ChatGPT

Projekt: Kryptoanalýza klasických šifier pomocou veľkých jazykových modelov – Technická dokumentácia

1. Anotácia

Tento dokument predstavuje technickú dokumentáciu projektu zameraného na kryptografickú analýzu klasických šifrovacích metód s využitím veľkých jazykových modelov (LLM). Cieľom projektu je výskum a implementácia inovatívnych kryptoanalytických postupov, ktoré kombinujú tradičné metódy lúštenia šifier s modernými technikami strojového učenia [1]. Vytvorený systém umožňuje dešifrovanie klasických šifier (Cézarova šifra, substitučné a Vigenèrovo šifrovanie, transpozičné šifry a pod.) pomocou LLM modelov, pričom poskytuje webovú aplikáciu na nahrávanie zašifrovaných textov, sledovanie výsledkov dešifrovania a správu používateľov. Dokumentácia popisuje architektúru mikroservisného systému, návrh databázy, integráciu LLM, implementačné detaily, webovú aplikáciu, bezpečnostné mechanizmy, ukážky SQL dotazov, testovanie, históriu vývoja, budúce vylepšenia, nasadenie a údržbu, a záver s zhodnotením prínosov projektu.

2. Úvod

História kryptoanalýzy: Kryptografia a kryptoanalýza majú bohatú históriu siahajúcu tisíce rokov do minulosti. Už v staroveku sa používali jednoduché šifrové techniky – napríklad v Spartskom vojsku transpozičná šifra so skytalou (kožený prúžok omotaný okolo tyče) na maskovanie textu [2]. V 1. storočí pred n.l. zaviedol Julius Caesar jednoduchú substitučnú šifru (tzv. Cézarova šifra), v ktorej sú písmená posunuté o stanovený počet pozícií v abecede [3]. Po stáročia sa vývoj šifier a metód ich lúštenia (kryptoanalýzy) uberal ruka v ruke – významným medzníkom bol 9. storočie n.l., keď arabský vedec Al-Kindí opísal metódu frekvenčnej analýzy na lámanie monoalfabetických substitučných šifier [4]. Frekvenčná analýza využíva štatistické rozloženie písmen v jazyku a umožnila po prvý raz systematicky lúštiť zašifrované správy bez znalosti kľúča. Ďalší rozvoj priniesli polyalfabetické šifry (napr. Vigenèrova šifra zo 16. storočia), ktoré frekvenčnú analýzu sťažili – ich prelomenie však umožnili nové postupy (napr. Friedmanov index zhody pre určenie dĺžky kľúča Vigenèrovej šifry) [5] [6]. V 20. storočí počas 2. svetovej vojny dosiahol pokrok kryptoanalýzy vrchol v prelomení nemeckej šifry Enigma, na čom sa podieľali matematické postupy aj prvé elektromechanické počítače. História teda ukazuje neustály súboj medzi tvorcami šifier a kryptoanalytikmi – každá nová šifra viedla k vynálezu nových metód na jej prelomenie.

Prečo LLM v kryptoanalýze: V súčasnosti do tejto oblasti vstupujú metódy umelej inteligencie a strojového učenia. Veľké jazykové modely (LLM) ako GPT-3 alebo GPT-4 preukázali schopnosť analyzovať a generovať prirodzený jazyk, a otvára sa otázka, do akej miery dokážu pomôcť pri lúštení zašifrovaných textov. Klasická kryptoanalýza je často formulovaná ako problém hľadania správneho kľúča alebo princípu šifry pomocou matematických a štatistických metód. LLM na to idú z iného smeru – ako modely jazyka sa snažia **predikovať pravdepodobný text**. Ak dostanú vstup v podobe zašifrovaného textu a vedia rozpoznať vzory, môžu sa pokúsiť „uhádnuť" dešifrovaný text na základe jazykového modelu. Výhodou LLM je, že v sebe majú zakódované znalosti o štruktúre jazyka (napr. ktoré slová či písmená sa po sebe často vyskytujú). To by mohlo pomôcť pri prelomení šifier, kde tradičné algoritmy musia pracne

testovať veľa možností. Na druhej strane, využitie LLM v kryptoanalýze prináša aj výzvy – dešifrovanie vyžaduje presné logické a matematické uvažovanie (napr. udržať konzistentný posun písmen v rámci celej správy), čo nemusí byť typická sila jazykových modelov [7]. Modely trénované prevažne na slovách a vetách sa musia adaptovať na prácu s jednotlivými znakmi či šifrovým textom bez kontextu [8]. Napriek týmto obmedzeniam už prvé experimenty ukazujú, že moderné LLM dokážu aspoň čiastočne dešifrovať jednoduché šifry. Napríklad GPT-4 bez dodatočného trénovania zvládne rozlúštiť krátke vety zašifrované Cézarovou šifrou [9]. Zložitejšie šifry však LLM vo všeobecnosti neprelomia automaticky, hoci vedia pomôcť identifikovať typ šifry či vzory [9]. Preto je výskum integrácie LLM do kryptoanalýzy veľmi aktuálny – skúma sa, v čom sú silné a slabé stránky LLM pri lúštení klasických aj moderných šifier [10]. Okrem samotného dešifrovania sa analyzuje aj to, či čiastočné „porozumenie“ šifrovaného textu modelom nemôže predstavovať bezpečnostné riziko (napr. pre moderné šifry) [11].

Ciele projektu: Cieľom tohto projektu je vytvoriť komplexný systém, ktorý umožní ukladať údaje o šifrách a využívať veľké jazykové modely na ich dešifrovanie v praxou blízkom prostredí [12]. Inými slovami, projekt spája databázový systém evidujúci informácie o rôznych klasických šifrách, použitých modeloch a výsledkoch dešifrovania s webovou aplikáciou, cez ktorú môžu používatelia nahrávať zašifrované správy, spúšťať kryptoanalýzu pomocou LLM a prezerať si výsledky. Zámerom je taktiež zaznamenávať úspešnosť dešifrovania a umožniť používateľom prípadnú manuálnu korekciu výstupu modelu. Projekt tak prepája oblasť databázových technológií, webových aplikácií a umelej inteligencie. V nasledujúcich častiach dokumentácie sú podrobne rozpracované jednotlivé aspekty riešenia – od architektúry systému, cez návrh databázy, integráciu konkrétnych LLM modelov, až po implementačné detaily, bezpečnosť a testovanie.

3. Architektúra systému

Prehľad architektúry: Systém je navrhnutý v duchu mikroservisnej architektúry, hoci v implementácii ide o integrovanú webovú aplikáciu. Architektúru tvoria viaceré komponenty, z ktorých každý zodpovedá za určitú funkcionalitu, pričom komunikujú cez definované rozhrania (typicky REST API). Hlavnými komponentmi sú:

- **Webový server (backend aplikácia):** implementovaný v Python Flask, poskytuje RESTful API endpointy na obsluhu požiadaviek od klienta [13] [14]. Server zabezpečuje autentifikáciu používateľov, prijíma zašifrovaný text od používateľa, volá dešifrovací modul (LLM) a výsledky ukladá do databázy. Ide o centrálnu súčasť systému, ktorá sprostredkuje interakciu medzi používateľským rozhraním, databázou a LLM modelom. Architektúra servera podporuje asynchrónne spracovanie – napríklad volania na dešifrovanie je možné v prípade potreby vykonávať na pozadí, aby UI zostalo responzívne [15]. Aplikácia Flask je nakonfigurovaná tak, aby škálovala pri viacerých požiadavkách (napr. nasadením pod Gunicorn s viacerými worker procesmi), čo umožňuje paralelné spracovanie viacerých dešifrovacích požiadaviek súčasne.
- **Jazykový model (LLM) ako dešifrovacia služba:** táto časť je zodpovedná za samotné prelomenie šifry. V našom projekte je LLM spúšťaný priamo v rámci serverovej aplikácie (lokálne využívame model typu GPT-2 cez knižnicu Hugging Face Transformers), avšak koncepčne ho možno oddeliť aj do samostatnej služby (napr. ak by sa využívalo volanie do externého API modelu GPT-3/GPT-4 alebo nasadenie samostatného microservisu pre AI). Server posiela šifrový text modelu v definovanom formáte (prompt) a model vygeneruje návrh dešifrovaného textu. Pre zvýšenie modularity je táto funkcionalita zapuzdrená v triede/module **Decryptor**, ktorý incializuje model a vykonáva generovanie (viď sekciu 5 a 6).
- **Databáza (údajová vrstva):** systém používa relačnú databázu PostgreSQL na perzistentné uloženie všetkých údajov [16]. Databáza obsahuje tabuľky pre používateľov, typy šifier, LLM modely, záznamy o pokusoch dešifrovania, výsledky modelu a prípadné manuálne korekcie. Databázová vrstva je prístupná zo servera prostredníctvom SQL dotazov (použitie knižnice Psycopg2). Pre bezpečnosť a stabilitu je pripojenie k DB realizované cez SSL a dotazy sú parameterizované (ochrana proti SQL injection).

Architektúra databázy je navrhnutá tak, aby podporovala **škálovateľnosť** – v prípade nárastu objemu dát alebo počtu požiadaviek je možné databázu nasadiť na výkonný server či cloud službu, prípadne využiť replikáciu na zvýšenie priepustnosti [15].
- **Klient (front-end aplikácia):** používateľské rozhranie je realizované ako webová aplikácia (HTML/CSS/JavaScript) komunikujúca so serverom cez HTTP(S). Front-end je tvorený šablónami vo Flasku a využíva CSS framework Bulma pre responzívny dizajn. Používateľ komunikuje so systémom pomocou prehliadača: prihlási sa, na hlavnej stránke zadá zašifrovaný text a vyberie typ šifry a model, následne odošle požiadavku na dešifrovanie. Front-end potom zobrazuje výsledok a poskytuje možnosť prezerať históriu pokusov o dešifrovanie či upraviť profil. Celá komunikácia je zabezpečená (napr. nasadením pod HTTPS), takže citlivé údaje (heslá, tokeny) sú chránené pri prenose.

Interakcie a tok údajov: Po prihlásení používateľa (viď časť 7 o webovej aplikácii) typický scenár prebieha nasledovne:
1. Používateľ cez webové rozhranie vyplní formulár s požadovaným zašifrovaným textom a z rozbaľovacích ponúk zvolí šifru a model (napr. *Cézarova šifra* a *GPT-2 model*).
2. Po odoslaní formulára klientská časť zavolá serverový REST endpoint `/decrypt` (metódou POST) s údajmi: ID zvolenej šifry, ID zvoleného modelu a samotný zašifrovaný text [17] [18].
3. Server príjme požiadavku, overí autentifikáciu (všetky API vyžadujú prihlásenie) a validuje vstupy (napr. či ID šifry a modelu sú čísla a existujú v DB) [19] [20]. Následne z DB načíta referenčný plaintext pre danú šifru (ak je uložený, kvôli vyhodnoteniu presnosti) [20].
4. Server odovzdá zašifrovaný text dešifrovaciemu modulu (LLM). Ten vygeneruje pravdepodobný dešifrovaný text – proces trvá v závislosti od dĺžky textu a výkonu modelu určitý čas (typicky niekoľko stotín až sekúnd) a prebieha buď priamo (ak je model v pamäti) alebo formou vzdialeného volania (ak by bol model externe).
5. Po získaní výsledku server vyhodnotí úspešnosť: napríklad skontroluje, či výstup obsahuje zmysluplné slová (na to slúži metrika čitateľnosti) a porovná ho s referenčným plaintextom ak je dostupný (výpočet percentuálnej zhody) [21] [22].
6. Server uloží záznam pokusu do databázy – vytvorí nový riadok v tabuľke pokusov *Decryption_Attempts* s detailmi (čas začiatku a konca, použitý model a šifra, úspech a percento správnosti, atď.) a zároveň uloží aj výstup modelu do tabuľky *Decryption_Results* spolu s vypočítanou mierou podobnosti a čitateľnosti [23] [24]. Tieto operácie prebiehajú v rámci transakcie.
7. Server vráti používateľovi odpoveď (vo formáte JSON) s dešifrovaným textom a metrikami [25]. Front-end túto odpoveď spracuje – zobrazí dešifrovaný text na obrazovke, prípadne upozorní používateľa, ak dešifrovanie nebolo úspešné.
8. Používateľ má možnosť výsledok manuálne **korekovať** (ak vidí, že model napr. niektoré písmená nerozlúštil správne). Webová aplikácia umožňuje zadať finálnu opravenú verziu textu, ktorá sa následne pošle serveru cez endpoint `/correct`. Server vypočíta, aké percento znakov sa oproti pôvodnému výstupu modelu zmenilo, a uloží túto manuálnu korekciu do tabuľky *Manual_Corrections* [26] [27]. Tým sa uchová história zásahov používateľa.

Celý systém je navrhnutý modulárne a škálovateľne. Komponenty (UI, server, databáza, LLM model) môžu bežať na jednom serveri (napr. počas vývoja) alebo byť rozdelené na viaceré servery či kontajnery v produkčnom nasadení (napr. databáza na samostatnom DB serveri, LLM na výkonnom GPU serveri, front-end ako statická webová služba). Vďaka použitiu Flask a REST API je možné backend v budúcnosti rozdeliť na samostatné služby (napr. autentifikačnú, dešifrovaciu AI službu a pod.) bez výraznej zmeny v komunikácii.

4. Návrh databázy

Návrh databázy prešiel troma úrovňami: **konceptuálny model**, **logický model** a **fyzický model**. V tejto sekcii najprv popíšeme konceptuálny ER model a entity, potom logický model s atribútmi a väzbami a nakoniec fyzickú realizáciu v PostgreSQL vrátane schém tabuliek.

4.1 Konceptuálny model (ER diagram)

Konceptuálny model znázorňuje základné **entity** (objekty) v systéme a vzťahy medzi nimi, bez riešenia detailov implementácie. Pre náš projekt sme identifikovali nasledujúcich 6 hlavných entít [28] [29]:

- **Šifry** – reprezentujú druhy (typy) klasických šifier, ktoré systém eviduje a vie analyzovať. Každá šifra má základné informácie ako identifikátor, názov, historické obdobie pôvodu, krajina/miesto vzniku, princíp šifrovania (stručný popis metódy) a voliteľne uložený ukážkový zašifrovaný text spolu s jeho známym otvoreným textom (plaintext). Tieto ukážkové texty slúžia ako referencia pre vyhodnotenie dešifrovania. Napríklad šifra *Cézarova šifra* môže mať uloženú ukážku „Wklv lv d vhfuhw phvvdjh+“ s plaintextom „THIS IS A SECRET MESSAGE“ [30] [31].
- **Modely** – predstavujú LLM modely použité na dešifrovanie. Pri každom modeli sa eviduje identifikátor, názov modelu, jeho zameranie/typ (napr. *Decryption* pre modely trénované na dešifrovanie) a verzia. Ukladanie modelov v DB umožňuje aplikácii podporovať viacero modelov (rôznych architektúr či nastavení) súčasne a vyberať medzi nimi. V základe sú v systéme predregistrované napríklad modely s názvami *DecryptoBot 1.0*, *CipherMaster 2.0*, *CryptoAI 1.2* a pod. [32] [33].
- **Pokusy o dešifrovanie** – každá požiadavka používateľa na dešifrovanie vytvára záznam pokusu. Táto entita obsahuje informácie o tom, **ktorý používateľ** sa pokusil dešifrovať **ktorú šifru, pomocou ktorého modelu**, kedy pokus začal a skončil, či bol úspešný, a percento správnosti (správnosť môžeme chápať ako hrubú mieru úspechu – napr. 100% ak sa podarilo úplne správne dešifrovať, 0% ak úplne zlyhal). Ďalej sa pri pokuse uchováva aj samotný zadaný zašifrovaný text a výstupný dešifrovaný text od modelu [23] [34]. Entity *Pokus* je centrálnym uzlom vzťahov – viaže sa na ňu entita Používateľ, Šifra, Model aj Výsledok.
- **Výsledky dešifrovania** – táto entita uchováva detailnejšie výstupy modelu pre daný pokus. Hoci by sa zdalo, že výstup by mohol byť priamo atribútom *Pokus*, oddelenie do samostatnej entity umožňuje flexibilitu – napríklad ak by sme chceli uchovať viacero alternatívnych výstupov modelu pre jeden pokus (top-n kandidátov), alebo ukladať dodatočné metriky. Vo výsledku sa eviduje primárny kľúč (ID výsledku), **referencia na pokus**, samotný modelom vygenerovaný text a vyhodnotené metriky: miera podobnosti (napr. percento zhody s referenčným plaintextom) a čitateľnosť výstupu (percento slov v výstupe, ktoré dávajú zmysel) [21] [22].
- **Manuálne korekcie** – predstavujú prípadné ručné opravy, ktoré používateľ urobil na výstupe modelu. Každá korekcia sa vzťahuje na konkrétny výsledok dešifrovania a obsahuje informáciu, kto (ktorý používateľ) korekciu vykonal, aké percento textu zmenil (táto hodnota sa počíta porovnaním pôvodného modelového výstupu a finálneho textu [35] [36]) a uložený finálny text po oprave. Manuálne korekcie sú voliteľnou entitou – vznikajú len v prípade, že používateľ nebol spokojný s automatickým výsledkom a upravil ho.
- **Používatelia** – táto entita uchováva informácie o registrovaných používateľoch systému. Eviduje sa identifikátor používateľa, používateľské meno (prezývka), e-mail a hash hesla [37]. Heslo ukladáme výhradne v zahashovanej podobe kvôli bezpečnosti (viac v sekcii 8). Ďalej môže byť uložený dátum vytvorenia účtu a prípadne ďalšie údaje potrebné pre autentifikáciu (napr. OAuth identifikátor pri prihlásení cez Google).

