FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Architektura systému pohledové inspekce kvality

Studijní program: N0613A140028 – Informační technologie
Studijní obor: N0613A140028AI – Aplikovaná informatika
Autor práce: **Bc. Kevin Daněk**
Vedoucí práce: Ing. Igor Kopetschke

Liberec 2025

Tento list nahraďte
originálem zadání.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.
Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.
Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.
Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.
Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.
Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.
Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

22. 1. 2026 Bc. Kevin Daněk

ARCHITEKTURA SYSTÉMU POHLEDOVÉ INSPEKCE KVALITY

ABSTRAKT

Tato zpráva popisuje třídu `tulthesis` pro sazbu absolventských prací Technické univerzity v Liberci pomocí typografického systému LaTeX.

Klíčová slova: LaTeX, třída, TUL

ABSTRACT

This report describes the `tulthesis` package for Technical university of Liberec thesis typesetting using the LaTeX typographic system.

Keywords: LaTeX, class, TUL

PODĚKOVÁNÍ

Dál bych chtěl poděkovat i zbytku akademické obce, se kterou jsem měl co dočinění. I když se s některými jejími členy, ozvláště pak z nadřízených pozic, neshodnu ani na počasí, tak díky každému jsem získal novou zkušenost. Někdy byla pozitivní, a někdy zase i negativní - to ale k životu patří.
A na závěr musím poděkovat svojí drahé polovičce, Simče Daňkové, která mě podporovala po celou dobu psaní této práce. Na začátku této práce byla mou přítelkyní, a nyní, na jejím konci, je již mojí manželkou. Díky ní si budu tuto práci pamatovat nejenom jako svůj *magnum opus*, ale jako životní milník, kterým se změnil skoro každý aspekt mého života. Pokud Vás zajímají regionální dějiny a příběhy z časů minulých, doporučuji si přečíst její bakalářskou práci, kterou tvořila souběžně s tou mojí, a zabývá se tématem sebevražd po druhé světové válce na Liberecku a obzvláště v Jablonci nad Nisou a jeho blízkém okolí.

OBSAH

SEZNAM ZKRATEK

CD	Continuous Deployment (Postupné nasazení)
CI	Continuous Integration (Postupná integrace)
DevOps	Development Operations
GPG	GNU Privacy Guard
JSON	Javascript Object Notation
JWT	JSON Web Tokens
SecOps	Security Operations
SSH	Secure Shell
TPM	Trusted Platform Module

1 Úvod

Cílem této práce není vytvářet nové softwarové funkce, ale podívat se na systém jako celek z pohledu softwarového inženýrství. Je totiž nezbytné věnovat se technickému dluhu, metodám pro nasazení softwaru a kvalitě kódu, vysvětlit na konkrétních příkladech, proč jsou

2 OSVĚDČENÉ TECHNIKY SOFTWAROVÉHO INŽENÝRSTVÍ

Softwarové inženýrství je disciplína, která formalizuje programovací postupy do sady rad, vzorů a doporučení, které chybováním ostatních inženýrů vykrystalizovali jako nejvhodnější. Programátor vybavený klávesnicí dokáže vytvořit téměř cokoliv, co ho napadne. Stejně jako malíř se štětcem dokáže nakreslit téměř cokoliv si představí. Je ovšem pozoruhodné, že, narozdíl od programátorů, malíři často úmyslně omezí své možnosti. Dodržením určitých pravidel totiž docílí toho, že obraz nebude pouze nic neříkající kaluž čar a barev. Z obrazu se stane myšlenka, kterou další člověk dokáže rozpoznat pomocí svých instinktů.

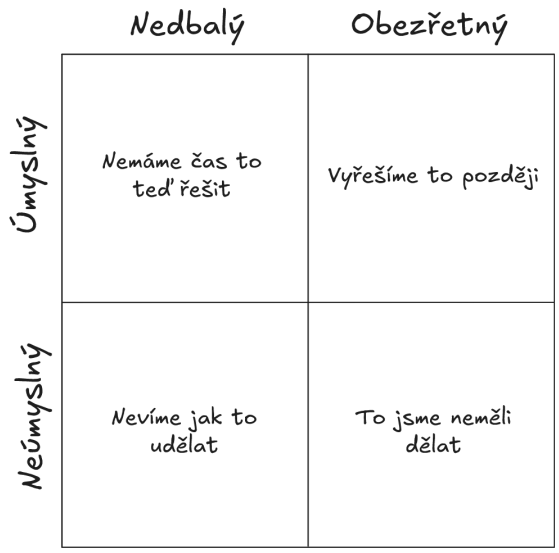
Programování nemá od této myšlenky zas tak daleko. Pokud se ve vývoji zavážeme dodržovat určitá pravidla, umožníme tím ostatním programátorům snáze pochopit, čeho se kód snaží docílit. Ze snáze pochopitelného a předvídatelného kódu benefituje i samotný autor, který může s odstupem času přijít ke kódu zpět a netrávit čas nad jeho luštěním.

Softwarové inženýrství je oborem obsáhlým, a proto jsou v této kapitole pouze vybraná témata, která jsou relevantní pro pozdější části práce.

2.1 ŘEŠENÍ TECHNICKÉHO DLUHU

Technický dluh vzniká v situaci, kdy došlo k nedodržení doporučených postupů či špatnému návrhu za účelem rychlejšího dokončení vývoje. Krátkodobě ušetřený čas se totiž vždy projeví jako dlouhodobé zdržení, obzvláště když je potřeba s kódem dále pracovat. Ušetření času nemusí být nutně negativní věc, je ovšem potřeba se vzniklým technickým dluhem nakládat opatrně.

Martin Fowler ve svém blogu z roku 2009 rozděluje technický dluh do čtyř kategorií, přičemž dluh klasifikuje podle toho, zda-li byl úmyslný či neúmyslný, a jestli byl vývojář v daný okamžit obezřetný či nedbalý. Dluh je úmyslný nebo neúmyslný podle toho, jestli si je programátor vědom toho, že dělá něco špatně. Na druhou stranu dluh je nedbalý nebo obezřetný, pokud programátor nemá, respektive má, vůli a prostředky technický dluh řešit (Fowler, 2009).

