Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky

Návrh a testování prostředí pro AI agenty v softwarovém vývoji

BAKALÁŘSKÁ PRÁCE

Studijní program: Aplikovaná informatika

Autor: Thanh An Nguyen
Vedoucí práce: Ing. Jiří Korčák

Praha, květen 2026

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

V Praze dne 22. února 2026

Thanh An Nguyen

Poděkování

Poděkování.

Abstrakt

TODO: Napsat až budou hotové výsledky.

Klíčová slova

AI agenti, softwarový vývoj, prostředí, scaffolding, kvalita kódu

Abstract

TODO: Write after results are complete.

Keywords

AI agents, software development, environment, scaffolding, code quality

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

TODO Doplňte zkratky a tento řádek smažte.

Úvod

1. Vymezení problému a cílů práce

1.1 Motivace

[DRAFT]

Velké jazykové modely (LLM) dnes umí generovat kód, ale to neznamená, že umí vyvíjet software. Randomizovaná studie METR (METR, 2025) na 16 zkušených vývojářích a 246 úlohách ukázala, že s AI nástroji byli vývojáři o 19 % pomalejší — přestože sami odhadovali zrychlení o 24 %. Když projekt roste, bývá těžší jej rozšiřovat — jak pro člověka, tak pro LLM. Vývojáři přicházejí o kontext a hlubokou znalost kódové základny, zatímco agenti jsou limitováni omezeným context window. Otázka tedy není, co modely umí, ale jak je zasadit do vývojového procesu tak, aby produkovaly kvalitní výstup.

Ukazuje se, že odpověď leží spíš v instrukcích než v samotných modelech. Breunig (Breunig, 2025) zjistil, že změna promptu mění chování agenta výrazněji než výměna modelu. Lulla et al. (Lulla et al., 2026) dokládají, že přidání architektonických konvencí do instrukčního souboru zkrátí dobu běhu o 28 % a sníží spotřebu tokenů o 20 %. Scaffolding — struktura instrukcí a procesních pravidel, která agenta provází vývojem — má na výsledek zásadní vliv. Ale nikdo systematicky nezkoumal, *které* složky tohoto scaffoldingu jsou pro dodržování softwarově-inženýrských praktik nezbytné a které jsou zbytečné.

Tato práce zkoumá právě to: jak musí být strukturovány instrukce pro AI coding agenta, aby dodržoval specifikací řízený vývoj, test-driven development a granulární správu verzí — a co z těch instrukcí je skutečně kritické.

[RAW]

Studie společnosti METR.ORG ukazuje, že LLM zkušené vývojáře spíše zpomaluje. S rychlým vývojem schopností modelů se situace pravděpodobně mění. Ale to neznamená, že je LLM samo o sobě dostatečné k vypracování dlouhotrvajících úkolů. Není to problém pouze LLM, když projekt roste, bývá těžší jej rozšiřovat jak pro člověka, tak i pro LLM. AI programování tenhle problém ještě více prohlubuje. Vývojáři přicházejí o kontext a hlubokou znalost kódové základny (codebase), zatímco velké jazykové modely (LLM) jsou limitovány omezenou pamětí (context window). Jak nastavit harness/scaffolding tak, aby v tom mohli fungovat agenti a lidé to stále měli pod kontrolou?

1.2 Cíle práce

[DRAFT]

1. Popsat, jak se řízení softwarových projektů mění v kontextu AI agentů, a zmapovat co

víme o vlivu instrukcí na jejich chování.
2. Iterativně navrhnout instrukce, které dovedou AI coding agenta k dodržování vývojového workflow na case study projektu (systém upomínek faktur).
3. Zjistit, které složky těchto instrukcí jsou pro dodržování workflow nezbytné, které jsou zbytečné, a jaký kontext agent potřebuje k tomu, aby vytvářel a využíval projektové artefakty.

[RAW]

1. Popsat jak se řízení SWE projektů mění v kontextu agentních systémů (teoretický rámec)
2. Navrhnout a implementovat experimentální prostředí (case study: systém upomínek faktur)
3. Prozkoumat vliv různých nastavení scaffoldingu na schopnost agenta provést kvalitní práci
4. Identifikovat jaký kontext je pro agenty klíčový a jak instrukce/procesy ovlivňují schopnost agenta tento kontext vytvářet a využívat

1.3 Rozsah práce

[DRAFT]

Práce zkoumá vliv struktury instrukcí na chování jednoho AI coding agenta při vývoji z formální specifikace. Case study je systém upomínek faktur — projekt s deterministickou logikou, jasnými pravidly správnosti a objektivně ověřitelnými výstupy. Experiment pokrývá fáze requirements, design a implementation.

Práce se zaměřuje na:

- Vliv struktury a obsahu instrukcí na to, jak agent dodržuje vývojový workflow
- Iterativní návrh instrukcí metodou Design Science Research
- Identifikaci toho, co z instrukcí je potřeba a co ne, prostřednictvím ablace

Práce se nezaměřuje na:

- Porovnávání LLM modelů
- Porovnávání programovacích jazyků a frameworků
- Projekty s převážně subjektivními výstupy (UI, kreativní tvorba, ML)

[RAW]

Poznámky k propojení:

- Řízení vyžaduje holistický pohled (vidět celek, ne jen část) - proto celý SDLC, ne jedna fáze
- Billing Reminder Engine jako case study: malý projekt, ale reálné nuance (state machine, business days, edge cases)
- Deterministická logika (stejný vstup = stejný výstup) + jasná pravidla správnosti = objektivně testovatelné
- Hraniční případy (víkendy, svátky, grace periods) = místa kde lze testovat kvalitu agentní práce

TODO: Vysvětlit termíny:

- state machine - ?
- edge cases / hraniční případy - ?
- SDLC - ?

Scope SDLC (k diskuzi):

- Primární plán: celý SDLC včetně deployment/maintenance
- Fallback: zúžit na implementation + testing (hlavní doména coding agents)
- Poznámka: i impl + testing má feedback loop (implementace → testy → chyba → úprava) = stále systémový pohled

2. Teoretická východiska

[DRAFT]

Tato kapitola vysvětluje teoretická východiska práce. Nejprve je popsáno softwarové inženýrství jako disciplína, následně životní cyklus a metodiky vývoje software. Poté je zkoumáno, jak se oblast mění díky AI coding agentům, a nakonec jsou představeny prvky scaffoldingu (podpůrných struktur), které agenti využívají.

2.1 Softwarové inženýrství

[DRAFT]

Pro pochopení toho, jak AI agenti mění vývoj software, je nezbytné nejprve porozumět zavedeným postupům a standardům softwarového inženýrství. Tato sekce definuje softwarové inženýrství, vysvětluje proč je software inherentně složitý a jak tato složitost vedla ke vzniku oboru.

2.1.1 Definice a vymezení oboru

[DRAFT]

Softwarové inženýrství je disciplína, která se zabývá celým životním cyklem software – od specifikace až po údržbu (IEEE Computer Society, 2024, s. xxxvii).

Na rozdíl od programování, které se soustředí na implementaci a technické aspekty jako algoritmy a datové struktury, softwarové inženýrství přistupuje k vývoji software holisticky – zahrnuje nejen technickou stránku, ale i organizační aspekty jako řízení projektů a rozpočty (Sommerville, 2016, s. 21).

[RAW]

CS vs SWE – vymezení:

Computer Science (informatika) – fundamentální, teoretické otázky:

- Algoritmy (sorting, searching, graph algorithms)
- Datové struktury (stromy, hashovací tabulky)
- Teorie výpočetnosti (Turingovy stroje, rozhodnutelnost)
- Formální jazyky a automaty
- Kompilátory, parsery, teorie typů

Software Engineering – praktické, systémové otázky:

- Jak spojit tisíce algoritmů do funkčního systému?
- Jak na tom pracovat v týmu?
- Jak to udržovat roky?

Rozdíl:

- CS = "Jak napsat správný algoritmus?"
- SWE = "Jak postavit a udržovat systém z tisíců algoritmů s týmem 50 lidí?"

Základní koncepty SWE:

K řešení těchto otázek SWE využívá tři klíčové koncepty:

1. Abstrakce

→ `colburn2000`:

Skrývání detailů, práce na vyšší úrovni bez znalosti implementace.

→ `swebok2024` s. 370 (Dijkstra):

"The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."

2. Modularita (information hiding)

→ `parnas1972`:

Rozdělení systému na nezávislé části s jasným rozhraním. Každý modul skrývá rozhodnutí která se mohou změnit.

3. Architektura

→ `perry1992`:

"Software architecture is a set of architectural elements that have a particular form."

Definují architekturu jako: elements (processing, data, connecting) + form + rationale.

→ `garlan1993`:

"As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system presents a new set of design problems."

High-level struktura systému – komponenty, konektory, konfigurace.

2.1.2 Historický kontext

[DRAFT]

Systémy jako software jsou čím dál složitější (Sommerville, 2016, s. 582).

[RAW]

Narativní flow (revidovaný):

1. **Složitost systémů roste** – software je čím dál komplexnější
   → `sommerville2016` s. 582:
   "The root cause of these problems is, as it was in the 1960s, that we are trying to build systems that are larger and more complex than before. We are attempting to build these 'mega-systems' using methods and technology that were never designed for this purpose."
2. **Abstrakce jako nástroj** – reakce na složitost:
   - Assembler → C → Java → frameworky
   - Každá vrstva skrývá detaily (information hiding)
     → `colburn2000` s. 1:
     "Abstraction through information hiding is a primary factor in computer science progress and success through an examination of the ubiquitous role of information hiding in programming languages, operating systems, network architecture, and design patterns."
   - Dijkstra: abstrakce pomáhá být přesný, ne vágní
     → `swebok2024` s. 370:
     "Dijkstra states: 'The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."'
3. **Přesto 1968 krize** – proč?
   - Složitost rostla rychleji než naše schopnost ji zvládat
   - NATO konference: "software crisis"
     → `nato1968` s. 78 (Perlis keynote):
     "I believe it is because we recognize that a practical problem of considerable difficulty and importance has arisen: The successful design, production and maintenance of useful software systems."
4. **Reakce: vznik SWE** – disciplína pro řízení složitosti
   → `sommerville2016` s. 592:
   "In software engineering, we have seen the incredibly rapid development of the discipline to help manage the increasing size and complexity of software systems... the approach that has been the basis of complexity management in software engineering is called reductionism."
5. **Dnes: AI jako nový problém**
   - Není to jen další vrstva abstrakce
   - Je to ztráta determinismu – "black box"

- Proto mechanistic interpretability
  → `bereska2024` s. 1:
  "Understanding AI systems' inner workings is critical for ensuring value alignment and safety. This review explores mechanistic interpretability: reverse engineering the computational mechanisms and representations learned by neural networks into human-understandable algorithms and concepts to provide a granular, causal understanding."

Klíčový insight: Abstrakce ≠ složitost. Abstrakce je NÁSTROJ pro zvládání složitosti. AI přináší kvalitativně nový problém (nedeterminismus), ne jen "více abstrakce".

2.1.3 Komplexita software

[RAW]

Co sem patří (Brooks – No Silver Bullet):

- **Essential complexity** – inherentní složitost problému
  → `brooks1987` s. 6:
  "The complexity of software is an essential property, not an accidental one. Hence descriptions of a software entity that abstract away its complexity often abstract away its essence."
  - Business logika, requirements, doménová znalost
  - Nelze odstranit – je to podstata toho co řešíme
- **Accidental complexity** – složitost kterou si přidáváme
  - Nástroje, technologie, jazyky, frameworky
  - Lze redukovat lepšími nástroji a abstrakcemi
- **Vlastnosti software** – proč je jiný než fyzické systémy:
  → `brooks1987` s. 6:
  "Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike."
  - Neviditelný, snadno měnitelný, nemá fyzická omezení
  - **Nelineární růst komplexity:** Na rozdíl od fyzických systémů (např. stavba zdi – cihla na cihlu, proces stále stejný), u software každý nový prvek interaguje s ostatními a přidává nové stavy.
    → `brooks1987` s. 6:
    "A scaling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly."
  - → `mcconnell2004` s. 127:
    "People use abstraction continuously. If you had to deal with individual wood fibers, varnish molecules, and steel molecules every time you used your front door, you'd

hardly make it in or out of your house each day. Abstraction is a big part of how we deal with complexity in the real world."

