Budapesti Műszaki és Gazdaságtudományi Egyetem
Mesterséges Intelligencia és Rendszertervezés Tanszék

LLM-vezérelt modellpéldány-generálás Refinery-alapú formális validációs kontrollal

BSc Önálló laboratórium Dokumentáció 2025/26. I. félév

Terék Zoltán Tamás (DKLNXS)

Mérnökinformatikus szakos hallgató
BSc Szoftverfejlesztés specializáció

Konzulens:
Semeráth Oszkár
(docens)
(Mesterséges Intelligencia és Rendszertervezés Tanszék)

Tartalomjegyzék

1. Bevezetés

A mai világban rendkívül népszerű lett a nagy nyelvi modellek (LLM-ek) használata. Kiemelkedő képességeket mutattak természetes nyelvű leírások értelmezésében és strukturált tartalmak generálásában. Ezek a modellek implicit módon jelentős mennyiségű háttérismerettel rendelkeznek, ami pedig vonzóvá teszi őket komplex rendszerek tervezésének támogatására, például akár modellezési feladatok során is. Ugyanakkor ma már közismert fogalom nagy nyelvi modelleknél a hallucináció, amikor magabiztosan ténybelileg hibás kimenetet tudnak adni. A „nem-determinisztikus" generálás, valamint a formális szemantika hiánya miatt az általuk előállított struktúrák önmagukban nem tekinthetők megbízhatónak kritikus vagy szabály vezérelt környezetekben. Különösen problémás ez modellalapú rendszertervezés eseteknél, hiszen a helyesség, konzisztencia és a teljesség alapvető követelmény a működéshez. Például az LLM által generált modellpéldány megsérthet olyan formális követelményeket, amelyek később nehezen visszavezethető hibákhoz vezethetnek. Ez azt mutatja, hogy az LLM-ek önmagukban bármennyire is fejlettek és kompetensek, sosem lesznek alkalmasak formális modellek megbízható előállítására, mivel működésük természetéből fakadóan a hallucináció kockázata nem zárható ki teljes mértékben. Ugyanakkor megfelelő szabályozási megoldásokkal kombinálva értékes komponenssé válhatnak.

Az önálló laborom célja egy olyan architektúra megtervezése és megvalósítása volt, amelyben az LLM nem önálló döntéshozóként működik, hanem a modellalkotási folyamatot támogató javaslatképző szerepet tölt be. A létrejövő modellpéldányok helyességét és konzisztenciáját egy formális, gráf-alapú solver, a Refinery ellenőrzi. A megközelítés alapja a felelősségek tudatos szétválasztása: míg az LLM a természetes nyelvű leírásokból kinyert háttérismeretet strukturált állítások formájában adja vissza, addig a Refinery formális szabályok és megszorítások alapján dönt arról, hogy ezek az állítások elfogadhatók-e. Az OnlabRefinery egy iteratív folyamatot valósít meg, ahol az LLM által javasolt kiegészítéseket a Refinery formálisan ellenőrzi. Sikertelen ellenőrzés esetén célzott visszajelzés alapján történik a javítás, így a végső kimenet a rögzített követelményeknek megfelelő modellt ad.

A megközelítés bemutatása főként egy egyszerű, mégis szemléletes videójátékos példán keresztül történik, ahol egy formálisan leírt világállapotot kell az LLM-nek kiegészítenie entitások, entitások és kapcsolatok figyelembevételével. Bár a példa szándékosan egyszerű, a mögöttes gondolatmenet jól általánosítható más területekre is, például termékkatalógusokra, konfigurációs rendszerekre vagy üzleti folyamatmodellekre. A dolgozat célja nem egy univerzális megoldás bemutatása, hanem egy olyan általános minta felvázolása, amelyben a természetes nyelvű tervezés és a formális modellalkotás ellenőrzött, megbízható módon kapcsolható össze.

2. Kontrollált LLM-alapú modellgenerálás architektúrája

Ebben a fejezetben a megvalósított (OnlabRefinery) projekt technikai felépítését és működését mutatom be. A célom nem az, hogy az LLM-ek általános képességeit újra felsoroljam, hanem hogy megmutassam, modellezési helyzetben milyen bizonytalanságok jönnek elő, ezek miért gondok formális környezetben, és milyen felépítéssel lehet úgy használni az LLM-et, hogy közben a végeredmény megbízhatósága kézben maradjon.

