Загварыг зөвшөөрөөгүй үйлдэл хийлгэх оролдлого.
- **Мэдээлэл задруулах**: Загварын сургалтын өгөгдөл эсвэл контекстын мэдээллийг задруулах оролдлого.

2.3 Retrieval-Augmented Generation (RAG)

RAG буюу Хайлтаар Дэмжигдсэн Генераци нь загваруудын мэдлэгийг гадаад эх сурвалжаар өргөтгөх арга юм. Энэ нь загварын дотоод мэдлэг хангалтгүй, хуучирсан эсвэл алдаатай байх асуудлыг шийддэг.

2.3.1 RAG хэрэгтэй болох шалтгаан

Хэдийгээр загваруудын контекстын урт улам нэмэгдэж байгаа ч RAG-ийн ач холбогдол алдагдахгүй байна:

1. Контекстын урт хэдий өсвөр байсан ч зарим аппликейшнд хангалтгүй байх болно. Өгөгдлийн хэмжээ байнга өсч байдаг.

2. Урт контекстыг боловсруулж чаддаг гэдэг нь тэр контекстыг сайн ашигладаг гэсэн үг биш. Контекст урт байх тусам загвар буруу хэсэгт анхаарал хандуулах магадлал өсдөг.

3. Контекстын токен бүр нэмэлт өртөг, нэмэлт хоцрогдол авчирдаг. RAG нь асуулт бүрт зөвхөн хамгийн холбогдолтой мэдээллийг ашиглах боломжийг олгоно.

Anthropic-ийн зөвлөмжөөр хэрэв таны мэдлэгийн сан 200,000 токеноос бага (ойролцоогоор 500 хуудас) бол RAG-ын оронд бүх мэдлэгийг prompt-д оруулж болно гэжээ.

2.3.2 RAG системийн бүтэц

RAG систем нь хоёр гол бүрэлдэхүүнтэй:

1. **Retriever (Хайгч)**: Асуултад хамгийн холбогдолтой баримтуудыг олж авдаг.

2. **Generator (Үүсгэгч)**: Олж авсан баримтууд болон асуултыг ашиглан хариулт үүсгэдэг.

2.3.3 Retrieval алгоритмууд

Нэр томьёо суурилсан хайлт (Term-based retrieval)

Энэ арга нь түлхүүр үгээр баримт хайдаг. TF-IDF (Term Frequency - Inverse Document Frequency) нь энэ аргын үндэс юм:

- **TF (Term Frequency)**: Нэр томьёо баримт доор хэдэн удаа гарч байгааг хэмждэг.

- **IDF (Inverse Document Frequency)**: Нэр томьёо хэдэн баримтад гарч байгааг үндэслэн түүний чухлыг хэмждэг.

Түгээмэл шийдлүүд: Elasticsearch, BM25. Эдгээр нь inverted index ашигладаг.

Embedding суурилсан хайлт (Embedding-based retrieval)

Semantic хайлт гэж нэрлэгддэг энэ арга нь утгын түвшинд холбоотой байдлыг тооцдог. Баримт бүр vector embedding болгон хувиргагдаж, vector database-д хадгалагдана. Асуулт ирэх үед түүний embedding-тэй хамгийн ойр векторуудыг хайдаг.

Vector search алгоритмууд:

- **k-NN (k-Nearest Neighbors)**: Энгийн арга боловч өгөгдөл их бол удаан.
- **ANN (Approximate Nearest Neighbors)**: Хурдан боловч ойролцоогоор хайна.
- **LSH (Locality-Sensitive Hashing)**: Ижил төстэй векторуудыг нэг bucket-д hash хийнэ.
- **HNSW (Hierarchical Navigable Small World)**: Олон давхаргын граф ашиглана.
- **IVF (Inverted File Index)**: K-means clustering ашиглан векторуудыг бүлэглэнэ.

Алдартай vector database-үүд: FAISS, Milvus, Pinecone, Weaviate, Qdrant.

2.3.4 RAG-ын үнэлгээ

RAG системийг үнэлэхэд дараах метрикүүд ашигладаг:

- **Context Precision**: Олж авсан баримтуудын хэдэн хувь нь асуулттай холбоотой вэ?
- **Context Recall**: Асуулттай холбоотой бүх баримтуудын хэдэн хувийг олж авсан бэ?
- **Answer Quality**: Эцсийн хариултын чанар хэр сайн вэ?

2.3.5 RAG-ыг сайжруулах аргууд

1. **Chunking стратеги**: Баримтуудыг хэрхэн хэсэглэх нь чухал. Тогтмол урттай хэсэглэх, өгүүлбэр/догол мөрөөр хэсэглэх, semantic хэсэглэх зэрэг аргууд байдаг.
2. **Reranking**: Анхны хайлтын үр дүнг дахин эрэмбэлэн илүү нарийвчлалтай болгох.
3. **Query rewriting**: Асуултыг дахин найруулж илүү сайн хайлт хийх.
4. **Hybrid хайлт**: Нэр томьёо болон embedding суурилсан хайлтыг хослуулах.
5. **Contextual retrieval**: Хэсэг бүрийг metadata, түлхүүр үг, холбогдох асуултуудаар баяжуулах.

2.3.6 Агентын RAG (Agentic RAG)

Уламжлалт RAG нь тогтмол workflow ашигладаг - асуулт ирэх бүрд ижил үйл явц дагана: хайлт хийх, топ-k баримт олох, контекст үүсгэх, хариулт үүсгэх. Гэвч энэ арга нь хатуу, нарийн төвөгтэй даалгаврууд хязгаарлагдмал байдаг.

Агентын RAG нь RAG-ыг илүү динамик, контекст дээр суурилсан болгож хөгжүүлдэг. Тогтмол workflow-д найдахын оронд, агентууд нь бодит цагт ямар өгөгдөл хэрэгтэйгээ, хаанаас олохоо, өгөгдсөн даалгаврын үндсэн дээр асуултаа хэрхэн боловсронгуй болгохоо шийддэг.

Агентын RAG vs Уламжлалт RAG:

- **Динамик хайлт**: Агент нь шаардлагатай бол олон эх сурвалжаас мэдээлэл цуглуулж, асуултаа боловсронгуй болгож, шинэ мэдээлэл гарч ирэхэд дасан зохицож чаддаг.
- **Олон алхам бүхий дүгнэлт**: Агент нь анхны хайлтын үр дүнд үндэслэн дараагийн асуултуудыг үүсгэж, илүү гүнзгий мэдлэг олж авч чаддаг.
- **Багажийн ашиглалт**: Агент нь зөвхөн баримтын хайлт биш, API дуудах, өгөгдлийн сангаас асуулт хийх, тооцоолол хийх зэрэг олон багаж ашиглаж болно.

- **Контекст санах**: Санах ойгоороо агент нь өмнөх асуултууд болон хариултуудыг хадгалж, дараагийн асуултдаа илүү нарийвчлалтай хандаж чаддаг.

Жишээлбэл, маркетингийн стратеги боловсруулж байгаа агент нь:

1. CRM-ээс харилцагчийн өгөгдөл татаж авна
2. API ашиглан зах зээлийн чиг хандлагыг цуглуулна
3. Шинэ мэдээлэл гарч ирэхэд өөрийн аргачлалаа боловсронгуй болгоно
4. Санах ой болон итераци ашиглан илүү нарийвчлалтай, холбогдолтой үр дүн гаргана

Агентын RAG нь хайлт, дүгнэлт, үйлдлийг нэгтгэдэг. Энэ нь RAG-ыг тогтмол конвейероос төвлөрсөн, дасан зохицох боломжтой систем болгон хувиргадаг.

2.4 ХОУ агентууд

2.4.1 Агент гэж юу вэ

Агент гэдэг нь өөрийн орчныг мэдрэх, түүн дээр үйлдэл хийх чадвартай систем юм. ХОУ-оор дэмжигдсэн агентууд нь суурь загваруудын хүч чадлаар дамжуулан бидний туслах, хамтран ажиллагч, сургагч байж чадна. Тэд вебсайт бүтээх, өгөгдөл цуглуулах, аялал төлөвлөх, зах зээлийн судалгаа хийх, харилцагчийн данс удирдах, өгөгдөл оруулалтыг автоматжуулах зэрэг олон зүйлд тусалж чадна.

2.4.2 Агентын бүрэлдэхүүн хэсгүүд

ХОУ агентыг тодорхойлдог гурван гол зүйл:

1. **Орчин (Environment)**: Агент ажиллах орчин нь түүний хэрэглээний тохиолдлоор тодорхойлогдоно. Жишээ нь: тоглоом (Minecraft, Go), интернэт, гал тогоо, зам.
2. **Үйлдлүүд (Actions)**: Агентын хийж чадах үйлдлүүд нь түүний хандах боломжтой багажуудаар өргөжинө.
3. **Даалгавар (Task)**: Хэрэглэгчээс өгөгдсөн зорилго, зорилт.

2.4.3 Багажууд (Tools)

Гадаад багажууд байхгүй бол агентын чадавхи маш хязгаарлагдмал байх болно. Багажууд нь агентыг илүү чадварлаг болгодог. Багажуудын гурван ангилал:

Мэдлэг нэмэгдүүлэх багажууд

RAG системийн бүрэлдэхүүн хэсгүүд:

- Текст retriever
- Зураг retriever
- SQL executor
- Интернэт хайлт API
- Дотоод хайлтын системүүд

Чадавхи өргөтгөх багажууд

- **Тооны машин**: ХОУ загварууд математикт сул байдаг. Тооны машин нь энгийн тооцоог гүйцэтгэнэ.
- **Код интерпретер**: Код бичиж, ажиллуулах, үр дүнг задлах. Энэ нь кодчилолын туслах, өгөгдөл шинжлэгч, судалгааны туслах боломжийг олгоно.