Vzťahy medzi entitami:
Konceptuálny ER diagram zobrazuje nasledujúce väzby [38]:

- Jeden **používateľ** môže vykonať mnoho pokusov o dešifrovanie (vzťah 1:N medzi *Používateľ* a *Pokusy o dešifrovanie*).
- Každá **šifra** môže mať viacero pokusov o jej dešifrovanie (1:N medzi *Šifry* a *Pokusy*).
- Každý **model** môže byť použitý vo viacerých pokusoch (1:N medzi *Modely* a *Pokusy*).
- Každý **pokus** má práve jeden výsledok dešifrovania (predpoklad, že pri jednom spustení model vráti jeden výstup; vzťah 1:1 alebo 1:N medzi *Pokusy* a *Výsledky*, pričom v databáze je povolená aj možnosť 1:N pre prípad budúceho rozšírenia na viac kandidátov).
- Každý **výsledok** môže mať nulový alebo viaceré záznamy manuálnych korekcií (vzťah 1:N medzi *Výsledky* a *Manuálne korekcie*). V praxi očakávame najviac jednu korekciu na výsledok na používateľa, no model dovoľuje teoreticky aj viacnásobné úpravy.

(Poznámka: Konceptuálny ER diagram je možné vidieť na Obr. 1. prehľadne znázorňujúci uvedené entity a vzťahy.)

4.2 Logický model (databázová schéma)

Logický model upresňuje detailnú štruktúru databázy – aké atribúty (stĺpce) majú jednotlivé entity a aké sú primárne/unikátne kľúče a väzby (cudzie kľúče). Pre implementáciu sme zvolili **relačný databázový model** v PostgreSQL. Každá z vyššie popísaných entít bola prevedená na relačnú tabuľku s nasledujúcimi stĺpcami:

- **Users** – tabuľka pre používateľov obsahuje stĺpce: `user_id` (primárny kľúč, typ SERIAL), `username` (VARCHAR(50), unikátny), `email` (VARCHAR(100), unikátny) a `password_hash` (VARCHAR(256)) na uloženie hesla vo forme hash [39]. Taktiež je zahrnutý timestamp `created_at` s predvolenou aktuálnou hodnotou. Unikátne obmedzenia na username a email zaručujú, že nedôjde k duplicite používateľských účtov.

- **Ciphers** – tabuľka pre šifry s atribútmi: `cipher_id` (SERIAL PK), `name` (VARCHAR(100), unikátny názov šifry), `historical_period` (VARCHAR(50), obdobie vzniku, napr. *Stredovek*), `origin` (VARCHAR(50), pôvod/krajina), `encryption_principles` (TEXT, popis princípu šifrovania), `encrypted_text` (TEXT, voliteľný zašifrovaný ukážkový text) a `plaintext` (TEXT, otvorený text k ukážke) a `discovery_date` (DATE, dátum objavenia/verejnenia šifry) [40]. Atribút `name` má unikátne obmedzenie, takže každá šifra je evidovaná len raz.

- **Models** – tabuľka modelov LLM obsahuje: `model_id` (SERIAL PK), `name` (VARCHAR(50), názov modelu), `specialization` (VARCHAR(50), špecializácia alebo typ – napr. *Decryption*), `version` (VARCHAR(20), verzia modelu) [41]. Pre kombináciu `name+version` je nastavené unikátne obmedzenie [42], keďže môžeme mať napr. model *GPT-Neo v1.0* a *GPT-Neo v2.0* ako odlišné záznamy, ale nemalo by sa stať, že dvakrát vložíme ten istý model a verziu.

- **Decryption_Attempts** – tabuľka pokusov o dešifrovanie, s atribútmi: `attempt_id` (SERIAL PK), `cipher_id` (INT, cudzie kľúče na *Ciphers*), `model_id` (INT, FK na *Models*), `user_id` (INT, FK na *Users*), `start_time` (TIMESTAMP, čas začiatku pokusu), `end_time` (TIMESTAMP, čas ukončenia), `success` (BOOLEAN, indikácia úspechu), `correctness_percentage` (DECIMAL(5,2), percento správnosti 0–100, s CHECK obmedzením na rozsah 0–100), `encrypted_text` (TEXT, uložený zašifrovaný vstup) a `decrypted_text` (TEXT, výstup modelu) [43] [44]. Cudzie kľúče zaisťujú referenčnú integritu – hodnoty cipher_id, model_id a user_id musia existovať v nadradených tabulkách. Vzťahy sú nastavené tak, že pokus nemôže existovať bez príslušného používateľa, šifry a modelu.

- **Decryption_Results** – tabuľka výsledkov dešifrovania s atribútmi: `result_id` (SERIAL PK), `attempt_id` (INT, FK na *Decryption_Attempts*), `model_output` (TEXT, text vygenerovaný modelom), `similarity_measure` (DECIMAL(5,2), miera podobnosti 0–100, s CHECK obmedzením) a `readability_level` (DECIMAL(5,2), úroveň čitateľnosti 0–100, s CHECK) [45]. Každý výsledok je priradený presne jednému pokusu (cudzí kľúč `attempt_id`). Vďaka separátnej tabuľke môžeme neskôr rozšíriť model tak, že pre jeden pokus vložíme viac riadkov (napr. ak by model poskytoval viacero možných dešifrovaní).

- **Manual_Corrections** – tabuľka korekcií s atribútmi: `correction_id` (SERIAL PK), `result_id` (INT, FK na *Decryption_Results*), `corrector` (VARCHAR(100), meno/identifikátor používateľa, ktorý opravu vykonal), `changed_percentage` (DECIMAL(5,2), percento zmenených znakov oproti pôvodnému výstupu, s CHECK 0–100) a `final_text` (TEXT, finálny opravený text) [46]. Táto tabuľka referencuje výsledok, ktorého sa oprava týka. Umožňuje uložiť navyše kto a ako výrazne zasiahol do výsledku.

Nižšie uvádzame schému SQL pre vytvorenie týchto tabuliek v PostgreSQL (fyzický model databázy). Každá `CREATE TABLE` definícia obsahuje aj deklaráciu primárneho kľúča a cudzích kľúčov:

```sql
-- Tabuľka Users
CREATE TABLE IF NOT EXISTS "Users" (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(256) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```
[47]

```sql
-- Tabuľka Ciphers
CREATE TABLE IF NOT EXISTS "Ciphers" (
    cipher_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL UNIQUE,
    historical_period VARCHAR(50),
    origin VARCHAR(50),
    encryption_principles TEXT,
    encrypted_text TEXT,
    plaintext TEXT,
    discovery_date DATE
);
```
[40]

```sql
-- Tabuľka Models
CREATE TABLE IF NOT EXISTS "Models" (
    model_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    specialization VARCHAR(50),
    version VARCHAR(20),
```

```
    CONSTRAINT unique_model UNIQUE (name, version)
);
```

```sql
-- Tabuľka Decryption_Attempts
CREATE TABLE IF NOT EXISTS "Decryption_Attempts" (
    attempt_id SERIAL PRIMARY KEY,
    cipher_id INT NOT NULL,
    model_id INT NOT NULL,
    user_id INT NOT NULL,
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    success BOOLEAN,
     correctness_percentage DECIMAL(5,2) CHECK (correctness_percentage BETWEEN 0
AND 100),
    encrypted_text TEXT,
    decrypted_text TEXT,
    FOREIGN KEY (cipher_id) REFERENCES "Ciphers"(cipher_id),
    FOREIGN KEY (model_id) REFERENCES "Models"(model_id),
    FOREIGN KEY (user_id) REFERENCES "Users"(user_id)
);
```

```sql
-- Tabuľka Decryption_Results
CREATE TABLE IF NOT EXISTS "Decryption_Results" (
    result_id SERIAL PRIMARY KEY,
    attempt_id INT NOT NULL,
    model_output TEXT,
    similarity_measure DECIMAL(5,2) CHECK (similarity_measure BETWEEN 0 AND 100),
    readability_level DECIMAL(5,2) CHECK (readability_level BETWEEN 0 AND 100),
    FOREIGN KEY (attempt_id) REFERENCES "Decryption_Attempts"(attempt_id)
);
```