2.1. Az LLM-alapú modellezés kihívásai

Modellezési feladatokban a legnagyobb nehézség nem az, hogy az LLM néha téved, hanem az, hogy a hiba gyakran nem nyilvánvaló. Egy szöveges válasznál sokszor azonnal látszik, ha valami nem stimmel, viszont egy formális modellpéldány esetén a hibák könnyen háttérben maradhatnak, ha nem figyelünk. Például a modell első ránézésre strukturáltnak tűnhet, mégis olyan állapotokat írhat le, amelyek formálisan értelmezhetetlenek vagy nem felelnek meg a rendszer belső szabályainak.
A „nem-determinisztikus" működés itt különösen kellemetlen. Ugyanarra a bemenetre az LLM nem feltétlenül ad ugyanolyan eredményt, ami a modellépítés folyamatában azt jelenti, hogy két futtatás során eltérő modellpéldányok születhetnek. Ez fejlesztési szemszögből több problémát tud okozni, mert nehezebb reprodukálni egy adott eredményt, nehezebb megérteni, miért romlott el egy korábban még jó megoldás, és nehezebb stabilan építeni egy olyan pipeline-t/rendszert, ahol a modellkimenet a későbbi lépések bemenete.
A hallucináció modellalapú környezetben tipikusan nem úgy történik, hogy egy látványos összeomlást vagy hibát okoz, hanem úgy, hogy csendben bekerülnek téves elemek. Például a modell tartalmazhat olyan entitásokat vagy kapcsolatokat, amelyek a leírás alapján logikusnak tűnnek, de a formális szabályok szerint nem megengedettek. A gond ezzel az, hogy az LLM meggyőző stílusa miatt a hibák meggyőzőnek látszanak, tehát nem feltétlenül buknak le azonnal emberi ellenőrzéssel.
Egy további alapvető különbség, amit megemlítenék, a szintaktika és a szemantika között van. Sok esetben könnyű elérni, hogy az LLM egy strukturált formátumot adjon (pl. JSON-t), de ettől még nem lesz garantált, hogy a tartalom megfelel a formális jelentésnek. Vagyis hiába néz ki jól a kimenet, attól még sérthet olyan követelményeket, amelyek a modell működőképességéhez szükségesek. Ezért egy igazán megbízható megoldásnál nem elég a kimeneti forma ellenőrzése, hanem a modell jelentését és belső konzisztenciáját is vizsgálni kell.

Összességében tehát a kihívás nem az, hogy az LLM-et „jó válaszra” bírjuk, hanem az, hogy a generált eredmény megfeleljen egy adott formális környezet szabályainak. Itt jön be a Refinery szerepe, ami egy olyan formális réteget ad a rendszerhez, amelyben a modell egy solverrel és további automatikus feldolgozó lépésekkel együtt garantáltan konzisztensen és ellenőrizhetően viselkedik.

2.2. Felelősségek szétválasztása: LLM és Refinery

A megvalósított megközelítés egyik központi döntése, hogy az LLM nem a végső szó a folyamatban, hanem egy olyan komponens, amely hasznos javaslatokat ad a modell kiegészítésére. Ez azért fontos, mert a modellezési folyamatban nagy kockázat, ha ugyanaz a komponens egyszerre generál, ellenőriz és véglegesít is. Már taglaltam bővebben az előző pontokban, hogy LLM-eknél ez különösen problémás, mivel nincs olyan beépített garancia, ami minden esetben kikényszerítené a belső szabályok következetes betartását.

Ezzel szemben a Refinery szerepe éppen az, amire egy formális rendszernek jónak kell lennie, explicit szabályok alapján, determinisztikusan eldönti, hogy egy adott modellpéldány elfogadható-e. A felelősségek szétválasztása így lényegében azt jelenti, hogy:

- az LLM „kitalál" jelölteket a modell kiegészítésére,
- a Refinery pedig „eldönti", hogy ezek a jelöltek kompatibilisek-e a formális szabályokkal.

Fontos, hogy a Refinery nem csak „utólagos ellenőrző", hanem a folyamat aktív része. Az érvénytelen kimenetet nem egyszerűen eldobja, hanem olyan hiba-információt ad, ami visszacsatolásként használható. Ez a visszacsatolás teszi lehetővé, hogy a generálás ne egy egyszeri alkalmas folyamat legyen, hanem egy lépésenként javuló iteráció, ahol a rendszer fokozatosan konvergál egy elfogadható modellhez.