Бичих үйлдлийн багажууд

Зөвхөн унших биш, өөрчлөлт оруулах багажууд:

- Өгөгдлийн санд өгөгдөл нэмэх, засварлах, устгах
- Email илгээх
- Банкны шилжүүлэг эхлүүлэх
- Календарт үйл явдал нэмэх

Анхааруулга: Бичих үйлдэл нь өндөр эрсдэлтэй. Code injection довтолгооноос болгоомжлох хэрэгтэй.

2.4.4 Төлөвлөлт (Planning)

Төлөвлөлт нь агентын гол үүрэг бөгөөд дараах үе шаттай:

1. **Төлөвлөгөө үүсгэх (Plan Generation)**: Даалгаврыг гүйцэтгэх дараалсан үйлдлүүдийн төлөвлөгөө гаргах. Үүнийг task decomposition буюу даалгавар задлах гэж нэрлэнэ.
2. **Эргэцүүлэн бодох ба алдаа засах (Reflection)**: Үүсгэсэн төлөвлөгөөг үнэлэх. Муу байвал шинэ төлөвлөгөө гаргах.
3. **Гүйцэтгэл (Execution)**: Төлөвлөгөөнд заасан үйлдлүүдийг хийх. Энэ нь ихэвчлэн функц дуудалт хийх явдал юм.
4. **Үр дүнг үнэлэх**: Үйлдлийн үр дүнг хүлээн авсны дараа зорилго биелсэн эсэхийг тодорхойлох. Алдааг тодорхойлж засах.

2.4.5 Суурь загварууд төлөвлөгч болж чадах уу

Зарим судлаачид LLM нь төлөвлөгч байж чадахгүй гэж үздэг. Учир нь төлөвлөлт нь үндсэндээ хайлтын асуудал бөгөөд autoregressive загвар нь зөвхөн урагш үйлдэл үүсгэж чаддаг гэж үздэг. Гэвч бодит байдал дээр загвар дахин эхлэж өөр зам сонгож чаддаг тул энэ нь төдийлөн хатуу хязгаарлалт биш юм.

Төлөвлөлтийг сайжруулах аргууд:

- Илүү сайн system prompt бичих, жишээ олноор өгөх
- Багажуудын тайлбарыг илүү сайн бичих
- Функцүүдийг хялбарчлах, задлах
- Илүү хүчтэй загвар ашиглах
- Төлөвлөлтөд зориулж загвар нарийвчлан сургах

2.4.6 Эргэцүүлэн бодох ба алдаа засах

Хамгийн сайн төлөвлөгөө ч байнга үнэлэгдэж, тохируулагдах шаардлагатай. Reflection нь агентын амжилтад чухал үүрэг гүйцэтгэнэ. Үүнийг хоёр аргаар хийж болно:

- **Self-critique**: Ижил загвар өөртэйгөө ярилцаж алдааг илрүүлнэ.
- **Тусдаа үнэлэгч**: Тусдаа загвар эсвэл функц үр дүнд оноо өгнө.

2.4.7 Агентын санах ой

ХОУ загвар нь гурван санах ойн механизмтай:

1. **Дотоод мэдлэг (Internal Knowledge)**: Загвар өөрөө нь санах ой юм. Сургалтын өгөгдлөөс олж авсан мэдлэг. Загварыг шинэчлэхгүй бол өөрчлөгдөхгүй.
2. **Богино хугацааны санах ой (Short-term Memory)**: Загварын контекст. Өмнөх мессежүүд контекстэд нэмэгдэж болно. Даалгавар дууссаны дараа устдаг. Хурдан боловч багтаамж хязгаарлагдмал.
3. **Урт хугацааны санах ой (Long-term Memory)**: Гадаад өгөгдлийн эх сурвалж (RAG). Даалгаврууд хооронд үргэлжилдэг. Өгөгдлийг устгаж болно.

2.4.8 Агентын дизайны загварууд

Агентууд нь зөвхөн үндсэн чадваруудаасаа биш, тэдгээрийн workflow болон харилцааг бүтэцлэх дизайны загваруудаас ч хүч авдаг. Эдгээр загварууд нь агентуудад нарийн төвөгтэй асуудлыг шийдэх, хувьсах орчинд дасан зохицох, үр дүнтэй хамтран ажиллах боломжийг олгодог.

Эргэцүүлэн бодох (Reflection)

Reflection нь агентуудад өөрсдийн шийдвэрийг үнэлж, үйлдэл хийх эсвэл эцсийн хариулт өгөхөөсөө өмнө гаралтаа сайжруулах боломжийг олгодог. Энэ чадвар нь агентуудад

алдаагаа олж засах, дүгнэлтээ боловсронгуй болгох, илүү өндөр чанартай үр дүн гаргах боломжийг олгоно.

Жишээлбэл, код бичиж байгаа агент нь эхлээд код үүсгээд, дараа нь өөрөө тухайн кодыг шалгаж, алдаа олж, сайжруулалт хийснийхээ дараа хэрэглэгчид хүргэнэ. Энэ нь эцсийн үр дүнгийн чанарыг мэдэгдэхүйц сайжруулдаг.

Багажийн ашиглалт (Tool Use)

Гадаад багажуудтай холбогдох нь агентын функционалыг өргөжүүлж, өгөгдөл авах, үйл явцыг автоматжуулах, эсвэл детерминист workflow-г гүйцэтгэх зэрэг даалгаврыг гүйцэтгэх боломжийг олгоно. Энэ нь математик тооцоолол эсвэл өгөгдлийн сангийн асуулга гэх мэт нарийвчлал шаардлагатай үйлдлүүдэд онцгой ач холбогдолтой.

Багажийн ашиглалт нь уян хатан шийдвэр гаргалт болон таамагладаг, найдвартай гүйцэтгэлийн хоорондох хоосон зайг нөхдөг.

Төлөвлөлт (Planning)

Төлөвлөлтийн чадвартай агентууд нь өндөр түвшний зорилгуудыг үйлдэл хийх боломжтой алхмуудад задалж, даалгавруудыг логик дарааллаар зохион байгуулж чаддаг. Энэ дизайны загвар нь олон алхам бүхий асуудлыг шийдэх эсвэл хамааралтай workflow-г удирдахад чухал юм.

Жишээлбэл, аялал төлөвлөх агент нь дараах төлөвлөгөө гаргаж болно:

1. Нислэг хайх
2. Зочид буудал захиалах
3. Үзвэр үйлчилгээний цэгүүдийг судлах
4. Өдөр бүрийн маршрут үүсгэх

Олон агентын хамтын ажиллагаа (Multi-Agent Collaboration)

Олон агентын системүүд нь асуудлыг шийдэхэд модуль чиглэсэн арга барил ашигладаг - тодорхой даалгавруудыг мэргэшсэн агентуудад хуваарилдаг. Энэ арга нь уян хатан байдлыг санал болгодог: та үр ашигтай байдлыг сайжруулахын тулд даалгавар тодорхой агентуудад илүү жижиг хэлний загварууд (SLMs) ашиглаж болно.

Модульчлагдсан дизайн нь тэдгээрийн контекстыг тусгай даалгавруудаа чиглүүлэх замаар агент тус бүрийн нарийн төвөгтэй байдлыг багасгадаг. Хамтран ажиллахаар эдгээр мэргэшсэн агентууд мэдээлэл солилцож, хариуцлагыг хуваарилж, нарийн төвөгтэй сорилтуудыг илүү үр дүнтэй шийдэхийн тулд үйлдлүүдээ зохицуулдаг.

Уламжлалт системийн дизайнтай адилаар, асуудлыг модульчлагдсан бүрэлдэхүүн хэсгүүдэд задлах нь тэднийг засварлах, өргөжүүлэх, дасан зохицуулахад илүү хялбар болгодог.

2.5 ХОУ инженерчлэлийн хэрэгжүүлэлт

Суурь загварууд ашиглан аппликейшн хөгжүүлэх нь уламжлалт ML инженерчлэлээс гурван чухал талаараа ялгаатай:

1. **Загвар дасан зохицуулалт**: Өөрөө загвар сургахын оронд бусдын сургасан загварыг ашиглана. Үүнээс болж загвар дасан зохицуулалт (model adaptation) илүү чухал болсон.

2. **Тооцоолол эрчимт**: Загварууд том, илүү их тооцоолол шаарддаг, хоцрогдол өндөр. Үр ашигтай сургалт болон inference optimization илүү чухал.

3. **Нээлттэй гаралт**: Загварууд нээлттэй төгсгөлтэй гаралт үүсгэдэг тул үнэлгээ илүү том асуудал болсон.

Загвар дасан зохицуулалтын хоёр гол арга:

- **Prompt-based techniques**: Загварын жин өөрчлөхгүйгээр зааварчилгаа, контекст өгч дасан зохицуулна. Хялбар, цөөн өгөгдөл шаардана. Prompt engineering энд хамаарна.