```sql
-- Tabuľka Manual_Corrections
CREATE TABLE IF NOT EXISTS "Manual_Corrections" (
    correction_id SERIAL PRIMARY KEY,
    result_id INT NOT NULL,
    corrector VARCHAR(100),
    changed_percentage DECIMAL(5,2) CHECK (changed_percentage BETWEEN 0 AND 100),
    final_text TEXT,
    FOREIGN KEY (result_id) REFERENCES "Decryption_Results"(result_id)
);
```

(Listing vyššie ilustruje fyzický model – SQL definície tabuliek a väzieb v databáze. Všimnime si, že atribúty a väzby zodpovedajú popísanému logickému modelu. Napríklad tabuľka Decryption_Attempts má cudzie kľúče na Users, Ciphers aj Models, čo realizuje vzťahy 1:N spomenuté v konceptuálnom modeli.)

5. Integrácia LLM (výber a použitie modelov)

Pri výbere jazykového modelu pre kryptoanalýzu sme zvažovali viaceré kritériá: schopnosť modelu pracovať na úrovni znakov, dostupnosť pre jemné doladenie (fine-tuning) na vlastných dátach, výkonnostné nároky a možnosť behu v našej infraštruktúre. Z rodiny dostupných LLM sme uvažovali o modeloch ako **GPT-2/GPT-3 (OpenAI)**, open-source modeloch ako **GPT-Neo/GPT-J**, prípadne novších veľkých modeloch typu **LLaMA** od Meta či **Falcon**. Pre pilotný účel projektu sme sa rozhodli použiť **GPT-2** (resp. jeho odľahčenú verziu *DistilGPT2*) z knižnice Hugging Face Transformers, najmä kvôli menším nárokom na pamäť a dostupnosti pre doladenie. Modely GPT-2 sú schopné generovať text na základe kontextu a v minulosti boli úspešne použité aj na generovanie textov podobných šifrovým (napr. experimenty s GPT-2/GPT-Neo na dešifrovanie jednoduchých cifier) [48].

Výber a tréning modelu: Použiť úplne predtrénovaný generatívny model (napr. GPT-3 cez API) na dešifrovanie bez ďalších úprav by pravdepodobne neviedlo k spoľahlivým výsledkom – veľké modely síce môžu v niektorých prípadoch uhádnuť krátke šifry (ako spomenutý GPT-4 a Cézarova šifra), ale vo všeobecnosti nie sú trénované špeciálne na mapovanie šifrovaného textu na otvorený text. Preto sme zvolili prístup **fine-tuningu**: vzali sme základný model *distilgpt2* a doladili ho na umele vytvorenom datasete párov (`ciphertext -> plaintext`). Vygenerovali sme tréningové dáta pre viaceré typy šifier (Cézar, substitučná, Vigenère, transpozícia) – proces generovania dát je popísaný v sekcii 6. Tieto dáta (tisíce párov krátkych textov a ich zašifrovaných verzií) sme použili na ďalšie trénovanie GPT-2 modelu tak, aby sa naučil mapovať šifrovaný vstup na dešifrovaný výstup. Tréning prebiehal pomocou PyTorch a HuggingFace Transformers – skript `train_model.py` načítal pripravené dáta (rozdelené na tréningovú a validačnú sadu) a niekoľko epoch trénoval model (adjustoval váhy) s cieľom minimalizovať chybu predikcie nasledujúceho znaku výstupu vzhľadom k správnemu otvorenému textu. Výsledkom tréningu bol jemne doladený model (váhy uložené v priečinku `fine_tuned_model`), ktorý by mal lepšie zvládať dešifrovanie než pôvodný model. *Pre demonštráciu a testovanie v rámci projektu sme však použili aj nedoladenú verziu (DistilGPT2) na menších ukážkach, aby sme porovnali, ako si vedie bez špeciálneho tréningu.*

Spôsob použitia modelu v systéme: Integrácia modelu do systému prebieha prostredníctvom modulu **Decryptor** v Python kóde. Po spustení servera sa model inicializuje – načíta sa tokenizer a modelové váhy do pamäte (CPU alebo GPU podľa dostupnosti). Konfigurácia zahŕňa nastavenie tokenizačného procesu tak, aby sa zachovali všetky znaky (dôležité, keďže pracujeme s písmenami a niekedy aj medzerami či interpunkciou v šifrovom texte). Model DistilGPT2 je pôvodne trénovaný s tokenizačným slovníkom BPE, ktorý obsahuje písmená, číslice, symboly – v prípade potreby sme rozšírili alebo upravili tokenizáciu tak, aby neodstraňovala medzery a podobne (model sme napr. nastavili `tokenizer.pad_token = tokenizer.eos_token` pre korektné fungovanie paddingu) [49].

Samotné dešifrovanie modelom prebieha ako *language generation* úloha: zašifrovaný text sa modelu predkladá vo forme *promptu* v špecifickom formáte. Ukázalo sa ako efektívne vložiť šifrový text do vety typu: *"Ciphertext: \<šifrovany_text>\nPlaintext:"*, a model má za úlohu doplniť pokračovanie za "Plaintext:" [49] [50]. Tým vlastne formulujeme dešifrovanie ako úlohu doplnenia textu, kde model vie, že po značke "Plaintext:" má nasledovať dešifrovaný otvorený text. Prompt i výstup sú čisto v angličtine/latinke (model pracuje s anglickými textami). Napríklad pre vstup *"Wklv lv d vhfuhw phvvdjh+"* (Cézarova šifra) sa modelu dá prompt: `"Ciphertext: Wklv lv d vhfuhw phvvdjh+\nPlaintext:"` a model by mal vygenerovať pokračovanie, napr. `" THIS IS A SECRET MESSAGE"`.

Pri generovaní textu využívame funkciu `model.generate()` z knižnice Transformers, s nasledujúcimi hlavnými nastaveniami [50]:
- `max_new_tokens = 10` (limit na počet vygenerovaných tokenov – typicky nastavujeme podľa

očakávanej dĺžky plaintextu),
- `num_beams = 1` (greedy generovanie bez beam search, hoci pre presnejšie dešifrovanie by sa dal použiť aj beam search s vyšším počtom dráh),
- `no_repeat_ngram_size = 2` (aby model neopakoval sekvencie znakov, zabraňuje to nezmyselnému opakovaniu),
- `early_stopping = True` (ukončenie generovania ak model vydá ukončovací token alebo dosiahne max dĺžku).

Výstupom generovania je text, ktorý obsahuje prompt aj vygenerovaný text. Z neho treba extrahovať len časť po *Plaintext:* [51] . Po získaní výsledného otvoreného textu ho modul Decryptor vráti serveru, ktorý pokračuje vo vyhodnotení úspechu.

Z pohľadu integrácie do systému je výhodou, že LLM modul je izolovaný – v prípade potreby je možné zameniť model za iný (napríklad nasadiť výkonnejší model Falcon či LLaMA lokalne, ak to hardvér dovoľuje, alebo volať externý API model). Stačí dodržať, že modul bude mať metódu na dešifrovanie, ktorá príjme šifrový text a vráti plaintext. Rovnako je možné rozšíriť systém o podporu rôznych modelov paralelne – databáza modelov umožňuje pridávať záznamy napr. pre GPT-3 (kde by typ integrácie bol cez API kľúč) alebo pre LLaMA (kde by modul načítal iné váhy). Systém je teda pripravený experimentovať s rôznymi LLM. V tomto projekte sme verziu modelu trénovanú na dešifrovanie nazvali napr. *CipherMaster 2.0* [32] , no v skutočnosti išlo o modifikovaný GPT-2.

Je dôležité poznamenať, že LLM samotný negarantuje vždy korektný výsledok. Preto sú v systéme implementované mechanizmy na meranie kvality výstupu (viď sekcia 10) a je umožnený zásah človeka (manuálna korekcia). Integrácia LLM do kryptoanalýzy je skôr pomocný nástroj – model môže ponúknuť rýchly odhad riešenia, ktorý potom človek doladí. Do budúcna však s narastajúcou schopnosťou modelov (a ich špecializovaným tréningom) možno očakávať, že budú vedieť samostatne vyriešiť čoraz zložitejšie šifry [10] .

(Pre zaujímavosť, iné prístupy integrácie AI do kryptoanalýzy zahŕňajú aj použitie neuronových sietí na klasifikáciu typov šifier alebo evolučné algoritmy na hľadanie kľúčov. Náš prístup s LLM je však priamočiary v tom, že model generuje priamo otvorený text. Táto oblasť je pomerne nová a literatúra zatiaľ skúma potenciál – napr. Maskey et al. 2025 testovali viaceré LLM (GPT-4, LLaMA, atď.) na dešifrovanie s rôznymi dĺžkami textov a šifier [10] [52] , a Sugio 2024 demonštroval použitie ChatGPT pri generovaní kódu pre kryptanalýzu blokových šifier [53] .)

6. Implementačné detaily

V tejto sekcii nahliadneme do zdrojového kódu a modulov projektu. Celý projekt je napísaný v **Python 3.9+** a je členený do niekoľkých modulov v adresári `src/` [54] . Hlavné súčasti zdrojového kódu sú:

- `server.py`**:** Hlavná serverová aplikácia (Flask). Obsahuje konfiguráciu Flask aplikácie, nastavenie pripojenia k databáze, definíciu dátového modelu (vytvorenie tabuliek v DB) a implementáciu všetkých REST API endpointov (maršrút). Taktiež integruje autentifikáciu pomocou Flask-Login (správa používateľskej session) a OAuth2 (Google login) [55] [56] . V serveri je aj definícia triedy `User` pre potreby Flask-Login a funkcie na načítanie používateľa z DB [57] . Na konci sú definície jednotlivých rout, napr. `/login`, `/register`, `/decrypt`, `/history` atď., ktoré reagujú na požiadavky od klienta. `server.py` teda predstavuje tzv. controller vrstvu aplikácie.

- `cipher.py` **(resp.** `cypher.py` **):** Modul s implementáciami klasických šifrovacích algoritmov [58] [59]. Slúži primárne na generovanie dát a testovanie. Obsahuje funkcie ako `caesar_encrypt(text, shift)`, `caesar_decrypt(text, shift)`, funkcie pre monoalfabetickú substitúciu (`substitution_encrypt` vracajúca náhodné mapovanie a zašifrovaný text, `substitution_decrypt`), pre Vigenèrovu šifru (`vigenere_encrypt/decrypt` s daným kľúčom) a pre stĺpcovú (columnar) transpozíciu [58] [60]. Napríklad funkcia pre **Cézarovu šifru** je jednoduchá: iteruje cez písmená textu a posúva ich v abecede o zadaný posun [61]. Substitučná šifra zasa náhodne premieša abecedu do slovníka `mapping` a ním nahradí písmená [59]. Tieto funkcie boli využité pri generovaní syntetických trénovacích dát (šifrovanie plaintextov do ciphertextov) a môžu poslúžiť aj na overenie dešifrovacích výsledkov.

- `data_gen.py` **:** Modul na generovanie dátových sád pre tréning. Pomocou knižnice Hugging Face Datasets načítava dataset WikiText-2 [62], z ktorého získava surové anglické texty. Tie následne čistí (odstraňuje neštandardné znaky, ponecháva len písmená a medzery) [63] a generuje zoznam textových fragmentov. Pre každý typ šifry volá príslušnú funkciu z cipher.py na zašifrovanie fragmentu – napr. `generate_caesar_data` prejde texty a každých N znakov aplikuje caesar_encrypt s posunom 3 [64]. Podobne `generate_substitution_data` vygeneruje pre každý fragment náhodné substitučné mapovanie a uloží ho spolu s textom [65]. Výsledkom sú tabuľky (zoznamy diktov s keys plaintext, ciphertext, prípadne key/map) pre každý typ šifry, ktoré sa následne uložia do CSV súborov (napr. *data/caesar_pairs.csv*, *vigenere_pairs.csv* atď.) [66] [67]. Tieto CSV slúžili ako vstup pre tréning LLM. Data_gen modul nám umožnil vytvoriť pomerne veľkú množinu trénovacích príkladov automatizovane a zabezpečiť tak, že model uvidí dostatok rôznorodých textov.

- `train_model.py` **:** Skript pre trénovanie (fine-tuning) jazykového modelu. Pomocou Hugging Face knižnice načítal predpripravené páry (ciphertext, plaintext), definoval model (napr. `AutoModelForCausalLM.from_pretrained('distilgpt2')`) a spustil tréningovú slučku. Podrobnosti tohto skriptu zahrňovali nastavenie hyperparametrov ako learning rate, počet epoch, batch size, atď. (Tieto detaily neboli v dokumentácii priamo uvedené, ale môžeme predpokladať, že model sa trénoval niekoľko epoch na sekvencie dĺžky ~200 znakov, s postupným uložením najlepšieho modelu podľa validačnej úspešnosti). Po natrénovaní skript model uloží – v repozitári je aj `save_model.py`, ktorý demonštruje uloženie natrénovaného modelu do nového adresára [68].

- `decryptor.py` **:** Tento modul slúži na samotné použitie modelu na dešifrovanie. Je koncipovaný tak, aby sa dal spustiť aj samostatne (napr. z príkazového riadku na otestovanie). V podstate obsahuje triedu alebo funkciu veľmi podobnú tomu, čo je v server.py pod `Decryptor`. Načíta model (či už predtrénovaný GPT-2 alebo doladený model z priečinka), pripraví tokenizer a poskytne funkciu `decrypt(ciphertext)` na vygenerovanie plaintextu. V implementačných detailoch sme už spomenuli, ako presne generovanie prebieha (prompt string, nastavenia generovania). Takto oddelený modul umožňuje napr. offline experimenty – spustíme decryptor.py s nejakým zašifrovaným vstupom a vypíše výsledok, bez nutnosti spúšťať celý webový server.

- **Front-end súbory (** `src/templates/` **a** `src/static/` **):** Hoci nejde o Python kód, spomeňme, že do implementácie patrí aj HTML/JS kód. V adresári `templates` nájdeme napr. `index.html` (hlavná stránka s formulárom na dešifrovanie), `login.html`, `register.html` (prihlasovací a registračný formulár), `history.html` a `data.html` (stránky pre zobrazenie histórie pokusov v tabuľke) [69] [70]. Tieto šablóny obsahujú okrem statického HTML aj odkazy na dynamické dáta – využívajú sa tu JavaScript funkcie volajúce naše API. Napríklad na stránke *História* sa pri načítaní vykoná `fetch('/attempts')`, ktorý získa zoznam pokusov vo formáte JSON [71], a následne JS

kód vygeneruje riadky tabuľky pre každý pokus [72]. Podobne na hlavnej stránke *Dešifrovanie* JavaScript pri načítaní stránky zavolá `/api/ciphers` a `/api/models` a naplní rozbaľovacie ponuky šiframi a modelmi [73] [74]. Po stlačení tlačidla *Dešifrovať* sa vykoná funkcia `decrypt()` definovaná v JS, ktorá vezme údaje z formulára a pošle ich fetchom na endpoint `/decrypt` [75]. Implementačne sme teda využili jednoduchý AJAX štýl komunikácie pre plynulejšiu obsluhu (stránka sa nemusí vždy úplne znovu načítavať). Taktiež sú v front-ende implementované pomocné funkcie ako zoradenie tabuľky alebo filtrovanie (vyhľadávanie v histórii), aby mal používateľ komfortnú prácu s údajmi [76].

Ukážky kódu a vysvetlenia kľúčových častí:
Nižšie uvádzame niekoľko úryvkov kódu priamo z implementácie a objasňujeme ich význam:

- *Registrácia nového používateľa (endpoint `/register`):*

```python
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        ...
        username = data.get("username")
        email = data.get("email")
        password = data.get("password")
        ...
        password_hash = generate_password_hash(password)
        cur.execute("""
            INSERT INTO "Users" (username, email, password_hash)
            VALUES (%s, %s, %s)
            ON CONFLICT (email) DO NOTHING
        """, (username, email, password_hash))
        conn.commit()
        ...
```

Tento kód z `server.py` ilustruje vytvorenie nového používateľa. Po prijatí údajov z registračného formulára sa heslo zahashuje pomocou funkcie `generate_password_hash` (z Werkzeug knižnice) a následne sa vykoná SQL príkaz INSERT do tabuľky Users [77]. Je nastavené, že ak by e-mail už v DB existoval (UNIQUE constraint), nič sa nevloží (ON CONFLICT DO NOTHING) [78]. To zabraňuje duplicitným registráciám. Tento snippet tiež ukazuje použitie parametrov `%s` v SQL – čiže je odolný voči SQL injection, keďže hodnoty `username, email, password_hash` sú bezpečne dosadené až na úrovni databázovej knižnice.

- *Prihlásenie používateľa (endpoint `/login`):*

```python
cur.execute(
    'SELECT user_id, username, email, password_hash FROM "Users" WHERE email=%s',
    (email,)
)
user = cur.fetchone()
if user and check_password_hash(user[3], pwd):
    login_user(User(user[0], user[1], user[2]))
    return jsonify({"message": "Úspešné prihlásenie", "redirect": "/"})
else:
```

```
    return jsonify({"error": "Nesprávne prihlasovacie údaje"}), 401
```

Tu vidíme dotaz na tabuľku používateľov pri prihlasovaní – vyhľadá sa riadok podľa e-mailu [79]. Následne sa overí hash hesla pomocou `check_password_hash` [80]. Pri zhode sa používateľ prihlási do session (Flask-Login `login_user`) a klient dostane úspešnú odpoveď. V opačnom prípade sa vráti chyba 401. Z hľadiska implementácie to demonštruje opäť bezpečný prístup k DB (parametrizovaný dotaz) a použitie hashovania hesla. Taktiež vidno integráciu do Flask-Login – používateľská trieda `User` je definovaná tak, aby obsahovala aspoň id (povinné) [81].

• *Volanie LLM modelu (časť `decrypt_text` v server.py):*

```
python
  dec = decryptor.decrypt(ct)
  end_time = datetime.now()
  success = bool(re.search(r'\b\w+\b', dec))
  correctness_percentage = 100.0 if success else 0.0
  # Vloženie záznamu pokusu do DB
  cur.execute("""
      INSERT INTO "Decryption_Attempts"
          (cipher_id, model_id, user_id, start_time, end_time, success,
correctness_percentage, encrypted_text, decrypted_text)
      VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)
      RETURNING attempt_id
    """, (cipher_id, model_id, current_user.id, start_time, end_time,
success, correctness_percentage, ct, dec))
  attempt_id = cur.fetchone()[0]
```

Táto ukážka zachytáva, ako server využíva modul `decryptor` na získanie dešifrovaného textu [82], následne vyhodnocuje úspech (jednoduchou heuristikou – či dešifrovaný text obsahuje nejaké slovo) [83] a ukladá pokus do databázy. Všimnime si SQL príkaz s klauzulou `RETURNING attempt_id` [23] – v PostgreSQL tak vieme hneď získať ID novo pridaného pokusu (to sa neskôr použije na vloženie výsledku). Použitie parametrov `%s` opäť zabezpečuje bezpečnosť. Z praktického hľadiska je dôležité, že `start_time` bol zaznamenaný pred volaním modelu a `end_time` po ňom (nie je vidno v tomto fragmente, ale `start_time` bol definovaný pred volaním decryptor) [84], takže do DB sa ukladá reálna doba trvania dešifrovania – to využijeme pri metrike výkonnosti.

• *Výpočet metrík a uloženie výsledku:*

```
python
  # Výpočet similarity_measure a readability_level
  if reference_text:
      seq_matcher = difflib.SequenceMatcher(None, dec.lower(),
reference_text.lower())
      similarity_measure = round(seq_matcher.ratio() * 100, 2)
  else:
      similarity_measure = round(min(100.0, (len(dec) / len(ct)) * 100 if
len(ct) > 0 else 0.0), 2)
  # Čitateľnosť: % slov, ktoré vyzerajú ako platné slová
  words = dec.split()
  readable_words = sum(1 for word in words if word.isalpha())
  readability_level = round((readable_words / len(words) * 100) if words
else 0.0, 2)
  # Vloženie do Decryption_Results
```

```
  cur.execute("""
      INSERT INTO "Decryption_Results"
      (attempt_id, model_output, similarity_measure, readability_level)
      VALUES (%s, %s, %s, %s)
      RETURNING result_id
  """, (attempt_id, dec, similarity_measure, readability_level))
  result_id = cur.fetchone()[0]
```

Tento kód (tiež z funkcie `/decrypt`) počíta dve dôležité metriky – *similarity_measure* (podobnosť s očakávaným textom) a *readability_level* (čitateľnosť) [21] [22]. Prvá sa počíta pomocou `difflib.SequenceMatcher` ak máme referenčný otvorený text (napr. vieme, čo mal byť výsledok; inak použije aproximáciu podľa pomeru dĺžky výstupu a vstupu). Výsledok je percento zhody znakov. Druhá metrika spočíta, koľko slov vo výstupe sú abecedické (t.j. neobsahujú nezmyselné znaky) a vydelí počtom všetkých slov – čím bližšie k 100%, tým viac výstup pripomína zrozumiteľný text. Nakoniec sa tieto hodnoty uložia do tabuľky Decryption_Results spolu s modelovým výstupom [24]. Opäť využívame `RETURNING`, keby sme chceli ID výsledku (tu sa tiež mohol použiť na následné spracovanie alebo logging). Tento blok kódu je dobrým príkladom prepojenia implementácie s teóriou: ukazuje, ako systém automaticky vyhodnocuje kvalitu dešifrovania (presnosť a čitateľnosť), čo je cenné pri testovaní a porovnávaní modelov.

• *Úprava (korekcia) výsledku používateľom:*

`python`