- **Evoluce a růst komplexity v čase** (Lehman's Laws):
  → `lehman1980` s. 9:
  "I. Continuing Change – A program that is used... undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version."
  "II. Increasing Complexity – As an evolving program is continually changed, its complexity increases unless work is done to maintain or reduce it."
  - Software musí být neustále adaptován (zákon I)
  - Komplexita roste s časem pokud se aktivně neredukuje (zákon II)
  - Doplňuje Brookse: Brooks říká PROČ je software složitý, Lehman říká že komplexita navíc ROSTE

Propojení s 2.1.2: Abstrakce (kompilátory, frameworky) řeší accidental complexity – ale essential zůstává. To je důvod proč "no silver bullet".

2.2 Životní cyklus a metodiky

[RAW]

Úvod sekce 2.2 (1-2 věty): Tato sekce popisuje jak se software vyvíjí v praxi: co se dělá (fáze), jak se to organizuje (metodiky), čím se to dělá (nástroje), co vzniká (artefakty), a kdo to dělá (role a komunikace).

2.2.1 Fáze životního cyklu

[RAW]

Různé přístupy k definici fází:

- Různé zdroje definují fáze životního cyklu různě:
  - SWEBOK/ISO 12207: Concept, Development, Production, Utilization, Support, Retirement
  - Waterfall klasický: Requirements, Design, Implementation, Testing, Deployment, Maintenance
  - Sommerville: 4 základní aktivity (abstrakce)
- Každá metodika (Scrum, Waterfall, XP...) má vlastní konkrétní fáze
- Pro účely této práce používáme Sommervillovu abstrakci – 4 aktivity které jsou společné všem přístupům

Sommerville – 4 základní aktivity:

→ `sommerville2016` s. 24:

"A software process is a sequence of activities that leads to the production of a software product. Four fundamental activities are common to all software processes:"

1. **Software specification** – zákazníci a inženýři definují co se má vytvořit
2. **Software development** – software se navrhuje a implementuje
3. **Software validation** – ověřuje se že software dělá co zákazník požaduje
4. **Software evolution** – software se mění podle měnících se požadavků

→ `sommerville2016` s. 55:

"The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved."

Klíčový bod: Tyto 4 aktivity jsou abstrakce – každá metodika je organizuje jinak (viz 2.2.2).

2.2.2 Modely a metodiky

[RAW]

Klasifikace metodik (Boehm & Turner 2003):

→ `boehm2003balancing`:

Metodiky nejsou binární volba, ale **spektrum** mezi plan-driven a agile.

5 dimenzí pro klasifikaci:

1. **Size** – velikost projektu/týmu
2. **Criticality** – kritičnost systému (life-critical vs comfort)
3. **Dynamism** – jak moc se mění requirements
4. **Personnel** – zkušenosti a schopnosti týmu
5. **Culture** – organizační kultura (chaos vs order)

→ `boehm2002agile`:

"Both agile and plan-driven approaches have their home grounds where each clearly works best, and a danger zone where each has problems."

Plan-driven metodiky:

1. Waterfall (sekvenční):

→ `royce1970` – **primární zdroj**, původní popis modelu

→ `boehm1988` s. 3:

"The waterfall model's basic scheme has encountered some fundamental difficulties, and these have led to the formulation of alternative process models."

2. Spiral Model (risk-driven):

→ `boehm1988` – kombinuje iterativní vývoj s analýzou rizik, reakce na problémy waterfall

---

Agile metodiky:

→ `agilemanifesto2001` – 4 hodnoty, 12 principů, reakce na heavyweight procesy

3. Extreme Programming (XP):

→ `beck2000` – pair programming, TDD, continuous integration, short iterations

4. Scrum:

→ `scrumguide2020`:

Framework pro komplexní problémy. Sprinty (2-4 týdny), role (PO, SM, Developers), artefakty (backlog, increment).

---

5. Spec-Driven Development (SDD):

→ `sdd2026` – Emerging metodika pro éru AI coding agentů:

Specifikace je source of truth, kód je odvozený/generovaný artefakt.

Tři úrovně rigidity:

1. **Spec-first** – specifikace před kódem, ale nemusí se udržovat po implementaci. Vhodné pro initial development s AI.
2. **Spec-anchored** – specifikace žije vedle kódu, testy vynucují synchronizaci (BDD, Cucumber). Sweet spot pro produkční systémy.
3. **Spec-as-source** – specifikace JE kód, nikdy se needituje přímo. Zatím jen specializované domény (embedded, Simulink).

SDD workflow: Specify → Plan → Implement → Validate (human review mezi každou fází).

Propojení s ostatními metodikami:

- SDD není náhrada agile/waterfall – je to vrstva nad nimi
- TDD = SDD na úrovni unit testů
- BDD = SDD s Gherkin scénáři
- DDD ubiquitous language = základ pro SDD specifikace

Thoughtworks Technology Radar 2025: SDD identifikován jako jeden z nejdůležitějších emerging trendů.

Klíčový bod: Většina reálných projektů používá hybrid – "the challenge is to balance the two approaches" `boehm2003balancing`. S příchodem AI agentů se přidává nová dimenze: SDD spec-first umožňuje "waterfall per increment" – sekvenční fáze (spec → implement → validate) v rámci malých incrementů, ale iterativní mezi incrementy.

2.2.3 Role a komunikace

[RAW]

1. SWE je fundamentálně týmová disciplína

Brooks's Law – komunikační overhead:

→ `brooks1975` s. 25:

"Adding manpower to a late software project makes it later."

→ `mcconnell2004` s. 713:

"Merely increasing the number of people increases the complexity and amount of project communication."

Komunikační kanály rostou: $n(n-1)/2$

---

2. Role a zodpovědnosti

Tradiční role:

- Developer – implementace, code review
- Tester/QA – validace, quality assurance
- Architekt – design, technická rozhodnutí
- Project Manager – plánování, koordinace
- Product Owner – requirements, priorities

Role v Scrum:

→ `scrumguide2020`:

- Product Owner – maximalizuje hodnotu produktu, spravuje backlog
- Scrum Master – efektivita týmu, odstraňuje překážky
- Developers – vytváří increment každý sprint

---

3. Conway's Law – organizace ↔ architektura

→ `conway1968`:

"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

Struktura týmu se odráží v architektuře systému. Důsledek: změna architektury často vyžaduje změnu organizace.

---

4. Koordinace práce

→ `malone1994`:

Coordination = managing dependencies between activities.

Typy závislostí: shared resources, producer-consumer, simultaneity constraints.

Tacit vs explicit knowledge:

→ `nonaka1995`:

"There are two types of knowledge: explicit knowledge, contained in manuals and procedures, and tacit knowledge, learned only by experience, and communicated only indirectly."

- **Tacit knowledge** – v hlavách lidí, těžko artikulovatelné, "know-how"
- **Explicit knowledge** – dokumentace, procesy, kód
- Konverze tacit → explicit = klíčový proces (externalization)
- Meetingy, pair programming, code review = mechanismy sdílení tacit knowledge

---

Propojení s BP – Quality gates pro agenty:

Každá role má kontrolní body (quality gates):

- Code review (senior/peer kontroluje kód)
- QA approval (tester ověřuje funkcionalitu)
- Acceptance criteria (PO schvaluje feature)

- Design review (architekt validuje návrh)

Pro agenty:

- Agent přebírá některé role (developer, částečně tester)
- Quality gates zůstávají – kdo je teď dělá?
- Implicitní znalosti z meetingů → explicitní instrukce pro agenta
- Agent nemá "kontext z meetingu" – vše musí být napsané

2.2.4 Nástroje a artefakty

[RAW]

Propojení s fázemi a rolemi:

Každá fáze (2.2.1) produkuje artefakty, role (2.2.3) používají nástroje k jejich tvorbě a správě.

---

1. Artefakty podle fází:

- **Specification** → Requirements (funkční, nefunkční), user stories, use cases, behavioral models (state diagrams pro event-driven systémy (Sommerville, 2016) kap. 5.4)
  Spektrum formátů specifikace: Specifikace může mít různou míru formálnosti (Sommerville, 2016, kap. 4):
  1. **Formální** – IEEE 830 SRS dokument, matematické specifikace
  2. **Structured** – šablony (Function/Inputs/Outputs/Pre/Post), structured natural language
  3. **Semi-structured** – GitHub Issues, user stories v backlogu, RFC dokumenty
  4. **Neformální** – konverzace, emaily, chat

  V open source komunitách se issue trackery staly de facto nástrojem pro requirements management – "software informalisms" nahrazují formální specifikace (Scacchi, 2002). V agilních týmech se user stories a backlog items používají místo SRS dokumentů (Cao & Ramesh, 2008). Viz také sekce 2.4 – pro AI agenty se GitHub Issues staly standardním vstupním formátem.
- **Design** → Architektura, design docs, diagramy (UML), API kontrakty (OpenAPI)
- **Implementation** → Zdrojový kód, konfigurace
- **Validation** → Testy (unit, integration, e2e), test reports
- **Evolution** → Change requests, release notes, patches

Traceability:

→ `gotel1994`:

"Requirements traceability refers to the ability to describe and follow the life of a requirement,

in both a forwards and backwards direction."

Propojení mezi artefakty: Requirement → Design → Code → Test.

Důležité pro quality gates – lze ověřit že každý requirement má test.

---

2. Nástroje podle činností:

- **Vývoj:** IDE (VS Code, IntelliJ), editory, debuggery
- **Verzování:** Git – distribuovaný version control
  → `chacon2014`: nejen kód, ale i dokumentace, konfigurace ("everything as code")
- **Koordinace:** Issue tracking (Jira, GitHub Issues), project management
- **Kvalita:** CI/CD, test frameworks, linters
  → `humble2010`: automatizace buildů, testů, deploymentu
  → `forsgren2018`: DevOps praktiky a jejich měřitelný dopad
- **Dokumentace:** Wikis, Markdown, generátory dokumentace

---

3. Komunikační nástroje:

→ `schmidt1992` – CSCW (Computer Supported Cooperative Work):

"Articulation work" – práce potřebná ke koordinaci spolupráce.

- Synchronní: Slack, Teams, meetings, pair programming
- Asynchronní: Email, code review comments, dokumentace

Navazuje na 2.2.3 – komunikační nástroje slouží k sdílení tacit knowledge. Problém: hodně kontextu zůstává v těchto kanálech a není explicitně zachycen.

---

4. Standardy a formáty:

Strukturované formáty usnadňují automatizaci a konzistenci:

- UML – diagramy architektury a designu
  → `fowler2004uml`: standardní jazyk pro modelování software
- OpenAPI/Swagger – API kontrakty (strojově čitelné specifikace)
- Markdown – dokumentace (lidsky i strojově čitelné)
- JSON Schema – validace dat
- Conventional Commits – standardizované commit messages

Propojení s BP:

Tato sekce popisuje tradiční nástroje a artefakty v SWE. V sekci 2.3 (Agentic coding) ukážeme jak agenti tyto nástroje používají a artefakty čtou/produkují. V sekci 2.4 (Scaffolding) ukážeme jak připravit artefakty aby byly pro agenty čitelné.

Zdroje:

- `gotel1994` – traceability definice
- `chacon2014` – Git/verzování
- `humble2010` – Continuous Delivery, CI/CD
- `forsgren2018` – Accelerate, DevOps metriky
- `schmidt1992` – CSCW, komunikační nástroje
- `fowler2004uml` – UML standardy

2.3 Agentic coding

[RAW]

Úvod sekce: Tato sekce definuje základní pojmy (LLM, agent, tool calls) a popisuje jak AI agenti mění softwarové inženýrství.

2.3.1 Základní pojmy

[RAW]

1. Large Language Model (LLM):

→ `vaswani2017` – Transformer architektura:

"Attention Is All You Need" – self-attention mechanismus umožňuje modelovat závislosti bez rekurence.