- **Finetuning**: Загварын жинг өөрчилж дасан зохицуулна. Илүү нарийн төвөгтэй, илүү их өгөгдөл шаардана. Гэвч чанар, хоцрогдол, өртгийг мэдэгдэхүйц сайжруулж чадна.

3. МИКРОСЕРВИС АРХИТЕКТУР

3.1 Монолитоос микросервис рүү

3.1.1 Монолитын эрин үе

Вэб аппликейшн хөгжүүлэлтийн эхэн үед бүх зүйлийг монолит хэлбэрээр бүтээдэг байсан - бүх бизнес логик, хэрэглэгчийн харилцаа, өгөгдлийн үйлдлүүд нэг том, нягт нэгдсэн кодын санд амьдардаг байв. Энэ арга нь эхэндээ утга учиртай байсан. Монолитууд нь хөгжүүлэх, deploy хийхэд энгийн байсан, ялангуяа аппликейшнууд хязгаарлагдмал нарийн төвөгтэй байдалтай, нэг сервер дээр ажилладаг үед.

3.1.2 Монолитыг өргөжүүлэх сорилт

Гэвч аппликейшнууд томрох тусам асуудлууд ч өсдөг. Монолитыг өргөжүүлэх нь бүх зүйлийг өргөжүүлэх гэсэн үг - системийн зөвхөн нэг хэсэгт илүү их нөөц шаардлагатай байсан ч гэсэн. Нэг модульд хийсэн жижиг өөрчлөлт бүх код санд өргөжиж болох бөгөөд энэ нь шинэчлэлийг удаан, эрсдэлтэй болгодог. Хамгийн муу нь, өөр өөр функц дээр ажиллаж байгаа багууд байнга бие биенийхээ хөлд гишгэж, хөгжлийг удаашруулж, алдаа гарах эрсдэлийг нэмэгдүүлдэг.

Монолит архитектураас эхэлсэн олон компани эцэст нь эдгээр хязгаарлалтад тулгарсан. Atlassian, Jira болон Confluence-ийн ард байгаа компани, тэдний нэг нь байв. 2018 оноос өмнө Atlassian нь эдгээр бүтээгдэхүүндээ монолит архитектурт найдаж байсан. Компани өргөжихийн хэрээр, дэлхийн таван хөгжлийн төвд тархсан багуудтай, энэхүү төвлөрсөн монолит бүтэц нь томоохон саад болов. Бүх бизнес логикийн нэг кодын санд нягт холбогдсон байдал нь жижиг өөрчлөлт ч гэсэн бүх stack-ийг дахин бүтээж, дахин deploy хийх шаардлагатай болгосон.

Хөгжүүлэгчид монолит руу хандах эрхийг зохицуулах ёстой байсан, энэ нь хоцрогдол, багуудын хоорондын үрэлт нэмэгдэхэд хүргэсэн. Хурдан хөдлөхийн оронд багууд бүх

системийг шинэчлэх, турших хүртэл хүлээх шаардлагатай байв. Энэ нь хурдан хөгжлөөр амьдардаг компанид үндсэн саад байсан.

3.1.3 Микросервис рүү шилжих

Эдгээр хязгаарлалтаас ангижрахын тулд Atlassian 2018 онд Jira болон Confluence-г микросервис архитектур руу шилжүүлсэн. Энэ өөрчлөлт нь тархсан багуудад бие даан ажиллах, өөрчлөлтийг хурдан deploy хийх, бүх платформыг дахин deploy хийхгүйгээр шинэчлэл гаргах боломжийг олгосон. Микросервис рүү шилжих нь зөвхөн өргөжүүлэх чадварыг сайжруулаад зогсохгүй, багуудад автономи өгч, зохицуулалтын ачааллыг бууруулж, инновацийг хурдасгасан.

3.1.4 Микросервисийн тодорхойлолт

Микросервис архитектур нь програм хангамжийг хөгжүүлэх арга бөгөөд аппликейшныг жижиг, бие даасан үйлчилгээнүүдэд хувааж хөгжүүлдэг.

Микросервисийн гол онцлогууд:

- **Бие даасан байдал**: Микросервис бүр бие даасан үүрэгтэй, өөрийн өгөгдлийн сантай.
- **Уян хатан хөгжүүлэлт**: Өөр өөр багууд өөр өөр технологи, хэл ашиглаж хөгжүүлж болно.
- **Өргөжих чадвар**: Зөвхөн шаардлагатай үйлчилгээг л өргөжүүлж болно.
- **Найдвартай байдал**: Нэг үйлчилгээ унавал бусад үйлчилгээнүүд ажиллаж үргэлжлэнэ.

3.2 Микросервисийн давуу тал

1. **Технологийн олон янз байдал**: Үйлчилгээ бүр өөрийн хэрэгцээнд тохирсон технологи сонгож болно. Жишээ нь: нэг үйлчилгээ Python, нөгөө нь Go, өөр нь Node.js ашиглаж болно.

2. **Багуудын бие даасан ажиллагаа**: Баг бүр өөрийн үйлчилгээг бие даан хөгжүүлж, deploy хийж чадна.

3. **Хурдан deployment**: Том системийг бүхэлд нь дахин deploy хийх шаардлагагүй. Зөвхөн өөрчлөгдсөн үйлчилгээг л deploy хийнэ.

4. **Илүү сайн өргөжих чадвар**: Ачаалал их үйлчилгээг л өргөжүүлэх нь бүх системийг өргөжүүлэхээс үр ашигтай.

5. **Алдааны тусгаарлалт**: Нэг үйлчилгээний алдаа бусад үйлчилгээнд дамжихгүй.

3.3 Микросервисийн сорилтууд

Микросервис архитектур олон давуу талтай ч дараах сорилтуудтай:

1. **Нарийн төвөгтэй байдал**: Олон үйлчилгээнүүдийг удирдах, тэдгээрийн харилцааг хянах нь монолит системээс илүү төвөгтэй.

2. **Өгөгдлийн консистенс**: Үйлчилгээ бүр өөрийн өгөгдлийн сантай байх нь өгөгдлийн нийцтэй байдлыг хангахад хэцүү болгодог.

3. **Сүлжээний хоцрогдол**: Үйлчилгээнүүд хоорондоо сүлжээгээр харилцах нь нэмэлт хоцрогдол авчирна.

4. **Алдаа илрүүлэх хэцүү байдал**: Олон үйлчилгээгээр дамжин явах хүсэлтийн алдааг олох, засах хэцүү.

5. **Үйлчилгээ хоорондын харилцаа**: Үйлчилгээнүүд хэрхэн харилцах, мэдээлэл солилцох нь нарийн асуудал.

6. **Transaction удирдлага**: Олон үйлчилгээгээр transaction явуулах нь үл нэгдэл өгөгдлийн сангийн transaction-аас илүү төвөгтэй.

3.4 Микросервис хоорондын харилцаа

Микросервисүүд хоорондоо хоёр гол аргаар харилцдаг:

3.4.1 Синхрон харилцаа (Synchronous Communication)

HTTP/REST API эсвэл gRPC ашиглан шууд хүсэлт илгээж хариу хүлээнэ. Энэ нь хялбар боловч:

- Tight coupling үүсгэдэг
- Нэг үйлчилгээ унавал бусад үйлчилгээнүүд алдаа гаргана
- Latency нэмэгддэг

3.4.2 Асинхрон харилцаа (Asynchronous Communication)

Message queue (RabbitMQ, Kafka) ашиглан мессеж солилцоно. Энэ нь:

- Loose coupling үүсгэнэ
- Илүү найдвартай
- Илүү төвөгтэй

3.5 Үйл явдлаар удирдагдах архитектур

3.5.1 Үйл явдлаар удирдагдах архитектур гэж юу вэ

Микросервисүүд гарч ирснээр шинэ сорилт бий болсон: эдгээр үйлчилгээнүүд хэрхэн үр дүнтэй харилцах вэ? Хэрэв бид үйлчилгээнүүдийг шууд RPC эсвэл API дуудлагаар холбовол бид асар том хамааралын сүлжээ үүсгэнэ. Хэрэв нэг үйлчилгээ унавал энэ нь холбогдсон замын дагуух бүх node-д нөлөөлнө.

Үйл явдлаар удирдагдах архитектур (Event-Driven Architecture, EDA) нь энэ асуудлыг шийдсэн. Нягт холбогдсон, синхрон харилцааны оронд EDA нь бүрэлдэхүүн хэсгүүдэд үйл явдлаар асинхрон харилцах боломжийг олгодог. Үйлчилгээнүүд бие биенийхээ хүлээхгүй - тэд бодит цагт юу болж байгаад хариу үйлдэл үзүүлдэг.

3.5.2 EDA-ын давуу тал

1. **Салангид байдал (Decoupling)**: Үйлчилгээнүүд үйл явдлаар харилцдаг тул тэд бие биенээсээ хараат бус байна. Нэг үйлчилгээ өөрчлөгдөх эсвэл унах нь бусад үйлчилгээнд шууд нөлөөлөхгүй.

2. **Өргөжүүлэх чадвар**: Үйлчилгээ бүр үйл явдлыг бие даан боловсруулах тул системийг өргөжүүлэх илүү хялбар.

3. **Уян хатан байдал**: Шинэ үйлчилгээ нэмэх эсвэл одоо байгаа үйлчилгээг өөрчлөх нь бусад үйлчилгээнд өөрчлөлт шаардахгүй.