Ebből a nézőpontból a Refinery tényleg egy „minőségkapuként" szolgál, és ez az elnevezés lényegében meghatározza a rendszer működését. Ha ez a kapu nem enged át hibás modellt, akkor a kimenet megbízhatósága nem azon múlik, hogy az LLM épp mennyire talált el mindent, hanem azon, hogy a formális szabályoknak tényleg megfelel-e. Ez adja a rendszer egyik legfontosabb értékét, a generálás lehet kreatív, de az eredmény csak akkor fogadható el, ha formálisan is rendben van.

2.3. Rétegezett modellfelépítés és technológiai megvalósítás

Ahhoz, hogy az LLM ne szerkessze szét a teljes modellt, a rendszerben célszerű volt pontosan kijelölni, hogy mi az, ami stabil és nem változhat, és mi az, amit a generáló komponens módosíthat. Ezért a megvalósításhoz rétegezett modellfelépítést használtam. A rétegek lényege, hogy a modellnek van egy olyan része, amely a magas szintű kereteket és a stabil szabályokat adja, és van egy olyan része, ahol a konkrét modellpéldányt lehet bővíteni. Továbbá fejlesztés közben is átláthatóbbá tette a projektet.

A megoldásban ez a rétegezés a forrás összeállításának módjában is megjelenik. A design réteg adja a magas szintű kontextust és kereteket, a model réteg írja le a konkrét sémát és szabályokat, míg a runtime réteg a futás során releváns kiegészítő információkat és állapotot tartalmazza. A bemutatott példákban a runtime réteg szerepe kisebb, mivel nem egy folyamatosan futó alkalmazásról volt szó, ugyanakkor a réteg elkülönítése előkészíti azt, hogy a megközelítés később állapottal rendelkező, dinamikusabb esetek felé is bővíthető legyen.

A gyakorlatban ez azt jelenti, hogy az LLM nem nyúlhat bele mindenbe, hanem egy kijelölt kimeneti területet kap. Azokat az állításokat adja hozzá, amelyek a modellpéldányt bővítik. Ez azért hasznos, mert ha validációs hiba történik, akkor a javítás is fókuszált marad, és nem kell attól tartani, hogy a rendszer alapvető szabályai vagy a már stabil részek véletlenül megváltoznak.

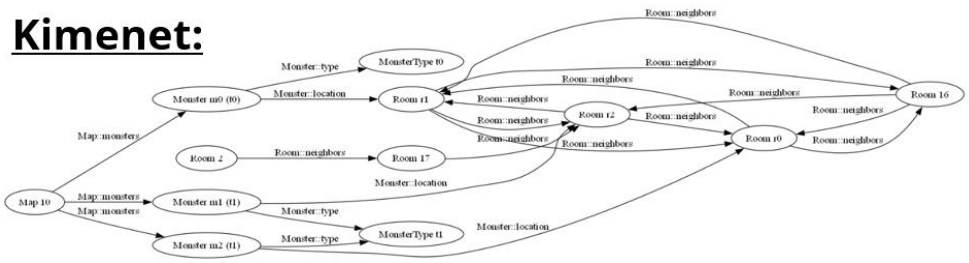
Technológiai oldalról a rendszer egy Java 21-es, konzolos alkalmazásként készült, Gradle build rendszerrel. Ez tudatos döntés volt, mivel hasonló LLM-es prototípusok gyakran JavaScript környezetben készülnek, én viszont a saját eszközkészletemhez és preferenciáimhoz jobban illeszkedő Java vonalon akartam ezt megvalósítani. A konzolos felépítés praktikus, mert támogatja a reprodukálható futtatást, a naplózhatóságot, és jól illik egy pipeline-szerű működéshez.

Az LLM-hívás OpenRouteren keresztül történik, ami a gyakorlatban azt jelenti, hogy a háttérben használt modell könnyen cserélhető, illetve a limit beállítása és kézenfekvőbb. A projekt során nem a lehető legerősebb modellt használtam, hanem egy kisebb, költséghatékonyabb változatot(chatgpt 4o-mini), mert a hangsúly nem a nyers generálási képességen volt, hanem azon, hogy a köré épített ellenőrzési és visszacsatolási logika mennyire stabilan tudja formális irányba terelni a kimenetet.