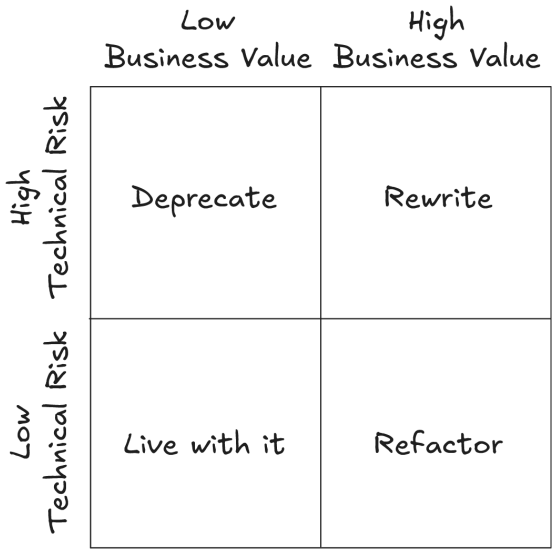


Obrázek 2.1: Kvadranty technického dluhu (Team Asana, 2025)

Tento model nám pomáhá pochopit vznik a druhy technického dluhu, avšak je technický dluh opravdu takový problém? Technický dluh je nazýván dluhem právě proto, že se chová stejně závazek vůči cizímu kapitálu. Čas, který si v daný moment ”půjčíme” pro zrychlení vývoje musí být v pozdější době někým splacen. Je pak na vývojáři, zda-li bude technický dluh splácet průběžně, nebo počká, než mu přímo znemožní další práci.

Jak technický dluh vypadá v praxi? Nízké pokrytí testy, nízká či žádná dokumentace, obří třídy a funkce, hodnoty definované ”na tvrdo” - to jsou jenom některé znaky technického dluhu. Všechny tyto znaky brzdí další vývoj a údržbu kódu.

Jak technický dluh řešit? Pro systém zatížený technickým dluhem je potřeba zvážit rozsah, riziko a rozpočet systému. Pokud systém není důležitý a nepředstavuje jeho výpadek riziko, není potřeba dluh řešit prioritně. Naopak pokud je systém kritický pro softwarový produkt, a je zanesen velkým rizikem z hlediska poruchovosti a udržitelnosti, je lepší celý systém přepsat od začátku. Touto úvahou lze analyzovat i každý dílčí podsystém. V nižších vrstvách abstrakce je potřeba hledět také na závislosti mezi částmi kódu. Pokud má jednotka kódu hodně závislostí, zvedá se tím její rizikový faktor a je potřeba pečlivě zvážit, jak dál postupovat.



Obrázek 2.2: Kvadranty řešení technického dluhu (ForrestKnight, 2025)

2.2 CLEAN CODE

Clean Code je souhrnný název pro sadu pravidel a doporučení, které mají za cíl snížit kongitivní zátěž při čtení a údržbě kódu. Na clean code je spíše pohlíženo jako na princip a myšlenkový pochod, který má za cíl usnadnit práci nejenom ostatním vývojářům v týmu, ale i autorovi v budoucnu, pokud se ke kódu vrátí s odstupem času. Jak píše i samotný Robert Martin ve své knize *Clean Code: A Handbook of Agile Software Craftsmanship*, každý může s jednotlivými zásadami pro *Clean Code* souhlasit či nesouhlasit, nejedná se o absolutní pravdu.

Je ovšem důležité si uvědomit, že cílem *Clean Code* není fanatické dodržování arbitrárních pravidel, která jsou vytesána do kamenných desek jako desatero Božích příkázání. Cílem je zmírnit dopad neudržitelného kódu, často psaného ve spěchu na koleni, na budoucí produktivitu celého týmu. Žádoucím výsledkem je minimální pokles produktivity v čase a vyhnout se velkým přepisům a refaktorům celého projektu.

Clean Code a technický dluh jdou ruku v ruce. Jedna z příčin technického dluhu je právě zanedbání čitelnosti a udržitelnosti za cenu momentálního zrychlení vývoje. Dodržení, či alespoň snaha o Clean Code, nese sebou cenu dočasného zpomalení vývoje. Vývojář dočasně investuje více kongitivní aktivity do práce s kódem, aby ji mohl v budoucnu ušetřit.

V této kapitole si dovolím vyzdvihnout několik zásad, která mají dle mého názoru největší dopad na čitelnost a udržitelnost kódu.

2.2.1 Identifikátory

V programech pojmenováváme mnohé konstrukce. Od proměnných, funkcí, návětší, tříd a různých struktur a jazykových specifik. Je důležité mít za názvem nějaký smysl, a právě o tom je tato kapitola. Jak tedy identifikátor smysluplně pojmenovat?

Název identifikátoru by měl zodpovídat otázky jako: ”Proč to existuje?”, ”Co to dělá?” a ”Co se s tím dělá?”. Samotný název by neměl záviset na explicitním kontextu, ale implicitně by nám měl prozradit odpovědi na tyto otázky. Například proměnné - jednopísmenný název nám nic nenapoví o funkci a důvodu existence této proměnné, ovšem pokud řešíme úlohu nalezení délky přepony pravoúhlého trojúhelníku, tak název `delkaPreponyCm` nám již prozradí, co se schovává za její hodnou.

Ke smysluplnosti názvu identifikátoru patří i zdánlivě jasné, ač z anekdotické zkušenosti často nedodržované, pravidlo - název nesmí lhát. Nenazývejme věc seznamem, pokud to skutečně není seznam. Od seznamu očekáváme možnost indexovat pomocí přirozených čísel, ovšem pokud se za názvem schovává slovník, je tento předpoklad rozbitý a kód nám *defacto* lže. S tím souvisí i zdánlivě podobné názvy. V dnešní době, kdy vývojová prostředí nabízejí možnosti v našeptávači, nebo agenti dokončují implementace, je malý rozdíl v jinak podobných názvech katastrofa čekající na dokončení našeptávačem.

Jednotlivé názvy by se měly od sebe lišit nejenom pro vyvarování se nahodilé záměny, ale také pro to, aby bylo možné v kódu sledovat, jakou roli vlastně daný identifikátor plní. Pokud argumenty funkce pojmenujeme stejným slovem a přidáme sufix v podobě číslovky, rozhodně nám to nepoví tolik, jako když každý argument důsledně pojmenujeme podle jeho role v implementaci. Kromě číslovek se lze často setkat s přístupem přidání afixů jako `manager` nebo `data`. Pokud názvy od sebe odlišíme akorát těmito afixy, ztrácí se smysl daných proměnných. Mnohdy totiž vývojář použije název s afixem jenom proto, že identifikátor již v kontextu existuje a nenapadá ho lepší název. Když názvy nesou různé významy, musí být taky různě pojmenovány.

Identifikátory by měli být hledatelné a vyslovitelné. Není žádným překvapením, že pokud chceme se zbytkem týmu diskutovat o kódu, budeme používat slova. *Clean Code* doporučuje mít identifikátory pojmenované tak, aby je bylo možné používat v diskuzi jako běžná slova. Vyhneme se tím dlouhému přemýšlení nad tím, co daná zkratka znamená, a můžeme se místo toho soustředit na podstatu věci. Navíc, pokud budeme chtít daný identifikátor v kódu vyhledat, musí být v názvu dostatečně odlišitelný od odstatních a zároveň nesmí být moc obecný. Je vhodné se vyvarovat magickým konstantám v řídících strukturách, a dát jim smysluplný název na vrcholu kontextu.