4. **Бодит цагийн боловсруулалт**: Үйл явдлууд тэр даруйдаа боловсруулагдах тул систем нь өөрчлөлтөд хурдан хариу үйлдэл үзүүлнэ.

3.5.3 Apache Kafka: Үйл явдлын хүчирхэг суурь

Apache Kafka нь салангид, өндөр дамжуулалттай, бага хоцрогдолтой үйл явдлын streaming платформ юм. Kafka нь EDA архитектурын төв мэдрэлийн систем болж чаддаг.

Kafka-ийн үндсэн ойлголтууд

- **Topic**: Үйл явдлуудыг ангилах логик channel. Жишээ нь: ”user-events”, ”order-events”
- **Producer**: Үйл явдлыг topic руу бичдэг аппликейшн
- **Consumer**: Topic-оос үйл явдлыг уншдаг аппликейшн
- **Partition**: Topic-ийг өргөжүүлэх, параллель боловсруулалт хийх боломжтой болгодог
- **Consumer Group**: Олон consumer нэг багаар ажиллаж ачааллыг хуваарилдаг

Kafka-ийн давуу тал

1. **Хэвтээ өргөжих чадвар**: Kafka-ийн салангид дизайн нь саадгүйгээр шинэ агент эсвэл consumer нэмэх боломжийг олгоно.

2. **Бага хоцрогдол**: Бодит цагийн үйл явдал боловсруулалт нь агентуудад өөрчлөлтөд шууд хариу үйлдэл үзүүлэх боломжийг олгоно.

3. **Loose Coupling**: Topic-уудаар харилцах нь агентууд хараат бус, өргөжүүлэх боломжтой байхыг баталгаажуулна.

4. **Үйл явдлын хадгалалт**: Тогтвортой мессежийн хадгалалт нь өгөгдөл дамжилтын явцад алга болохгүй гэдгийг баталгаажуулна.

5. **Дахин тоглуулах боломж (Replayability)**: Kafka нь үл хөдлөх салангид log учраас үйл явдал бүр хадгалагдаж, debug, үнэлгээ, дахин сургалтад дахин тоглуулж болно.

3.5.4 Apache Flink: Streaming боловсруулалтын хөдөлгүүр

Apache Flink нь stateful тооцоолол, үйл явдал цагийн боловсруулалт хийх чадвартай салангид stream processing framework юм. Flink нь Kafka-тай хамт ашиглах замаар хүчирхэг бодит цагийн өгөгдөл боловсруулалтын конвейер бүтээх боломжийг олгоно.

Flink-ийн давуу тал

- **Stateful боловсруулалт**: Flink нь state-г найдвартай удирдаж, нарийн төвөгтэй тооцоолол хийх боломжийг олгоно.
- **Цаг цагийн боловсруулалт**: Үйл явдлын цаг дээр үндэслэн бодит цагийн шинжилгээ хийх.
- **Өндөр дамжуулалт**: Секундэд сая сая үйл явдлыг боловсруулж чаддаг.
- **Exactly-once семантик**: Мэдээлэл нэг удаа л боловсруулагдахыг баталгаажуулна.

Flink AI Inference

Flink нь LLM-тэй ажиллах чадвартай. Flink AI inference нь өгөгдлийг авч, LLM рүү илгээж, хариу авах боломжийг олгоно. Энэ нь төлөвлөгч агентыг Flink app болгон хөгжүүлэх боломжийг олгодог.

Жишээлбэл:

- Kafka topic-оос үйл явдал уншина
- LLM ашиглан контекст ойлгох, төлөвлөгөө гаргах
- Үр дүнг өөр Kafka topic руу бичих
- Бусад агентууд энэ үр дүнг уншиж гүйцэтгэнэ

Flink болон RAG

Hallucination-ийг багасгахын тулд LLM-ийг бодит өгөгдөлд суурилуулах хэрэгтэй. Flink нь RAG pattern-ийг бүтээхэд тусална:

1. Flink нь өгөгдлийг боловсруулна
2. LLM inference ашиглан өгөгдлийг embedding болгон хөрвүүлнэ
3. Embedding-ыг Kafka topic руу бичнэ
4. Kafka Connect ашиглан vector database руу синхрончлоно
5. Агентууд RAG ашиглан бодит өгөгдөл дээр суурилсан хариулт өгнө

3.5.5 Агентууд ба EDA

ХОУ агентуудыг өргөжүүлэх нь үндсэндээ салангид системийн асуудал юм. Агентууд нь шийдвэр гаргаж, үйлдэл хийхийн тулд олон эх сурвалж, бусад агентууд, багажууд, гадаад системүүдээс мэдээлэл цуглуулах шаардлагатай.

Агентууд яагаад EDA шаарддаг:

- **Асинхрон шинж чанар**: Агентууд нь хүн шиг ажилладаг - тэд олон эх сурвалжаас мэдээлэл цуглуулж, өгөгдлийг шинжилж, үйлдэл хийхээсээ өмнө шийдвэр гаргах хэрэгтэй. Эдгээр үйл явцууд нь асинхрон шинжтэй.
- **Мэдээллийн хамаарал**: Микросервисүүд ихэвчлэн салангид үйлдлүүдийг боловсруулдаг бол агентууд нь контекст баялаг, хуваалцсан мэдээлэлд найддаг. Энэ нь хамаарлыг удирдах, бодит цагийн өгөгдлийн урсгалыг баталгаажуулах онцгой шаардлагыг бий болгоно.
- **Олон хэрэглэгчдэд үйлчлэх**: Агентын гаралт нь зөвхөн AI app руу буцахгүй - тэд өгөгдлийн агуулах, CRM, CDP, customer success платформ зэрэг бусад чухал системүүд рүү урсах ёстой.

Тийм ээ, та агентуудыг RPC болон API-аар холбож болно, гэхдээ энэ нь нягт холбогдсон системүүд бий болгох жор юм. Нягт холбоос нь өргөжүүлэх, дасан зохицох, эсвэл ижил өгөгдлийн олон хэрэглэгчдийг дэмжихэд хэцүү болгодог. Агентууд нь уян хатан байдал шаарддаг. Тэдний гаралт нь бусад агентууд, үйлчилгээнүүд, платформуудад хатуу хамааралгүйгээр үр дүнгүй дамжих ёстой.

4. АСУУДЛЫН ТОДОРХОЙЛОЛТ

4.1 Микросервис архитектур дахь сорилтууд

Микросервис архитектурт дараах гол асуудлууд байдаг:

4.1.1 Үйлчилгээ хоорондын нарийн төвөгтэй логик

Хэрэглэгчийн нэг хүсэлт нь олон үйлчилгээгээр дамжин явах ёстой. Жишээ нь онлайн худалдааны систем дээр:

1. Хэрэглэгч бараа захиална
2. Захиалгын үйлчилгээ захиалга үүсгэнэ
3. Агуулахын үйлчилгээнээс бараа байгаа эсэхийг шалгана
4. Төлбөрийн үйлчилгээгээр төлбөр хийнэ
5. Хүргэлтийн үйлчилгээнд мэдэгдэнэ
6. Notification үйлчилгээгээр хэрэглэгчид мэдэгдэнэ

Эдгээр алхам бүрт алдаа гарч болох бөгөөд алдаа гарах үед юу хийх нь тодорхойгүй байдаг. Уламжлалт аргаар энэ workflow-г хатуу кодлох нь:

- Уян хатан бус
- Шинэ үйлчилгээ нэмэхэд хэцүү
- Алдаа засалт хэцүү
- Бизнес дүрмийн өөрчлөлт хийхэд том код өөрчлөлт шаардлагатай

4.1.2 Өгөгдлийн нэгтгэл ба шинжилгээ

Микросервис бүр өөрийн өгөгдлийн сантай байх нь:

- Олон өгөгдлийн сангийн өгөгдлийг нэгтгэх хэцүү
- Cross-service шинжилгээ хийх төвөгтэй
- Тайлангийн системд өгөгдөл цуглуулах асуудалтай
- Хэрэглэгчийн бүрэн дүр төрхийг харах хэцүү

4.1.3 Динамик routing ба шийдвэр гаргалт

Хэрэглэгчийн хүсэлтийг аль үйлчилгээ рүү чиглүүлэх, хэдэн үйлчилгээг дуудах шаардлагатайг тодорхойлох нь:

- Бизнес логик өөрчлөгдөх бүрд код өөрчлөх шаардлагатай
- A/B testing хийхэд төвөгтэй
- Орчны өөрчлөлтөд дасан зохицох чадваргүй

4.1.4 Мониторинг ба алдаа илрүүлэлт

- Олон үйлчилгээний логийг нэгтгэх хэцүү
- Хэрэглэгчийн хүсэлт аль үйлчилгээнд алдаа гаргасныг тодорхойлох хэцүү
- Алдааны язгуур шалтгааныг олох төвөгтэй

4.1.5 Агентын монолит

Агент суурилсан систем хөгжүүлэхдээ анхны хувилбар нь ихэвчлэн монолит хэлбэртэй байдаг. Controller, Planner, SQL Handler, Streaming Handler зэрэг агентууд нэг аппликейшн дотор ажилладаг. Энэ арга нь эхэндээ ажилладаг ч дараах асуудлуудтай:

- **Deployment хамаарал**: Planner-ийн v2 гаргахын тулд бүх системийг дахин deploy хийх шаардлагатай.