LLM = velký jazykový model založený na transformer architektuře, trénovaný na velkém množství textu.

---

2. LLM-based Agent:

→ `liu2024llmagents`:

Agent = LLM rozšířený o schopnost interagovat s prostředím.

4 klíčové komponenty:

- **Planning** – dekompozice komplexních úkolů na pod-úkoly
- **Memory** – krátkodobá (kontext) a dlouhodobá (RAG, databáze)
- **Perception** – vnímání prostředí (čtení souborů, výstupů)
- **Action** – provádění akcí (tool calls, zápis souborů)

---

3. Tool Calls (Function Calling):

Klíčová schopnost agentů – volání externích nástrojů.

→ `yao2022react` – ReAct framework:

"Synergizing Reasoning and Acting in Language Models"

Cyklus: Thought → Action → Observation (opakuje se).

Agent přemýšlí, provede akci, pozoruje výsledek, přemýšlí znovu.

→ `schick2023toolformer`:

LLM se může naučit kdy a jak použít nástroje (API, kalkulačka, vyhledávání).

Příklady tool calls pro coding agenty:

- Read/Write soubory
- Bash příkazy (git, npm, make...)
- Grep/Glob pro hledání v kódu
- Web search, fetch dokumentace

---

4. Prompt Engineering:

[DOPLNIT] Definovat prompt engineering jako disciplínu formulování instrukcí pro LLM. Klíčové techniky: zero-shot, few-shot, chain-of-thought, role prompting. Zdroje: najít vhodnou citaci (survey on prompt engineering).

Důležité rozlišení pro BP:

- **Prompt engineering** = jak formuluješ konkrétní instrukci/zadání pro LLM
- **Context engineering** = co všechno agent dostane jako kontext (viz sekce 2.4)
- **Scaffolding** = struktury v prostředí které agenta vedou (soubory, konvence, workflow)

Tato práce se primárně zabývá context engineeringem a scaffoldingem, ne prompt engineeringem – nejde o to jak formulovat prompt, ale jaké podpůrné struktury agentovi poskytnout.

2.3.2 Typy coding agentů

[RAW]

Evoluce podle autonomie:

→ `guo2025benchmarks`:

Tři paradigmata s rostoucí autonomií:

1. **Prompt-based** – člověk instruuje, LLM odpovídá (ChatGPT, copilot)
2. **Fine-tune-based** – model adaptován na SE doménu (CodeLlama, StarCoder)
3. **Agent-based** – autonomní plánování, akce, učení (Devin, Claude Code, OpenHands)

Copilot vs Autonomous Agent:

- **Copilot** – asistuje člověku, návrhy, autocomplete (GitHub Copilot, Cursor Tab)
- **Autonomous agent** – samostatně plánuje a vykonává úkoly (Devin, Claude Code agentic mode)

Taxonomy agentů:

→ `liu2024llmagents`:

- **Reasoning-enhanced** – chain-of-thought, tree-of-thought
- **Tool-augmented** – externí API, knowledge bases
- **Multi-agent** – spolupráce více agentů
- **Memory-augmented** – persistentní kontext mezi interakcemi

Příklady nástrojů (2024-2025):

- Devin (Cognition AI) – první "AI software engineer"
- Claude Code (Anthropic) – CLI agent
- Cursor – IDE s integrovaným agentem
- OpenHands – open-source platforma
- Agentless – alternativní přístup bez agentního cyklu → `xia2025agentless`

2.3.3 Jak agenti mění SDLC

[RAW]

Které fáze agenti ovlivňují:

→ `jin2024` – 6 klíčových oblastí:

1. **Requirement Engineering** – generování user stories, analýza požadavků
2. **Code Generation** – implementace z popisu, autocomplete
3. **Autonomous Decision-making** – plánování, výběr přístupu
4. **Software Design** – návrh architektury, API design
5. **Test Generation** – unit testy, test cases
6. **Software Maintenance** – bug fixing, refactoring, code review

---

Co zůstává stejné:

- Fáze životního cyklu (specification, development, validation, evolution)
- Artefakty (requirements, kód, testy, dokumentace)
- Quality gates (code review, testing, acceptance)
- Potřeba jasných requirements

Co se mění:

- **Rychlost** – minuty místo hodin/dnů
- **Explicitnost** – vše musí být napsané (agent nemá tacit knowledge)
- **Role** – developer se stává "reviewer" a "specifikátor"
- **Batch size** – menší úkoly, častější iterace

Propojení s 2.2:

- **Nástroje (2.2.4):** Agent používá stejné nástroje (Git, CI/CD) přes tool calls
- **Artefakty (2.2.4):** Agent čte/píše stejné artefakty (kód, testy, docs)
- **Role (2.2.3):** Agent přebírá část rolí (developer, částečně tester)
- **Tacit knowledge (2.2.3):** Musí být převedeno na explicit (viz 2.4 Scaffolding)

2.3.4 Trade-offs a výzvy

[RAW]

1. Tacit → Explicit Knowledge:

→ `nonaka1995`:

Agent nemá přístup k tacit knowledge (meetingy, tribal knowledge).

Vše musí být explicitně zapsáno: CLAUDE.md, README, specs, komentáře.

Důsledek: Kvalita výstupu agenta závisí na kvalitě dokumentace a instrukcí.

---

2. Agenti vs Týmy (Brooks's Law):

→ `brooks1975`:

Velké týmy = velká komunikační režie ($n(n-1)/2$ kanálů).

S agenty:

- Agenti nepotřebují meetingy
- Ale nerozumí nuancím a kontextu
- Hypotéza: malé týmy + agenti mohou být efektivnější než velké týmy bez agentů

---

3. Micro-waterfall hypotéza:

Nevracíme se s AI agenty zpět k waterfall, jen v menší časové škále?

- **Waterfall:** Requirements → Design → Implementation → Testing [měsíce]
- **Agile:** Planning → Dev → Testing → Review [týdny]
- **AI agent:** Prompt → Generate → Review → Fix [minuty]

Sekvenční kroky zůstávají – jen se zmenšuje batch size a zrychluje feedback loop.

Tento pattern je formalizován jako Spec-Driven Development (viz 2.2.2): "waterfall per increment" – detailní specifikace v rámci každého malého incrementu, ale iterativní přístup mezi incrementy (Piskala, 2026).

Empirická podpora:

→ `watanabe2025agentprs` (ACM TOSEM, under review): 567 Claude Code PRs, 157 projektů. 83.8% acceptance rate (vs 91% human PRs). 54.9% merged bez úprav. Agenti dělají víc refactoring (25% vs 15%), testing (19% vs 5%), docs (22% vs 14%). Ale 45% potřebovalo revizi – hlavně bug fixes a project-specific standardy.

→ `ehsani2026failedprs` (MSR 2026): 33k agent PRs od 5 agentů. Agent PRs by měly být malé a focused – velké PRs mají víc CI selhání. Documentation a build úkoly mají nejvyšší úspěšnost (84%, 74%), bug fixes a performance nejnižší (64%, 55%).

Důsledek pro workflow: Malé, focused incrementy s jasnou specifikací vedou k lepším výsledkům než velké, komplexní úkoly. Podporuje SDD spec-first přístup.

---

4. Další výzvy:

- **Halucinace** – agent může generovat neexistující API, chybný kód
- **Context window** – omezená velikost kontextu
- **Bezpečnost** – agent má přístup k systému (bash, soubory)
- **Evaluace** – jak měřit kvalitu agenta? (viz metriky v praktické části)

Zdroje sekce 2.3:

- `vaswani2017` – Transformer/LLM
- `yao2022react`, `schick2023toolformer` – Tool calls
- `jin2024`, `liu2024llmagents`, `guo2025benchmarks` – Surveys
- `nonaka1995`, `brooks1975` – Knowledge, týmy

2.4 Scaffolding pro agenty

[RAW]

Úvod sekce: Tato sekce navazuje na problém identifikovaný v 2.3.4 – agenti nemají přístup k tacit knowledge. Popisuje state of the art přístupy k řešení tohoto problému a připravuje půdu pro praktickou část, kde bude vybrán a evaluován konkrétní přístup.

2.4.1 Problém kontextu

[RAW]

Navazuje na 2.3.4:

Agent nemá tacit knowledge → potřebuje explicitní kontext. Kvalita výstupu agenta přímo závisí na kvalitě poskytnutého kontextu.

---

1. Tacit → Explicit Knowledge:

→ `nonaka1995`:

Articulation = převod tacit knowledge na explicit.

Pro agenty: vše co člověk "prostě ví" musí být explicitně zapsáno.

---

2. Dualita SE4H vs SE4A:

→ `hassan2025sase` – SASE framework:

"SE for Humans (SE4H) vs SE for Agents (SE4A)"

- **SE4H** – člověk jako "Agent Coach", specifikuje intent, mentoruje
- **SE4A** – strukturované prostředí pro agenty, explicitní artefakty

Klíčový posun: člověk se stává specifikátorem a reviewerem, ne implementátorem.

---

3. Co agent potřebuje vědět:

Propojení s artefakty z 2.2.4:

- **Requirements** – co má být výsledkem
- **Architektura** – kde se změna má udělat, jaké moduly
- **Konvence** – coding standards, patterns, "gotchas"
- **Historie** – proč je kód takový jaký je (rationale)

Propojení s BP: Tato sekce definuje *problém*. Následující sekce popisuje různé *přístupy* k řešení. V praktické části bude vybrán a evaluován konkrétní přístup.

2.4.2 Přístupy k memory a kontextu

[RAW]

State of the art – jaké přístupy existují:

Tato sekce popisuje různé přístupy k poskytování kontextu agentům. Kategorizace podle typu řešení.

---

1. Memory systémy pro agenty:

→ `memorymechanism2024` – Survey on Memory Mechanism:

Systematický přehled memory mechanismů pro LLM agenty.

→ `amem2025` – A-MEM:

Agentic memory založená na Zettelkasten metodě – propojené knowledge networks.

Kategorie memory:

- **Short-term** – kontext aktuální konverzace
- **Long-term** – persistentní mezi sessions

- **Parametric** – v model weights
- **Non-parametric** – external storage (RAG, databáze)

---

2. Context Engineering:

→ `contextengsurvey2025` – Survey of Context Engineering:

"Context Engineering as a formal discipline that transcends simple prompt design"

1400+ papers analyzováno – definuje context engineering jako disciplínu.

→ `ace2025` – Agentic Context Engineering (ACE):

Context jako "evolving playbooks" – akumulace, refinement, organizace strategií.

Komponenty context engineering:

- Context retrieval and generation
- Context processing and management
- RAG (Retrieval-Augmented Generation)
- Tool-integrated reasoning

---

3. Strukturované artefakty:

→ `hassan2025sase` – SASE artifacts:

- **BriefingScript** – mission plan (Goal, What, Context, Blueprint, Validation)
- **MentorScript** – tribal knowledge jako kód, pravidla a normy
- **LoopScript** – workflow orchestration, SOP

BriefingScript sekce:

1. Goal & Why – business value, purpose
2. What & Success Criteria – definition of done, testable properties
3. All Needed Context – curated information, "Known Gotchas"
4. Implementation Blueprint – strategic guidance, constraints
5. Validation Loop – acceptance testing plan

---

4. Repository-level context:

→ `contextmodule2024` – ContextModule:

Edit history jako kontext pro code completion.

Tři typy kontextu: user behavior, similar code, symbol definitions.

→ `bugfixcontext2025` – Bug Fixing with Broader Context:

"Historical commit information" pro bug fixing.

"Co-occurring files" – soubory často commitované společně.

---

5. Git-based přístupy:

→ `gcc2025` – Git Context Controller:

Git jako *metafora* pro agent memory – COMMIT/BRANCH/MERGE operace.

Alternativní pohled (pro praktickou část):

Git není jen metafora – je to existující infrastruktura pro:

- Institutional memory (commit history, blame)
- Explicitní knowledge (commit messages, PR descriptions)
- Rationale (proč byla změna udělána)
- Koordinace (branches, merges)

Hypotéza: Místo vymýšlení nových memory systémů lze efektivně využít Git.