S odlišováním musí být člověk opatrný, protože to může svádět k porušení dalšího doporučení, které radí nekódovat informaci o typu nebo kontextu do

názvu proměnné. V době našeptávačů a statických analyzérů již není potřeba zakódovat typ proměnné do jejího identifikátoru. Stejně jako není potřeba do názvu rozhraní zakódovat, že se jedná o rozhraní. Pokud nastane případ, kdy máme kolizi názvu rozhraní a třídy, je to pro nás indikátor, že je třeba se nad názvy lépe zamyslet. Ač bychom mohli mít třídu `Camera` a k tomu rozhraní `ICamera`, je lepší třídu pojmenovat `CameraImplementation` a nebo podle návrhového vzoru `CameraFactory`. V tomto případě totiž dáváme do názvu informaci o způsobu, jakým třída funguje a co dělá, a nekódujeme akorát typ do identifikátoru.

2.2.2 Funkce

Funkce a procedury jsou většinou první úrovní abstrakce a členění kódu, se kterou se vývojář setká. Není programátor, který by nenapsal ve své kariéře funkci. Na funkcích se ostatně zakládají celá paradigmata. *Clean Code* nemůže takto důležitou konstrukci opomenout a přichází s řadou doporučení, která mají za cíl udělat funkce snadno testovatelné a pochopitelné i pro ostatní členy týmu.

Pravidla pro funkce by se dala shrnout do tří následujících vět: Funkce by měla být malá, měla by dělat jednu věc a neměla by ovlivnit nic mimo svůj kontext. To, že by funkce měla být malá, je varování před vysokou mírou zanoření programu. Pokud se ve funkci vyskytuje několikanásobné zanoření, jedná se s velikou pravděpodobností o nekonzistentní úroveň abstrakce a je vhodné se zamyslet, zda-li by neměly být některé bloky kódu refaktorovány do samostatných funkcí. Jak ostatně řekl Linus Torvalds na počet zanoření: *"... if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program."*

Co znamená, že by funkce měla dělat jednu věc? V rámci jedné funkce by měla panovat jedna úroveň abstrakce a jedna úloha, kterou řeší. Buďto funkce něco upravuje, nebo něco zodpovídá, ale nikdy by neměla dělat obojí.

Pokud funkce provádí nízkoúrovňový výpočet, nebude potřebovat volat abstrakci pro zpracování dat. Naopak vysokoúrovňová abstrakce by neměla provádět surový výpočet a místo toho řešení problému delegovat. Ředitel firmy se nezabývá balením zásilek a vyřizováním objednávek, ale chceme, aby dobře řídil firmu. Skladník zase řeší naskladnění a vyskladnění zboží, a ne logistický proces. Jak tedy efektivně rozdělit problém na úrovně abstrakce? *Clean Code* doporučuje postup zhora dolů, kdy program by měl být čitelný jako posloupnost odstavců začínající anglickou předponou *to* (česká předpona *k*). Tato pomůcka nám dovoluje intuitivně rozdělit problém na jednotlivé úrovně abstrakce a zůstat v maximální míře konzistentní.

- K vyřízení objednávky potřebujeme zabalit zboží, odečíst ho ze skladu a vytisknout štítek
- K zabalení zboží potřebujeme zboží najít na skladě, a najít vhodnou krabici

- K odečtení ze skladu potřebujeme najít, na které pozici se zboží nachází
- K vytisknutí štítku potřebujeme určit přepravce, vyplnit jeho šablonu a odeslat k tisku
- ...

Funkce mají ovšem kromě implementace jeden další bod, který určuje jejich komplexitu. Jedná se o její argumenty. Ideální funkce je niladická, tudíž nemá žádné argumenty. Monadická funkce má přesně jeden argument a je zcela běžná. Dyadické funkce, které mají dva argumenty, jsou přijatelné. Přítomnost triadické funkce (tři argumenty) nebo polyadické funkce (více než 3 argumenty) by měla být varovným signálem, že je s návrhem něco špatně.

S rostoucím počtem parametrů roste také počet testovacích případů, pokud předpokládáme čtyři parametry, které jsou pravdivostní hodnoty, jedná se o $2^4 = 16$ kombinací, které je potřeba otestovat. Podobně můžeme postupovat i u spojitých proměnných, kde vybíráme zástupce z jednotlivých tříd ekvivalence. Menší počet argumentů nejenom zjednodušejí použití funkce, ale i její testování.

V čem tkví problém dvou a více argumentů? Jedná se o případy, kdy jednotlivé argumenty nejsou přirozeně seřazeny. Pokud bychom vytvářeli instanci bodu, je v pořádku mít dva argumenty, protože přirozeně řadíme body do uspořádaných dvojic (x, y). Naopak pokud bychom měli funkci, která porovnává očekáváný výsledek a skutečný výsledek jednotkového testu, je pořadí argumentů dané konvencí a ne přirozeným řazením. Existuje ovšem metoda, jak počet argumentů redukovat. Jednotlivé argumenty je možné seskupit dohromady do tříd nebo jiných datových struktur.

Clean Code varuje před přepínači - pokud funkce přijímá argument v podobě pravdivostní hodnoty, znamená to, že dělá více jak jednu věc, a to v závislosti na daném přepínači. V takovém případě je lepší jednotlivé cesty rozdělit do samostatných funkcí.

2.2.3 Clean Code jako programátorova mantra

Clean Code ať už to jako koncept nebo jako kniha nemá být vnímána, ani v rámci v této práce, jako konkrétní body, které se v programu kontrolují podobně jako když se hledají běžné zranitelnosti (CVE). Jedná se spíše o mantru, která vede k hlubšímu porozumnění problematiky čitelnosti kódu. Každý vývojář dospěje do fáze, kdy chce vše dělat podle pravidel, aby následně zjistil, že je potřeba uvážit situace, kdy pravidla a doporučení více škodí, než pomáhají. Stejně tak *Clean Code* není konceptem, který by měl být vštípán do hlavy začátečníkům, ale slouží jako horizontální rozšíření dovedností jednotlivce. Není tak důležité znění daných pravidel, ale jejich smysl a jaký problém se snaží řešit. V takový moment je možné uvažovat, která pravidla nasadit, a naopak která vynechat, pokud nás daný

problém netrápí. Stejně jako mluvené slovo, jazyky jsou expresivní záleží na jednotlivci, komu je jeho slovo určeno a jak se chce vyjadřovat.

2.3 NÁVRHOVÉ VZORY

2.4 TESTOVÁNÍ

2.5 VERZOVÁNÍ

2.6 POSTUPNÁ INTEGRACE A POSTUPNÉ NASAZENÍ

2.7 MONITOROVÁNÍ NASAZENÉHO SYSTÉMU

Kritickou součástí vývoje softwaru je jeho testování a ladění (debugging). Testy, například jednotkové nebo integrační, simulují chování systému za různých podmínek a následné ladění pomocí debuggerů odhaluje jeho vnitřní stav instrukce po instrukci. V praxi ovšem není možné odhalit všechny chyby a dojde na situace, kdy se v nasazeném řešení chyba objeví. Při lokálním vývoji by se program odkrokoval debuggerem a zjistili se okolnosti, které chybu způsobily.