Az LLM válasza JSON-alapú formában érkezik, mert így a kimenet automatikusan feldolgozható és könnyebben integrálható a további lépésekbe. Ugyanakkor nem feltételezem, hogy a modell minden esetben hibátlanul formázott adatot ad vissza, ezért a rendszerben van egy egyszerű, hibatűrő értelmezési réteg is, ami a tipikus strukturális hibákat minimális mértékben kezeli. Ez nem helyettesíti a formális validációt, inkább azt szolgálja, hogy a feldolgozás ne törjön el azonnal kisebb formázási eltéréseken.

A vizualizáció szintén fontos része a technológiai megvalósításnak. Egy modellezési folyamatnál sokat segít, ha a felhasználó nem csak szövegként látja az eredményt, hanem gráfként is. Ezért érdemes a modell állapotát DOT/Graphviz formátumba exportálni, és képként megjeleníteni. Ilyenkor nem az a cél, hogy minden részlet látszódjon, hanem hogy gyorsan ellenőrizhető legyen, hogy a struktúra nagyvonalakban azt adja-e, amit várunk.

Kimenet:

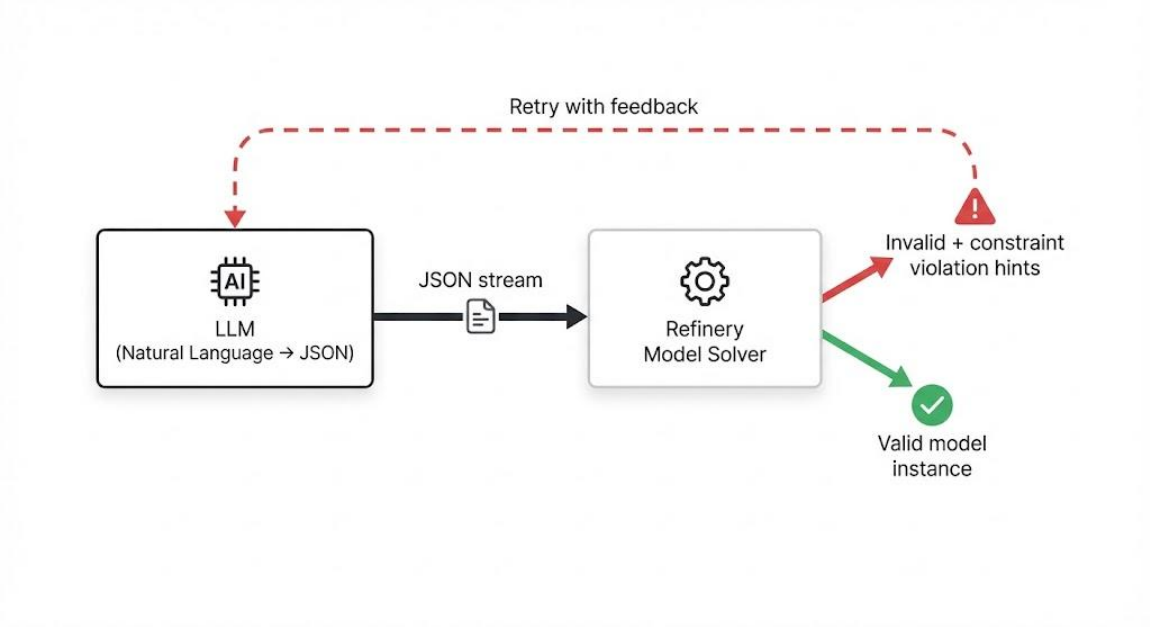



```
=== Assertions ===
MonsterType(troll).
strength(troll):15..20.
health(troll):15..20.

MonsterType(goblin).
strength(goblin):1..5.
health(goblin):1..5.

type(m0, troll).
type(m1, goblin).
type(m2, goblin).
```