---

6. Specifikace jako vstup pro agenty (SWE-bench):

→ `swebench2024` – SWE-bench (ICLR 2024):

De facto benchmark pro AI coding agenty. 2294 úloh z reálných GitHub repozitářů.

Specifikace úlohy = **GitHub Issue** (natural language popis problému) + přístup k codebase.

Důsledky pro scaffolding:

- GitHub Issues se staly **standardním formátem specifikace** pro AI agenty v akademickém výzkumu
- Agent čte issue description → plánuje → implementuje → testuje
- Issue propojuje specifikaci s git workflow (branch, commits, PR)
- Traceability: Issue #N → branch → commits → PR → merge (propojení s (Gotel & Finkelstein, 1994))

---

7. Co dělá specifikaci efektivní pro LLM agenty:

Empirický výzkum ukazuje které vlastnosti specifikace vedou k lepším výstupům agentů.

Klíčový nález – kvalita requirements = kvalita výstupu:

→ `rope2024` (ACM TOCHI): Silná korelace mezi kvalitou requirements a kvalitou LLM výstupu. Trénink zaměřený na psaní requirements (ROPE) zvedl kvalitu o 20%, zatímco běžný prompt engineering trénink jen o 1%.

→ `ullrich2025` (RE 2025): 18 praktiků z 14 firem – requirements ve stávající podobě jsou **příliš abstraktní** pro přímý LLM vstup. Musí se manuálně dekomponovat a obohatit o design decisions a architekturální kontext.

→ `yang2025underspec` (CMU): Pod-specifikované prompty jsou **2× náchylnější k regresi** při změnách modelu, s poklesy přesnosti přes 20%.

a) 10 alignment rules – empirické pořadí důležitosti:

→ `specine2025` (ICSE 2026): Odvozeno 10 pravidel zarovnání specifikace z RE principů, testováno na 4 LLM a 5 benchmarcích. Tři s nejvyšším dopadem:

1. **Příklady s vysvětlením** (∼14.5% řešených problémů) – konkrétní I/O s explanací
2. **Účel specifikace** (∼13.5%) – co má kód dělat a proč
3. **Výstupní požadavky** (∼11.6%) – jak má vypadat output

Průměrné zlepšení Pass@1: 29.60%. Dalších 7 pravidel (background, key concepts, input requirements, edge cases, APIs, error handling, hints) má nižší ale stále měřitelný dopad.

b) Testy jako součást specifikace:

→ `ticoder2024` (IEEE TSE): Formalizace záměru přes testy (pass/fail feedback) dosáhla 45.97% zlepšení v Pass@1. Navíc výrazně snížila kognitivní zátěž pro člověka.

→ Důsledek: acceptance criteria mapovatelná na testy jsou **empiricky nejdůležitější element**.

c) Doménový kontext (ubiquitous language):

→ `domaincodegen2024` (ACM TOSEM): Zahrnutí domain-specific API knowledge do promptů vedlo k profesionálnějšímu kódu. Podporuje DDD koncept sdíleného doménového slovníku.

d) Pre/post conditions jako design constraints:

→ `newcomb2025prepost` (IEEE Access): Explicitní pre/postconditions "significantly boost initial generation accuracy", zejména u menších modelů.

e) I/O specifikace na různých úrovních abstrakce:

→ `wen2024io`: NL popisy I/O fungují lépe než samotné konkrétní příklady, výrazně snižují

"executable but incorrect" výstupy.

f) Klarifikace ambiguity:

→ `clarifygpt2024` (FSE 2024): Když LLM před generováním kódu klarifikuje nejednoznačné requirements, GPT-4 Pass@1 stoupne z 70.96% na 80.80%.

g) RE metodologie aplikovaná na prompty:

→ `reprompt2025`: 4 RE fáze (elicitation, analysis, specification, validation) mapovány na optimalizaci promptů. Ukazuje že tradiční RE postupy jsou aplikovatelné i pro LLM.

h) Strukturované requirements v benchmarcích:

→ `swebenchpro2025`: Rozšiřuje SWE-bench o explicitní "Requirements" a "Interface" sekce psané lidmi. Jasnější specifikace korelují s vyšší úspěšností agentů.

Syntéza – empiricky podložené tiers specifikačních elementů:

Tier 1 (nejvyšší dopad):

1. **Description / účel** – co a proč (Sommerville, 2016; Tian & Chen, 2026)
2. **Acceptance criteria s příklady** – Given/When/Then, mapovatelné na testy (Fakhoury et al., 2024; Tian & Chen, 2026)
3. **Výstupní specifikace** – formát a struktura výstupu (Tian & Chen, 2026; Wen et al., 2024)

Tier 2 (silně doporučené):

4. **Vstupní specifikace** – datové typy, formáty, constraints
5. **Doménový slovník** – klíčové pojmy z business domény (X. Gu et al., 2024)
6. **Edge/corner cases** – hraniční podmínky

Tier 3 (hodnotné pro složité úlohy):

7. **Pre/postconditions** – stavové podmínky (Newcomb et al., 2025)
8. **Error handling** – chování při nevalidním vstupu
9. **Behavioral model** – state machines pro event-driven logiku (Sommerville, 2016, kap. 5.4)

Porovnání přístupů:

Přístup	Výhody	Nevýhody
Memory systémy	Flexibilní, agent-specific	Nová infrastruktura
Context engineering	Systematické	Komplexní setup
Strukturované artefakty	Explicitní, auditovatelné	Overhead pro člověka
Repository-level	Využívá existující kód	Jen aktuální stav
Git-based	Existující infrastruktura	Vyžaduje disciplínu

Propojení s BP: V praktické části bude evaluován Git-based přístup jako memory layer pro agenty.

2.4.3 Praktické techniky

[RAW]

Konkrétní implementační techniky:

---

1. Agent Harness:

→ Anthropic – "Effective harnesses for long-running agents":

Komponenty harnessu:

- `init.sh` – environment setup
- `claude-progress.txt` – tracking co je hotovo
- Git commits jako checkpointy
- JSON pro feature list (odolnější než Markdown)

Doporučení:

- Incremental progress – jedna feature per session
- Explicit end-to-end testing
- Consistent startup procedures

---

2. Meta-prompt soubory:

→ `hassan2025sase` – AGENT.md pattern:

Soubory jako CLAUDE.md, .clinerules, AGENT.md.

"Employee handbook" pro AI teammates.

Obsah:

- Project-specific konvence
- Architectural decisions
- Known gotchas a warnings
- Coding standards

---

3. Living Documents:

→ `hassan2025sase`:

"Briefing Pack must be a living document, not a static one."

Principy:

- Version-controlled (Git)
- Iterativní refinement na základě feedback
- Auditable history změn
- Single source of truth

---

4. Human-AI Collaboration:

→ `humanai2024` – empirická studie:

22 profesionálních vývojářů, 3 hodiny s ChatGPT.

Klíčové poznatky:

- AI zlepšuje efektivitu code generation
- Human oversight zůstává kritický (complex problem-solving, security)
- Transition from "tool" to "collaborative partner"

Zdroje sekce 2.4:

- `nonaka1995` – tacit → explicit
- `hassan2025sase` – SASE, BriefingScript, MentorScript
- `contextengsurvey2025`, `ace2025` – context engineering
- `memorymechanism2024`, `amem2025` – memory systémy
- `contextmodule2024`, `bugfixcontext2025` – repository context
- `gcc2025` – Git-based přístupy
- `humanai2024` – human-AI collaboration

[RAW]

Formát specifikace: GitHub Issues

Specifikace referenční implementace je strukturována jako GitHub Issues. Volba tohoto formátu vychází z:

1. **Akademický standard** – SWE-bench (Jimenez et al., 2024), de facto benchmark pro AI coding agenty (ICLR 2024), používá GitHub Issues jako specifikaci. 2294 úloh z reálných repozitářů.
2. **Agilní RE praxe** – v agilních týmech user stories a backlog items nahrazují formální SRS dokumenty (Cao & Ramesh, 2008).
3. **Open source praxe** – issue trackery fungují jako de facto requirements management (Scacchi, 2002).
4. **Nativní čitelnost pro agenty** – agent čte issues přes GitHub API nebo CLI, propojuje je s branches a PR.
5. **Traceability** – Issue #N → branch → commits → PR → merge. Přirozená provázanost specifikace s implementací (Gotel & Finkelstein, 1994).

Struktura každého issue vychází z empirického výzkumu o optimální specifikaci pro LLM agenty (viz sekce 2.4.2, bod 7). Studie ukazují, že kvalita requirements přímo koreluje s kvalitou LLM výstupu (Ma et al., 2024) a že tradiční user stories jsou příliš abstraktní pro přímý vstup do LLM (Ullrich et al., 2025) – je nutná dekompozice a obohacení o konkrétní kontext.

Dvě vrstvy specifikace:

Původní návrh obsahoval tři vrstvy (requirements, specification, architecture). Analýza redundance (viz níže) ukázala, že prostřední vrstva (specification: inputs/outputs, pre/postconditions) je implicitně obsažena v acceptance criteria a invariantech. Výsledná šablona proto obsahuje dvě vrstvy:

1. **Requirements** (problémová doména – CO business potřebuje):
   - Title, Description – účel a kontext funkcionality
   - Acceptance criteria – Given/When/Then s konkrétními hodnotami (Fakhoury et al., 2024). Implicitně obsahují vstupy/výstupy (Given/Then), pre/postconditions (Given = precondition, Then = postcondition) i přechody stavů. Jsou přímo mapovatelná na unit testy – explicitní test cases tedy nejsou nutnou součástí specifikace, ale odvozitelným artefaktem
   - Domain glossary – sdílený slovník z business domény (X. Gu et al., 2024)
2. **Architecture** (struktura – JAK je řešení organizované):
   - Type definitions – datové typy, interfaces, enums (Tian & Chen, 2026; Wen et al., 2024)
   - Invariants – business pravidla která musí vždy platit (Newcomb et al., 2025)
   - Behavioral model – state diagram, sekvenční logika (Sommerville, 2016, kap. 5.4)
   - Technické constraints – tech stack, patterns, rozhraní

Referenční implementace používá plnou specifikaci (obě vrstvy). Experimentální běhy používají pouze requirements vrstvu (bez architecture) — úroveň specifikace je fixní proměnná, ne experimentální dimenze.

Zdůvodnění redukce z tří na dvě vrstvy:

Původní třívrstvý návrh obsahoval prostřední vrstvu *Specification* (inputs/outputs, pre/postconditions). Analýza ukázala překryv s ostatními vrstvami: acceptance criteria implicitně obsahují vstupy/výstupy (Given/Then), pre/postconditions (Given = precondition, Then = postcondition) i přechody stavů. Tato redundance představuje problém: pro člověka vyšší cognitive overload, pro LLM agenta plýtvání vzácným context window duplicitními informacemi.

Anthropic (Anthropic, 2025a) zavádí pojem **context rot** – s rostoucím počtem tokenů klesá schopnost modelu přesně vzpomínat informace. Doporučuje "nejmenší možnou sadu high-signal tokenů". IEEE 830 (IEEE, 1998) upozorňuje, že "redundance sama o sobě není chyba, ale snadno k chybám vede". Bockeler (Bockeler, 2025) kritizuje spec-kit (GitHub) za to, že specifikační soubory jsou "repetitive, both with each other, and with the code" – označuje to jako *Verschlimmbesserung* (zhoršení snahou o zlepšení).

Obsah zrušené vrstvy byl absorbován: vstupy/výstupy do acceptance criteria (konkrétní hodnoty v Given/When/Then), datové typy do *type definitions* a pre/postconditions do *invariants* v architektonické vrstvě.

Dvě publikum, různé potřeby:

Specifikace slouží dvěma publikům současně: **AI agentovi** (implementuje z ní kód) a **lidskému vývojáři** (rozumí co se staví a kontroluje co agent vytvořil). Kruchtenův 4+1 model (Kruchten, 1995) argumentuje, že více pohledů je komplementárních **pro různá publika**. Diagramy jsou pro člověka "high-bandwidth" komunikace (rychlé pochopení celkové struktury), zatímco LLM zpracovávají Mermaid diagramy jako text. Konkrétní Given/When/Then scénáře mohou být pro agenta účinnější než vizuální model, ale pro člověka méně přehledné u komplexních systémů.