V produkčním nasazení ovšem luxus odkrokování programu zpravidla nemáme, ať už kvůli chybějícím pomocným strukturám (například *symbol table*) a nebo kvůli optimalizacím kompilátoru, který program převede do jazyka symbolických adres či byte kódu. Když tedy nelze pozorovat stav programu zevnitř, nezbývá nám nic jiného, než ho pozorovat zvenčí - a konkrétně přes jeho výstupy. Schopnost pozorovat chování systému na základě jeho výstupů nazýváme **observabilitou**.

Mnoho autorů a spolků, kteří se observabilitou zabývají, jí rozdělují na pilíře. Cílem těchto pilířů je zodpovědět na otázky: *Co se to děje?*, *Proč se to děje?* a *Kde se to děje?*. Každý pilíř tak doplňuje informace k těm dalším, a dohromady tvoří kompletní vhled do chování systému. (Elastic Observability Team, 2024) Tato práce ovšem bude na observabilitu nahlížet trochu jinak. Paradigma pilířů indikuje, že každý pilíř je disjunktním článkem, které, když se spojí dohromady, tvoří observabilitu.

2.7.1 Logy

Jedním z nejčastějších výstupů programu, který se používá pro ladění nasazeného řešení, jsou logy. Jedná se o textové záznamy, které nesou

informaci o událostech v programu. Logování má své zásady a doporučení, jejichž dodržení či nedodržení ovlivňuje míru, kterou nám logy pomohou s diagnostikou problému. Ačkoliv je logování velmi užitečné, nedává nám komplexní obraz o chování systému, protože se jedná jenom o jeden z možných výstupů programu. Program může vytvářet další telemetrická data, která mohou být následně dále zpracována.

2.7.2 Metriky

Metriky jsou kvantitativní veličiny, které měří konkrétní vlastnosti běžícího programu. Můžeme tak měřit například počet požadavků na konkrétní instanci či počet I/O operací. Metriky máme přímo měřené a odvozené, přičemž odvozené metriky můžeme zjišťovat z logů (například počet chyb v čase)

Je nutné podotknout, že metriky má smysl zaznamenávat dlouhodobě jakožto časové řady, abychom určili chování systému v čase. Je nutné tak metriky zaznamenávat na základě události, nebo pravidelného měření. Naopak zaznamenávat metriky vycházející ze statické analýzy smysl nedává, protože pokud bychom chtěli například analyzovat složitost programu pomocí Halsteadových vzorců, tak se tato metrika nemůže měnit v čase běhu programu. Naopak zatížení konkrétní instance programu nebo využití hardwarových prostředků proměnlivé čase běhu programu je.

3 ARCHITEKTURA NASAZENÉHO ŘEŠENÍ

3.1 ANALÝZA STÁVAJÍCÍHO ŘEŠENÍ

Cílem této kapitoly je provést podrobnou analýzu stávajícího řešení a identifikovat jeho případné nedostatky či zásadní chyby v návrhu, které mohou negativně ovlivnit běh, stabilitu a celkovou spolehlivost systému. K tomuto účelu byly využity tzv. C4 diagramy, což je metodika vizualizace softwarové architektury, která umožňuje nahlížet na systém z několika úrovní abstrakce – od celkového kontextu systému až po detailní zobrazení jednotlivých komponent. Tato strukturovaná forma modelování pomáhá nejen pochopit architekturu z pohledu různých cílových skupin (např. vývojáři, architekti, zadavatelé), ale také usnadňuje identifikaci slabých míst návrhu a jejich následnou optimalizaci.

3.1.1 Diagram systémového kontextu

Prvním diagramem je diagram systémového kontextu. Ten nám dovoluje modelovat systém jako střed diagramu, který je obklopen uživateli a dalšími systémy, se kterými interaguje. Cílem není tedy modelovat samotný systém, ale prostředí, ve kterém je nasazen. V kontextu CLOUCODE AITechDetect umožňuje modelovat, jací uživatelé se systémem interagují a jaké další systémy nasazené řešení potřebuje. V diagramu [doplň odkaz] je znázorněn diagram systémového kontextu pro AITechDetect. První věcí, kterou můžeme z diagramu zjistit, je fakt, že celé řešení není nadstavbou nad dalším řešením spravované jinou společností.

4 MITIGACE CHYB V ARCHIKTUŘE

Po analýze celého systému je možné začít jednotlivé neduhy napravovat. Tato kapitola dokumentuje jednotlivé snahy o napravení nalezených problémů a slabin, pričemž zvláštní důraz je kladen na empirický přístup k implementaci nejlepšího řešení. Je tedy možné, že potenciální náprava problému se po měření ukáže jako ta, která působí více škody než užitku. Nelze takto ale měřit všechny problémové aspekty, a u takových případů je kladen důraz na *best-practices* vyjmenované v konkrétních CWE.

4.1 DEFINICE SOFTWAROVÝCH POŽADAVKŮ

Před samotným vývojem softwaru je velmi užitečné sepsat dokument, který jasně vymezí, co se od výsledného řešení očekává. Norma ISO 29148 takový dokument nazývá specifikace softwarových požadavků. Tato norma popisuje požadavky a související aspekty do velkého detailu, takže kompletní využití celé osnovy se v praxi obvykle nepoužívá. To ale neznamená, že by se měl tento přístup úplně zavrhnout – i stručná a základní verze specifikace pomáhá předejít chybným interpretacím zadání a následným nedorozuměním mezi vývojáři a zákazníky.

Specifikace by však neměla být jen seznamem funkcí. Důležitý je také kontext výsledného produktu. Dokument proto neslouží jen programátorům, ale i dalším zúčastněným stranám, jako jsou testeři nebo designeři. Ti potřebují znát prostředí, ve kterém bude software používán, a úkony, které má plnit. Typická specifikace softwarových požadavků se skládá ze tří hlavních částí:

- **Přehled produktu**, ve které se popíše produkt, jeho kontext, uživatelé a možná omezení.
- **Přehled požadavků**, která definuje požadavky na funkce, systémy, rozhraní, návrh či kvalitu.
- **Přehled ověřovacích metod**, která definuje způsoby ověření skutečnosti, že byl software vytvořen v souladu se zadáním.

4.1.1 Přehled produktu

Přehled o produktu, který je předmětem této práce, již částečně poskytla kapitola 3.1.1 s diagramem systémového kontextu. Tyto diagramy dobře zobrazují vztahy mezi produktem, jeho uživateli a externími systémy. Na druhou stranu se však příliš nevěnují samotnému produktu – chybí jim pohled dovnitř, tedy zaměření na jeho funkce a omezení.