- **Hardware хязгаарлалт**: Planner агент нь LLM ашиглаж GPU шаарддаг бол SQL Handler нь жижиг SLM ашиглан CPU дээр ажиллаж болно. Монолит систем энэ хоёрыг салгаж чадахгүй.
- **Өргөжүүлэх боломжгүй**: Нэг агентад их ачаалал ирсэн ч бүх системийг өргөжүүлэх шаардлагатай болно.
- **Нягт холбоос**: Агентууд хоорондоо шууд дуудлагаар холбогдсон байх нь алдааны дамжлага, хоцрогдлын асуудал бий болгоно.

4.2 ХОУ агентууд хэрхэн туслах вэ

ХОУ агентууд нь дээрх асуудлуудыг дараах аргаар шийдэж чадна:

1. **Ухаалаг орчуулагч (Intelligent Orchestrator)**: Агент нь хүсэлтийг ойлгож, аль үйлчилгээнүүдийг дуудах хэрэгтэй, ямар дарааллаар дуудах, алдаа гарвал яах талаар төлөвлөгөө гарган гүйцэтгэнэ.
2. **Контекст бүтээгч (Context Builder)**: RAG ашиглан олон өгөгдлийн сангийн мэдээллийг нэгтгэж, ойлгомжтой контекст бүтээнэ.
3. **Динамик routing**: Хэрэглэгчийн хүсэлтийг ойлгож, хамгийн тохиромжтой үйлчилгээ рүү чиглүүлнэ.
4. **Өгөгдлийн задлагч (Data Analyst)**: Олон үйлчилгээний өгөгдлийг цуглуулж, шинжилж, ойлгомжтой тайлан гаргана.
5. **Алдаа засагч (Error Handler)**: Алдаа гарсан тохиолдолд шалтгааныг олж, засах төлөвлөгөө санал болгоно.

4.3 Үйл явдлаар удирдагдах архитектурын шаардлага

Агентуудыг үр дүнтэй өргөжүүлэхийн тулд тэдгээрийг микросервис болгон салгах шаардлагатай. Гэхдээ микросервисүүд нь RPC эсвэл API дуудлагаар харилцах нь дахин монолиттой адилхан асуудал үүсгэнэ - зөвхөн салгагдсан хэлбэртэй.

Үйл явдлаар удирдагдах архитектур (EDA) нь дараах асуудлуудыг шийднэ:

- **Салангид байдал**: Агентууд хоорондоо Kafka topics-оор харилцах тул шууд хамаарал байхгүй.
- **Асинхрон**: Агентууд бие биенийхээ хүлээхгүй, бодит цагт хариу үйлдэл үзүүлдэг.
- **Олон хэрэглэгч**: Нэг агентын гаралтыг олон агент, систем ашиглаж болно.
- **Өргөжих чадвар**: Агент бүр бие даан өргөжиж болно.
- **Найдвартай байдал**: Kafka-ийн үйл явдлын хадгалалт нь мэдээлэл алдагдахгүйг баталгаажуулна.

5. ШИЙДЭЛ БА САНАЛ БОЛГОЖ БУЙ ЗАГВАР

5.1 ХОУ агент суурилсан микросервис архитектур

Энэхүү судалгаанд микросервис архитектурт ХОУ агентуудыг нэвтрүүлэх загварыг санал болгож байна. Уг загвар нь дараах гол бүрэлдэхүүнтэй:

5.1.1 Агентын архитектур

Orchestrator Agent (Орчуулагч агент)

Энэ агент нь хэрэглэгчийн хүсэлтийг хүлээн авч, ямар үйлчилгээнүүдийг дуудах, ямар дарааллаар дуудах талаар төлөвлөгөө гаргадаг. Үүний бүтэц:

- **Intent Understanding**: LLM ашиглан хэрэглэгчийн хүсэлтийг ойлгох
- **Service Discovery**: Боломжтой үйлчилгээнүүдийн жагсаалтыг мэдэх
- **Execution Planning**: Дуудах үйлчилгээнүүдийн дараалал, параметрүүдийг тодорхойлох
- **Error Handling**: Алдаа гарвал алдааг засах төлөвлөгөө гаргах

Service Agents (Үйлчилгээний агентууд)

Үйлчилгээ бүр өөрийн агенттай байж болно. Энэ агент нь:

- Үйлчилгээний API-г тайлбарлаж өгнө
- Хүсэлтийг тухайн үйлчилгээнд тохирсон форматруу хөрвүүлнэ
- Үр дүнг ойлгомжтой хэлбэрт хөрвүүлнө
- Үйлчилгээний статус, чадавхийг мэдээлнэ

Knowledge Agent (Мэдлэгийн агент)

RAG систем ашиглан:

- Үйлчилгээнүүдийн баримт бичгийг хадгална
- Өмнөх хүсэлт, хариултуудыг санана
- Бизнес дүрэм, журмуудыг мэднэ
- API спецификацийг хадгална

Monitoring Agent (Хяналтын агент)

- Логуудыг цуглуулж, шинжилнэ
- Алдаануудыг илрүүлнэ
- Системийн эрүүл байдлыг хянана
- Сайжруулах санал өгнө

5.1.2 Системийн архитектур

Санал болгож буй үйл явдлаар удирдагдах архитектур нь:

```
    |
    v
API Gateway
    |
    v
[user-requests] Kafka Topic
    |
    v
Orchestrator Agent (Event Consumer/Producer)
```

```
|
+---> [planning-tasks] Kafka Topic
|           |
|           v
|       Planner Agent (Flink App with LLM)
|           |
|           v
|       [execution-plans] Kafka Topic
|
+---> [knowledge-queries] Kafka Topic
|           |
|           v
|       Knowledge Agent (RAG + Vector DB)
|           |
|           v
|       [knowledge-results] Kafka Topic
|
+---> [service-calls] Kafka Topic
          |
          +---> Service Agent 1 --> Microservice 1
          |
          +---> Service Agent 2 --> Microservice 2
          |
          +---> Service Agent N --> Microservice N
          |
          v
      [service-results] Kafka Topic
          |
          v
```

```
Response Aggregator
    |
    v
[user-responses] Kafka Topic
    |
    v
API Gateway -->
```

5.1.3 Kafka Topics ба хамаарал

Системд дараах Kafka topics ашиглагдана:

- **user-requests**: Хэрэглэгчийн хүсэлтүүд
- **planning-tasks**: Төлөвлөгөө гаргах даалгаврууд
- **execution-plans**: Гүйцэтгэх төлөвлөгөөнүүд
- **knowledge-queries**: RAG хайлтын асуултууд
- **knowledge-results**: RAG-ийн үр дүнгүүд
- **service-calls**: Микросервис дуудлагууд
- **service-results**: Микросервисийн үр дүнгүүд
- **user-responses**: Хэрэглэгчид буцах хариултууд
- **monitoring-events**: Системийн логууд ба мониторинг

5.1.4 Үйл явдлаар удирдагдах Workflow

1. **Хүсэлт publish**: API Gateway хүсэлтийг `user-requests` topic руу бичнэ
2. **Orchestrator consume**: Orchestrator Agent үйл явдлыг уншиж, intent-ийг ойлгоно
3. **Төлөвлөлт үйл явдал**: Хэрэв төлөвлөлт хэрэгтэй бол `planning-tasks` topic руу үйл явдал илгээнэ

4. **Flink Planning**: Planner Agent (Flink app) үйл явдлыг уншиж, LLM ашиглан төлөвлөгөө гаргаж, `execution-plans` topic руу бичнэ
5. **Knowledge хайлт**: Orchestrator нь `knowledge-queries` topic руу хайлтын үйл явдал илгээнэ
6. **RAG боловсруулалт**: Knowledge Agent үйл явдлыг уншиж, vector хайлт хийж, үр дүнг `knowledge-results` topic руу бичнэ
7. **Service дуудлага**: Төлөвлөгөөний дагуу `service-calls` topic руу үйл явдлууд илгээгдэнэ
8. **Параллель гүйцэтгэл**: Service Agent-ууд параллель байдлаар үйл явдлуудыг уншиж, микросервисүүдийг дуудаж, үр дүнг `service-results` topic руу бичнэ
9. **Aggregation**: Response Aggregator бүх үр дүнг цуглуулж нэгтгэнэ
10. **Хариулт publish**: Эцсийн хариултыг `user-responses` topic руу бичнэ
11. **Хэрэглэгчид хүргэх**: API Gateway хариултыг уншиж хэрэглэгчид буцаана
12. **Мониторинг**: Бүх агент `monitoring-events` topic руу лог мэдээлэл бичнэ

Асинхрон онцлог: Агентууд бие биенийхээ хүлээхгүй. Тэд үйл явдлыг publish хийж, өөрийн ажилдаа үргэлжлүүлнэ. Энэ нь системийг өргөжүүлэх боломжтой, найдвартай болгодог.

5.2 Техникийн нарийн ширийн зүйлс

5.2.1 Prompt загвар

Orchestrator Agent-ийн system prompt:

. :

1.

2.

3.