Zasazení do SASE frameworku:

Hassan et al. (Hassan et al., 2025) navrhují framework Structured Agentic Software Engineering (SASE), který rozlišuje **SE4H** (SE for Humans – člověk jako "Agent Coach" zaměřený na intent, strategii a mentoring) a **SE4A** (SE for Agents – strukturované prostředí pro agenty). Definují tři typy artefaktů: **BriefingScript** (mission brief – co agent má udělat), **LoopScript** (workflow playbook – jak má postupovat) a **MentorScript** (quality normy – jaké standardy dodržovat).

Naše specifikační šablona odpovídá BriefingScript: obsahuje intent (Description), ověřitelná kritéria (Acceptance Criteria) a doménový kontext (Glossary). Soubor `agents.md` se scaffoldingem odpovídá LoopScript a MentorScript: definuje workflow (git conventions, testování)

a kvalitativní normy (code quality).

Kruchtenův Scenarios (+1) view (Kruchten, 1995), který sloužil jako validační most mezi všemi pohledy pro všechny stakeholdery, nachází paralelu v acceptance criteria – ty fungují jako most mezi záměrem člověka a exekucí agenta.

Empirické pořadí důležitosti:

Studie Specine (Tian & Chen, 2026) empiricky měřila dopad jednotlivých elementů na kvalitu generovaného kódu (Pass@1, 4 LLM, 5 benchmarků):

Tier 1 – nejvyšší dopad:

- Příklady s vysvětlením (∼14.5%) → Acceptance criteria
- Účel specifikace (∼13.5%) → Description
- Výstupní požadavky (∼11.6%) → Outputs

Tier 2 – silně doporučené: vstupní požadavky, klíčové pojmy, edge/corner cases.

Tier 3 – hodnotné pro složité úlohy: pre/postconditions (Newcomb et al., 2025), error handling, behavioral model.

Tato šablona kombinuje přístupy podložené výzkumem: structured natural language (Sommerville, 2016, kap. 4.4), test-driven specifikaci (Fakhoury et al., 2024), doménový kontext (X. Gu et al., 2024), redukci specification misalignment (Tian & Chen, 2026), design constraints (Newcomb et al., 2025) a klarifikaci ambiguity (Mu et al., 2024).

Multi-issue gap:

Stávající benchmarky ukazují výrazný propad výkonu agentů při přechodu od single-issue k multi-issue úlohám: SWE-EVO (Wei et al., 2025) reportuje pouze 21 % úspěšnost na evolučních úlohách (průměrně 21 souborů) oproti 65 % na single-issue SWE-Bench; FeatureBench („FeatureBench: Benchmarking Agentic Coding for Complex Feature Development“, 2026) měří 11 % na feature-level úlohách; ACE-Bench („ACE-Bench: End-to-End Feature Development Benchmark“, 2025) 7,5 % na end-to-end feature development. Tato case study (5 issues, celý dunning system) cílí přesně do tohoto rozsahu, kde scaffolding může přinést měřitelný rozdíl.

Test oracle problem — teoretické východisko:

Klíčovým rizikem LLM-generovaných testů je tzv. **test oracle problem** (Mathews & Nagappan, 2024). Mathews et al. ukázali, že nástroje pro automatické generování testů (CoverAgent, CoverUp) systematicky filtrují failing testy a ponechávají pouze passing – výsledkem je, že až

68,1 % vygenerovaných test suites **validuje chybné chování** místo jeho odhalení. Příčinou je, že expected values jsou odvozeny z pozorování kódu, ne ze specifikace.

Chen et al. (Chen et al., 2025) empiricky potvrdili na 500 úlohách SWE-bench, že agent-generované testy slouží primárně jako **observační feedback** (value-revealing prints), ne jako validační nástroj – 83,2 % úloh má stejný výsledek bez ohledu na to, zda agent testy píše. Relační a boundary kontroly (nejcennější pro detekci chyb) tvoří pouze 3–8 % assertions.

Obrana proti těmto anti-patterns:

- **TDD ze specifikace** – expected values vycházejí z acceptance criteria, ne z pozorování kódu. Agent píše testy **před** implementací (red → green → refactor).
- **Failing test = opravit kód, ne test** – zabraňuje selection biasu, kde se zahazují testy odhalující chyby.

3. Metodika

3.1 Výzkumný přístup: návrhový výzkum (DSR)

[DRAFT]

Hevner et al. (Hevner et al., 2004) rozlišují v informatickém výzkumu dva přístupy. Behaviorální výzkum popisuje a vysvětluje existující jevy — například jak vývojáři pracují s AI nástroji. Návrhový výzkum (Design Science Research, DSR) naopak navrhuje nové artefakty a ověřuje, jestli řeší daný problém. Rozdíl je v tom, co je výstupem: popis světa, nebo nástroj který ho mění.

Tato práce spadá do návrhového výzkumu. Artefaktem je sada instrukcí pro AI coding agenta. Instrukce se navrhnou, agent s nimi projede vývojový úkol, z výstupu (kód, testy, git log) se vyhodnotí co fungovalo a co ne, a instrukce se podle toho upraví. Hevner tento postup formalizuje jako build-evaluate cyklus — iterativní smyčku návrhu a vyhodnocení, která se opakuje dokud artefakt nedosáhne požadované kvality.

Dvě fáze výzkumu

Výzkum má dvě fáze, které odpovídají DSR cyklu:

1. **Pilotní iterace** — opakovaně navrhujeme a vyhodnocujeme instrukce. Každá iterace je jeden build-evaluate cyklus: spustíme agenta, analyzujeme výstup, diagnostikujeme problém, upravíme instrukce. Zaznamenáváme co se měnilo a proč. Způsob měření popisuje sekce 3.3, exit kritéria sekce 3.4.4.
2. **Komparativní variace** — z fungujících instrukcí systematicky měníme jednotlivé komponenty a měříme dopad na chování agenta. Změna může být odebrání (ablace — funguje agent bez této instrukce?) nebo nahrazení alternativou (substituce — funguje jiná formulace stejného záměru?). Ablace ukáže *jestli* komponenta záleží, substituce ukáže *proč* a *v jaké formě*.

Case study a generalizace

Výzkum probíhá formou case study na jednom projektu (systém upomínek faktur). Yin (Yin, 2018) rozlišuje dva typy generalizace: statistickou (ze vzorku na populaci) a analytickou (z případu na teorii). Case study neumožňuje říct "tento scaffolding funguje vždy" — k tomu by byl potřeba velký vzorek projektů. Umožňuje ale identifikovat principy a mechanismy: *proč* určité instrukce fungují a jiné ne, a za jakých podmínek.

Více experimentálních běhů na jednom projektu odpovídá tomu, co Yin nazývá embedded

single-case design — jeden případ (projekt) s více vnořenými jednotkami analýzy (jednotlivé běhy). DSR poskytuje celkový rámec (jak iterovat artefakt), case study poskytuje kontext (reálný projekt s deterministickou logikou a ověřitelnými výstupy).

3.2 Výběr projektu pro case study

[DRAFT]

Experiment potřebuje projekt, na kterém lze spustit více běhů s různým nastavením instrukcí a objektivně měřit výsledky. Zvolený projekt musí splňovat:

- **Hard logic** — jasná business pravidla, ne subjektivní výstupy
- **Jasné invarianty** — deterministické chování, ověřitelná správnost
- **Testovatelné** — kvalitu výstupu lze měřit objektivně (testy, mutation score)
- **Přiměřená velikost** — menší projekt umožňuje více experimentálních běhů
- **Reálný use case** — prakticky využitelné, ne umělý příklad

Systém upomínek faktur

[DRAFT]

Systém pro automatické odesílání připomínek k nezaplaceným fakturám. Obsahuje stavový automat pro sledování stavu faktury (nová, po splatnosti, upomínaná, eskalovaná), časové výpočty (pracovní dny, ochranné lhůty), pravidla pro eskalaci a plánování odesílání upomínek.

3.3 Způsob měření

[RAW]

[RAW] Metriky experimentu jsou organizovány podle taxonomie Fenton a Biemana (Fenton & Bieman, 2014): **procesní metriky** (jak agent pracuje), **produktové metriky** (co agent vyrobil) a **metriky efektivity** (za jakou cenu). Toto členění je standardní v softwarovém inženýrství a umožňuje oddělit hodnocení procesu od hodnocení výstupu.

Všechny metriky mají jednoznačný kód pro referenci v tabulkách a across-run srovnáních.

Měření kombinuje tři přístupy: automatické skripty (git log, testy, statická analýza), LLM-as-judge s fixním rubrikem (J. Gu, Xu et al., 2024), a manuální review transkriptu.

Primární zdroje dat:

- **Git log + GitHub API** — commity, branches, PRs, issues

- **Session transcript** (`opencode export`) — kompletní sekvence tool calls a rozhodnutí agenta
- **Zdrojový kód agenta** — vstup pro testy a statickou analýzu

3.3.1 Procesní metriky

[RAW]

[RAW] Procesní metriky (Fenton & Bieman, 2014) měří *jak* agent pracuje — dodržuje instrukce? Postupuje systematicky? Je transparentní?

P1 — TDD compliance. Podíl test-first cyklů ku celkovým implementačním cyklům. Z git logu: poměr sekvencí `test:`→`feat:` ku celkovým implementačním commitům.

P2 — commit granularity. Počet commitů a průměrný počet řádků na commit (Ehsani et al., 2026).

P3 — test integrity. Podíl commitů kde agent změnil assertion v existujícím testu aby matchoval implementaci (místo opravy kódu). Měří sycophantic test modification — reward hacking přes test suite místo řešení problému (Song et al., 2025; Tao et al., 2026). *Ověření:* git diff na test files — klasifikace: nový test vs. změněný assertion (automatické).

P4 — completion honesty. Podíl případů kdy agent deklaruje dokončení ale testy neprochází. Z transkriptu: completion claim vs. poslední test output (semi-automatické).

P5 — kvalita procesních artefaktů. Commit messages (popisnost, atomicita, konvenční prefix), issue descriptions (scope, AC, kontext), PR descriptions (co a proč, odkaz na issue). Hodnocení LLM-as-judge (sekce 3.3.4).

3.3.2 Produktové metriky

[RAW]

[RAW] Produktové metriky (kód Q — quality) (Fenton & Bieman, 2014) měří *co* agent vyrobil — funguje to? Jsou testy kvalitní? Je kód udržovatelný?

Funkční korektnost (Q1–Q2)

Q1 — referenční test pass rate. 40 behavioral testů odvozených z acceptance criteria metodou TDD ze specifikace (Mathews & Nagappan, 2024) (viz sekce 3.4.1). Testují chování přes veřejné API — nezávislé na interní struktuře agentovy implementace. Odpovídá přístupu SWE-bench (Jimenez et al., 2024). Pass rate = projité testy / 40. Binární varianta (pass all / fail) odpovídá metrice % Resolved v SWE-bench.

Q2 — API contract match. `tsc` import + type-check referenčních typů proti agentovu kódu. Ověřuje kompatibilitu s definovaným rozhraním.

Kvalita testů (Q3–Q4)

Q3 — mutation score (Stryker). Podíl zabitých mutantů — měří jestli agentovy testy skutečně detekují chyby, nebo jsou tautologické. Silnější prediktor než strukturální coverage — 36 % chyb odhalitelných pouze mutation testingem (Papadakis et al., 2019). Harman et al. (Harman et al., 2025) potvrdili v produkčním nasazení na Meta, že 70 % mutantů zůstává neodhalených i při plném coverage.

Q4 — AC coverage. Kolik z 24 acceptance criteria má odpovídající test. Měří úplnost pokrytí požadavků, ne jen korektnost implementace. Mapování test → AC manuální nebo LLM-as-judge.