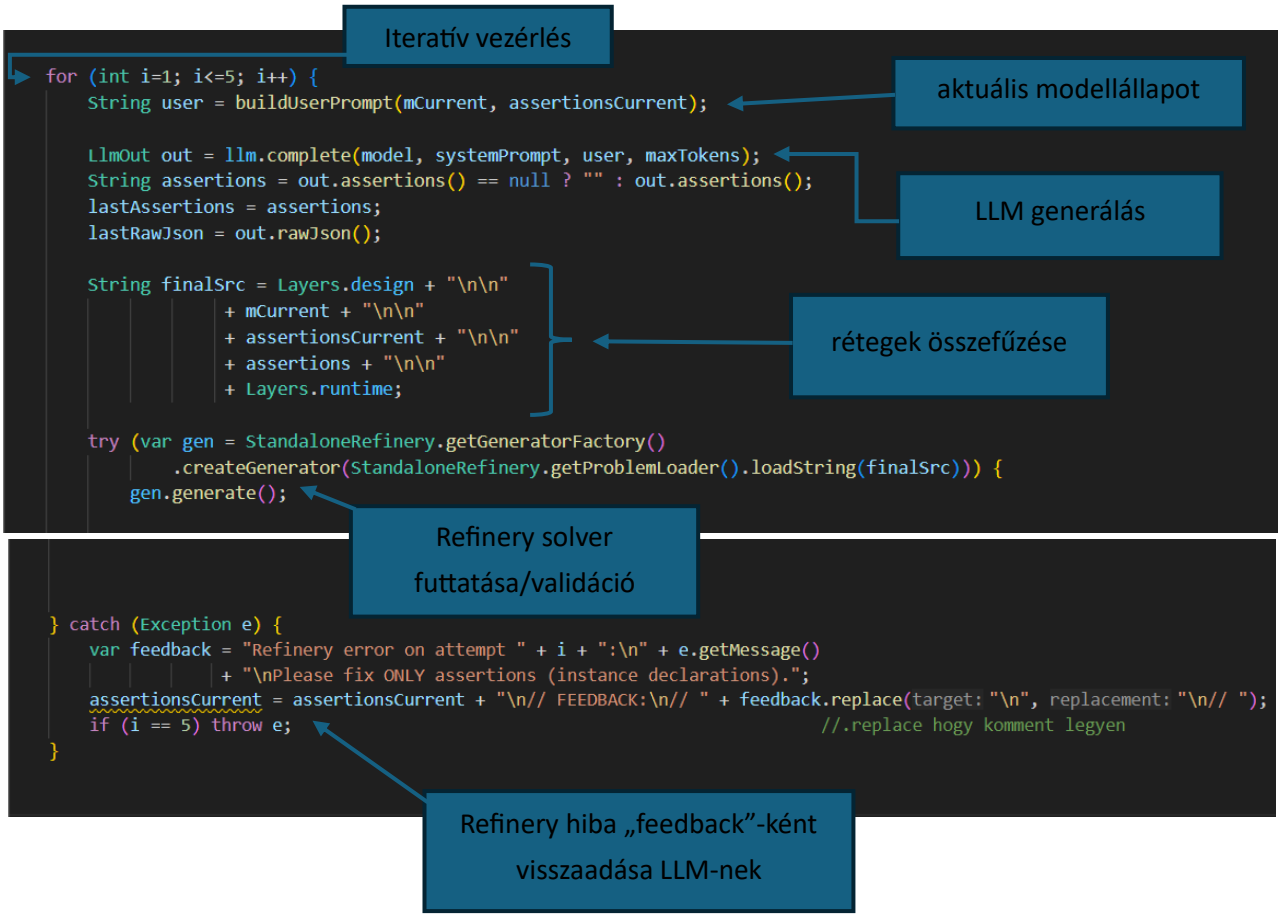
2.4. Iteratív vezérlési kör és visszacsatolás

A rendszer működését egy iteratív vezérlési kör írja le. A folyamat lényege, hogy az LLM nem egyszer próbálkozik, hanem több lépésben, fokozatosan közelít egy olyan modellpéldányhoz, amely a formális szabályoknak megfelel. Minden iterációban ugyanaz történik: az LLM javaslatot tesz a modell bővítésére, a rendszer összeállítja a teljes modellt a rétegek alapján, majd a Refinery ellenőrzi a kapott eredményt.

Ha a validáció sikeres, akkor a folyamat leáll, és a modell elfogadhatóként kerül ki a rendszerből. Ha viszont a validáció sikertelen, akkor a Refinery hibaüzeneteiből a rendszer egy célzott visszajelzést képez, amelyet az LLM megkap a következő próbálkozás előtt. A fontos megkötés itt az, hogy az LLM ilyenkor csak a kijelölt részt (az állításokat) módosíthatja. Vagyis a rendszer nem azt várja, hogy pl. újra írja az egészet, hanem hogy javítsa ki azt a konkrét részt, ami a formális ellenőrzés szerint hibás.