:

{service_list}

:

{business_rules}

5.2.2 *Service Registry*

Үйлчилгээ бүрийг JSON форматаар тодорхойлно:

```
{
  "name": "user-service",
  "description": "      ␣    ␣    ␣     ",
  "endpoints": [
    {
      "path": "/users/{id}",
      "method": "GET",
      "description": "      ␣    ␣  ",
      "parameters": {
        "id": "      ␣ID"
      }
    }
  ]
}
```

Код 5.1: Service Registry Example

5.2.3 *RAG Систем*

Knowledge Agent нь vector database ашиглана:

- **Index**: API баримт бичиг, өмнөх хүсэлтүүд, бизнес дүрэм

- **Embedding Model**: Sentence-Transformers эсвэл OpenAI embeddings
- **Vector DB**: FAISS, Pinecone, эсвэл Qdrant
- **Retrieval**: Асуулт бүрт top-k хамгийн холбогдолтой баримтуудыг олох

5.2.4 Алдаа засалт

Алдаа гарах үед агент:

1. Алдааны төрөл, шалтгааныг тодорхойлно
2. Retry стратеги сонгоно (exponential backoff)
3. Альтернатив үйлчилгээ байгаа эсэхийг шалгана
4. Хэрэглэгчид ойлгомжтой алдааны мэдэгдэл өгнө
5. Алдааны логийг хадгална

5.3 Санал болгож буй загварын давуу тал

5.3.1 ХОУ агентын онцлогууд

1. **Натурал хэл интерфейс**: Хэрэглэгч натурал хэл дээр хүсэлт илгээж болно
2. **Уян хатан орчуулалт**: Бизнес дүрэм өөрчлөгдөхөд код өөрчлөх шаардлагагүй
3. **Ухаалаг алдаа засалт**: Агент өөрөө алдааг таньж, засах арга санал болгоно
4. **Түүх санах**: Өмнөх харилцааг санаж, контекст хадгална
5. **Өгөгдлийн нэгтгэл**: Олон үйлчилгээний өгөгдлийг нэгтгэж ойлгомжтой болгоно
6. **Өөрөө сайжрах**: Логуудаас суралцаж, цаашид илүү сайн ажиллана

5.3.2 Үйл явдлаар удирдагдах давуу тал

1. **Салангид байдал (Decoupling)**:
    - Агентууд хоорондоо шууд хамааралгүй
    - Нэг агент унах нь бусдад шууд нөлөөлөхгүй
    - Агентыг өөрчлөх, солих хялбар
2. **Өргөжүүлэх чадвар (Scalability)**:
    - Агент бүр бие даан өргөжиж болно
    - Kafka partition ашиглан параллель боловсруулалт
    - Consumer group-оор ачааллыг хуваарилах
3. **Найдвартай байдал (Resilience)**:
    - Kafka-ийн replication алдагдал хамгаална
    - Үйл явдал дахин тоглуулах боломжтой
    - Агент тасалдах ч үйл явдал алдагдахгүй
4. **Бодит цагийн хариу үйлдэл**:
    - Үйл явдал тэр даруйдаа дамждаг
    - Streaming боловсруулалт
    - Бага хоцрогдол
5. **Олон хэрэглэгч дэмжлэг**:
    - Нэг үйл явдлыг олон систем ашиглаж болно
    - CRM, CDP, analytics руу автоматаар урсах
    - Шинэ consumer нэмэхэд producer өөрчлөх шаардлагагүй
6. **Observability ба Debugging**:

- Бүх үйл явдал хадгалагдана
- Lineage tracking: үйл явдал хаашаа явсныг мэдэх
- Үйл явдал дахин тоглуулж bug олох
- Дахин сургалтад ашиглах

6. ТУРШИЛТ БА КОДЫН ЖИШЭЭ

6.1 Прототип систем

Санал болгосон загварыг туршихын тулд энгийн e-commerce системийн прототип бүтээсэн. Систем нь дараах микросервисүүдтэй:

- User Service: Хэрэглэгчийн бүртгэл, нэвтрэлт
- Product Service: Бүтээгдэхүүний мэдээлэл
- Order Service: Захиалга удирдлага
- Payment Service: Төлбөр боловсруулалт
- Notification Service: Мэдэгдэл илгээх

6.2 Orchestrator Agent хэрэгжүүлэлт

6.2.1 Үндсэн бүтэц

```python
import openai
from typing import List, Dict
import json

class OrchestratorAgent:
    def __init__(self, service_registry: Dict,
                 knowledge_base: KnowledgeBase):
        self.service_registry = service_registry
        self.knowledge_base = knowledge_base
        self.llm_client = openai.OpenAI()

    def process_request(self, user_query: str) -> str:
```

```
        # 1. Knowledge retrieval
        relevant_docs = self.knowledge_base.retrieve(
            user_query,
            top_k=5
        )

        # 2. Build context
        context = self._build_context(
            user_query,
            relevant_docs
        )

        # 3. Generate plan
        plan = self._generate_plan(context)

        # 4. Execute plan
        results = self._execute_plan(plan)

        # 5. Generate response
        response = self._generate_response(
            user_query,
            results
        )

        return response

    def _build_context(self, query: str,
                       docs: List[str]) -> str:
        services_desc = json.dumps(
            self.service_registry,
```

```
            indent=2
        )
        docs_text = "\n\n".join(docs)

        return f"""
         : {query}

         :
        {services_desc}

          :
        {docs_text}
        """

    def _generate_plan(self, context: str) -> Dict:
        system_prompt = """
           .

           .

          JSON         :
        {
          "steps": [
            {
              "service": "service-name",
              "action": "endpoint-path",
              "parameters": {...},
              "description":   "         "
            }
          ]
```

```
        }
        """

        response = self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": context}
            ],
            temperature=0.1
        )

        plan_json = response.choices[0].message.content
        return json.loads(plan_json)

    def _execute_plan(self, plan: Dict) -> List[Dict]:
        results = []

        for step in plan["steps"]:
            try:
                result = self._call_service(
                    step["service"],
                    step["action"],
                    step["parameters"]
                )
                results.append({
                    "step": step["description"],
                    "status": "success",
                    "data": result
                })
```

```
        except Exception as e:
            results.append({
                "step": step["description"],
                "status": "error",
                "error": str(e)
            })

            # Error recovery
            recovery_plan = self._handle_error(
                step,
                str(e)
            )
            if recovery_plan:
                results.extend(
                    self._execute_plan(recovery_plan)
                )

    return results

def _call_service(self, service: str,
                  action: str,
                  params: Dict) -> Dict:
    # Service agent
    service_agent = ServiceAgent(service)
    return service_agent.call(action, params)

def _handle_error(self, failed_step: Dict,
                  error: str) -> Dict:
    #
    # recovery
```

```
        error_context = f"""
            :
        {json.dumps(failed_step, indent=2)}

        : {error}


        None    .
        """

        # LLM     recovery plan
        # ...
        return None

    def _generate_response(self, query: str,
                           results: List[Dict]) -> str:
        results_summary = json.dumps(results, indent=2)

        response_prompt = f"""
         : {query}

          :
        {results_summary}

            .
        """

        response = self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[
```

```
                {"role": "user", "content": response_prompt}
            ],
            temperature=0.7
        )

        return response.choices[0].message.content
```

Код 6.1: Orchestrator Agent Implementation

6.2.2 Knowledge Base хэрэгжүүлэлт

```
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np
from typing import List

class KnowledgeBase:
    def __init__(self):
        self.model = SentenceTransformer(
            'paraphrase-multilingual-mpnet-base-v2'
        )
        self.documents = []
        self.embeddings = None
        self.index = None

    def add_documents(self, docs: List[str]):
        self.documents.extend(docs)

        # Generate embeddings
        new_embeddings = self.model.encode(docs)

```

```
        if self.embeddings is None:
            self.embeddings = new_embeddings
        else:
            self.embeddings = np.vstack([
                self.embeddings,
                new_embeddings
            ])

        # Build FAISS index
        dimension = self.embeddings.shape[1]
        self.index = faiss.IndexFlatL2(dimension)
        self.index.add(self.embeddings.astype('float32'))

    def retrieve(self, query: str,
                 top_k: int = 5) -> List[str]:
        # Query embedding
        query_embedding = self.model.encode([query])

        # Vector
        distances, indices = self.index.search(
            query_embedding.astype('float32'),
            top_k
        )

        #
        results = [
            self.documents[idx]
            for idx in indices[0]
        ]

```

```
        return results
```

Код 6.2: RAG Knowledge Base Implementation

6.2.3 Service Agent хэрэгжүүлэлт

```
import requests
from typing import Dict

class ServiceAgent:
    def __init__(self, service_name: str):
        self.service_name = service_name
        self.base_url = self._get_service_url()

    def _get_service_url(self) -> str:
        # Service discovery  - URL
        service_map = {
            "user-service": "http://localhost:8001",
            "product-service": "http://localhost:8002",
            "order-service": "http://localhost:8003",
            "payment-service": "http://localhost:8004",
        }
        return service_map.get(self.service_name)

    def call(self, endpoint: str,
             params: Dict) -> Dict:
        url = f"{self.base_url}{endpoint}"

        try:
            response = requests.post(
                url,
```

```
            json=params,
            timeout=10
        )
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        raise Exception(
            f"{self.service_name}␣  :␣{str(e)}"
        )
```

Код 6.3: Service Agent Implementation

6.3 Туршилтын үр дүн

6.3.1 Туршилтын тохиргоо

Прототип системийг дараах нөхцөлд туршсан:

- **Загвар**: GPT-4 (OpenAI API)
- **Embedding Model**: paraphrase-multilingual-mpnet-base-v2
- **Vector Database**: FAISS
- **Програмчлалын хэл**: Python 3.10
- **Микросервисүүд**: FastAPI framework ашигласан 5 микросервис

6.3.2 Туршилтын тохиолдлууд

Систем нь дараах тохиолдлуудыг амжилттай гүйцэтгэсэн:

Тохиолдол 1: Энгийн захиалга

Хэрэглэгчийн хүсэлт: ”Би утас захиалмаар байна. iPhone 15 Pro байгаа юу?”