Referenční hodnoty (z referenční implementace):

- Mutation score: $\geq$ 74,3 %
- AC coverage: 24/24

Kvalita kódu (Q5–Q8)

Automatické metriky:

Q5 — lint warnings. `eslint -format json`, počet varování a chyb.

Q6 — typecheck errors. `tsc -noEmit`, počet chyb. Strict mode compliance — počet `any` v kódu.

Q7 — složitost kódu. Cyklomatická složitost per funkce (ESLint `complexity` rule), maximální délka funkce. SWE-bench používá McCabe jako jednu z metrik (Jimenez et al., 2024).

Q8 — kvalita kódu (LLM-as-judge): Naming conventions, separation of concerns, idiomatický TypeScript, zbytečná komplexita. Hodnocení LLM-as-judge (sekce 3.3.4).

3.3.3 Metriky efektivity

[RAW]

[RAW] Metriky efektivity měří zdroje spotřebované při vývoji — standardní dimenze v agent benchmarcích (SWE-bench, AgentBench).

E1 — tokeny. Input/output tokeny: `opencode export` → JSON parsing.

E2 — trvání. Wall-clock time v minutách (session timestamps).

E3 — kompletní dokončení. Crash / no crash + důvod. Počet restartů auto-continue pluginu jako proxy pro autonomii.

S N=1 per run jsou tyto metriky deskriptivní (porovnání napříč běhy), ne inferenční (žádné p-hodnoty).

3.3.4 Hodnocení LLM-as-judge

[RAW]

[RAW] Metriky P5 (procesní artefakty) a Q8 (kvalita kódu) hodnotí aspekty které nelze měřit deterministicky. Pro obě používáme metodu LLM-as-judge (Zheng et al., 2023) — LLM hodnotí výstup agenta podle fixního rubriku.

Self-preference bias a volba modelu

Panickssery et al. (Panickssery et al., 2024) prokázali kauzální vztah mezi schopností LLM rozpoznat vlastní výstupy a preferencí těchto výstupů při hodnocení. Model který generoval hodnocený text má tendenci ho hodnotit výše — mechanismem je nižší perplexita vlastních výstupů. Zheng et al. (Zheng et al., 2023) identifikovali self-enhancement bias jako jeden ze tří systematických biasů LLM-as-judge (spolu s position bias a verbosity bias).

Z toho plyne: model hodnotitele musí být z **jiné rodiny** než model který generoval hodnocený kód (Verga et al., 2024). V našem experimentu generuje kód minimax-m2.5-free (rodina MiniMax). Jako hodnotitele volíme GLM-5 (rodina Zhipu AI) — odlišná architektura i trénovací data, čímž je eliminován perplexitní mechanismus self-preference.

Design hodnocení

- **Model:** GLM-5 (Zhipu AI) přes OpenCode
- **Škála:** 1–3 per dimenze (poor / acceptable / good) — nižší granularita je spolehlivější při malém N
- **Rubric:** fixní prompt s definicí per úroveň a příklady, totožný pro všechny běhy (příloha X)
- **Separátní prompty** per dimenze — model nehodnotí více aspektů najednou
- **Validace:** autor ohodnotí podmnožinu běhů manuálně, shoda s LLM-as-judge vyjádřena Cohenovým κ

Hodnocené dimenze

P5 — procesní artefakty: commit messages (popisnost, atomicita, konvenční prefix), issue descriptions (scope, AC, kontext), PR descriptions (co a proč, odkaz na issue).

Q8 — kvalita kódu: naming conventions, separation of concerns, idiomatický TypeScript, zbytečná komplexita (inverzně).

Rozdíl: P5 měří jak agent *komunikuje* (procesní dimenze), Q8 měří co agent *vyrobil* (produktová dimenze).

[RAW] Rubriky jsou definovány v experimentální infrastruktuře (`infra/judge/p5-process-artifacts.md` a `infra/judge/q8-code-quality.md`). Kompletní znění rubriků včetně příkladů bude v příloze.

3.4 Experimentální design

3.4.1 Referenční implementace

[RAW]

[RAW] Referenční implementace slouží dvěma účelům: validaci specifikace a vytvoření **behavioral test suite** pro měření funkční korektnosti (metrika Q1).

Metoda: TDD z acceptance criteria. Postup odpovídá spec-first TDD (Mathews & Nagappan, 2024): nejprve se napíší behavioral testy přímo z AC ve formátu Given/When/Then — testy jsou zpočátku červené (implementace neexistuje). Následně se implementuje dunning system tak, aby testy postupně zelenaly. Tato sekvence zajišťuje, že expected values pochází ze specifikace, nikoliv z pozorování kódu (test oracle problem (Mathews & Nagappan, 2024)).

Behavioral testy, ne unit testy. Referenční testy testují chování systému přes veřejné API (vstupy a výstupy definované specifikací), nikoliv interní implementaci. Tento přístup odpovídá black-box testování funkčních požadavků (IEEE Computer Society, 2024) a metodice SWE-bench (Jimenez et al., 2024). Výhodou je přenositelnost: stejné testy lze spustit na implementaci libovolného agentního běhu bez znalosti jeho interní struktury.

Validace specifikace. Implementace referenčního řešení odhaluje nejednoznačnosti a mezery v acceptance criteria dříve než experimentální běhy.

Výstup: 40 behavioral testů (TypeScript/Vitest), spustitelné na libovolné implementaci se standardizovaným API.

3.4.2 Fixní proměnné

[RAW]

[RAW] Všechny běhy sdílejí stejný setup — jedinou proměnnou je obsah `AGENTS.md`.

- Prázdné GitHub repo (`AGENTS.md`, `.opencode/config.json`, `.opencode/agents/build.md`, auto-continue plugin)
- Specifikace v GitHub Issue #1 — 24 acceptance criteria, doménový glossary, API contract, out of scope
- Auto-continue plugin (`session.idle` hook s počítadlem restartů a build/test kontrolou)
- Model: minimax-m2.5-free přes OpenCode

- System prompt `build.md` (`mode: primary`, nahrazuje defaultní `qwen.txt` — kódové konvence, žádné procesní instrukce)

3.4.3 Konstrukce instrukcí

[RAW]

[RAW] Jediná proměnná experimentu je obsah `AGENTS.md` — soubor s instrukcemi pro AI agenta. Konstrukce výchozí verze (baseline) vychází z empirických frameworků pro návrh prompt šablon a agentních instrukcí.

Struktura a pořadí komponent. Mao et al. (Mao et al., 2025) analyzovali 2 163 produkčních prompt šablon a identifikovali 7 komponent s empiricky odvozeným pořadím: Role/Directive → Context → Workflow → Output → Constraints. Directive (imperativní instrukce) je nejfrekventovanější komponenta (86,7 %) a patří na začátek dokumentu. Constraints typu exclusion (“nikdy nedělej X”) jsou nejefektivnější typ omezení (46 % z identifikovaných constraints).

Balance tří typů obsahu. Hassan et al. (Hassan et al., 2025) rozlišují tři “scripty” v agentním scaffoldingu: BriefingScript (co a proč — cíl, kontext, success criteria), LoopScript (jak — workflow, kroky) a MentorScript (co nedělat + recovery). Efektivní instrukce kombinují všechny tři; dominance LoopScriptu (jen workflow) vede k agentovi, který neumí reagovat na odchylky.

Obsah instrukcí. Lulla et al. (Lulla et al., 2026) zjistili, že `AGENTS.md` snižuje runtime o 28,6 % a výstupní tokeny o 20 %. Efektivní obsah je architektura projektu a kódové konvence — snižují explorativní navigaci. Verbose procesní instrukce naopak přidávají tokeny bez přínosu. Chatlatanagulchai et al. („Agent READMEs: An Empirical Study of Context Files for Agentic Coding“, 2025) potvrzují: medián délky je 335–535 slov; nejčastější kategorie jsou Testing (75 %), architektura (67,7 %) a vývojový proces (63,3 %).

Meta-princip. Každý řádek instrukcí projde filtrem: “Kdyby tento řádek chyběl, udělal by agent neočividnou chybu?” Pokud ne, řádek přidává tokeny bez benefitu a měl by být odstraněn („Agent READMEs: An Empirical Study of Context Files for Agentic Coding“, 2025).

Mapování očekávaného chování na instrukce. Výchozí verze `AGENTS.md` je zkonstruována tak, aby přímo podporovala chování požadovaná v exit kritériích pilotní fáze (sekce 3.4.4):

- Přečtení specifikace → Directive: “Read Issue #1 for the full specification” (Mao et al., 2025)
- Dekompozice do sub-issues → Workflow Step 1 s bash ukázkou `gh issue create` (Mao et al., 2025)
- Architecture issue před kódem → Workflow Step 1: první issue je architektura (Hassan

et al., 2025; Lulla et al., 2026)
- TDD test-first → Workflow Step 2 (bash blok s red/green cyklem) + Constraint "never implement without failing test" (Hassan et al., 2025; Mathews & Nagappan, 2024)
- Branch per issue → Workflow outer loop (`git checkout main`) + exclusion constraint s negative example (Mao et al., 2025; Razavi & Fard, 2025)
- Průběžné commity → Workflow: explicitní `git add` + `git commit` per fáze (Anthropic, 2025b)
- Lint + typecheck → Workflow: `npm run lint` a `tsc -noEmit` před PR + Constraint: opravit nalezené problémy (Hassan et al., 2025)

Breunig (Breunig, 2025) ukazuje, že když agent opakovaně ignoruje instrukci, řešením je přestrukturování (jiný formát, jiná pozice), nikoliv opakování stejné formulace. Razavi (Razavi & Fard, 2025) potvrzuje, že cílená negative example z konkrétního pozorovaného selhání dramaticky snižuje prompt sensitivity.

3.4.4 Pilotní iterace

[RAW]

[RAW] DSR build-evaluate cyklus aplikovaný na instrukce.

Každá iterace:

1. Spustit agenta s aktuální verzí `AGENTS.md`
2. Analyzovat výstup (git log, session trace, kód, testy)
3. Diagnostikovat selhání — kde a proč agent nedodržel očekávané chování
4. Upravit instrukce s odkazem na literaturu
5. Opakovat dokud agent konzistentně splní exit kritéria

Výstup každé iterace:

- Aktualizovaný `AGENTS.md`
- CHANGELOG záznam: co se změnilo, proč, jaká evidence vedla ke změně
- Behavioral trace z git logu a session trace

Exit kritéria pilotní fáze. Pilot končí když agent v posledním běhu bez manuálního zásahu splní následující požadavky. Kvantitativní metriky odpovídají kódům ze sekce 3.3.

Procesní požadavky:

- Přečte specifikaci (Issue #1) před kódem (transcript — první tool call)
- Decomponuje práci do sub-issues (`gh issue list`)
- Vytvoří architecture issue před implementací (GitHub timeline)
- TDD test-first: P1 $> 80\,\%$
- Průběžné commity: P2 — separátní commity pro test a implementaci

- Branch per issue + PR workflow (`git branch -a` vs. issue count)

Produktové minimum:

- Q1 (referenční testy) > 80 %
- Q2 (API contract) = match
- Q5 + Q6 (lint + typecheck) = 0 chyb

"Konzistentně" = v posledním běhu bez manuálního zásahu. Ostatní metriky (P3–P5, Q3–Q4, Q7–Q8, E1–E3) se zaznamenávají pro across-run srovnání, ale nejsou exit kritéria.

3.4.5 Komparativní variace

[RAW]

[RAW] Z fungující sady instrukcí (výstup pilotní fáze) systematicky měníme jednotlivé komponenty a měříme dopad na chování agenta.

Dva typy změn:

1. **Ablace** — odebrání komponenty úplně. Měří *nutnost*: funguje agent bez této instrukce? Příklad: odebrat TDD instrukci → dělá agent TDD přirozeně?
2. **Substituce** — nahrazení komponenty alternativou. Měří *efekt obsahu*: která varianta produkuje lepší výsledky? Příklad: "strict TDD" vs. "impl-first, testy po stabilizaci".