Jádrem systému AITechDetect je proces inspekce. Ten probíhá ve třech krocích: pořízení fotografie, její vyhodnocení a následné uložení. Na tento základní proces navazují rozšiřující funkce, například vícepohledová inspekce, anotace snímků, archivace kontrol nebo správa specifikací a projektů.

4.1.2 Přehled produktu

4.1.3 Přehled ověřovacích metod

4.2 MITIGACE CHYB V ARCHIKTUŘE

4.2.1 Rozbití monolitu

Monolitický návrh řešení je pro začátek vývojového cyklu velmi výhodnou cestou. Umožňuje totiž velmi snadno komunikovat mezi částmi aplikace a přidávat nové funkcionality, které staví na těch předchozích. Výhodou je také to, že řešení se chová a pouští jako celek, který nepotřebuje k jeho běhu další závislosti.

Každý větší projekt se ovšem jednou dostane do fáze, kdy je třeba zvážit, jestli monolitický návrh projektu spíše neškodí. Neošetřená chyba v jedné části programu sebou stáhne celý program a části, které nijak s chybovým kódem nesouvisí, jsou kvůli pádu programu nedostupné. Pokud je monolit dostatečně velký, nese sebou i jistou míru komplexity, která stěžuje hledání chyb a jejich reprodukci. Navíc pokud je monolit orientovaný kolem určité technologie, např. kombinace Flask a Python, jsou do této technologie vývojáři uzamčeny, i když zrovna nemusí být ta nejideálnější, a v horším případě i aktivně nedoporučovaná.

Důvodů pro zachování monolitu i pro jeho rozbití je v případě CLOUDCODE AITechDetect mnoho, k jeho rozbití bylo nakonec přistoupeno s ohledem na budoucí implementaci CI/CD, monitoringu a způsobu nasazení pomocí kontejnerů, enginu Docker a Docker Compose pro orchestraci kontejnerů. Do kontejnerového diagramu tudíž přibudou nové kontejnery, které se převážně odštěpí od Flask Serveru.

- CCM-ML
- CCM-INSPEKCE

- CCM-FRONTEND
- CCM-BACKEND
- CCM-BROKER

Jedna věc je monolit rozbít, ale druhou je jednotlivé části opět spojit zpátky. K tomu má sloužit primárně kontejner CCM-BROKER, se kterým lze komunikovat pomocí HTTP nebo pomocí Websockets.

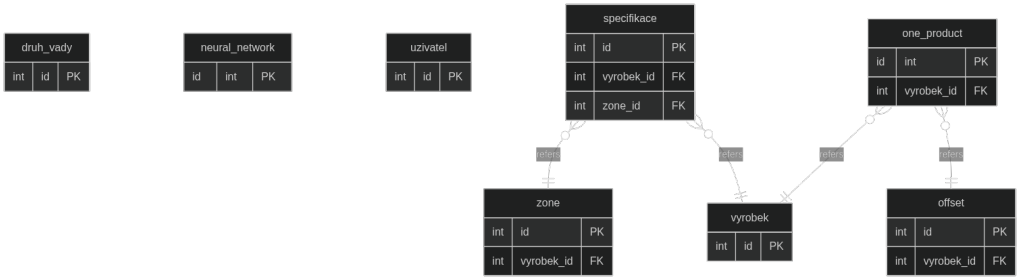
4.2.2 Přidání definic funkcí a životních cyklů v řešení

4.2.3 Přidání brokeru pro systémové zprávy a události

https://kafka.apache.org/documentation/

4.2.4 Normalizování schématu relační databáze

AITechDetect využívá relační databází SQLite pro uchovávání některých dat systému. V této kapitole je cílem podívat se na schéma této relační databáze a normalizovat ho na úroveň, kdy nebude docházet k aktualizačním anomáliím. Databáze obsahuje celkově 8 tabulek, které jsou ovšem svými atributy natolik rozsáhlé, že nebudou zakresleny do ER diagramu, ale podrobně popsány v následujícím textu.



Obrázek 4.1: Zkrácený ER Diagram původního schématu databáze

4.3 MITIGACE CHYB V NÁVRHU DATABÁZE

Normální formy

Návrh schématu

Při návrhu schématu relační databáze je třeba dbát na fakt, že databáze má modelovat vztahy mezi entitami z reálného světa. Je proto důležité mít výčet těchto entit a jaké mají mezi sebou závislosti. Mezi podstatné entity modelu AITechDetect patří:

- **Projekty**, neboli logická jednotka pro konkrétní druh výrobku a jeho souvisejících kontrol,
- **Vady**, které je možné nalézt na výrobku,
- **Kamery**, které jsou rozmístěné na kontrolní stanici,
- **Specifikace**, která určuje zásady pro vyhodnocení kontroly (kolik vad je ještě v pořádku),
- a **Kontroly**, které zaznamenávají výsledek jednotlivých kontrol inspekce.

Každá inspekce se odvíjí od projektů, které reprezentují jednotlivé druhy výrobků, které systém kontroluje.

Druhy vad

Tabulka `druh_vady` je minimálně v BCNF, a to čistě z toho důvodu, že obsahuje tři atributy - identifikační číslo vady, název a přepínač, jestli je vada považována na smazanou.

Vytvoření vlastního systému migrací

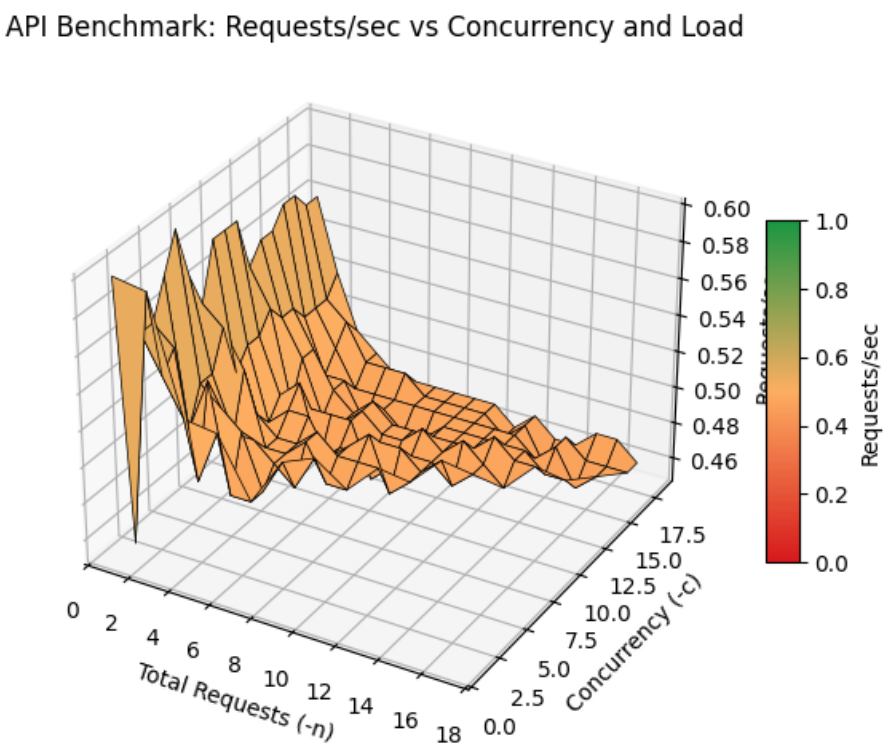
- `druh_vady`, která reprezentuje vadu na výrobku
- `uzivatel`, která reprezentuje přihlášeného uživatele v systému
- `neural_network`, která reprezentuje natrénovanou neuronovou síť
- `specifikace`, která definuje kolik vad a jakého druhu může být na jednom výrokbu.