Агентын төлөвлөгөө:

1. Product Service: Бүтээгдэхүүн хайх
2. Product Service: Үлдэгдэл шалгах
3. Order Service: Захиалга үүсгэх (хэрэв хэрэглэгч баталвал)

Үр дүн: Агент амжилттай бүтээгдэхүүнийг олж, үнэ болон үлдэгдлийн мэдээллийг хэрэглэгчид ойлгомжтой байдлаар хүргэсэн.

Тохиолдол 2: Нарийн төвөгтэй захиалга

Хэрэглэгчийн хүсэлт: ”Би өчигдөр iPhone 15 захиалсан. Хаягаа солих боломжтой юу?”

Агентын төлөвлөгөө:

1. User Service: Хэрэглэгчийн мэдээлэл авах
2. Order Service: Хэрэглэгчийн сүүлийн захиалгыг олох
3. Order Service: Захиалгын статус шалгах
4. Order Service: Хүргэлтийн хаяг шинэчлэх

Үр дүн: Агент амжилттай өмнөх захиалгыг олж, хаягийг шинэчилсэн.

Тохиолдол 3: Алдаа засалт

Хэрэглэгчийн хүсэлт: ”Та Samsung Galaxy S24-г санал болгож чадах уу?”

Тохиолдол: Product Service түр зуур унасан.

Агентын үйлдэл:

1. Анхны product service дуудлага амжилтгүй
2. Агент алдааг таньж, 2 секундын дараа дахин оролдсон
3. Хоёр дахь оролдлого амжилттай
4. Үр дүнг хэрэглэгчид хүргэсэн

Үр дүн: Агент алдааг амжилттай засч, хэрэглэгч алдаа гарснийг мэдэхгүй.

6.3.3 Гүйцэтгэлийн үнэлгээ

Туршилтын үр дүнгээс:

Table 6.1: Агент суурилсан системийн гүйцэтгэл

Үзүүлэлт	**Утга**
Дундаж хариултын хугацаа	2.3 секунд
Зөв төлөвлөгөө гаргах хувь	94%
Амжилттай гүйцэтгэл	89%
Алдаа засах чадвар	78%

6.3.4 Харьцуулалт

Уламжлалт хатуу кодлосон (hard-coded) орчуулагчтай харьцуулбал:

Table 6.2: Агент vs Уламжлалт систем

Үзүүлэлт	**Агент**	**Уламжлалт**
Код хэмжээ (мөр)	450	1,200
Шинэ үйлчилгээ нэмэх хугацаа	5 минут	2 цаг
Уян хатан байдал	Өндөр	Бага
Дундаж хариултын хугацаа	2.3с	0.8с
Натурал хэл дэмжлэг	Тийм	Үгүй

6.3.5 Олдсон асуудлууд

Туршилтын явцад дараах асуудлуудыг олсон:

1. **Хоцрогдол**: LLM дуудлага нь нэмэлт хоцрогдол авчирдаг. Энэ нь prompt кэш, streaming response ашиглан сайжруулж болно.

2. **Өртөг**: GPT-4 API үнэтэй. Энгийн даалгавруудад илүү хямд загвар (GPT-3.5) ашиглаж болно.

3. **Тогтворгүй байдал**: Ховорхон тохиолдолд агент буруу төлөвлөгөө гаргадаг. Few-shot examples нэмж сайжруулж болно.

4. **Хэт ерөнхий төлөвлөгөө**: Заримдаа агент хэтэрхий их үйлчилгээ дуудах төлөвлөгөө гаргадаг.

6.4 Хэлэлцүүлэг

6.4.1 Судалгааны үр дүн

Энэхүү судалгааны ажил нь ХОУ агентуудыг микросервис архитектурт амжилттай нэвтрүүлж болохыг харуулсан. Гол үр дүнгүүд:

1. ХОУ агентууд нь микросервис хоорондын нарийн төвөгтэй логикийг удирдаж чадна
2. RAG систем нь олон үйлчилгээний мэдээллийг нэгтгэхэд үр дүнтэй
3. Натурал хэл интерфейс нь хэрэглэгчийн туршлагыг сайжруулна
4. Агент нь алдаа засалтын чадвартай
5. Систем нь уян хатан, өргөжүүлэх боломжтой

6.4.2 Хязгаарлалтууд

Санал болгосон загвар нь дараах хязгаарлалтуудтай:

1. **Хоцрогдол**: LLM inference хоцрогдолтой. Бодит цагийн шаардлагатай системд тохиромжгүй.
2. **Өртөг**: API дуудлага бүр өртөгтэй. Өндөр ачаалалтай системд үнэтэй.
3. **Найдвартай байдал**: LLM-ийн найдвартай байдал 100% биш. Чухал системд баталгаат тохиргоо шаардлагатай.
4. **Аюулгүй байдал**: Prompt injection довтолгооноос хамгаалах шаардлагатай.

6.4.3 Ирээдүйн судалгаа

Цаашид дараах чиглэлээр судалгааг үргэлжлүүлж болно:

1. **Хоцрогдол бууруулах**: Жижиг, хурдан загварууд (Llama 3, Mistral) туршиж үзэх
2. **Finetuning**: Тодорхой domain-д зориулж загвар нарийвчлан сургах
3. **Олон агентын систем**: Агентууд хоорондоо хамтран ажиллах
4. **Өөрөө сайжрах**: Логуудаас суралцаж агентыг автоматаар сайжруулах
5. **Аюулгүй байдал**: Prompt injection, алдаа засах механизмыг сайжруулах
6. **Том хэмжээний туршилт**: Илүү олон микросервис, илүү их ачаалал дээр турших

Дүгнэлт

Энэхүү судалгааны ажил нь ХОУ агентуудыг микросервис архитектурт нэвтрүүлэх боломжийг судалж, практик загвар санал болгосон. Судалгааны явцад дараах гол үр дүнд хүрсэн:

Онолын хувьд:

ХОУ-н инженерчлэл нь програм хангамж хөгжүүлэлтийн шинэ салбар болж хөгжиж байгаа нь тодорхой. Суурь загваруудын гарч ирэх нь аппликейшн хөгжүүлэлтийн саад бэрхшээлийг эрс багасгасан. Хэл загваруудаас том хэлний загвар, цаашлаад суурь загвар руу хөгжих үйл явц нь хэдэн арван жилийн технологийн дэвшлийн үр дүн юм. Өөрийгөө удирдсан сургалт, Transformer архитектур, post-training аргууд зэрэг гол түлхүүр цэгүүд нь өнөөгийн хүчирхэг ХОУ системүүдийг бий болгосон.

ХОУ агентуудын онол нь орчин, үйлдэл, даалгавар гэсэн гурван гол бүрэлдэхүүн дээр суурилдаг. Агентууд нь төлөвлөлт, багаж ашиглалт, эргэцүүлэн бодох чадвартайгаараа уламжлалт програмуудаас давуу талтай. RAG систем нь агентуудын мэдлэгийг өргөтгөж, илүү найдвартай, бодит мэдээлэл дээр суурилсан хариулт өгөх боломжийг олгодог.

Практик хувьд:

Микросервис архитектур дахь үйлчилгээ хоорондын нарийн төвөгтэй логик, өгөгдлийн нэгтгэл, динамик routing зэрэг асуудлуудыг ХОУ агентууд ашиглан шийдэж болох нь тодорхой болсон. Санал болгосон Orchestrator Agent, Service Agent, Knowledge Agent, Monitoring Agent зэрэг бүрэлдэхүүн хэсгүүд нь уян хатан, өргөжүүлэх боломжтой системийг бий болгодог.

Гэхдээ агентуудыг үйлдвэрлэлийн түвшинд өргөжүүлэхийн тулд зөвхөн агентын ур чадвар биш, тэдгээрийг холбодог архитектур чухал гэдгийг ойлгосон. Монолит болон RPC/API суурилсан холболт нь монолит архитектурт тулгарсан асуудалтай адил саад

болдог. Үүнийг шийдэхийн тулд үйл явдлаар удирдагдах архитектур (EDA) ашиглан агентуудыг салангид микросервис болгон хөгжүүлэх шаардлагатай.

Үйл явдлаар удирдагдах архитектурын үр дүн:

Apache Kafka болон Apache Flink ашиглан үйл явдлаар удирдагдах агентын систем бүтээх нь:

- **Салангид байдал**: Kafka topics-оор харилцах нь агентууд хоорондоо шууд хамааралгүй байхыг баталгаажуулна
- **Параллель боловсруулалт**: Kafka partitions ашиглан олон агент зэрэг ажиллаж, системийн дамжуулалтыг нэмэгдүүлнэ
- **Найдвартай байдал**: Үйл явдлын хадгалалт нь мэдээлэл алдагдахгүй, дахин тоглуулж болохыг баталгаажуулна
- **Водит цагийн боловсруулалт**: Streaming архитектур нь тэр даруй хариу үйлдэл үзүүлэх боломжийг олгоно
- **Олон хэрэглэгч**: Агентын гаралт нь CRM, CDP, analytics зэрэг олон системд автоматаар урсах боломжтой

Flink AI Inference ашиглах нь төлөвлөгч агентыг stream processing app болгон хөгжүүлэх боломжийг олгож, Flink-ийн stateful боловсруулалт, exactly-once семантик зэрэг давуу талыг ашиглах боломжтой болгоно.