Ablace ukáže *jestli* komponenta záleží, substituce ukáže *proč* a *v jaké formě*. Konkrétní dimenze variace budou určeny na základě výsledků pilotní fáze — identifikujeme komponenty kde má smysl testovat alternativy.

Prompt sensitivity (Razavi & Fard, 2025) je přiznaná limitation: výsledky mohou záviset na konkrétní formulaci, ne jen na přítomnosti/absenci komponenty.

4. Praktická část

[RAW]

[RAW] Kostra kapitoly — zde budou výsledky experimentu.

4.1 Referenční implementace

[RAW]

[RAW] Výsledky referenční implementace — konkrétní čísla, ne design rozhodnutí (ta jsou v kap03 sekce 3.3).

TODO: Přesunout sem z experimentu:

- Počet testů v behavioral test suite
- Mutation score referenční implementace (Stryker)
- Pokrytí acceptance criteria
- Případné problémy nalezené při implementaci a úpravy specifikace

4.2 Pilotní iterace

[RAW]

[RAW] Výsledky pilotních běhů — DSR build-evaluate cykly.

Pro každou iteraci:

- Co se změnilo v instrukcích oproti předchozí iteraci (a proč)
- Co agent udělal (behavioral trace)
- Compliance score
- Co se naučilo (jaké úpravy instrukcí z toho vyplynuly)

TODO: Popsat pilot-r1 až pilot-r5 (nebo kolik jich bude).

4.3 Ablační fáze

[RAW]

[RAW] Výsledky ablačních běhů — systematické odebírání komponent.

Pro každý ablační běh:

- Která komponenta byla odebrána
- Výsledné chování agenta
- Změna v metrikách oproti baseline (pilotní fungující konfigurace)
- Interpretace: je komponenta nutná / redundantní?

4.4 Přehled výsledků

[RAW]

[RAW] Souhrnná tabulka výsledků všech běhů.

- Tabulka: běh × metriky (mutation score, test pass rate, tokeny, compliance, ...)
- Grafy pokud relevantní
- Konkrétní hodnoty metrik — definice metrik jsou v kap03 sekce 3.4

5. Vyhodnocení a diskuse

[RAW]

[RAW] Kostra kapitoly — zde bude interpretace výsledků z kap04.

5.1 Interpretace výsledků

[RAW]

[RAW] Co výsledky znamenají — propojení s výzkumnými otázkami.

- Které složky scaffoldingu jsou nezbytné a které redundantní?
- Jaký kontext agent potřebuje k dodržování SWE praktik?
- Odpovědi na cíle práce (viz kap01)

5.2 Porovnání s literaturou

[RAW]

[RAW] Zasazení výsledků do kontextu existujícího výzkumu.

- Porovnání s Lulla et al. (AGENTS.md efekt)
- Porovnání s Breunig (prompt vs. model)
- Porovnání s ablačními studiemi (SWE-agent, CCA, RePrompt)
- Kde naše výsledky potvrzují / rozporují existující evidence

5.3 Limity a hrozby validity

[RAW]

[RAW] Přiznání omezení studie.

- Prompt sensitivity (Razavi & Fard, 2025) — malá změna formulace může změnit výsledky
- Jeden model, jeden projekt — analytická generalizace, ne statistická
- Single run per podmínka — omezená reprodukovatelnost
- Case study generalizuje na teorii, ne na populaci (Yin, 2018)

5.4 Doporučení pro praxi

[RAW]

[RAW] Praktické závěry pro vývojáře pracující s AI coding agenty.

- Co zahrnout do AGENTS.md / instrukčních souborů
- Co je zbytečné / kontraproduktivní
- Minimální efektivní scaffolding

5.5 Náměty pro další výzkum

[RAW]

[RAW] Future work.

- Více modelů, více projektů
- Systematické porovnání formátů specifikace
- Multi-agent vs. single-agent scaffolding
- Longitudinální studie (více issues, evoluce projektu)

Závěr

Bibliografie

ACE-Bench: End-to-End Feature Development Benchmark [212 tasks from 16 repos; Claude 4 Sonnet + OpenHands: 7.5% on feature-level tasks]. (2025). https://openreview.net/forum?id=41xrZ3uGuI

Agent READMEs: An Empirical Study of Context Files for Agentic Coding [1925 repos analyzed; AGENTS.md behaves as dynamic configuration; median FRE 16.6 (very difficult reading level)]. (2025). *arXiv preprint arXiv:2511.12884*. https://arxiv.org/abs/2511.12884

Anthropic. (2025a). *Effective Context Engineering for AI Agents* [Introduces "context rot" – model recall degrades with token count. Recommends minimal high-signal token sets]. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

Anthropic. (2025b). *Effective Harnesses for Long-Running Agents* [Two-agent architecture; ablatable: feature list, progress file, init scripts, testing instructions]. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents

Bockeler, B. (2025). *Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl* [Critical analysis of SDD tools. Spec-kit files "repetitive, both with each other, and with the code"]. https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html

Breunig, D. (2025). Don't Fight the Weights [Defines fighting-the-weights: when prompt instructions oppose trained model behavior; recognition signs and mitigation strategies]. https://www.dbreunig.com/2025/11/11/don-t-fight-the-weights.html

Cao, L., & Ramesh, B. (2008). Agile Requirements Engineering Practices: An Empirical Study [16 organizations, 7 agile RE practices identified]. *IEEE Software*, *25*(1), 60–67. https://doi.org/10.1109/MS.2008.1

Ehsani, R., et al. (2026). Where Do AI Coding Agents Fail? An Empirical Study of Failed Agentic Pull Requests in GitHub [33k agent PRs, 5 coding agents, taxonomy of failures]. *Proceedings of MSR 2026*. https://arxiv.org/abs/2601.15195

Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., & Lahiri, S. K. (2024). TiCoder: LLM-Based Test-Driven Interactive Code Generation [Test-driven spec, 45.97% improvement in Pass@1, Microsoft Research]. *IEEE Transactions on Software Engineering*. https://arxiv.org/abs/2404.10100

FeatureBench: Benchmarking Agentic Coding for Complex Feature Development [Agents struggle when scope exceeds single-issue; feature-level tasks significantly harder]. (2026). *arXiv preprint arXiv:2602.10975*.

Fenton, N. E., & Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach* (3. vyd.) [Process vs. product vs. resource metrics taxonomy; GQM framework]. CRC Press.

Gotel, O. C., & Finkelstein, A. C. (1994). An Analysis of the Requirements Traceability Problem. *Proceedings of IEEE International Conference on Requirements Engineering*, 94–101.

Gu, J., Xu, X., et al. (2024). A Survey on LLM-as-a-Judge [Comprehensive survey on using LLMs as evaluators - reliability, bias mitigation, evaluation scenarios]. *arXiv preprint arXiv:2411.15594*. https://arxiv.org/abs/2411.15594

Gu, X., Chen, M., Lin, Y., Hu, Y., Zhang, H., Wan, C., Wei, Z., Xu, Y., & Wang, J. (2024). On the Effectiveness of Large Language Models in Domain-Specific Code Generation [Domain context in prompts improves code quality, supports DDD ubiquitous language]. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3697012

Harman, M., Mao, K., Moran, K., Tufano, R., Gligoric, M., Shi, A., & Havrilla, A. (2025). Mutation-Guided LLM-Based Test Generation at Meta: A White Paper [70% mutants unkilled despite full coverage; 73% engineer acceptance rate; deployed on 10,795 classes].

Hassan, A. E., Li, H., Lin, D., Adams, B., Chen, T.-H., Kashiwa, Y., & Qiu, D. (2025). Agentic Software Engineering: Foundational Pillars and a Research Roadmap [SASE framework, BriefingScript, MentorScript, SE4H/SE4A duality]. *arXiv preprint arXiv:2509.06216*. https://arxiv.org/abs/2509.06216

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research [Foundational DSR framework: build-evaluate cycle, 7 guidelines for design science research]. *MIS Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

Chen, J., Lin, J., Xiong, Y., Lu, J., Zhang, H., & Xie, T. (2025). Rethinking the Value of Agent-Generated Tests in Autonomous Code Repair [83.2% tasks same outcome regardless of test writing; tests serve as observational feedback]. *arXiv preprint arXiv:2505.21615*.

IEEE. (1998). *IEEE Std 830-1998: Recommended Practice for Software Requirements Specifications* (tech. zpr.) ("Redundancy itself is not an error, but it can easily lead to errors"). IEEE.

IEEE Computer Society. (2024). *Guide to the Software Engineering Body of Knowledge* (Version 4.0). https://www.computer.org/education/bodies-of-knowledge/software-engineering

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? [2294 tasks from real GitHub repositories, de facto benchmark for AI coding agents]. *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2310.06770

Kruchten, P. B. (1995). The 4+1 View Model of Architecture [Multiple architectural views are complementary for different stakeholders]. *IEEE Software*, *12*(6), 42–50. https://doi.org/10.1109/52.469759

Lulla, K., Zhu, M., Kalliamvakou, E., & Mohylevskyy, Y. (2026). On the Impact of AGENTS.md Files on AI Coding Agents [124 PRs across 10 repos: -28% runtime, -20% output tokens with AGENTS.md; architectural info and coding conventions most effective]. *arXiv preprint arXiv:2601.20404*. https://arxiv.org/abs/2601.20404

Ma, Y., Peng, Z., & Wu, T. (2024). What Should We Engineer in Prompts? Training Humans in Requirement-Driven LLM Use [ROPE: 20% vs 1% improvement, strong corre-

lation between requirement quality and LLM output quality]. *ACM Transactions on Computer-Human Interaction*. https://doi.org/10.1145/3731756

Mao, Y., He, J., & Chen, C. (2025). From Prompts to Templates: A Systematic Prompt Template Analysis for Real-world LLM Applications [Merged 4 frameworks (Google Cloud, Elavis Saravia, CRISPE, LangGPT); 7 component types; Directive 87% prevalence]. *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*. https://arxiv.org/abs/2504.02052

Mathews, N. S., & Nagappan, M. (2024). LLM-Based Test Generation as Bug Validation: Insights from Reproducing Real-World Bugs [68.1% generated test suites validate bugs instead of detecting them]. *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware)*. https://doi.org/10.1145/3664646.3664766

METR. (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity [RCT: 16 developers, 246 tasks; AI tools slowed experienced developers by 19%; developers believed they were 24% faster]. *arXiv preprint arXiv:2507.09089*. https://arxiv.org/abs/2507.09089

Mu, F., Shi, L., Wang, S., Yang, Z., Li, B., Wang, S., & Yu, Q. (2024). ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification [Clarifying ambiguous requirements improves GPT-4 Pass@1 from 70.96% to 80.80%]. *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/3660810

Newcomb, E., Newcomb, J. L., & Ochoa, M. (2025). Preconditions and Postconditions as Design Constraints for LLM Code Generation [Design constraints significantly boost initial generation accuracy, especially for smaller models]. *IEEE Access*. https://doi.org/10.1109/ACCESS.2025.3573889

Panickssery, A., Bowman, S. R., & Feng, S. (2024). LLM Evaluators Recognize and Favor Their Own Generations [Causal link: self-recognition → self-preference; linear correlation; fine-tuning pushes recognition to 90%+]. *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2404.13076

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2019). Mutation Testing Advances: An Analysis and Survey [Mutation score stronger predictor of fault detection than structural coverage]. *Advances in Computers*, *112*, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

Piskala, D. B. (2026). *Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants* (tech. zpr.) (Technical report. Three levels: spec-first, spec-anchored, spec-as-source. SDD workflow: Specify → Plan → Implement → Validate). arXiv. https://arxiv.org/abs/2602.00180

Razavi, S. P., & Fard, F. H. (2025). What Did I Do Wrong? Quantifying LLMs' Sensitivity and Consistency to Prompt Engineering [Up to 76 accuracy points difference from subtle formatting changes in few-shot settings]. *Proceedings of NAACL*. https://arxiv.org/abs/2406.12334

Scacchi, W. (2002). Understanding the Requirements for Developing Open Source Software Systems [Software informalisms in OSS – issue trackers as de facto requirements]. *IEE Proceedings – Software*, *149*(1), 24–39. https://doi.org/10.1049/ip-sen:20020202

Sommerville, I. (2016). *Software Engineering* (10th). Pearson.