4.3.1 Přepsání API rozhraní do kompilovaného jazyka

Webové API poskytující služby pro uživatelské rozhraní a další součásti systému AITechDetect je napsán v jazyce Python, který je interpretovaný. To sebou přináší několik obtíží a *overhead* v podobě interpreteru. Dalším problémem je fakt, že zdrojový kód aplikace je čitelný běžnému smrtelníkovi, což je v rámci politiky společnosti stav nežádoucí. Nabízela se tedy otázka, zdali by nebylo výhodné tuto část apliakce zkompilovat a jaký jazyk k tomu využít. Pro python existují kompilační nástroje, ovšem kompilace trvají dlouho a navíc přinášejí další zátěž do celého procesu.

Pro účely měření byl vytvořen testovací dataset o velikosti 1000 souborů JSON. Cílem úlohy bylo tyto data načíst a vrátit jako jeden velký JSON payload. Měření probíhalo pomocí nástroje Apache Benchmark, kde byl postupně zvyšován počet celkových požadavků a počet celkově souběžných požadavků. Měření byla provedena na stejném hardware se

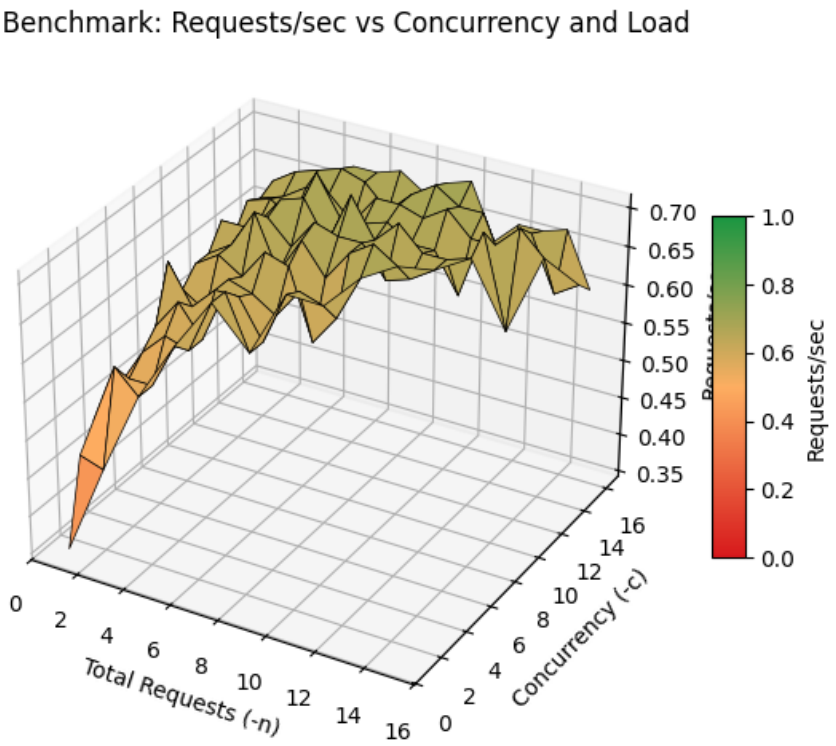
srovnatelně stejným zatížením od operačního systému Ubuntu 24.02. První měření proběhlo na implementaci v jazyce Python



Obrázek 4.2: Porovnání rychlosti čtení datasetu

Z grafu 4.2 můžeme vidět, že nízké počty celkových požadavků byly obslouženy decentně, ovšem s rostoucím sekvenčním zatížením výkon python serveru klesá. Lze spekulovat nad důvodem takového poklesu, ozvláště když je pokles vázaný na počet celkových požadavků a ne na počet souběžných požadavků. Jedním z možných vysvětlení je manipulace paměti při alokování a dealokování proměnných, které mohou negativně ovlinit následné požadavky. Problémy se správou paměti a *garbage collection* algoritmy je u intepretovaných jazyků znám již dlouho, a je to jedna z věcí, kterou je třeba zvážit při výběru platformy. Další možností, či pouze faktorem, je prodleva při manipulaci s vlákny, které Flask používá k obsluze požadavků. V některých případech je pohodlí vývoje na úkor menší výkonové ztráty přijatelná. V případě systému AITechDetect ovšem není, a proto bylo přistoupeno k implementaci v jiném, kompilovaném, programovacím jazyce.

Volbou pro novou implementaci byl jazyk Go. Tento jazyk skvěle pasuje do prostředí kontejnerů, kdy výstupní obrazy mají minimální velikosti, a zároveň je optimalizovaný na webové aplikace. Navíc nemá natolik komplikovanou syntaxi, aby přechod z jazyka Python byl pro zbytek vývojového týmu složitý. Byl tedy implementován koncový bod pro načtení datasetu, který využívá gorutiny a synchronizační primitiva, a bylo provedeno další měření.



Obrázek 4.3: Porovnání rychlosti čtení datasetu

Normalizace webového API

Původní webové API postrádalo konzistentní strukturu. CWE-1099 (Inconsistent Naming Conventions for Identifiers) upozorňuje na riziko, které s tím souvisí – konkrétně obtížnější lokalizaci chyb a slabin. Koncové body API byly nepředvídatelné a nebyly ani řádně dokumentované.