Прототип системийн туршилт нь 94% зөв төлөвлөгөө гаргах, 89% амжилттай гүйцэтгэл, 78% алдаа засах чадварыг харуулсан. Энэ нь уламжлалт hard-coded орчуулагчтай харьцуулахад код хэмжээг 62% бууруулж, шинэ үйлчилгээ нэмэх хугацааг 24 дахин богиносгосон. Гэвч дундаж хариултын хугацаа 2.9 дахин удаан болсон нь анхаарал хандуулах асуудал юм.

Цаашдын хөгжил:

Хоцрогдол болон өртгийн асуудлыг шийдэхийн тулд дараах чиглэлүүд чухал:

- Жижиг, хурдан загварууд (Llama 3, Mistral, Gemma) туршиж ашиглах

- Тодорхой domain-д зориулж загвар нарийвчлан сургах
- Prompt кэш, streaming response ашиглан хоцрогдол бууруулах
- Олон агентын систем бүтээж илүү нарийн төвөгтэй даалгавар гүйцэтгүүлэх

Судалгааны ач холбогдол:

Энэхүү судалгаа нь ХОУ агентууд болон микросервис архитектурын уялдааг судалсан анхны судалгаануудын нэг юм. Chip Huyen-ийн ”AI Engineering” номонд дурдагдсан онол, практик аргуудыг Монгол хэл дээр нэгтгэн авч үзэж, микросервис орчинд хэрэглэх загвар санал болгосон нь энэхүү ажлын гол хувь нэмэр юм.

Эцсийн дүгнэлт:

ХОУ-н инженерчлэлийн эрин үе эхэлж байна. Том хэмжээний хөрөнгө оруулалт, хурдацтай технологийн хөгжил, өргөн хүрээний хэрэглээ зэрэг нь энэ салбарын ирээдүй гэрэлтэй байгааг харуулж байна. Гэвч ХОУ агентуудын жинхэнэ боломжийг нээхийн тулд зөв архитектурын шийдэл чухал юм.

Энэхүү судалгаа нь ХОУ агентууд зөвхөн ухаалаг програм биш, үндсэндээ өргөжүүлэх боломжтой микросервисүүд болохыг харууллаа. Гэхдээ уламжлалт микросервисээс ялгаатай нь агентууд нь контекст баялаг, хуваалцсан мэдээлэлд найддаг. Энэ нь тэднийг үр дүнтэй холбохын тулд үйл явдлаар удирдагдах архитектур (EDA) шаардлагатай болгодог.

HubSpot-ийн CTO Dharmesh Shah-ийн хэлснээр ”Агентууд бол шинэ апп юм.” Хэрэв энэ нь үнэн бол агентуудыг микросервис болгон хөгжүүлэх хэрэгтэй. Хэрэв та агентуудыг үйл явдлаар харилцуулахгүй бол тэд зүгээр л LLM тархитай, хуучирч болзошгүй программууд хэвээр үлдэнэ.

Apache Kafka болон Apache Flink зэрэг технологиудыг ашиглан агентууд нь:

- Бие даан өргөжиж чадна
- Бодит цагт өгөгдөл боловсруулна
- Системийн хэмжээнд мэдээллээ хуваалцана
- Алдаанаас сэргэж чадна

- Дахин тоглуулах замаар сайжирна

Микросервис архитектурт ХОУ агентууд нэвтрүүлэх нь системийг илүү ухаалаг, уян хатан, хэрэглэгчид ээлтэй болгох боломжийг олгоно. Хэдийгээр одоогоор хязгаарлалтууд байгаа ч - хоцрогдол, өртөг, найдвартай байдал - технологи хурдацтай хөгжиж байгаа тул ойрын ирээдүйд эдгээр асуудлууд шийдэгдэх болно гэж найдаж байна.

Энэхүү дипломын ажил нь ХОУ-н инженерчлэлийн үндсийг тавьж, микросервис архитектурт агент суурилсан шийдлийг практикт хэрхэн хэрэглэж болохыг харуулсан. Хамгийн чухал нь, үйл явдлаар удирдагдах архитектур нь агентуудыг үйлдвэрлэлийн түвшинд өргөжүүлэх цорын ганц зам болохыг тогтоосон. Цаашдын судалгаа, хөгжүүлэлтээр энэ чиглэлийг улам бүр боловсронгуй болгож, бодит системүүдэд нэвтрүүлэх боломжтой гэж үзэж байна.

Bibliography

[1] Huyen, Chip. *AI Engineering*. O'Reilly Media, 2024.

[2] Goldman Sachs Research. "Generative AI Could Raise Global GDP by 7%", 2023. `https://www.goldmansachs.com/intelligence/pages/generative-ai-could-raise-global-gdp-by-7-percent.html`

[3] Vaswani, A., et al. "Attention Is All You Need". *Advances in Neural Information Processing Systems*, 2017.

[4] Devlin, J., et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *NAACL-HLT*, 2019.

[5] Brown, T., et al. "Language Models are Few-Shot Learners". *Advances in Neural Information Processing Systems*, 2020.

[6] Touvron, H., et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models". *arXiv preprint arXiv:2307.09288*, 2023.

[7] Touvron, H., et al. "The Llama 3 Herd of Models". *arXiv preprint arXiv:2407.21783*, 2024.

[8] Lewis, P., et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". *Advances in Neural Information Processing Systems*, 2020.

[9] Gao, Y., et al. "Retrieval-Augmented Generation for Large Language Models: A Survey". *arXiv preprint arXiv:2312.10997*, 2023.

[10] Wei, J., et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". *Advances in Neural Information Processing Systems*, 2022.

[11] Yao, S., et al. "ReAct: Synergizing Reasoning and Acting in Language Models". *ICLR*, 2023.

[12] Schick, T., et al. "Toolformer: Language Models Can Teach Themselves to Use Tools". *arXiv preprint arXiv:2302.04761*, 2023.

[13] Ortega, P.A., et al. "Shaping Representations Through Communication: Community Size and Stability Determine the Emergence of Shared Symbols". *DeepMind Technical Report*, 2021.

[14] Schulman, J. "Reinforcement Learning from Human Feedback: Progress and Challenges". *Berkeley EECS Colloquium*, 2022.

[15] OpenAI. "Prompt Engineering Guide", 2023. `https://platform.openai.com/docs/guides/prompt-engineering`

[16] Liu, P., et al. ”Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. *ACM Computing Surveys*, 2023.

[17] Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2015.

[18] Richardson, Chris. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[19] Fowler, Martin and Lewis, James. ”Microservices: A Definition of This New Architectural Term”, 2014. `https://martinfowler.com/articles/microservices.html`

[20] Johnson, J., Douze, M., and Jégou, H. ”Billion-scale Similarity Search with GPUs”. *IEEE Transactions on Big Data*, 2019.

[21] Malkov, Y.A. and Yashunin, D.A. ”Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[22] Reimers, N. and Gurevych, I. ”Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. *EMNLP-IJCNLP*, 2019.

[23] Muennighoff, N., et al. ”MTEB: Massive Text Embedding Benchmark”. *EACL*, 2023.

[24] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[25] Python Software Foundation. ”Python 3 Documentation”. `https://docs.python.org/3/`

[26] Ramírez, Sebastián. ”FastAPI Documentation”. `https://fastapi.tiangolo.com/`

[27] OpenAI. ”OpenAI API Reference”. `https://platform.openai.com/docs/api-reference`

[28] Anthropic. ”Claude 3 Model Card”, 2024. `https://www.anthropic.com/claude`

[29] LangChain. ”LangChain Documentation”. `https://python.langchain.com/docs/`

[30] Facebook AI Research. ”FAISS: A Library for Efficient Similarity Search”. `https://github.com/facebookresearch/faiss`

[31] Falconer, Sean. ”AI Agents are Microservices with Brains”. Medium, March 2025. `https://medium.com/@seanfalconer`

[32] Falconer, Sean. ”The Future of AI Agents is Event-Driven”. BigDataWire, March 2025.

[33] Polak, Adi. ”Building AI Agents with Event-Driven Microservices”. Confluent Developer Advocate, 2025.

[34] Apache Kafka Documentation. ”Apache Kafka: A Distributed Streaming Platform”. `https://kafka.apache.org/documentation/`

[35] Kreps, Jay, Narkhede, Neha, and Rao, Jun. ”Kafka: A Distributed Messaging System for Log Processing”. *Proceedings of the NetDB*, 2011.

[36] Apache Flink Documentation. ”Stateful Computations over Data Streams”. `https://flink.apache.org/`

[37] Carbone, Paris, et al. ”State Management in Apache Flink: Consistent Stateful Distributed Processing”. *Proceedings of the VLDB Endowment*, 2017.

[38] Wooldridge, Michael. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009.

[39] Park, Joon Sung, et al. ”Generative Agents: Interactive Simulacra of Human Behavior”. *arXiv preprint arXiv:2304.03442*, 2023.

[40] Wolff, Eberhard. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[41] Nadareishvili, Irakli, et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O’Reilly Media, 2016.

[42] Anthropic. ”Model Context Protocol: A Universal Standard for AI Integration”, 2024. `https://www.anthropic.com/news/model-context-protocol`

А. НҮҮР ХУУДАС