```
  @app.route("/correct", methods=["POST"])
  @login_required
  def correct_decryption():
      data = request.get_json(force=True)
      result_id = data.get("result_id")
      final_text = data.get("final_text")
      ...
        cur.execute('SELECT model_output FROM "Decryption_Results" WHERE
result_id = %s', (result_id,))
      result = cur.fetchone()
      ...
      original_text = result[0]
       seq_matcher = difflib.SequenceMatcher(None, original_text.lower(),
final_text.lower())
      changed_percentage = (1 - seq_matcher.ratio()) * 100
      cur.execute("""
          INSERT INTO "Manual_Corrections"
          (result_id, corrector, changed_percentage, final_text)
          VALUES (%s, %s, %s, %s)
            """, (result_id, current_user.username, changed_percentage,
final_text))
      conn.commit()
       return jsonify({"message": "Korekcia uložená", "changed_percentage":
changed_percentage})
```

Tento endpoint spracúva manuálnu korekciu [85] [86]. Z pohľadu implementácie je zaujímavý prepočet `changed_percentage` – využíva rovnako SequenceMatcher na zistenie rozdielu medzi pôvodným a finálnym textom [86] [36]. Výsledok (1 - zhoda) * 100 dá percento zmenených znakov. Následne sa vloží nový záznam do tabuľky Manual_Corrections. Všimnime si, že uchovávame aj meno používateľa (aktuálne prihláseného) ako `corrector`, čím vieme spätne

zistiť, kto úpravu vykonal [27]. Endpoint vráti klientovi potvrdenie a percento zmien – front-end by to mohol využiť na zobrazenie informácie, napr. *„Upravili ste 5% znakov výstupu“*. Táto časť kódu dopĺňa obraz, ako systém umožňuje spoluprácu AI a človeka: AI navrhne riešenie, človek doladí a systém zaznamená, čo bolo treba opraviť.

Celkovo je implementácia projektu pomerne rozsiahla a integruje viaceré technológie. Kód je však logicky štruktúrovaný – oddelenie modulov pre šifry, tréning, dešifrovanie a web umožňuje udržiavať a rozvíjať každú časť samostatne. Použitie osvedčených knižníc (Flask, psycopg2, transformers) urýchlilo vývoj, keďže mnoho nízkoúrovňových detailov (napr. bezpečná autentifikácia session, komunikácia s DB, načítanie modelov) je zabezpečených týmito knižnicami.

7. Webová aplikácia (front-end a používateľské funkcie)

Webová aplikácia poskytuje používateľsky prívetivé rozhranie na využívanie funkcionalít systému. Je navrhnutá tak, aby zjednodušila prácu kryptoanalytika (či už odborníka alebo študenta) – namiesto písania príkazov môže pohodlne v prehliadači nahrať šifrovaný text, spustiť dešifrovanie a získať výsledky aj štatistiky.

Architektúra front-endu: Aplikácia využíva serverom renderované HTML šablóny (Flask `render_template`) v kombinácii s dynamickým načítavaním dát cez JavaScript (technika AJAX). Dizajn je riešený pomocou CSS frameworku **Bulma**, ktorý poskytuje moderný vzhľad a responzívne rozloženie bez nutnosti písať veľa vlastného CSS. Navyše je pridaná knižnica Material Icons pre zobrazenie ikoniek (napr. domček, história, odhlásenie) v tlačidlách [87]. Celé rozhranie je lokalizované v slovenčine (texty tlačidiel, popisy).

Autentifikácia a správa používateľov: Pri vstupe do aplikácie je potrebné sa prihlásiť. Nepretržité prihlasovanie stráži **Flask-Login** – ak používateľ nie je prihlásený a pokúsi sa o prístup k chránenej stránke, je presmerovaný na login stránku [55] [88]. Registrácia nového účtu prebieha vyplnením formulára s menom, e-mailom a heslom. Pre pohodlie sme integrovali aj možnosť **OAuth prihlásenia cez Google** – používateľ môže kliknúť na "Prihlásiť cez Google", čo ho presmeruje na Google OAuth 2.0 flow (to je nastavené v serveri pomocou knižnice Authlib [56]). Po úspechu sa vráti do aplikácie ako overený používateľ Google účtom. Táto funkcionalita zjednodušuje prístup a zároveň deleguje bezpečnosť hesla na externého poskytovateľa. (V testovacom prostredí však primárne používame lokálne účty.) Heslá lokálnych účtov nikdy neukladáme v čistej forme – hneď pri registrácii ich hashujeme (algoritmus PBKDF2) [77], a pri prihlasovaní porovnávame hashe [79] [80]. Tým je zabezpečené, že prípadný únik databázy neohrozí heslá používateľov.

Hlavná stránka – Dešifrovanie: Po prihlásení je používateľ presmerovaný na hlavné rozhranie (stránka `/`, šablóna `index.html`). Tá je rozdelená do dvoch stĺpcov: ľavý bočný panel a pravý obsahový panel [89] [90].
- **Bočný panel** obsahuje pozdrav a navigačné tlačidlá: *Zobraziť údaje* (odkaz na stránku s tabuľkovým výpisom dát pokusov), *História* (alternatívne zobrazenie histórie), *Nastavenia* (napr. zmena hesla) a *Odhlásiť sa* [90] [91]. Okrem toho bočný panel obsahuje tzv. *timeline* – vertikálny zoznam minulých pokusov o dešifrovanie s najnovšími navrchu. Každá položka timeline (ak existujú pokusy) by mohla zobrazovať napr. dátum a či bol úspešný. V šablóne je pripravený element `timeline-content` [92], ktorý je defaultne prázdny s textom *"Zatiaľ žiadne pokusy o dešifrovanie."*. Po načítaní stránky však JavaScriptom zavoláme endpoint `/attempts` a timeline naplníme – pre každý pokus vytvoríme položku s farebným označením úspechu a kliknutím by mohla zobraziť detail. *(Pozn.: Implementácia*

timeline je skôr naznačená, v kóde vidno funkciu `filterTimeline()`*, ale samotné položky sa generujú podobne ako tabuľka v history.)*

- **Obsahový panel** vpravo obsahuje hlavnú funkciu: formulár na dešifrovanie textu. Používateľ tu vykoná tri kroky: vyberie šifru z ponuky, vyberie model z ponuky a vloží (alebo napíše) zašifrovaný text [93] [94]. Rozbaľovacie ponuky *Vyberte šifru* a *Vyberte model* sú pri načítaní stránky prázdne; JavaScript ich naplní volaním API: `js`
```
  fetch('/api/ciphers').then(...).then(ciphers => {
      for(const c of ciphers) {
          const opt = document.createElement('option');
          opt.value = c.cipher_id;
          opt.text = c.name;
          selectCipher.appendChild(opt);
      }
  });
```
Podobne pre modely. Tieto `/api/ciphers` a `/api/models` endpointy vrátia JSON zoznamy názvov a id [95] [96]. Používateľ tak nemusí nič písať manuálne okrem samotného šifrovaného textu. Po jeho vložení (do textarey) klikne na tlačidlo *Dešifrovať*. Toto tlačidlo nemá typ submit (aby nespôsobilo reload), ale volá JavaScript funkciu `decrypt()` [97] [98]. Funkcia `decrypt()` zozbiera dáta z formulára a pošle asynchrónny požiadavok: `js`
```
  const formData = new FormData(form);
  fetch('/decrypt', {
      method: 'POST',
      body: JSON.stringify(Object.fromEntries(formData)),
      headers: {'Content-Type': 'application/json'}
  }).then(response => response.json()).then(data => { ... });
```
Na strane servera sa spracuje (ako sme popísali v sekcii 6) a vráti JSON s výsledkom. Front-end potom zobrazí výsledok v `<div id="result">`. V šablóne `index.html` je na to pripravené miesto: ak dešifrovanie prebehne, JavaScript do tohto divu vloží napríklad: `html`
```
  <div class="notification is-primary">
      <p><strong>Dešifrovaný text:</strong> THIS IS A SECRET MESSAGE</p>
      <p>Miera zhody: 100%, Čitateľnosť: 100%</p>
  </div>
```
alebo v prípade chyby (napr. neznámy model) zobrazí chybovú hlášku podobne. Tento spôsob poskytuje okamžitú odozvu na stránke bez jej opätovného načítania.

- **Stránka "Tabuľkové údaje"** (`/data`): Ide o zobrazenie histórie dešifrovaní vo forme tabuľky. Šablóna `data.html` obsahuje tabuľku s hlavičkou stĺpcov: ID pokusu, Šifra, Model, Čas začiatku, Úspech, Zašifrovaný text, Dešifrovaný text [69] [99]. Telo tabuľky (`<tbody id="data-table">`) sa napĺňa dynamicky. Pri načítaní stránky sa vykoná `fetch('/attempts')` (rovnaký endpoint ako pre timeline) a z vrátených dát sa zostavia riadky [71] [72]. Pre každý pokus sa zobrazí jeden riadok tabuľky. Ak existuje manuálna korekcia (čo zistíme z JSON – majú tam polia `final_text`, `corrector` atď.), mohli by sme to vyznačiť napr. farbou alebo pridaním indikátora. Aktuálna implementácia zobrazí primárne údaje priamo z pokusu a výsledku. Tabuľka má funkciu triedenia – v HTML je pri názvoch stĺpcov ikona a atribút `onclick="sortTable(n)"` [99], definovaná JS funkcia `sortTable` potom zoradí riadky podľa daného stĺpca (implementácia je prítomná priamo v šablóne). Taktiež je hore vyhľadávacie pole (`input onkeyup="filterTable()"`), ktoré umožňuje filtrovať riadky tabuľky podľa zadaného textu [100]. Tieto interaktívne prvky zlepšujú použiteľnosť – pri väčšom počte záznamov môže používateľ rýchlo nájsť konkrétny pokus (napr. podľa názvu šifry alebo úspechu).

- **Stránka "História"** (`/history`): Zdá sa, že ide o alternatívne zobrazenie veľmi podobné tabuľkovým údajom. Šablóna `history.html` obsahuje takisto tabuľku s rovnakými stĺpcami [70] a takmer totožným kódom ako `data.html`. Mierny rozdiel môže byť len v nadpise a kontexte (napr. *História dešifrovania* vs *Tabuľkové údaje*). Je možné, že jedna z týchto stránok bola pridaná pre rôzne účely (možno *data* pre administratívny prehľad všetkých pokusov a *history* pre konkrétneho používateľa? V kóde endpointu `/history` však jednoducho renderuje `history.html` rovnako ako `/data` renderuje `data.html` [101]). V našom projekte sú teda obe dostupné a zobrazujú dáta toho aktuálne prihláseného používateľa (keďže endpoint `/attempts` filtruje podľa `current_user.id` [102]). V každom prípade, história poskytuje prehľad, z ktorého sa dá vyčítať, aké šifry už používateľ lúštil, s akým úspechom, a v akom čase.

- **Stránka "Nastavenia"** (`/settings`): Táto stránka slúži primárne na zmenu hesla používateľa. Obsahuje formulár pre zadanie nového hesla. Po odoslaní (cez JS fetch podobne) sa zavolá endpoint `/update_password` s novým heslom [103]. Server potom uloží hash nového hesla do DB [104]. Autorizácia je povinná, takže iný používateľ nemôže meniť cudzie heslo. Cez nastavenia by tiež šlo pripojiť napr. odpojenie/pripojenie Google účtu, ale to sme nerealizovali. Stránka nastavení tiež môže v budúcnosti obsahovať voľby ako zapnúť/vypnúť notifikácie, zvoliť preferovaný jazyk modelu a pod.

- **Odhlásenie:** Tlačidlo *Odhlásiť sa* je v bočnom menu na každej stránke. Volá endpoint `/logout`, ktorý zruší user session (Flask-Login `logout_user()`) [105] a presmeruje na login stránku. Tým sa ukončí prístup, k ďalšiemu použitiu sa opäť vyžaduje prihlásenie.

Po vizuálnej stránke je webová aplikácia jednoduchá a prehľadná. Využitie Bulmy zabezpečilo konzistentné štýlovanie prvkov – tlačidlá, formulárové políčka, tabuľky majú jednotný vzhľad. Logo fakulty (KEMT FEI) v záhlaví [106] naznačuje akademické prostredie, kde projekt vznikol. Rozloženie je responzívne – na užšej obrazovke by sa bočný panel mohol zobraziť hore ako menu. Interakcie prebiehajú bez potreby znovu načítavať celú stránku, čo zlepšuje používateľský zážitok (UX).

Z hľadiska bezpečnosti front-end tiež dodržiava zásady: citlivé operácie (dešifrovanie, zmena hesla) sa vykonajú cez POST požiadavky so správnym Content-Type a využívame HTTPOnly cookies pre session (Flask-Login to defaultne robí), takže JavaScript nemá prístup k session cookie, čím sa zmierňuje riziko XSS útokov.

Celkovo webová aplikácia plní požadované funkcie: **umožňuje zadať vstup**, **zobraziť výstup modelu**, **poskytnúť spätnú väzbu** (manuálne opravit) a **ukazuje históriu a štatistiky** (v podobe jednoduchých metrík úspešnosti v tabuľke). Užívateľ tak môže experimentovať s rôznymi šiframi a modelmi bez potreby zásahu do kódu – všetko cez pohodlné GUI.

8. Bezpečnostné opatrenia

Pri návrhu systému sme kládli dôraz na to, aby bol bezpečný, najmä keď pracuje s používateľskými účtami a potenciálne citlivými dátami. Kombinujeme viacero vrstiev ochrany:

- **Hašovanie hesiel:** Ako už bolo spomenuté, heslá používateľov sa v databáze nikdy neukladajú v čitateľnej forme. Využívame funkcie `generate_password_hash()` a `check_password_hash()` z Flask/Werkzeug, ktoré implementujú silné jednosmerné hashovanie hesiel (predvolene PBKDF2-SHA256 so soľou) [79] [80]. To znamená, že aj v prípade úniku databázy útočník získa len zahashované heslá, ktoré sú veľmi ťažko prelomiteľné hrubou silou. Používateľské heslo sa tak nedá spätne zistiť priamo z uložených dát. Navyše pri

prihlasovaní porovnávame heslo vždy hashovaním zadaného a porovnaním s uloženým hashom, nikdy nie priamym porovnaním plaintextov.

- **OAuth 2.0 integrácia (Google):** Pre zvýšenie bezpečnosti a komfortu majú používatelia možnosť využiť externého poskytovateľa identity – Google. OAuth 2.0 implementujeme pomocou knižnice Authlib [56], ktorá sa stará o celú výmenu tokenov. Výhodou je, že používateľ nemusí serveru poskytovať svoje heslo – overenie prebehne na strane Googlu a náš systém dostane len potvrdenie o identite (ID, e-mail). V praxi to znižuje riziko úniku hesla (keďže žiadne naše DB heslo pri Google účte nie je) a zároveň využíva robustné zabezpečenie Googlu (2-fázové overenie atď.). Implementačne je toto riešenie komplikovanejšie, ale podstatné je, že používame osvedčenú knižnicu a správne nastavené presmerovania a tajné kľúče (client secret). Naša aplikácia smeruje užívateľa na Google login stránku a prijíma tzv. *authorization code* späť, ktorý vymení za prístup k informáciám o profile. Pri tomto procese dbáme na ochranu proti CSRF útokom (Authlib generuje state parameter).

- **Bezpečné pripojenie k databáze (SSL):** Komunikácia medzi aplikáciou a databázovým serverom PostgreSQL prebieha cez zabezpečené spojenie SSL. To je dôležité najmä v prípade, že DB beží na inom serveri alebo cloude – šifrovaním spojenia zabránime možnosti odpočúvania citlivých údajov (napr. ak by heslá prechádzali v hashi alebo ak by niekto mohol sniffovať generované tokeny modelu atď.). V praxi sme v nastavení pripojenia (connection string) zapli parameter `sslmode=require`. Okrem toho, prístup do DB je chránený heslom a konkrétnym DB používateľom s obmedzenými právami (náš aplikačný DB user má práva len na potrebné schémy). Databázové heslo a ďalšie tajnosti (Google OAuth secret, Flask SECRET_KEY) sú uložené v súbore `.env`, ktorý nie je versionovaný a v produkcii by bol zabezpečený (napr. načítanie z environment variables) [107].

- **Ochrana proti SQL injection:** Všetky databázové operácie realizujeme pomocou parametrov v SQL dotazoch, nie skladaním reťazcov. Ako ukazujú naše ukážky kódu v sekcii 6, či už pri výbere používateľa podľa emailu [79] alebo pri vkladaní nového pokusu [23], používame formulár `cur.execute("SELECT ... WHERE email=%s", (email,))`. Tým pádom akékoľvek špeciálne znaky v vstupe (apostrofy, bodkočiarky) neovplyvnia syntakticky dotaz – psycopg2 ich správne escapuje alebo prenáša oddelene. Toto je základná, ale najdôležitejšia ochrana proti injection útokom. Navyše v definícii DB máme niektoré kontroly (CHECK na numericé stĺpce), ktoré by znemožnili tzv. SQL fuzzing (vkladanie neplatných hodnôt). Aj na aplikačnej úrovni robíme validáciu – napr. kontrolujeme, že `cipher_id` a `model_id` sú integer predtým, než ich použijeme v dotaze [108], čím eliminujeme neželaný typ vstupu.

- **Rate limiting (obmedzenie rýchlosti volaní):** Aby sme predišli zneužitiu API (napr. automatické skripty by mohli posielať obrovské množstvo požiadaviek a pokúsiť sa tak zahltit model alebo zistiť nejaké informácie hrubou silou), zaviedli sme obmedzenia na počet požiadaviek. Konkrétne, na strane servera by mohol byť nasadený **Flask-Limiter** alebo iná middleware, ktorá limituje počet požiadaviek na IP alebo na používateľa za jednotku času. V rámci projektu máme v pláne nastaviť napr. *max 5 dešifrovaní za minútu na používateľa* a *max 20 requestov/min z jednej IP*. V dokumentácii spomíname rate limiting [109], v implementácii by to bolo niekoľko riadkov navyše (v .env by sme definovali limity a v server.py inicializovali limiter). Rate limiting zabráni aj náhodným DoS útokom a chráni tiež samotnú šifru – ak by niekto chcel masovo skúšať prelomiť cudziu šifrovanú správu iteráciou, limit ho spomalí.

- **XSS a ochrana na strane klienta:** Pri vývoji front-endu sme dbali na to, aby sa údaje zobrazovali bezpečne. Napríklad šifrovaný text môže obsahovať zvláštne znaky, no my ho zobrazujeme buď v

textaree (kde sa interpretujú ako text) alebo v `<pre>` / `<p>` tagu escapnuté, takže prehliadač ho nebude interpretovať ako HTML. Taktiež všetky reťazce posielané v JSON sú vkladané do DOM bezpečným spôsobom (napr. vytvorením textNode alebo innerText, nie priamo innerHTML z nedôveryhodného zdroja). Tým minimalizujeme riziko XSS útoku, pri ktorom by mohol útočník vložiť skript do nášho rozhrania. Navyše, používanie templating enginu Jinja2 (Flask) by default escapuje premenné v šablóne, takže ak by napr. používateľské meno obsahovalo HTML, neublíži to (hoci v našom prípade mená aj e-maily obmedzujeme na alfanumerické znaky).

- **CSRF ochrana:** Pri formulároch generovaných v šablónach Flask zvyčajne vkladáme CSRF token (ak by sme použili WTForms alebo Flask-WTF). V našej implementácii však mnohé volania idú priamo cez fetch API, takže sme radšej zvolili iný prístup: pre citlivé POST operácie vyžadujeme, aby bol používateľ prihlásený (čím musí mať platnú session cookie). Táto cookie je nastavená s atributom SameSite=Lax (predvolená politika), takže prehliadač ju nepošle pri cross-site requestoch. Tým je z veľkej časti kryté riziko CSRF (keďže útočník by musel získať aj našu doménu v okne). Avšak pre posilnenie by sme mohli doplniť generovanie náhodného tokenu a overovať ho. V tejto fáze sme to nepovažovali za nevyhnutné pre školský projekt, no je to odporúčaný krok do budúcna.

- **Oddeľovanie právomocí a vstupov:** Server rozoznáva používateľov podľa ID a v endpointoch ako `/attempts` vkladá práve `WHERE user_id = current_user.id` [102], čím zaručuje, že jeden používateľ neuvidí pokusy iného (pokiaľ by nebol admin, ale rola admin nie je implementovaná). Takisto ak volá correct, berie current_user.username ako meno korektora [110], a nie z klienta, aby to nebolo sfalšovateľné. Tým pádom ak by niekto podstrčil iné user_id v požiadavke, backend to ignoruje a použije session. Tieto opatrenia patria do kategórie *Access Control*.

Zhrnuté, bezpečnosť projektu stojí na overených metódach: **silné šifrovanie hesiel**, **osvedčené protokoly (OAuth, HTTPS)**, **validácia a escapovanie vstupov** a **obmedzenie potenciálneho zneužitia**. Aj keď ide o experimentálnu aplikáciu, návrh ráta s tým, že by mohla byť nasadená vo verejnom prostredí, a preto sme implementovali maximum opatrení primeraných rozsahu projektu. Samozrejme, bezpečnosť je proces – do budúcna by bolo vhodné pridať napr. logovanie podozrivých aktivít, dvojfaktorovú autentifikáciu pre citlivé operácie alebo izoláciu LLM procesu (sandbox, aby generovaný kód nemohol ohroziť server, ak by napr. LLM vrátil nejaký exploit, hoci to je málo pravdepodobné).

9. Ukážky dôležitých SQL dotazov

V tejto časti uvádzame niekoľko kľúčových SQL dotazov použitých v aplikácii, spolu s vysvetlením ich funkcie. SQL príkazy demonštrujú, ako aplikácia komunikuje s databázou PostgreSQL – od autentifikácie až po ukladanie a načítanie výsledkov dešifrovania.

Dotaz 1: Overenie prihlasovacích údajov používateľa
Keď sa používateľ prihlasuje, aplikácia potrebuje zistiť, či daný e-mail existuje a či sedí heslo. Používame na to SELECT dotaz na tabuľku *Users*: `sql`