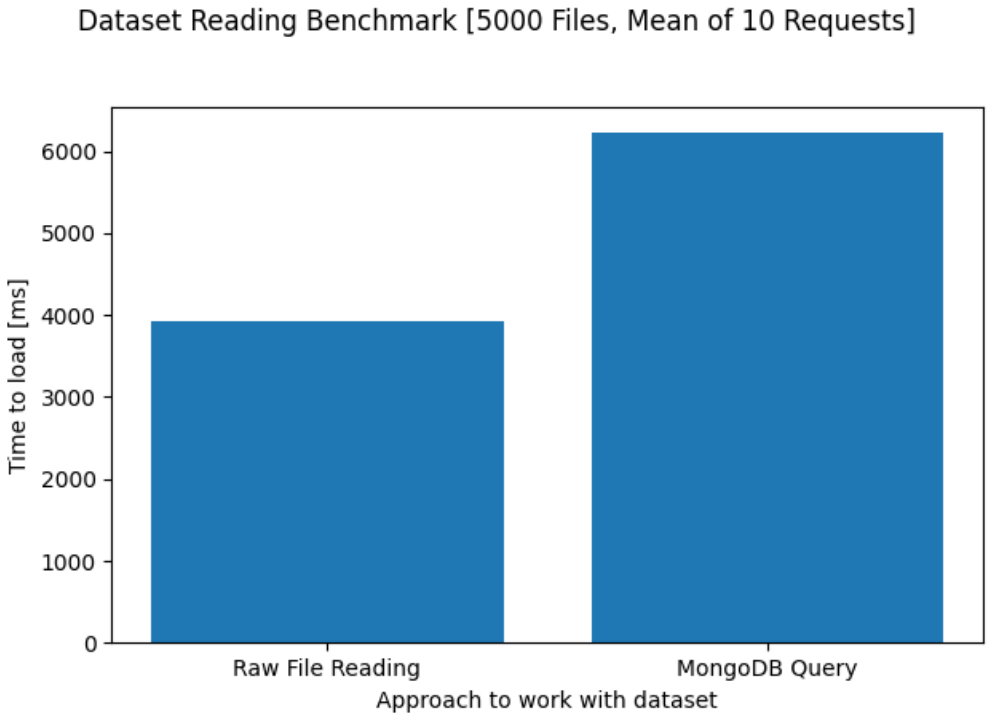
Řešením bylo co nejvíce normalizovat webové rozhraní podle standardu REST. Pro jednotlivé entity byly vytvořeny patřičné CRUD operace a byla zavedena architektura MVC s menšími úpravami. `Controller` zajišťuje obsluhu jednotlivých cest, `model` se stará o interakci s databází a místo tradičních pohledů jsou vytvořeny `services`, které poskytují obecné funkce, například připojení k databázi nebo validaci dat. Využití architektury MVC přivedlo konzistenci při vytváření nových koncových bodů, protože lze jednoznačně oddělit odpovědnosti jednotlivých částí.

Deklarativní validace vstupních dat

Generování dokumentace z anotací

4.3.2 Využití dokumentové databáze pro práci s JSON soubory

Jak bylo již řečeno v [[[[[Bakalářské práci, tady musíme dát proper odkaz]]]]], systém AITechDetect pracuje převážně nad soubory JSON. Každý dataset může mít od nižších desítek a po nižší desetitisíce takových souborů, a obzvláště na té větší spektra dochází k výrazném zpomalení při práci s datasety.



Obrázek 4.4: Porovnání rychlosti čtení datasetu

4.3.3 Předělání systému projektů

4.3.4 Přidání systému specifikací pro definici inspekce

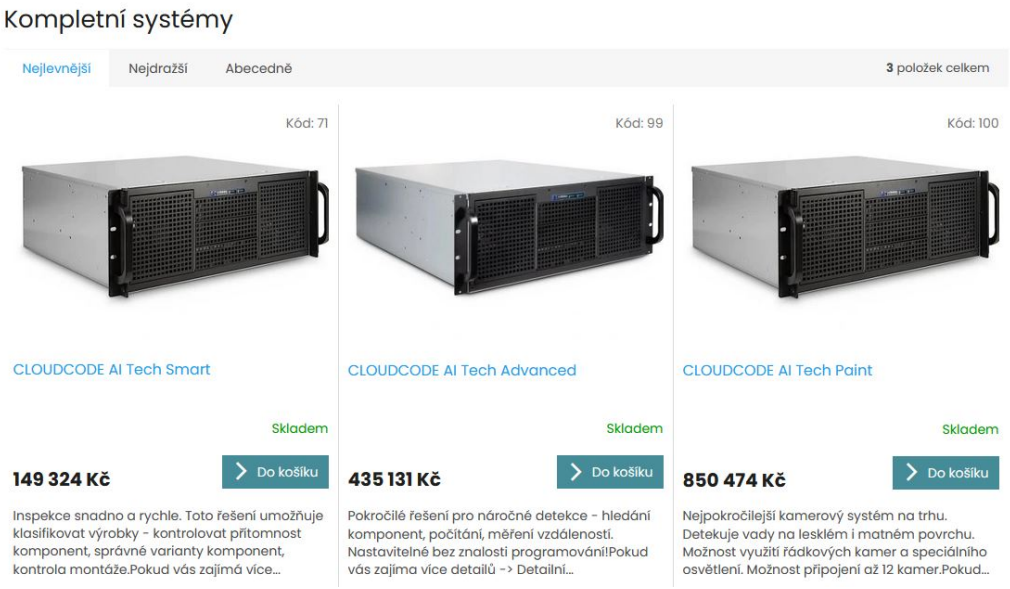
Lorem ipsum už něco máme z Hajdíku, ale je to hard coded.

4.3.5 Přidání systému pro správu toolů

- Seznam toolů a jejich verzí - V seznamu projektů vidět nasazené verze - Evaluační metriky z https://en.wikipedia.org/wiki/F-score

4.3.6 Přidání důkladné autentizace a autorizace

Jak bylo popsáno v analýze původního stavu řešení, aplikace nyní postrádá autentizaci a autorizaci, čímž se otevírají dveře k slabinám, které jsou popsané v CWE-284. Než vymýšlet vlastní řešení a řešit všechny bezpečnostní problémy s tím spojené, včetně auditů, je výhodnější využít již existující řešení, které je prověřené a aktivně používané. Z tohoto důvodu se pro autentizaci a autorizaci využil Keycloak. Ten je distribuován pod licencí Apache License 2.0, která umožňuje komerční užití bez licenčních poplatků a jiných právních omezení, které by jinak bránily jeho využití v komerční sféře.



Obrázek 4.5: Nasazení Keycloak kontejneru do systému AITechDetect

Keycloak umožňuje autentizaci pomocí protokolu OpenID, přičemž služby následně mohou uživatele autorizovat na základě dotazu, ve kterém přiloží informaci o chráněném zdroji a uživatelově tokenu. Keycloak pak rozhoduje, jestli má být uživatel autorizován či nikoliv. Služba má tak možnost vytvořit vlastní autorizační logiku, nebo ji delegovat na Keycloak.