Ez a visszacsatolásos működés azért hasznos, mert az LLM-ek jellemzően jobban tudnak javítani egy konkrét, lokalizált hibát, mint elsőre tökéletesre megcsinálni egy modellt. Ez a feedback egy egyfajta korlátozó erő, mivel nem engedi, hogy a rendszer végtelenül elsodródjon, hanem mindig visszahúzza a formális követelmények irányába. Az iterációszám limitje pedig egy praktikus mérnöki korlát. Ha a modell túl nehéz vagy a helyzet túl bizonytalan, akkor a rendszer nem fut örökké, hanem kontrolláltan megáll. Tapasztalataim során 3-5 iteráció már bőven elegendőnek látszott, ami nagyrészt a példa egyszerűségének és az LLM kompetenciájának köszönhető.



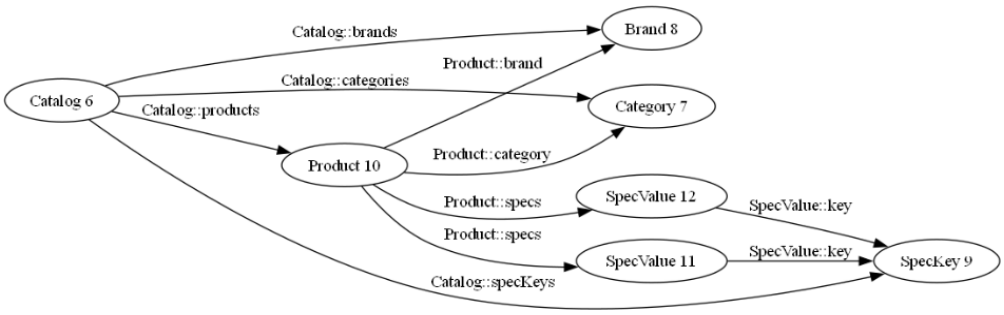
3. Modellalapú tervezés tágabb értelmezése

Az előző fejezetben a rendszer konkrét architektúráját és működését bontottam ki. A következőkben egy absztraktabb nézőpontot szeretnék felvenni, és az implementáció helyett azt a tervezési mintát emelem ki, amit ez a projekt megvalósít. A videójátékos példa részben preferencia volt, mert így a megvalósítás és a tesztelés is élvezhetőbb és szemléletesebb lett. A lényeg viszont nem maga a játék, hanem az, hogy természetes nyelvű leírásból hogyan lehet kontrollált módon formális modellt felépíteni úgy, hogy közben a generatív komponens bizonytalansága is kezelhető legyen.

3.1. Általánosíthatóság

A videójátékos keret azért működik jól demonstrációként, mert egyszerre egyszerű és jól vizualizálható. Könnyen megfoghatók benne az alap modellezési elemek, vannak entitások (például szereplők és tárgyak), vannak kapcsolatok (ki hol van, mi mivel van összekötve), és vannak olyan szabályok, amelyeknek a világállapotnak meg kell felelnie. Ugyanez a felépítés viszont nem játékhoz kötött, hanem sok valós problémában is megjelenik, ahol a rendszer állapotát és a rá vonatkozó megszorításokat formális módon szeretnénk leírni.