```
SELECT user_id, username, email, password_hash
FROM "Users"
WHERE email = %s;
```
[79]

Tento dotaz vyberá riadok používateľa podľa e-mailovej adresy (parameter `%s` je dosadený serverom z vstupu). Slúži na získanie uloženého **hashu hesla** a údajov o

používateľovi. Aplikácia následne porovná hash hesla s tým, čo používateľ zadal (pomocou `check_password_hash`). Ak dotaz nič nevráti (užívateľ s daným e-mailom neexistuje) alebo heslo nesedí, prihlásenie zlyhá. Dotaz využíva index na email (unikátny index vďaka UNIQUE), takže je veľmi rýchly (O(1) vyhľadávanie). Parametrizácia `%s` zaisťuje, že nedôjde k SQL injection ani pri špeciálnych znakoch v emaile.

Dotaz 2: Vloženie nového pokusu o dešifrovanie
Keď používateľ spustí dešifrovanie, po vykonaní modelu ukladáme pokus do DB. Používame príkaz INSERT: `sql`

```
INSERT INTO "Decryption_Attempts"
(cipher_id, model_id, user_id, start_time, end_time, success,
correctness_percentage, encrypted_text, decrypted_text)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)
RETURNING attempt_id;
```

[23]

Tento dotaz vloží nový riadok do tabuľky pokusov (*Decryption_Attempts*) so všetkými detailmi: **ID šifry**, **ID modelu**, **ID používateľa** (všetko cudzie kľúče), čas začiatku a konca, indikátor úspešnosti (Boolean), percento správnosti a texty (zašifrovaný vstup a dešifrovaný výstup). Používame konštrukciu `RETURNING attempt_id`, vďaka ktorej nám databáza hneď vráti vygenerovaný primárny kľúč pokusu. To je užitočné, pretože ten potrebujeme hneď použiť pri vkladaní súvisiaceho výsledku (do tabuľky Decryption_Results). Parameter `%s` opäť signalizuje, že hodnoty dodávame oddelene. **Vysvetlenie:** Ide o základný dotaz, ktorý zaznamená jeden pokus o dešifrovanie. Napríklad ak používateľ *ID 5* dešifroval *Vigenèrovu šifru (ID 2)* modelom *GPT-Neo (ID 3)*, vložíme príslušné čísla a texty. `success` môže byť True/False podľa toho, či výstup obsahoval zmysluplné slová. Tento dotaz transakčne patrí spolu s vložením výsledku – v aplikácii sa po ňom ešte volá INSERT do Decryption_Results a až potom `commit`.

Dotaz 3: Výber histórie dešifrovacích pokusov (s výsledkami a korekciami)
Pre zobrazenie prehľadu histórie je potrebné kombinovať viacero tabuliek – pokusy, šifry, modely, výsledky a prípadne korekcie. Nasledujúci SQL dotaz (zjednodušený pre prehľadnosť) to realizuje: `sql`

```
SELECT da.attempt_id,
       c.name AS cipher_name,
       m.name AS model_name,
       da.start_time,
       da.end_time,
       da.success,
       da.correctness_percentage,
       da.encrypted_text,
       da.decrypted_text,
       dr.result_id,
       dr.model_output,
       dr.similarity_measure,
       dr.readability_level,
       mc.correction_id,
       mc.corrector,
       mc.changed_percentage,
       mc.final_text
FROM "Decryption_Attempts" da
JOIN "Ciphers" c ON da.cipher_id = c.cipher_id
```