Jední

4.3.7 Přidání procesu sestavení řešení

4.3.8 Předělání procesu nasazení

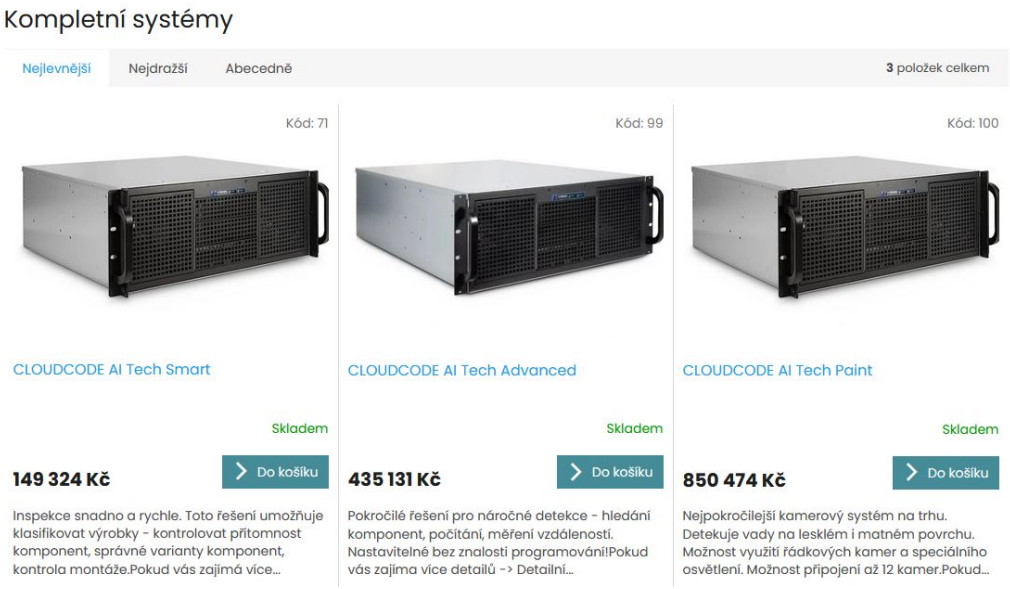
4.3.9 Přidání observability a monitoringu

4.3.10 Přidání licenčního systému

Jedním z požadavků společnosti CLOUDCODE bylo vytvořit systém licencování a zajistit, že software bude použit pouze oprávněnými uživateli

a v souladu s podmínkami licence. Tento systém musí řešit dva problémy - jak zabránit neoprávněné manipulaci se soubory poskytující informaci o licenci, a jak zabránit neoprávněné reprodukci.

AITechDetect je rozdělen do tří licencí, které se liší svými možnostmi a cenou. Licence jsou koncipované do hiearchie, kdy každá další úroveň obsahuje všechny funkce té předchozí. Cílem je poskytnout možnost levnější integrace výměnou za omezenější možnosti, které jsou pro danou problematiku dostačující. Na toto rozdělení funkcí je ale potřeba připravit i samotné řešení, a zabezpečit, že software bude odolný vůči účelné manipulaci zvenčí.



Obrázek 4.6: AITechDetect licence

Integrita a platnost licence je řešena pomocí kryptograficky podepsaného souboru, čímž se zabrání jeho manipulaci. Aby vše bylo bezpečné, byla využita asymetrická kryptografie. Licenční soubor je podepsán privátním klíčem zaměstnance CLOUDCODE a aplikace AITechDetect využívá veřejný klíč k ověření daného podpisu.

Samotná implementace licenčního systému byla koncipována jako webová služba, která implementuje rozhraní nad *GNU Privacy Guard* (GPG). Ten slouží ke správě klíčů a šifrování dat, a běžně se používá pro generování klíčů například pro SSH. GPG ovšem není jenom na vytvoření páru klíčů. Umožňuje také z privátního klíče generovat podklíče. Ty jsou využity jako další bezpečnostní opatření, ale tentokrát na straně samotné společnosti. Licenční systém totiž pro dané zaměstance vygeneruje unikátní podklíče, čímž je zajištěno, že původní privátní klíč nikdy nepřijde do styku se zaměstanci ani se zákazníky a je tak razantně nižší šance jeho kompromitace.

Posledním bezpečnostním opatřením bylo napojení licenčního systému na autorizační kontejner. Aby mohl zaměstnanec vytvářet licence, musí mít k tomu patřičná oprávnění definovaná v aplikaci Keycloak. Pokud tak není učiněno, licenční systém odmítne vystavit zaměstnanci jak podklíč, tak i podepsanou licenci. Tím je zajištěno, že k vytváření licencí má přístup pouze kontrolovaný počet zaměstnanců, kteří jsou tímto úkolem pověřeni a pro které případně plyne odpovědnost z nekalého počínání.

Podepsáním licenčního souboru je zajištěna integrita dat o zakoupené licenci, ovšem je potřeba také řešení ochránit před neoprávněnou reprodukcí. Počet možností zásadně snižuje fakt, že ověření licence musí kvůli bezpečnostním zásadám klientů fungovat offline. Nad tímto problémem byla vedena dlouhá úvaha. Prvotní myšlenka byla vytvořit otisky zařízení, ovšem řešení nasazováno v kontejnerech a tak nejde tak efektivně vytvářet otisky, nemluvě o situaci, kdy s dostatečným úsilím lze otisky zfalšovat. Další možností byly hardwarové klíče, které sice představují bezpečnější variantu, ale vyžadují fyzické umístění čipu do stanice nebo využití TPM. Tento způsob byl kvůli svému poměru náročnosti ku výsledku zamítnut.

Finálním řešením této kopírovací otázky byla změna úhlu pohledu. Než-li se snažit zamezit neoprávněné reprodukci, je z hlediska času i zdrojů lepší ji ošetřit právně. Pokud je zákazníkem zakoupena licence na AITechDetect, může sice software a licenci zkopírovat, ale bude muset čelit potenciálním právním důsledkům. Mezi ně může patřit například smluvní pokuta ve výši násobku ceny projektu. Na podobném principu fungují dohody o mlčenlivosti, které sdílení utajené informace nezastaví, ale potrestají odstrašující pokutou. Dalším faktorem je, že i při zatajení neoprávněné reprodukce se musí viník obejít bez podpory ze strany společnosti CLOUDCODE, protože žádáním o podporu by odhalil svůj přečin vůči smlouvě.

5 ZÁVĚR

Analýzy přiložené k textu práce nejsou nijak redigovány, protože neodpovídají nynějšímu skutečnému stavu.

BIBLIOGRAFIE

ELASTIC OBSERVABILITY TEAM, 2024 [online]. 2024-10-10. [cit. 2025-08-09]. Dostupné z: https://www.elastic.co/blog/3-pillars-of-observability.

FORRESTKNIGHT, 2025 [online]. 2025-07-24. [cit. 2025-08-10]. Dostupné z: https://www.youtube.com/watch?v=ukgmp6uxQJc.

FOWLER, Martin, 2009 [online]. 2009-10-14. [cit. 2025-08-10]. Dostupné z: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

TEAM ASANA, 2025 [online]. 2025-06-25. [cit. 2025-08-10]. Dostupné z: https://asana.com/resources/technical-debt.