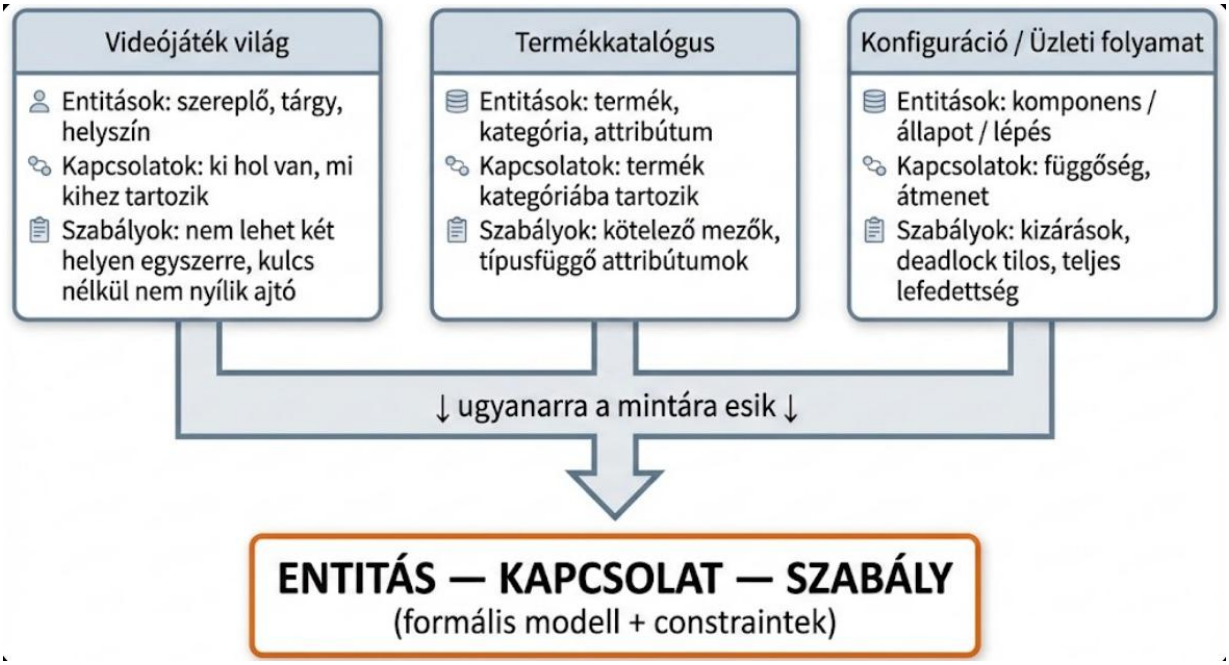
Egy kézenfekvő párhuzam például egy termékkatalógus. Ennél a példánál is van egy „világ”, amit entitások alkotnak (termékek, kategóriák, márkák stb.), és ezek között sok kapcsolat van (egy termék egy márkához tartozik, több kategóriában is szerepelhet, több paraméterrel írható le). Emellett itt is megjelennek formális követelmények, például bizonyos paraméterek kötelezőek, egy specifikáció csak adott típus mellett érvényes, vagy egy kategória csak adott attribútumokat enged. Ha az LLM ezeket elrontja, abból nagyon gyorsan lesznek hibás vagy hiányos adatok, később pedig üzleti logikai hibák is következhetnek.




```
=== Assertions ===
Category(ultrabook).
categories(cat1, ultrabook).

Brand(dell).
brands(cat1, dell).

SpecKey(cpu).
specKeys(cat1, cpu).

Product(p1).
products(cat1, p1).
category(p1, ultrabook).
brand(p1, dell).

SpecValue(sv1).
specs(p1, sv1).
key(sv1, cpu).

SpecValue(sv2).
specs(p1, sv2).
key(sv2, cpu).
```

Ezen katalógusos példa logikája kiterjeszthető konfigurációs rendszerekre is, amelyek szintén jó példát adnak hasonló kihívásokra. Ilyen esetekben elemekből kell összeállítani egy működő konfigurációt úgy, hogy közben kompatibilitási szabályok és kizáró feltételek is érvényesülnek, például bizonyos komponensek csak adott más komponensekkel használhatók együtt. Ezek tipikusan megszorításos jellegű problémák, ahol az egyszerű intuíció gyakran nem elég, mert a lehetséges kombinációk száma gyorsan megnő. Ilyen környezetben a generatív komponens hasznos lehet a jelölt megoldások javaslásában, de a végső ellenőrzést érdemes formális szabályokra bízni, különben a rendszer időnként hibás összeállításokat is ajánlhat.

Ugyanez a gondolat az üzleti folyamatmodellezés területén is jól alkalmazható. Ilyen esetekben a modell gyakran nem csupán adatokat ír le, hanem a rendszer működését is, például állapotokat, átmeneteket, kivételeket és jogosultságokat. Az LLM hatékonyan támogathatja azt, hogy egy természetes nyelvű folyamatleírásból kiindulva gyorsan megszülessen egy első strukturált modell, ugyanakkor a formális korlátok ellenőrzése nélkül könnyen maradhatnak benne rejtett hibák, például nem kezelt kivételek vagy hiányos állapotátmenetek.



(Google Nano Banana Pro)

A fenti példákat együtt nézve levonható, hogy a modell nem pusztán szemléltető eszköz, hanem egy működő rendszer alapját képezi. Ha a modell hibás, akkor vagy nem létezik rá érvényes megoldás, vagy a rendszer hibás megoldást állít elő, illetve a problémák csak később, nehezen visszakövethető módon jelentkeznek. Emiatt az projektem által képviselt minta, amely a generálást formális validációval és célzott visszacsatolással kombinálja, széles körben alkalmazható olyan helyzetekben, ahol egyszerre van szükség gyors vagy kreatív modellkiegészítésre, miközben a formális helyesség nem engedhető el.