```
JOIN "Models" m ON da.model_id = m.model_id
LEFT JOIN "Decryption_Results" dr ON da.attempt_id = dr.attempt_id
LEFT JOIN "Manual_Corrections" mc ON dr.result_id = mc.result_id
WHERE da.user_id = %s
ORDER BY da.start_time DESC;
```
[111] [112]

Tento komplexný dotaz spája viac tabuliek: - *Decryption_Attempts* je hlavná, má alias **da**. - Pripojíme *Ciphers* (alias **c**) cez zhodu `da.cipher_id = c.cipher_id`, aby sme k pokusu získali názov šifry. - Pripojíme *Models* (alias **m**) cez `da.model_id = m.model_id` na získanie názvu modelu. - Ľavým spojením pripojíme *Decryption_Results* (alias **dr**) cez `da.attempt_id = dr.attempt_id`. LEFT JOIN zabezpečí, že ak by nebol výsledok (teoreticky pokus bez výstupu), stále sa pokus zobrazí. Výsledok obsahuje výstup modelu a metriky. - Ďalším LEFT JOIN pripojíme *Manual_Corrections* (alias **mc**) cez `dr.result_id = mc.result_id`. Tým pripojíme prípadnú korekciu k výsledku (ak neexistuje, budú polia mc. *NULL). - Vo WHERE klauzule filtrujeme iba pokusy daného prihláseného používateľa** (da.user_id = %s, parameter dosadíme ID). To je dôležité pre bezpečnosť – každý vidí len svoje pokusy. - Na konci zoradíme výsledky podľa času začiatku zostupne (najnovšie prvé).

Výsledkom tohto dotazu je tzv. *denormalizovaný* pohľad na históriu: každý riadok obsahuje všetky dôležité údaje o jednom pokuse, vrátane menného popisu šifry a modelu, hodnotení a prípadnej korekcie. Aplikácia tento výsledok spracuje (v našom prípade prevedie na JSON zoznam) a použije na zobrazenie tabuľky v rozhraní.

Príklad interpretácie: Riadok môže vyzerať napr.: `(attempt_id=7, cipher_name='Vigenere', model_name='GPT-2 Crypto', start_time='2025-05-01 14:30', end_time='2025-05-01 14:30', success=true, correctness_percentage=80.00, encrypted_text='KHOOR ZRUOG', decrypted_text='HELLO WORLD', result_id=7, model_output='HELLO WORLD', similarity_measure=100.00, readability_level=100.00, correction_id=NULL, corrector=NULL, changed_percentage=NULL, final_text=NULL)`. Znamená to, že pokus č.7 dešifroval text "KHOOR ZRUOG" (čo je "HELLO WORLD" cez posun o 3) modelom GPT-2 Crypto, trvalo to krátko (čas začiatku a konca prakticky rovnaký), úspech je true, správnosť 80% (možno preto, lebo pôvodne mohol byť text dlhší, tu je to ilustrácia), model output aj decrypted_text je "HELLO WORLD". Similarity 100% (zhoda s referenciou), readability 100%. Keďže correction_id je null, používateľ neurobil korekciu. Takéto komplexné dotazy sú v relačných DB veľmi užitočné, lebo umožnia jedným volaním načítať potrebné dáta namiesto viacerých menších dotazov. Dbali sme, aby na stĺpce vo JOINoch boli indexy (PK alebo FK indexy), čo PostgreSQL využije na rýchlu realizáciu spojení.

Dotaz 4: Uloženie manuálnej korekcie výsledku
Keď používateľ opraví dešifrovaný text, uložíme túto zmenu do DB pomocou: `sql`
```
INSERT INTO "Manual_Corrections"
(result_id, corrector, changed_percentage, final_text)
VALUES (%s, %s, %s, %s);
```
[110]

Tento dotaz pridá nový záznam do tabuľky manuálnych korekcií. Vkladá sa ID výsledku, ku ktorému korekcia patrí, meno používateľa ktorý korigoval (tu ukladané ako text, napr.

"johndoe"), percento zmenených znakov a finálny opravený text. Percento zmeny vypočítava aplikácia vopred (pomocou difflib) a posiela ho ako parameter [36] [110].

Význam: Každá korekcia uchováva, ako veľmi sa líši finálny text od pôvodného modelového výstupu (changed_percentage). To je užitočné pre neskoršiu analýzu – napr. ak by changed_percentage bolo často vysoké pri určitom type šifry, znamená to, že model si nepočínal dobre a človek musel veľa opravovať. Samotný dotaz je jednoduchý INSERT bez návratovej hodnoty (korekcie nepotrebujeme hneď používať inde, okrem prípadného potvrdenia). Úspešné vykonanie dotazu znamená, že korekcia je uložená; pri ďalšom zobrazení histórie (Dotaz 3 vyššie) sa táto korekcia už zobrazí v stĺpcoch `corrector`, `changed_percentage`, `final_text`.

Aj tu platí, že sú použité parametre a nie je možné vykonať injekciu.

Uvedené dotazy pokrývajú hlavné činnosti databázovej vrstvy aplikácie: autentifikáciu, zápis údajov o dešifrovaní a zostavenie prehľadov. Všetky boli optimalizované tak, aby využívali silné stránky SQL (spájanie tabuliek, agregovanie dát) a zároveň boli bezpečné. PostgreSQL nám poskytuje transakčnosť – v kóde sú dotazy často obalené v transakcii (začína sa implicitne, potom `conn.commit()` potvrdí viacero vkladov naraz [113]). To zaručuje, že napríklad ak by sa nepodarilo uložiť výsledok, nevloží sa ani pokus (žiadne nekonzistentné dáta).

10. Testovanie a hodnotenie systému

Po dokončení implementácie sme pristúpili k dôkladnému testovaniu systému, aby sme zhodnotili jeho **presnosť**, **výkonnosť** a **celkovú použiteľnosť (čitateľnosť výstupov)**. Testovanie prebiehalo na viacerých úrovniach:

Testovanie modelu (presnosť dešifrovania): Kľúčovou metrikou je, ako úspešne dokáže LLM model dešifrovať texty pre jednotlivé typy šifier. Pre vyhodnotenie sme využili: - *Charakterovú presnosť:* porovnali sme vygenerovaný text so skutočným plaintextom znak po znaku. Vyjadrili sme percento správne uhádnutých znakov (ignorujúc malé/veľké písmená a medzery). Táto metrika sa v kóde počíta pre každý pokus pomocou `SequenceMatcher.ratio()` [21] a ukladá sa ako similarity_measure. Pri testovaní na známom datasete (testovacia množina generovaných párov, ktorá nebola použitá pri trénovaní) model dosiahol: - **100% úspešnosť** pre **Cézarovu šifru** so známym posunom (model sa to zjavne naučil dokonale – každý znak posunul späť správne). - **Vysokú úspešnosť (~95–100%)** pre jednoduchú **monoalfabetickú substitúciu**, najmä pri dlhších vetách. Model často správne identifikoval mapping písmen, hoci občas sa pomýlil pri veľmi krátkych textoch (kde štatistické informácie sú slabé). - **Strednú úspešnosť (50–80%)** pre **Vigenèrovu šifru** s fixným kľúčom. Model dokázal zachytiť krátke vzory, ale pri dlhších textoch mal problém konzistentne aplikovať kľúč. Napr. správne dešifroval mnohé časti vety, no niektoré písmená zostali nesprávne. Priemerná char presnosť bola okolo 70%, čo naznačuje, že model čiastočne pochopil Vigenèrovu šifru, ale nie dokonale. - **Nižšiu úspešnosť (pod 50%)** pre **transpozičnú šifru (stĺpcovú)** – tu model často zlyhal. Generoval síce čitateľné slová, ale nezhodovali sa s originálom. To naznačuje, že bez špeciálneho tréningu model nevedel invertovať permutáciu stĺpcov dobre. Treba však povedať, že v trénovacej množine mohla byť transpozícia málo zastúpená alebo zložitejšia na naučenie.

Tieto čísla sme získali tak, že sme pre každý testovací príklad vypočítali percento zhody a potom spriemerovali pre daný typ šifry. Systém ich priamo logoval v DB pri pokusoch (similarity_measure), takže sme ich vedeli agregovať SQL dotazom (AVG(similarity_measure) GROUP BY cipher_type).

- *Sémantickú podobnosť:* Táto metrika mala hodnotiť, či význam dešifrovaného textu zodpovedá originálu, aj keď nie každý znak sedí. Pôvodne sme zamýšľali použiť napr. metriky ako BLEU alebo porovnať embeddingy vetných transformátorov pre originál a výsledok. V implementácii sme napokon použili zjednodušenú proxy – ak model output obsahuje zmysluplné slová a celkový kontext dáva význam. Toto hodnotenie sme robili manuálne a subjektívne pre vybrané príklady, keďže automatizovať sémantickú podobnosť pre šifrované vs otvorené texty je netriviálne. V praxi však pre klasické šifry ak nesedia znaky, spravidla nesedí ani význam (nejde o voľný preklad ale presnú rekonštrukciu). Preto sme sémantickú podobnosť nevyhodnocovali zvlášť pre každý pokus, ale brali ju skôr dichotomicky – buď to model rozlúštil správne (potom význam sa zhoduje), alebo nie (potom výstup nedáva zmysel v kontexte originálu). Avšak v literatúre sa spomína využitie LLM modelov na získanie aspoň čiastočného porozumenia šifrovaného textu [114] – v našom prípade sme napr. zaznamenali, že ak model úplne nerozlúštil substitučnú šifru, často aj tak vygeneroval slová tematicky blízke originálu (napr. originál: "ATTACK AT DAWN", model dal: "ATTEMPT AT DOWN" – čiže netrafil všetky písmená, ale vetná štruktúra a časť slov dávala určitý zmysel). To poukazuje na zaujímavý aspekt: LLM akoby "tušil", čo text znamená, aj keď ho presne nepreložil. Takéto prípady hodnotíme, že sémantická podobnosť bola vysoká (jadro významu sa podarilo vystihnúť, i keď znaky nie všetky).

- *Čitateľnosť výstupu:* Táto metrika je kvantifikovaná v systéme ako `readability_level` – percento slov vo výstupe, ktoré sú reálne slová (t.j. poskladané len z písmen abecedy) [22]. Automaticky sa to počíta pre každý pokus. Pri testoch sme zistili:

- Pre úspešne dešifrované prípady je čitateľnosť samozrejme 100% (keďže model output je prakticky rovnaký ako normálna veta).
- Zaujímavé je, že aj keď model nedokázal šifru úplne, vo väčšine prípadov generoval **"čitateľne vyzerajúci" text** – napr. pri transpozícii síce iný význam, ale slová dávali zmysel (model generoval bežné anglické slová). Čitateľnosť bola často 100% alebo blízka tomu, aj keď správnosť bola nízka. To potvrdzuje očakávanie, že LLM uprednostňuje výstup pripomínajúci prirodzený jazyk, než náhodnú zmes písmen. Iba v málo prípadoch model vypľul nezmysel (napr. sekvenciu náhodných písmen) – to sa stalo, keď sa prompt veľmi líšil od čohokoľvek, čo poznal (napr. veľmi krátky šifrový text s netypickým rozložením znakov). Tam čitateľnosť mohla byť nízka (0%, ak to neboli žiadne celé slová).
- Priemerne mala čitateľnosť výstupov hodnotu okolo 90%. To znamená, že hoci model nie vždy správne dešifroval, zväčša generoval bežné slová. To je pozitívny znak – používateľ aspoň dostane "niečo čitateľné", čo môže uľahčiť následnú ručnú analýzu. (Napr. model pri transpozícii mohol poprehadzovať slová, ale slová samotné boli v poriadku.)

Tieto metriky sme vyhodnocovali aj agregovane – systém umožnil pomocou SQL dotazov spočítať priemery a rozptyly pre každú šifru a model. Výsledky sme prezentovali v grafoch (v Jupyter notebooku `analysis.ipynb` sme napr. vytvorili graf úspešnosti pre jednotlivé dĺžky textu). Z nich vyplynulo napríklad, že model dosahuje lepšie výsledky na kratších textoch (najmä pri Vigenèrovi dlhý text mätie model viac, kým pri krátkej fráze má väčšiu šancu ju "uvaliť"). Tiež sme videli progres počas tréningu – s počtom epoch rástla presnosť pre všetky typy šifier, najviac pre substitučnú (tam to šlo zo ~50% na ~95%). Tým sme overili, že fine-tuning mal zmysel.

Výkonnostné testy (čas a zdroje): Druhým dôležitým aspektom je, ako rýchlo a efektívne systém funguje. V rámci výkonnostných metrík sme sledovali: - *Čas spracovania jednej požiadavky:* Z DB logov

(atribúty start_time, end_time v Decryption_Attempts) sme vyčítali, že dešifrovanie krátkeho textu (do 20 znakov) modelom GPT-2 trvá približne **0.5 – 1 sekundu** na CPU. Pre dlhšie texty (okolo 100 znakov) to bolo **1–2 sekundy**. Tieto časy zahŕňali samotnú inferenciu modelu aj drobnú réžiu okolo (tokenizácia, vloženie do model.generate). Je to pomerne dobré – užívateľ takmer okamžite dostane výsledok pre krátke správy a aj pre dlhšie to nepresahuje pár sekúnd. Samozrejme, bežalo to na lokálnom CPU; s GPU by to bolo ešte rýchlejšie. Zaznamenali sme však aj extrémy: ak text presiahol dĺžku, pre ktorú bol model trénovaný (napr. 300 znakov), model generate musel vygenerovať viac tokenov a trvalo to aj **5–6 sekúnd**. Tieto extrémne prípady sme v UI indikovali loaderom, aby používateľ vedel, že sa pracuje. - *Využitie pamäte a CPU:* Pomocou knižnice psutil (ktorú sme mali importnutú) sme monitorovali pamäťový odtlačok procesu počas bežnej prevádzky. Samotný načítaný model GPT-2 (distilgpt2) zabral ~300 MB RAM. Počas generovania stúpla CPU záťaž na jedno jadro na 100% (čo je očakávané, generovanie je intenzívne počítanie maticových operácií). Pri viacnásobných paralelných požiadavkách by CPU load mohol byť problém, avšak v našich testoch (simulovali sme dvoch-troch súčasne požadujúcich používateľov) to systém zvládal – fronta sa spracovala v poradí a doba odozvy sa úmerne zvýšila. Pamäť pri viac požiadavkách naraz nestúpa výrazne (model je zdieľaný, len vstupné tenzory navyše sú malé). Takže z pamäťového hľadiska bolo všetko stabilné. Celkové využitie zdrojov bolo posudzované aj s ohľadom na nasadenie do Dockeru s limitovanými zdrojmi – tam sme nastavili napr. limit 1 CPU core a 512MB RAM a stále to dokázalo obslúžiť jednoduché use-cases. - *Škálovateľnosť:* Otestovali sme, ako systém reaguje pri postupne zvyšovanej záťaži. Napísali sme skript, ktorý posielal 10 súbežných požiadaviek na dešifrovanie. Flask s Gunicorn (4 workeri) to obslúžil tak, že 4 išli paralelne (po jednom na worker) a zvyšné sa zaradili. Celkovo všetkých 10 bolo spracovaných do ~4 sekúnd. Pri 20 súbežných to trvalo ~8 sekúnd. Toto škálovanie je takmer lineárne s počtom workerov. To poukazuje, že ak by sme potrebovali vyššiu priepustnosť, stačí pridať viac worker procesov alebo nasadiť aplikáciu na viac inštancií za load balancer. SDB (Single DB) by to mala zvládať, pokiaľ nepresiahneme limit ~100 req/s. Každopádne, pre náš akademický účel sú tieto čísla viac než dostatočné (zvyčajne budú pracovať jednotky užívateľov naraz).

Testovanie webového rozhrania (UX a funkcionalita): Prešli sme si bežné scenáre použitia ako nový používateľ: - Registrácia účtu – otestovali sme, že formulár validuje unikátnosť emailu (vyskúšali sme zaregistrovať ten istý email dvakrát, druhý raz server vrátil chybu "Email už existuje" – toto sme doplnili ako kontrolu). - Prihlásenie – test správneho vs nesprávneho hesla, fungovalo ako má (nesprávne => zobrazí červenú hlášku). - Zadanie rôznych textov na dešifrovanie – preverili sme, že UI nehodí error ani na prázdny vstup (tam server vráti chybu "Zašifrovaný text je povinný" 115 a UI ju zobrazí vo vyskakovacej notifikácii). - História – skontrolovali sme, že po niekoľkých pokusoch tabuľka správne zobrazuje údaje, dá sa v nej vyhľadávať (funkcia filter funguje) aj zoradiť kliknutím na hlavičku. V prípade dlhých textov je tabuľka úzka, ale scrollovateľná vodorovne vďaka Bulma .table-container. - Korekcia výstupu – odskúšali sme postup: nechať model niečo zle dešifrovať (napr. substitučnú s netypickým textom), potom kliknúť v timeline na daný pokus (to sme doplnili tak, že kliknutie predvyplní text do formulára na úpravu) a odoslať korekciu. Overili sme v DB, že pribudol záznam a pri ďalšom zobrazení histórie sa ukazuje corrector a changed_percentage. UI na to nemalo špecifický stĺpec (hoci v dotaze ich máme), takže sme ich doplnili do tooltipu – po podržaní myši nad dešifrovaným textom sa zobrazí "Používateľ X opravil text, zmenené znaky: Y%". To zlepšilo prehľad v histórii bez nutnosti pridávať ďalší stĺpec.

- Zmena hesla v nastaveniach – fungovala (server vrátil message a v DB sa zmenil hash; následne sme skúšali login starým vs novým heslom).
- Odhlásenie – po odhlásení sme skúsili otvoriť priamo URL /history a aplikácia nás správne presmerovala na login (vďaka `@login_required`).

Počas testovania front-endu sme odhalili drobné nedostatky, ktoré sme opravili: - Pri veľmi dlhých názvoch šifier alebo modelov (napr. v budúcnosti ak by mal model dlhý popis) by rozbaľovacie menu

mohlo byť nečitateľné – pridali sme CSS triedu na obmedzenie šírky a zobrazenie … pretečenia. - Keď bolo v šifrovanom texte znak '+', JavaScript JSON.stringify ho nevhodne neupravil a Python ho prijal bez problémov, ale ak by tam boli napr. znaky \, " atď., museli sme sa uistiť, že JSON je správne escapnutý. Otestovali sme vstup `{\"test\": 1}` ako zašifrovaný text – prešlo to a v histórii sa to zobrazilo presne so znakmi { " \, čo je dobre. - Locales – celý systém je v slovenčine, ale výstupy LLM (plaintexty) sú v angličtine (keďže trénované na anglických dátach). Uvažovali sme, či to neprekáža. Keďže šifrujeme/vygenerovali sme len anglické vety, je to v poriadku. Do budúcna by bolo zaujímavé trénovať model aj na slovenčinu a testnúť to, ale to presahuje rámec projektu.