Song, Z., Zhu, Z., Su, Y., & Yu, T. (2025). ImpossibleBench: How Good Are LLMs at Exploiting Test Cases? [GPT-5 cheating rate 76% on impossible tasks; 4 test exploitation strategies]. *arXiv preprint*.

Tao, Y., et al. (2026). Are Coding Agents Generating Over-Mocked Tests? [1.2M commits, 2,168 repos; agents more likely to modify tests and add mocks]. *Proceedings of MSR*.

Tian, Z., & Chen, J. (2026). Aligning Requirement for Large Language Model's Code Generation [10 alignment rules from RE, empirical: 29.60% improvement in Pass@1 across 4 LLMs and 5 benchmarks]. *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. https://arxiv.org/abs/2509.01313

Ullrich, C., Koch, M., & Vogelsang, A. (2025). From Requirements to Code: Understanding Developer Practices in LLM-Assisted Software Engineering [RE 2025, 18 practitioners from 14 companies: requirements too abstract for direct LLM input]. *arXiv preprint arXiv:2507.07548*. https://arxiv.org/abs/2507.07548

Verga, P., Hofstätter, S., Cer, D., & Thorne, J. (2024). Replacing Judges with Juries: Evaluating LLM Generations with a Panel of Diverse Models [Panel of LLM evaluators (PoLL) outperforms single GPT-4 judge; disjoint model families reduce intra-model bias; 7x cheaper]. *arXiv preprint arXiv:2404.18796*. https://arxiv.org/abs/2404.18796

Wei, Y., et al. (2025). SWE-EVO: Long-Horizon Software Evolution Tasks [Multi-step modifications spanning avg 21 files; GPT-5 + OpenHands only 21% vs 65% on single-issue SWE-Bench]. *arXiv preprint arXiv:2512.18470*. https://arxiv.org/abs/2512.18470

Wen, J., Liu, Y., Xie, D., Shi, J., Zhu, W., & Pei, K. (2024). Grounding Data Science Code Generation with Input-Output Specifications [NL I/O summaries better than concrete examples alone, reduces executable-but-incorrect outputs]. *arXiv preprint arXiv:2402.08073*. https://arxiv.org/abs/2402.08073

Yin, R. K. (2018). *Case Study Research and Applications: Design and Methods* (6. vyd.) [Embedded single-case design; analytic generalization (to theory, not population)]. SAGE Publications.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena [Foundational LLM-as-judge paper; position bias, verbosity bias, self-enhancement bias; GPT-4 >80% human agreement]. *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2306.05685

[RAW]

Zdroje k zpracování do BibTeX (původně z notes/sources.md)

1. PRIMÁRNÍ ZDROJE (Frameworky a Metodiky)

GITHUB NEXT. Spec Kit: Spec-Driven Development for AI Agents [online]. 2024. https://github.com/github/spec-kit – Klíčový zdroj pro koncept "Spec-Driven Development".

BMAD-CODE-ORG. BMAD METHOD: Breakthrough Method for Agile AI-Driven Develo-

pment [online]. GitHub, 2025. https://github.com/bmad-code-org/BMAD-METHOD – Metodika pro řízení AI agentů v agilním vývoji.

ANTHROPIC. Effective Harnesses for Long-Running Agents [online]. Anthropic Engineering Blog, 2024. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents – Technický popis problémů s dlouhodobou pamětí agentů.

METR. Model Evaluation and Threat Research [online]. 2024. https://metr.org/ – Standardy pro hodnocení bezpečnosti a schopností modelů.

THEDOTMACK. claude-mem: Persistent Memory System for Claude Code [online]. GitHub, 2025. https://github.com/thedotmack/claude-mem – Příklad implementace hooks v CLI agentech - persistentní paměť přes session.

1.2 SYSTÉMOVÉ MYŠLENÍ V SWE

PETKOV, Doncho et al. Information Systems, Software Engineering, and Systems Thinking: Challenges and Opportunities. International Journal of Information Technologies and Systems Approach. https://www.igi-global.com/gateway/article/2534 – Mapuje historii systémového přístupu v IS a SWE. Propojení systémového myšlení s praxí SWE je stále nedotažené.

MONAT, Jamie a GANNON, Thomas. Systems Thinking: A Review and Bibliometric Analysis. MDPI Systems, 2020. https://www.mdpi.com/2079-8954/8/3/23 – Přehled co systémové myšlení je a kde se používá. Interdisciplinární - SWE, management, vzdělávání.

ALHARTHI, Sultan et al. A Systems Thinking Approach to Improve Sustainability in Software Engineering. MDPI Sustainability, 2023. https://www.mdpi.com/2071-1050/15/11/8766 – Praktická aplikace - dívají se na vývoj jako systém (developeři, zákazníci, stakeholders).

2. ODBORNÉ STUDIE

2.1 Přehledové studie (Surveys) - LLM agenti v SE

LIU, Junwei et al. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv preprint arXiv:2409.02977. 2024. https://arxiv.org/abs/2409.02977 – Komplexní přehled 106 prací o LLM agentech v SE, kategorizace z pohledu SE i agentů.

JIN, Haolin et al. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv preprint arXiv:2408.02479. 2024. https://arxiv.org/abs/2408.02479 – Pokrývá requirements, code generation, testing, maintenance - celý SDLC.

Comprehensive Survey on Benchmarks and Solutions in Software Engineering of LLM-Empowered Agentic System. arXiv preprint arXiv:2510.09721. 2025. https://arxiv.org/html/2510.09721 – Přes 150 paperů, taxonomie řešení a benchmarků.

A Survey on Code Generation with LLM-based Agents. arXiv preprint arXiv:2508.00083. 2025. https://arxiv.org/abs/2508.00083 – Single-agent a multi-agent architektury, aplikace

napříč SDLC.

2.2 Agentic Software Engineering - Klíčové práce

Agentic Software Engineering: Foundational Pillars and a Research Roadmap. arXiv preprint arXiv:2509.06216. 2025. https://arxiv.org/html/2509.06216v2 – Přehodnocení SE pro spolupráci člověk-agent. Framework podobný SAE úrovním autonomie. Rozlišuje SE 2.0 (AI-augmented) vs SE 3.0 (Agentic SE).

AKBAR, Muhammad Azeem et al. Agentic AI in Software Engineering: Practitioner Perspectives Across the Software Development Life Cycle. SSRN. 2025. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5520159 – Rozhovory s 21 experty, pokrývá celý SDLC. Zjištění: agenti redefinují hranice mezi fázemi SDLC.

Autonomous Agents in Software Development: A Vision Paper. Springer, 2024. https://link.springer.com/chapter/10.1007/978-3-031-72781-8_2 – 12 LLM agentů spolupracujících na celém SDLC.

2.3 Evaluace a produktivita

METR. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. 2025. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/ – RCT studie: zkušení vývojáři s AI jsou o 19% pomalejší - překvapivé zjištění.

2.4 Metriky kvality software a AI agentů

ISO/IEC. ISO/IEC 25010:2023 - Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. 2023. https://www.iso.org/standard/78176.html – Industry standard pro kvalitu software. 8 charakteristik, Functional Suitability obsahuje Completeness, Correctness, Appropriateness.

ISO 25000. ISO 25010 Software Quality Model. 2023. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 – Přehledný popis ISO 25010 modelu kvality.

LXT. AI Agent Evaluation: Comprehensive Framework for Measuring Agent Performance. 2024. https://www.lxt.ai/blog/ai-agent-evaluation/ – Moderní framework pro evaluaci AI agentů: Task completion, Accuracy, Safety/trust (policy compliance, transparency), Tool usage.

Weights & Biases. AI Agent Evaluation: Metrics, Strategies, and Best Practices. 2024. https://wandb.ai/onlineinference/genai-research/reports/AI-agent-evaluation-Metrics-strategies-and-best-practices--VmlldzoxMjM0NjQzMQ – Praktický průvodce metrikami pro AI agenty.

SOMMERVILLE, Ian. Software Engineering. 10th ed. Pearson, 2016. Chapter 24: Quality Management (s. 705–728). – Rozlišení control metrics (procesní) vs. predictor metrics (produktové). Vztah proces-produkt u SW. Relevantní pro oporu dimenzí Functional Quality a Compliance v kap03.

McCONNELL, Steve. Code Complete: A Practical Handbook of Software Construction. 2nd ed. Microsoft Press, 2004. Chapter 28 (s. 715, Table 28-2). – Tabulka “Useful Software-Development Measurements” – kategorie Size a Overall Quality (defekty, defekty/KLOC, mean time between failures).

IEEE COMPUTER SOCIETY. SWEBOK: Guide to the Software Engineering Body of Knowledge. Version 4.0. IEEE, 2024. Chapter 12: Software Quality (s. 248–256), Chapter 6: Software Maintenance (s. 176). – Software Quality Measurement (s. 253) – kvantifikace atributů pro rozhodování. Míry údržby (s. 176): complexity, maintainability, testability, reliability.

JIMENEZ, Carlos E. et al. SWE-bench: Can Language Models Resolve Real-world Github Issues? In: ICLR. 2024. Appendix C.7 (s. 28). – Software Engineering Metrics v benchmarku – cyklomatická složitost (McCabe), Halstead measures pro hodnocení kódu agentů.

2.5 Základní LLM studie

JIMENEZ, Carlos E. et al. SWE-bench: Can Language Models Resolve Real-world Github Issues? In: The Twelfth International Conference on Learning Representations (ICLR). 2024. https://arxiv.org/abs/2310.06770 – Hlavní benchmark pro hodnocení schopností programovacích agentů.

LIU, Nelson F. et al. Lost in the Middle: How Language Models Use Long Contexts. arXiv preprint arXiv:2307.03172. 2023. https://arxiv.org/abs/2307.03172 – Klíčová studie vysvětlující, proč pouhé zvětšení kontextového okna nestačí.

VASWANI, Ashish et al. Attention Is All You Need. Advances in Neural Information Processing Systems, 2017. https://arxiv.org/abs/1706.03762 – Základní paper definující Transformer architekturu a mechanismus pozornosti (self-attention).

WEI, Jason et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems, 2022. https://arxiv.org/abs/2201.11903 – Základ pro techniky prompt engineeringu používané v práci.

3. TEORIE SOFTWAROVÉHO INŽENÝRSTVÍ A ARCHITEKTURY

RICHARDS, Mark a FORD, Neal. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media, 2020. ISBN 978-1492043454. – Moderní přehled architektonických stylů a charakteristik ("ilities").

FOWLER, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. ISBN 978-0321127426. – Katalog základních návrhových vzorů pro podnikové aplikace.

KHONONOV, Vlad. Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. 1. vyd. O’Reilly Media, 2021. ISBN 978-1098100131. – Definuje pojmy

jako "Bounded Context"a "Ubiquitous Language", které jsou analogií pro kontext LLM.

ISO/IEC. ISO/IEC/IEEE 42010:2011 Systems and software engineering — Architecture description. 2011. – Mezinárodní standard definující základní pojmy popisu architektury.

IEEE COMPUTER SOCIETY. SWEBOK: Guide to the Software Engineering Body of Knowledge. Version 3.0. IEEE, 2014. https://www.swebok.org/ – Standardní taxonomie softwarového inženýrství.

BROWN, Simon. The C4 model for visualising software architecture. 2024. https://c4model.com/ – Metodika pro hierarchický popis architektury, vhodná pro strojové zpracování.

ARC42. arc42 Template for Software Architecture Documentation. 2024. https://arc42.org/ – Pragmatická šablona pro strukturování architektonické dokumentace.

4. DALŠÍ ZDROJE (Historický kontext a Procesy)

NATO SCIENCE COMMITTEE. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 1968. – Historický kontext vzniku disciplíny.

KAUR, Rupinder a SENGUPTA, Jyotsna. Software Process Models and Analysis on Failure of Software Development Projects. In: arXiv preprint arXiv:1306.1068. 2013.

Přílohy

A. Formulář v plném znění

B. Zdrojové kódy výpočetních procedur