3.2. LLM-ek szerepe modellalapú rendszertervezésben

Ebben a pontban csak megemlítenék pár szót az LLM-ek szerepéről, ami pedig az, hogy a természetes nyelvű, gyakran hiányos vagy informális leírásokat gyorsan strukturált modelljavaslatokká alakítsák. Erősségük nem a formális helyesség garantálása, hanem az, hogy hatékonyan támogatják a modell kezdeti kialakítását és iteratív finomítását. Modellalapú rendszertervezésben ez akkor válik igazán értékessé, ha a generálás egy olyan környezetbe van beágyazva, ahol a formális ellenőrzés kiszűri a hibákat, és a visszacsatolás célzott javításokat tesz lehetővé. Így az LLM bizonytalansága nem kockázatként jelenik meg, hanem egy kontrollált folyamat részeként, amelyben a kreatív javaslatképzés és a formális döntési mechanizmus együtt biztosítja a megbízható végeredményt.

4. Összefoglalás

Az OnlabRefinery projektem célja az volt, hogy megmutassam, hogy a nagy nyelvi modellek hasznosak lehetnek formális modellezési feladatok támogatásában, de csak akkor, ha nem „igazságforrásként” kezeljük őket. Az önálló labor során bemutatott architektúra ezt a problémát úgy kezeli, hogy az LLM-et javaslatképző komponensként használja, miközben a végső döntést és a kimenet helyességének garantálását egy formális solverre(Refinery-re) bízza. A rétegezett modellfelépítés és az iteratív visszacsatolásos működés együtt olyan kontrollált folyamatot ad, ahol az LLM bizonytalansága nem tud észrevétlenül átcsúszni a végeredménybe.
A projekt videójátékos példán keresztül szemléltette a megközelítést, de a bemutatott minta általánosítható olyan területekre is, ahol entitások, kapcsolatok és szabályok együtt alkotnak egy formálisan kezelendő rendszert. Ilyen lehet például termékkatalógusok konzisztenciájának biztosítása, konfigurációs problémák kezelése vagy üzleti folyamatok formális leírása és ellenőrzése. A fő üzenet az, hogy a generatív komponens és a formális validáció együttműködése nem csak hibaszűrésre jó, hanem egy mérnöki értelemben stabilabb fejlesztési mintát is ad.

4.1. Értékelés

A projekt eredménye egy olyan felelősséget megosztott architektúra, ahol az LLM modellkiegészítési javaslatokat ad, a Refinery pedig formális szabályok alapján ellenőrzi ezek elfogadhatóságát. Ez a szétválasztás átláthatóvá teszi a működést, mert a hibák nem vélemény kérdések, hanem konkrét formális megsértések, amelyek visszacsatolhatók és javíthatók.
A megoldás másik erőssége az iteratív működés, amelyben a rendszer a Refinery hibáin keresztül fokozatosan tereli helyes irányba a kimenetet. Tudatos korlát, hogy a folyamat nem garantál mindig konvergenciát, ezért az iterációszám limitje kontrollált leállást biztosít túl összetett vagy bizonytalan esetekben. Összességében a bemutatott minta azért releváns, mert a generatív komponensek használata terjed, egyre népszerűbb, és ezért pedig egyre fontosabb, hogy a kimenet megbízhatósága formális eszközökkel kézben tartható legyen.

4.2. Továbbfejlesztési lehetőségek és jövőbeli tervek

A projekt több irányban is továbbfejleszthető. Elsősorban skálázás, tökéletesítés a komplexebb metamodellek és szabályrendszerek támogatásához, ahol különösen fontossá válik a szabályok kezelhetősége és a visszacsatolás informatívsága. Emellett érdemes lehet javítani a visszajelzés minőségét, például a hibák kategorizálásával és célzottabb instrukciók generálásával, ami csökkentheti a szükséges iterációk számát és javíthatja a helyes eredmény esélyét. További fejlesztési irány a megfigyelhetőség és a vizualizáció erősítése, például az iterációk közti változások kiemelésével és egyszerű hibastatisztikák bevezetésével, valamint a jelenlegi technikai hiányosságok kezelése, ideértve a modellvizualizáció finomítását, az állapotkezelés és összehasonlítás bővítését, továbbá az adatszerkezetek általánosítását és az egységesebb névkezelést, amelyek együttesen robusztusabbá és könnyebben bővíthetővé tehetik a megoldást. Végül egy valósabb demonstrációs domain választása, például termékkatalógus vagy konfigurációs feladat, még jobban megmutatná, hogy a megközelítés nem csak szemléltető példákon működik, hanem iparibb jellegű problémákra is átültethető.