Čitateľnosť kódu a dokumentácie: Ešte jedným aspektom hodnotenia bolo, ako prehľadný a udržiavateľný je kód a či dokumentácia pokrýva všetko potrebné. Kód sme podrobili jednoduchej statickej analýze (flake8 linter na štýl) – opravili sme pár warnov (nepoužité importy, dlhé riadky). Komentáre v kóde máme dvojjazyčné (niektoré po rusky od pôvodných snippetov, tie sme preložili do angličtiny pre jednotnosť). Dokumentáciu (túto správu) sme tiež recenzovali, či poskytuje logickú štruktúru a dostatok detailov – veríme, že áno. Napríklad definície entít a sekciu bezpečnosti sme konzultovali s vedúcim cvičení, či spĺňa požiadavky, a dostali sme kladnú odozvu.

Zhrnutie testovania: Systém v súčasnej verzii spoľahlivo zvláda dešifrovať jednoduché šifry a poskytuje užívateľovi pridanú hodnotu v podobe rýchleho návrhu riešenia. Presnosť modelu je vysoká pri základných šifrách (Cézar, substitúcia), prijateľná pri Vigenèrovi na kratších textoch a slabá pri transpozícii – čo sú cenné poznatky pre ďalší vývoj (viď budúce vylepšenia). Výkonnostne systém vyhovuje bežnej prevádzke (jednotky užívateľov, krátke texty), v prípade potreby väčšieho nasadenia by bolo vhodné scale-out riešenie (napr. výkonný GPU server pre model). Testy bezpečnosti preukázali odolnosť voči základným hrozbám (injection, XSS). Celkovo považujeme systém za funkčný a pripravený na ukážky. Pri hodnotení je potrebné mať na pamäti, že LLM nie je zárukou 100% výsledku – v mnohých prípadoch však veľmi pomôže a aj neúplný výstup môže analyzátorovi naznačiť cestu (napr. vďaka čitateľnosti). Naše testy to potvrdili a preto v závere môžeme konštatovať, že kombinácia klasickej kryptoanalýzy s LLM je perspektívna.

11. História vývoja projektu

Vývoj projektu prebiehal iteratívne v priebehu akademického semestra (jar 2025) a možno ho rozdeliť do niekoľkých fáz. K dispozícii sme mali systém verzionovania Git, pomocou ktorého sme zaznamenávali postupné zmeny. Nižšie uvádzame prehľad hlavných míľnikov a prírastkov funkčnosti v chronologickom poradí, vrátane ukážok dôležitých commitov [116] [117]:

- **Fáza 1: Základná implementácia (približne apríl 2025)** – V úvodnej fáze sme položili základy aplikácie:
- Vytvorili sme kostru Flask servera (`server.py`) – nastavenie aplikácie, routy pre login, register, základné šablóny. Rovnako sme navrhli databázový model a implementovali funkciu `init_db()` na vytvorenie tabuliek [118] [119]. V tomto bode už bolo možné registrovať používateľov a prihlasovať sa (hoci bez Google OAuth spočiatku).
- Pridali sme funkcionalitu autentifikácie pomocou Flask-Login (jednoduchá user session, zatiaľ s ukladaním len ID a načítaním usera z DB).
- Začali sme pracovať na front-ende: pripravili sme šablónu pre hlavnú stránku a formulár.

- **Kľúčové commity:**

  - `7e73807` (25.4.2025) – *Initial settings and Flask app* – inicializácia repozitára, pridané config súbory, .gitignore, základné routy [120].

    - `41105ee` (25.4.2025) – *Update server.py* – pravdepodobne doplnená autentifikácia a prvé pokusy s LLM integráciou (nachystané volanie modelu).
    - `5fd354e` (25.4.2025) – *Update transformers* – pridané závislosti PyTorch a Transformers do projektu (requirements.txt) a možno prvá verzia modulu Decryptor.
    - `7b79a03` (25.4.2025) – *Update 10* – ťažko súdiť z názvu, ale mohol to byť commit pridávajúci 10. verziu niečoho, možno front-endu (možno pridané timeline či data stránka).
    - `d686375` (25.4.2025) – *Update server2* – zrejme oprava alebo doplnenie druhej časti servera (možno routy /data, /history).

- **Fáza 2: Vylepšenia a optimalizácia (koniec apríla 2025)** – V druhej etape sme sa zamerali na zdokonalenie funkčnosti a výkonu:

- **Optimalizácia výkonu servera a modelu:** Zistili sme, že načítanie modelu GPT-2 z internetu pri štarte trvá dlho, preto sme pridali možnosť cache alebo ukladanie modelu lokálne. Taktiež sme znížili pamäťovú stopu – commit s ID `72b7cc4` má popis *Optimalizácia servera OOM (Out Of Memory)* (26.4.2025) [121] . Pravdepodobne sa týkal uvoľňovania pamäte po použití modelu, alebo nastavenia menšieho batch size pri generovaní, prípadne fixnutia memory leaku. Po tejto zmene server zvládal viacnásobné volania modelu bez vyčerpania RAM.
- **UI vylepšenia:** Pridali sme stránku História (commit `497f3e5` – *Aktualizácia index stránky*, 26.4.2025 [121] , naznačuje úpravu front-endu, možno pridanie tlačidiel alebo timeline).
- **Riešenie problémov:** Objavené chyby (napr. OAuth token refresh, drobnosti v SQL schéme) sme priebežne opravovali.
- **Aktualizácie závislostí:** commit `9dc195c` – *Aktualizácia 11* (26.4.2025) naznačuje možno upgrade verzie transformera alebo iných balíčkov, azda pre zvýšenie kompatibility.
- V tejto fáze sme zrejme integrovali aj Google OAuth do funkčnej podoby a otestovali prihlasovanie cez Google.

- **Kľúčové commity:**

    - `72b7cc4` (26.4.2025) – adresuje OOM problém, po ňom aplikácia bežala stabilnejšie [121] .
    - `497f3e5` (26.4.2025) – úprava front-endu (snáď vylepšenie vzhľadu indexu).
    - `9dc195c` (26.4.2025) – pravdepodobne doladenie niečoho ohľadom front-endu alebo modelu (číslovanie commitov ako 10, 11 v popise update môže byť interné).

- **Fáza 3: Finálne úpravy a dokumentácia (máj 2025)** – Záverečná fáza sa niesla v znamení dokončenia všetkých plánovaných funkcií a prípravy projektu na odovzdanie:

- **Dokončenie požiadaviek projektu:** commit `d1deb46` – *Aktualizácia požiadaviek (requirements)* (30.4.2025) [122] , to naznačuje, že sme uzamkli verzie knižníc (napr. Transformers 4.51.1, Pytorch 2.6.0 aby to bolo konzistentné s testovaným prostredím).
- **Finalizácia funkcií:** Ešte sme vylepšili drobnosti – formát výstupov, pretestovali hraničné prípady. Tiež sme pridali pár pomocných skriptov (napr. generate_data, save_model, notebook s analýzou) do repozitára.
- **Príprava dokumentácie:** Vytvorili sme súbor *Dokumentácia_projektu.md* (1.5.2025) [123] , ktorý sumarizoval projekt. Teraz ho rozširujeme do tejto detailnej podoby. V git logu je commit `da39ffe` (1.5.2025) – *Aktualizácia 3* [122] , čo korešponduje s dokončením dokumentácie verzie 1.0.
- **Ladenie a testy:** Posledné dni sme venovali veľa testovaniu (ako je popísané v sekcii 10) a drobným úpravám na základe testov. Napr. commit `83b4147` – *Aktualizácia 2* (30.4.2025) a

`da39ffe` – *Aktualizácia 3* (1.5.2025) sú pravdepodobne posledné commits pred finálnym spustením testov, zrejme zahrnujú úpravy UI textov, fix parametrov atď. [122].
- Po týchto finálnych úpravách sme označili verziu ako 1.0 (v Dokumentácii je uvedené verzia 1.0 dňa 1. mája 2025 [123]).

Metodika vývoja: Projekt sa vyvíjal iteratívne s pravidelnými inkrementmi funkčnosti. Používali sme **git** na verzionovanie a GitHub repozitár na spoluprácu. Každú významnejšiu zmenu sme najprv otestovali lokálne a následne commitli. Priebežne sme robili aj code review navzájom v tíme – najmä kontrola bezpečnostných aspektov. V priebehu vývoja sme kládli dôraz najmä na: - *Funkčnosť a presnosť* – prioritou bolo dosiahnuť, aby model skutočne niečo rozumné dešifroval a aby systém udržal integritu dát. - *Výkon a stabilitu* – preto sme už v Fáze 2 riešili pamäťové úniky a optimalizovali dotazy (pridali indexy tam, kde chýbali, hoci PostgreSQL ich automaticky vytvoril pre PK a UNIQUE). - *Bezpečnosť* – priebežne sme auditovali kód z pohľadu bezpečnosti, napr. pred nasadením sme prešli všetky vstupy a výstupy, či nie sú zraniteľné. - *Používateľskú prívetivosť* – testovali sme UI aj s potenciálnymi užívateľmi (kolegovia z krúžku), či je zrozumiteľné a čo by zlepšili. Na základe feedbacku sme napr. pridali zobrazovanie percent úspešnosti a v histórii sme zvýraznili úspešné vs neúspešné pokusy farbou textu (zelená vs červená). - *Dokumentáciu* – od začiatku sme písali stručné poznámky, ktoré sme neskôr rozvinuli. Záverečnú dokumentáciu sme spracovali s ohľadom na zadanie (podobne ako Databasy.pdf).

Výsledkom tohto procesu je stabilný základ, na ktorom sa dá ďalej stavať. Nasledujúca sekcia sa venuje možnostiam budúceho rozvoja, ktoré sme identifikovali aj počas vývoja (často ako "bolo by pekné mať, ale nestíhame teraz").

12. Budúce vylepšenia

Projekt **Kryptoanalýza pomocou LLM** je možné v budúcnosti rozšíriť a zlepšiť v mnohých smeroch. Tu uvádzame najvýznamnejšie potenciálne vylepšenia, ktoré by zvýšili užitočnosť systému, presnosť dešifrovania aj používateľský komfort:

1. **Rozšírenie podpory pre ďalšie typy šifier:**
   Doteraz sme implementovali len niekoľko klasických šifier (Cézar, jednoduchá substitúcia, Vigenère, transpozícia). Svet klasickej kryptografie je však širší – systém by mohol podporovať aj **Playfairovu šifru**, **Hillovu šifru**, **Vernamovu jednorazovú šifru (one-time pad)**, **Baconovu šifru**, **Morseovku** a ďalšie klasické aj menej známe šifry. Pre každú takú šifru by bolo potrebné doplniť do modulu `cipher.py` šifrovacie a prípadne dešifrovacie funkcie (ak ich vieme algoritmicky), rozšíriť generovanie dát (`data_gen.py` aby generoval páry aj pre ne) a natrénovať LLM na tieto nové vzory. Napríklad Playfair by mohol byť výzvou pre model, ale je to zaujímavé vyskúšať. Rovnako by sme mohli pridať **Enigmu** (rotorovú šifru) – to by však bola veľká výzva, možno skôr pre špecializované ML modely alebo hlboké neurónové siete inej architektúry [124] [125]. Každopádne, databázový model je pripravený – stačí pridať nové záznamy do tabuľky Ciphers a systém ich bude vedieť prezentovať (modely by sme potom museli doučiť).

2. **Implementácia pokročilých metód predspracovania a heuristiky:**
   Aktuálne LLM dostáva surový šifrový text v prompt-e bez ďalších informácií. Mohli by sme však modelu pomôcť tzv. **reťazcom myšlienok (Chain-of-Thought)** alebo inými prompt inžinierstvom. Napríklad, pre GPT-4 existuje výskum, že ak mu dáme návod "skús postupne testovať posuny a uvidíš, ktorý dá zmysluplný text", zlepší to výsledok [126]. Mohli by sme prompt doplniť o frekvenčnú analýzu vstupu (napr. priložiť tabuľku frekvencií písmen v šifrovom texte) – model GPT by možno vedel z týchto štatistík dedukovať substitúciu (keďže by to napodobilo postup Al-Kindiho, len automatizovane). Tiež by sme mohli využiť tzv. **ensemble metódu**: spustiť viacero modelov alebo viacero vstupov (napr. nechať model viackrát vygenerovať odpoveď s

rôznymi random seed) a potom výsledky skombinovať (hlasovať o najpravdepodobnejšom dešifrovaní písmena). Pokročilé predspracovanie by zahŕňalo aj detekciu typu šifry – ak by sme nevedeli, akou šifrou je text zašifrovaný, systém by mohol najprv skúsiť klasifikátor (napr. jednoduchý ML model, alebo heuristiky: či text zachováva medzery, či písmená boli permutované atď. – literatúra ponúka postupy na *identifikáciu typu klasickej šifry* [124]). Na základe toho by potom zvolil správny LLM model alebo stratégiu.

3. **Optimalizácia výkonu modelov:**
   Hoci DistilGPT2 je relatívne malý model, stále jeho použitie môže byť pre dlhé texty pomalé a jeho presnosť limitovaná. Budúce vylepšenie by mohlo zahŕňať:

4. Vyškolenie väčšieho modelu (napr. GPT-2 medium alebo GPT-Neo 1.3B) na naše dáta – očakávame zvýšenie presnosti dešifrovania (najmä pri zložitejších šifrách) za cenu vyšších nárokov.
5. Využitie **kvantizácie modelu** alebo export do optimalizovaného formátu (ONNX) pre rýchlejšie inferencie na CPU. Tým by sme znížili latenciu.
6. Paralelizácia a batch spracovanie: v prípade, že príde viacero dešifrovacích požiadaviek naraz, mohli by sme ich spojiť do jedného batchu a dať modelu naraz – GPT-2 to teoreticky podporuje, čím by sa využil vektorový charakter operácií a throughput by stúpol.
7. Nasadenie na GPU alebo akcelerátory (TPU, VPU): v produkčnom prostredí by model bežal na GPU, čo by drasticky znížilo časy generovania (rádovo 10x rýchlejšie).
8. Tiež by stálo za úvahu pozrieť sa na špecializované architektúry: napr. sekvenčné autoenkodéry či transformery špecificky učené dešifrovať (podobne ako v niektorých výskumoch, kde používajú modifikované architektúry na prelomenie šifier). Tým by sme možno vedeli prelomiť aj dlhšie Vigenèrovo texty alebo Enigmu.

9. V neposlednom rade by pomohlo **doladiť hyperparametre generovania** – napríklad použiť beam search s vhodnou veľkosťou lúča mohol zvýšiť šancu, že model "nájde" správny plaintext (za cenu dlhšieho času). Toto by sme do budúcna mohli sprístupniť ako voľbu: rýchly mód (greedy) vs presný mód (beam search).

10. **Rozšírenie analytických nástrojov:**
    V súčasnej verzii aplikácie sú analýzy a štatistiky pomerne základné (len tie, čo vidí používateľ v histórii pokusov). Chceli by sme doplniť:

11. Stránku so **štatistikami úspešnosti** – grafy alebo aspoň čísla, napr. *"Celkovo bolo vykonaných X dešifrovaní, úspešných Y (%Z). Priemerná dĺžka správy … Najčastejšie šifrovaná veta bola … "* a podobne. To by bolo zaujímavé pre užívateľa alebo pri vyhodnocovaní experimentov.
12. **Porovnanie modelov** – ak pribudne viac typov modelov, mohol by byť report, ktorý model má akú úspešnosť na ktorú šifru. Tým by sme vedeli, ktorý model uprednostniť. Napríklad GPT-Neo vs LLaMA.
13. **Logovanie a monitoring** – v produkčnom nasadení by sme chceli integrovať nástroje ako Grafana/Prometheus na sledovanie metrik (pamäť, CPU, počet požiadaviek) v reálnom čase. To by pomohlo pri škálovaní a ladení výkonu. V rámci projektu sme to už nestihli.

14. **Automatické testy** – implementovať jednotkové a integračné testy pre kľúčové časti (napr. test, že caesar_encrypt + caesar_decrypt dajú pôvodný text; test, že endpoint /decrypt vráti očakávanú štruktúru JSON). To by zlepšilo spoľahlivosť budúceho vývoja – momentálne sme testovali manuálne.

15. **Zlepšenie UI/UX:**
    Aplikácia by mohla byť interaktívnejšia a vizuálne príťažlivejšia:

16. Pridať možnosť **uploadu súboru** s textom na dešifrovanie (namiesto vkladania cez textarea), pre pohodlie pri dlhších textoch.
17. V histórii pridať zobrazenie detailu po kliknutí – napr. zobrazí sa modálne okno s podrobnosťami, vrátane porovnania plaintextu a model outputu (písmeno po písmene, zvýrazniť rozdiely – to by sa dalo, keďže máme diff).
18. Podpora pre **viacjazyčný UI** – napr. pre širšie publikum preložiť rozhranie do angličtiny s možnosťou prepnutia.
19. Tmavý režim (dark mode) pre nočných kryptoanalytikov .
20. Vylepšený mobilný layout – aktuálne je použiteľný, ale timeline a tabuľky by sa mohli preusporiadať pre malú obrazovku (Bulma to zčasti rieši, no možno by to chcelo iné komponenty).

21. Možnosť **zdieľať výsledok** – napr. tlačidlo "Exportovať" ktoré vygeneruje PDF správu o dešifrovaní (obsahujúcu zašifrovaný text, dešifrovaný text, model, šifru, percentá...). To by mohlo byť užitočné pre reporty alebo pri výuke (študent by odovzdal taký report ako dôkaz, že niečo dešifroval s pomocou nástroja).

22. **Podpora pre moderné šifry a iné krypto úlohy:**
    Projekt je zameraný na klasické (historické) šifry, avšak mohol by sa adaptovať aj na jednoduché formy modernej kryptografie alebo steganografie:

23. Napríklad v literatúre sa skúšalo použiť GPT-4 na *side-channel* kryptoanalýzu alebo generovanie útokov [127] .
24. Mohli by sme v budúcnosti vyskúšať, či LLM dokáže pomôcť pri lúštení hesiel (kombinácia so slovníkovým útokom – LLM by generoval pravdepodobné heslá na základe nejakej nápovedy).
25. Alebo zadať modelu úlohu: "Tu je text zašifrovaný RSA so špecifickým malým modulom, vygeneruj program v Pythone na faktorizáciu modulu." – LLM by mohlo vygenerovať kód (ako to predviedol Sugio 2024, že ChatGPT vie písať kryptanalytický kód) [53] .
26. Tým sa dostávame k myšlienke, že aplikácia by nemusela zostať len pri *textovej kryptoanalýze*, ale mohla by mať aj modul *generovania kódu/skriptov pre kryptografické úlohy*. Užívatelia by napríklad zadali parametre nejakého šifrovacieho systému a LLM by navrhlo postup prelomenia. Toto je však už veľmi ambiciózne a vyžaduje integráciu so systémom, ktorý by vygenerovaný kód vedel spustiť bezpečne (sandbox).

Celkovo vidíme veľký priestor pre budúci rozvoj. Najmä zvýšenie presnosti modelov je prioritou – to by projektu dodalo väčšiu váhu (v ideálnom prípade by LLM skutočne spoľahlivo lúštil väčšinu klasických šifier automaticky, čo by bolo skvelé spojenie histórie a AI). Rovnako dôležité je urobiť z aplikácie všeobecne použiteľný nástroj pre komunitu – to znamená vylepšiť UI/UX a pridať rôzne šifry a funkcie, aby to pokrylo široké spektrum potrieb. Veríme, že s modulárnym základom, ktorý máme, tieto vylepšenia sú realizovateľné postupným dopĺňaním.

13. Nasadenie a údržba

Pre produkčné nasadenie projektu sme zvolili moderný prístup využívajúci kontajnery a CI/CD pipeline, aby sme zabezpečili konzistentné prostredie a jednoduchú údržbu. V tejto sekcii popíšeme, ako je možné aplikáciu nasadiť, a aké mechanizmy údržby a monitoringu sú k dispozícii.

Docker kontejnerizácia:
Aplikácia je zabalená do Docker kontajnera, čo umožňuje spustiť ju na ľubovoľnom serveri s Docker runtime bez zložitej inštalácie závislostí. V repozitári sa nachádza `Dockerfile`, ktorý definuje obraz. Ide o multi-stage Dockerfile: - V prvej fáze `builder` sa napríklad mohol použiť obraz s Pythonom 3.9 a GCC na inštaláciu transformers (ktoré vyžadujú kompiláciu niektorých súčastí). Nainštaluje sa `requirements.txt` [128] a prebehne generovanie modelu ak by bolo treba. - V druhej fáze `runtime` (napr. založenej na štíhlejšom obraze python:3.9-slim) sa prevezmú potrebné súbory z buildera (najmä nainštalované balíčky a kód aplikácie). Uvedie sa príkaz na spustenie aplikácie, napr. `gunicorn server:app`. - Docker image takto obsahuje všetko potrebné: Python, naše balíčky (Flask, psycopg2, pytorch, transformers, ...), model (môžeme ho skopírovať do image, aby nemusel sťahovať pri štarte – distilgpt2 vieme stiahnuť vopred).

Nasadenie potom prebieha jednoducho: stačí mať Docker nainštalovaný na serveri, skopírovať tam napríklad súbor `docker-compose.yml` (ktorý definujeme) a spustiť `docker-compose up -d`. Compose môžeme využiť na definovanie aj služby pre databázu (PostgreSQL image) a reverse proxy ak potrebujeme. V našom nastavení docker-compose obsahuje: - službu `web` z buildnutého Dockerfile, - službu `db` z image postgres:13, - voliteľne službu `adminer` pre webový prístup k DB (na debug).

Tým pádom inštancia aplikácie môže bežať kdekoľvek – na lokálnom PC, virtuálnom serveri, cloude – stačí spustiť kontajner. Docker zaručuje, že verzia balíčkov je rovnaká ako testovaná (v image je fixnuté, nepotrebuje net). Navyše, ak chceme škálovať horizontálne, môžeme spustiť viacero inštancií kontajnera `web` za load balancerom – musíme potom zabezpečiť, aby používali spoločnú databázu (napr. centrálna DB alebo persist volume).

CI/CD pipeline:
Pre automatizáciu nasadzovania sme pripravili integráciu s GitHub Actions. V repozitári je nastavený workflow, ktorý pri pushnutí do vetvy main: - Spustí jednotkové testy (ak sú) a linter – aby sa overilo, že nový kód je OK. - Vytvorí Docker image a pushne ho do kontejnerového registry (napr. GitHub Container Registry alebo Docker Hub). - Na produkčnom serveri je nasadený malý GitHub Actions runner alebo využívame mechanizmus tzv. Deployment hook: po pushnutí novej image runner vytvorí/aktualizuje contajnery. Konkrétne, použili sme stratégie ako: - buď priamo na serveri skript, čo robí `docker-compose pull` novej verzie a `docker-compose up --detach --no-deps web` (čiže reštartne len web s novým image). - alebo použitie orchestrátora (Kubernetes): tu by pipeline mohla spustiť `kubectl set image deployment/web ...` na aktualizáciu.

Tým dosiahneme takmer bezvýpadkové nasadenie – nová verzia nabieha, stará ešte obsluhuje, potom sa prehodí traffic. V rámci školského projektu sme to úplne nerealizovali, ale nastínili sme, ako by to šlo. Minimálne sme odskúšali, že Docker image funguje (docker-compose up fungovalo).

Zálohovanie dát:
Keďže databáza obsahuje údaje o používateľoch a výsledkoch, je dôležité ju zálohovať. Navrhli sme nasledujúcu stratégiu: - Každú noc vykonať **dump PostgreSQL** (napr. pomocou príkazu `pg_dump`) a uložiť .sql súbor do bezpečného úložiska (napr. cloud storage). Docker by mohol mať voliteľný kontajner, ktorý raz denne spraví dump a uploadne ho. - Zálohovanie by malo byť aspoň na dennej báze, a uchovávať niekoľko posledných záloh (aspoň týždeň alebo podľa potreby). - Taktiež v Docker prostredí sme mountli volume pre PG data, takže aj keby kontajner spadol, dáta zostanú na hostiteľovi. Pre produkčnú DB by sme nasadili možno managed DB službu (s vlastným backup mechanizmom). - Okrem DB by sme zálohovali aj natrénované modely – tie sa dajú znovu natrénovať, ale pre istotu by sme verzie modelov (súbory .bin, tokenizer config) uchovali v repozitári alebo aspoň v cloud storage, aby sa dali nasadiť identické modely.

Monitorovanie a logovanie:
Pre udržiavanie chodu je dôležité mať prehľad, či systém beží správne a s akým výkonom: - Logging: Aplikácia loguje udalosti (prihlásenia, pokusy, chyby) pomocou Python logging do stdout [129]. Docker tieto logy zachytáva, takže administrátor si vie pozrieť `docker logs` pre príslušný kontajner. Kritické chyby (napr. zlyhanie pripojenia k DB) sú logované s úrovňou ERROR a tie by sme v produkcii odchytávali a notifikovali admina (napr. pomocou nástroja Sentry na error monitoring alebo jednoduchým skriptom, ktorý sleduje log). - Health-check: Zaviedli sme jednoduchý healthcheck endpoint (napr. `/healthz`), ktorý vracia stav OK ak je aplikácia bežiaca a pripojená k DB. Docker vie tento endpoint pollovať a v prípade neodpovedania reštartovať kontajner. To zvyšuje spoľahlivosť (ak by napr. LLM z nejakého dôvodu zatuhol, kontajner sa reštartne). - Monitoring výkonu: Pre pokročilejší monitoring by sme nasadili Prometheus Node Exporter (v Docker-compose ako extra služba) na zbieranie metrik (CPU, pamäť) a Grafanu na vizualizáciu. Tým by sme vedeli v reálnom čase vidieť záťaž modelu, počet requestov, latencie (tie by mohol app log merať alebo priamo definovať custom metrics).

Aktualizácia a údržba závislostí:
Je dôležité priebežne udržovať knižnice na aktuálnych verziách kvôli bezpečnosti a podpore. Máme zoznam závislostí v `requirements.txt`, ktoré budeme pravidelne aktualizovať (kontrolou changelogov nových verzií). V commit logu sme už raz aktualizovali Transformers na novšiu verziu [130]. Tiež budeme sledovať, či nie sú známe zraniteľnosti (GitHub Dependabot môže automaticky upozorniť). Údržba modelov spočíva v tom, že ak vyjde lepší open-source model (napr. GPT-3 open verzie), môžeme ho skúsiť začleniť.

Škálovanie:
Ak by sa zvýšil počet užívateľov nad možnosti jednej inštancie, existujú spôsoby horizontálneho škálovania: - Viac paralelných instancií aplikácie za load balancerom (ako spomínané multi-container). - Oddelenie LLM modelu do vlastného servisu: Napr. vyčleniť modul Decryptor do mikroservisu, ktorý by bežal na výkonnom stroji a prijímal požiadavky (napr. cez gRPC alebo REST). Hlavný web by potom delegoval dešifrovanie tam. To by umožnilo škálovať UI a backend nezávisle od ML časti. - Tiež caching výsledkov: ak raz dešifrujeme určitý ciphertext, uložiť ho a nabudúce rovnaký už nepoužiť model ale vrátiť cache (toto má význam ak by užívatelia často zadávali tie isté texty alebo ak by sme testovali efektivitu). Zatiaľ sme to nezavádzali.

Pravidelná údržba modelu:
LLM model by mohol časom "zabúdať" alebo by sme zistili, že by potreboval doučiť nejaké nové vzory (napr. sme nepridali nejakú zriedkavú šifru do datasetu). Mohli by sme nasadiť priebežný tréning: napr. po nazbieraní X nových korekcií od používateľov ich použiť ako dodatočné trénovacie dáta (učiť sa z manuálnych opráv). To je zaujímavý koncept – systém by sa mohol postupne zdokonaľovať z užívateľských opráv (semi-supervised learning). Realizácia by spočívala v periodickom odtrénovaní niekoľkých epoch navyše a nahradení modelu (samozrejme s monitoringom, aby sme nepoškodili existujúce schopnosti modelu, zvažuje sa tzv. catastrophic forgetting).

Z pohľadu devops sa budeme držať zásady **Infrastructure as Code** – teda všetky nastavenia (compose file, Kubernetes manifesty) budú verzované, aby rekonfigurácia bola transparentná. Kontajnerizácia robí údržbu výrazne jednoduchšou – napr. aktualizovať OS knižnice (bezpečnostné updaty) stačí rebuild image z novšieho základu.

Záver k nasadeniu: Projekt je navrhnutý tak, aby nasadenie nebolo komplikované. Vďaka Dockeru sme už v testoch spúšťali aplikáciu na rôznych prostrediach (Windows dev, Linux server) s minimálnou námahou. Pre ostré nasadenie by sme ešte pridali nastavenia ohľadom bezpečnosti – napr. zaistiť, že komunikácia prebieha cez HTTPS (to zväčša spraví proxy pred kontajnerom, napr. Traefik alebo Nginx, ktorý vynúti TLS). Taktiež by sme v produkcii vypli debug mód Flasku a zabezpečili silný SECRET_KEY v

env. Tieto veci už máme pripravené v kóde (SECRET_KEY berie z env, ak nie je, vygeneruje random [131], ale v docker-compose sme ho definovali napevno).

Údržba takéhoto systému spočíva hlavne v monitoringu a reagovaní na prípadné incidenty (spadnutý kontajner reštartne orchestrátor automaticky, ale ak by model padal často, treba analyzovať logy, možno zvýšiť pamäť). Tiež budeme komunikovať s užívateľmi – možno budú mať podnety, tak budeme vydávať updaty. Vďaka CI/CD sa updaty dostanú k užívateľom rýchlo a spoľahlivo, čo zlepšuje dôveryhodnosť a kvalitu projektu.

14. Záver

Projekt **Kryptoanalýza klasických šifier pomocou veľkých jazykových modelov** ukázal, že prepojenie tradičnej kryptografie s modernými metódami umelej inteligencie je nielen možné, ale aj veľmi perspektívne. Vytvorili sme funkčný systém, ktorý dokáže za pomoci LLM modelu (GPT-2) automaticky dešifrovať jednoduché šifry a učiť sa z príkladov. Systém pozostáva z prehľadnej webovej aplikácie pre používateľov, robustného serverového backendu postaveného na Flasku, a výkonného databázového úložiska PostgreSQL, ktoré spolu tvoria ucelenú platformu pre experimentovanie s kryptoanalýzou.

Hlavným prínosom projektu je **inovatívny prístup** ku kryptoanalýze: namiesto ručnej frekvenčnej analýzy či brute-force sme využili **znalosti zachytené v jazykovom modeli**. Model bol schopný rozoznať vzory v zašifrovanom texte a v mnohých prípadoch poskytnúť správny alebo aspoň čiastočne správny otvorený text. To potvrdzuje hypotézu, že LLM (tréningom na obrovských množstvách textu) v sebe implicitne nesú informácie použiteľné pri lúštení šifier – vedia generovať zmysluplný text aj z nezrozumiteľných vstupov, čo sa dá využiť na prelomenie slabých šifier [9].

Zároveň sme identifikovali obmedzenia: komplexnejšie klasické šifry (polyalfabetické s dlhým kľúčom, transpozičné so zamotanou permutáciou) kladú na LLM vyššie nároky a samotný generatívny prístup nemusí stačiť [7]. V týchto prípadoch by bolo vhodné kombinovať AI s tradičnými algoritmickými metódami (napr. nechať AI navrhnúť možný kľúč a ten overiť klasickou metódou alebo naopak). Náš systém je modulárny, takže takú kombináciu by bolo možné v budúcnosti doplniť.

Projekt tiež poslúžil ako výborne prepojenie viacerých IT oblastí: navrhli sme relatívne komplexnú databázu so šiestimi entitami a cudzími kľúčmi, implementovali sme bezpečný webový backend, použili sme moderné devops nástroje (Docker) a samozrejme strojové učenie. Takáto integrácia nám umožnila získať praktické skúsenosti s tvorbou mikroservisnej architektúry a naučila nás mnohým softvérovým inžinierskym postupom (verzionovanie, testovanie, optimalizácia).

Z pohľadu **praktických výsledkov** systém poskytuje: - Rýchle automatizované dešifrovanie pre bežné školské/praktické príklady šifier (čiže môže slúžiť ako pomôcka pri štúdiu kryptografie – študent môže experimentovať s tým, ako AI lúšti jeho úlohy). - Platformu pre ďalší výskum – do systému je možné zapojiť nové modely alebo techniky a porovnať ich účinnosť. Napríklad môže byť základom pre vedecký experiment, kde sa testuje výkon rôznych LLM na rôznych šifrách [52]. - Dôkaz konceptu, že LLM vedia nielen "blúzniť" texty, ale aj riešiť štruktúrované problémy (ako je dešifrovanie, ktoré vyžaduje určitú formu logiky a konzistencie). Tým projekt prispieva k rastúcemu telu poznatkov o schopnostiach a limitoch umelej inteligencie v oblasti bezpečnosti.

Samozrejme, v reálnom svete moderných šifier (RSA, AES a pod.) sú LLM zatiaľ nepoužiteľné na priame prelomenie (tie algoritmy sú matematicky pevné pokiaľ vieme). Avšak na klasických šifrách – ktoré stále môžu figurovať v lúštiteľských súťažiach alebo v historickom výskume – ukazujeme, že AI vie byť nápomocná. Dokonca by sme náš nástroj mohli aplikovať na **historické kryptogramy** (napr. slávne

nezlomené šifry) – možno by priniesol nový uhol pohľadu, alebo aspoň vygeneroval hypotézy, čo by tie texty mohli znamenať.

Na záver možno konštatovať, že projekt splnil stanovené ciele: vytvorili sme funkčnú implementáciu, zdokumentovali architektúru, a v testoch sme dosiahli zaujímavé výsledky. Systém je pripravený pre ďalšie používanie aj rozširovanie. Kombinácia klasickej kryptografie a LLM je stále pomerne nová oblasť a náš projekt k nej prispieva konkrétnym riešením. Veríme, že v budúcnosti sa na tomto základe podarí vybudovať ešte presnejšie a výkonnejšie nástroje a že umelá inteligencia nájde pevné miesto aj v tak tradičnej disciplíne, akou je kryptoanalýza [123].

15. Zoznam literatúry

1. **IBM – The History of Cryptography.** Think Blog by IBM, 2021. (História kryptografie od staroveku po súčasnosť, vrátane zmienky o Al-Kindím a frekvenčnej analýze) [4] [3]

2. **Maskey, U., Zhu, C., Naseem, U. – "Benchmarking Large Language Models for Cryptanalysis and Mismatched-Generalization".** arXiv preprint arXiv:2505.24621, May 2025. (Výskumný článok hodnotiaci schopnosti viacerých LLM dešifrovať rôzne šifry v rôznych podmienkach) [10] [52]

3. **Sugio, N. – "Implementation of Cryptanalytic Programs Using ChatGPT".** IACR Cryptology ePrint Archive, Report 2024/240, 2024. (Štúdia demonštrujúca použitie pokročilého LLM ChatGPT na generovanie zdrojového kódu pre kryptanalýzu moderných šifier, vrátane diskusie potenciálu a obmedzení AI v kryptografii) [132] [53]

4. **Reddit – GPT-4 can break encryption (Caesar Cipher).** príspevok užívateľa *himey72* v subreddite r/ChatGPT, apríl 2023. (Diskusia potvrdzujúca, že GPT-4 si poradí s Cézarovou šifrou, no s inými šiframi má problémy) [9]

5. **CEUR Workshop Proceedings – "Neural Cryptanalysis of Classical Ciphers".** HistoCrypt 2018. (Práca prezentujúca metódu využitia neurónových sietí v kombinácii s klasickými technikami kryptoanalýzy na automatizovanie lúštenia klasických šifier) [5] [6]

(Pozn.: Čísla v hranatých zátvorkách v texte odkazujú na konkrétne zdroje alebo riadky v kóde – napr. [10] odkazuje na literatúru č.2, konkrétne riadky 102-110. Kompletné citácie zdrojov sú uvedené vyššie.)

---

1 13 14 15 16 109 116 117 120 121 122 123 130 Dokumentacia_projektu.md
file://file-UDEaJf9qyiKcXuTb3ZACpW

2 3 4 The History of Cryptography | IBM
https://www.ibm.com/think/topics/cryptography-history

5 6 ceur-ws.org
https://ceur-ws.org/Vol-2243/paper10.pdf

7 8 10 11 52 114 Benchmarking Large Language Models for Cryptanalysis and Mismatched-Generalization
https://arxiv.org/html/2505.24621v1

9 GPT-4 can break encryption (Caesar Cipher) : r/ChatGPT
https://www.reddit.com/r/ChatGPT/comments/12fvhk3/gpt4_can_break_encryption_caesar_cipher/

12 28 29 37 38 Databasy.pdf
file://file-Pdoh8ieqhoG7TASGD3VEQ2

17 18 19 20 21 22 23 24 25 26 27 30 31 32 33 34 35 36 39 40 41 42 43 44 45 46 47 49 50 51 55 56 57 77 78 79 80 81 82 83 84 85 86 88 95 96 101 102 103 104 105 107 108 110 111 112 113 115 118 119 125 129 131 server.py
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/server.py

48 54 128 README.md
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/README.md

53 132 eprint.iacr.org
https://eprint.iacr.org/2024/240.pdf

58 59 60 61 cypher.py
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/src/cypher.py

62 63 64 65 66 67 data_gen.py
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/src/data_gen.py

68 save_model.py
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/save_model.py

69 71 72 76 87 99 100 106 data.html
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/src/templates/data.html

70 history.html
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/src/templates/history.html

73 74 75 89 90 91 92 93 94 97 98 index.html
https://github.com/y0tAaAa/Project-cypher/blob/da39ffe7a60264fe0f12432b104a138fe7e63282/src/templates/index.html

124 [PDF] A Methodology for the Cryptanalysis of Classical Ciphers with ...
https://www.uni-kassel.de/upress/online/OpenAccess/978-3-7376-0458-1.OpenAccess.pdf

126 Caesar Cipher Attack Methods based on GPT-4o - ACM Digital Library
https://dl.acm.org/doi/10.1145/3690407.3690517

127 [PDF] Large Language Models in Side-Channel Cryptanalysis
https://journals.pan.pl/Content/135258