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. Richard Antonín Novák, Ph.D.

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 15. 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]

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 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

[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

[DRAFT]

Práce pokrývá celý SDLC na jednoduchém projektu. Součástí je vlastní implementace, která poskytuje hloubku porozumění potřebnou pro návrh experimentů.

Práce se zaměřuje na:

- Nastavení a použití existujících nástrojů (GitHub, CLI agents)
- Exploratory case study - hledání vzorů a doporučení

Práce se nezaměřuje na:

- Programování nových nástrojů od nuly
- Porovnávání různých LLM modelů
- Porovnávání různých programovacích jazyků
- Vývoj produkčního nástroje/produktu

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 (**scacchi2002**). V agilních týmech se user stories a backlog items používají místo SRS dokumentů (**cao2008**). 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 (**sdd2026**).

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 (**gotel1994**))

---

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 (**specine2025**; Sommerville, 2016)
2. **Acceptance criteria s příklady** – Given/When/Then, mapovatelné na testy (**ticoder2024**; **specine2025**)
3. **Výstupní specifikace** – formát a struktura výstupu (**specine2025**; **wen2024io**)

Tier 2 (silně doporučené):

4. **Vstupní specifikace** – datové typy, formáty, constraints
5. **Doménový slovník** – klíčové pojmy z business domény (**domaincodegen2024**)
6. **Edge/corner cases** – hraniční podmínky

Tier 3 (hodnotné pro složité úlohy):

7. **Pre/postconditions** – stavové podmínky (**newcomb2025prepost**)
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

3. Metodika

3.1 Výběr projektu pro case study

[RAW]

Práce se zaměřuje na řízení a scaffolding - jde více do šířky než do hloubky. Proto potřebujeme menší projekt, na kterém můžeme spustit více běhů s různými nastaveními scaffoldingu a měřit výsledky.

Pro experiment potřebujeme projekt který:

- **Hard logic** - jasná business pravidla, ne subjektivní výstupy (např. generování textu)
- **Jasné invarianty** - deterministické chování, matematicky ověřitelné správnost
- **Testovatelné** - lze objektivně měřit kvalitu výstupu
- **Přiměřená velikost** - menší projekt umožňuje více experimentálních běhů
- **Reálný use case** - prakticky využitelné, ne umělý příklad

Systém upomínek faktur

[RAW]

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 (kdy poslat další upomínku, kdy předat k vymáhání)
- Plánování odesílání upomínek

3.2 Zvolená metodika: Spec-Driven Development

[RAW]

Pro referenční implementaci i experimenty je zvolena metodika **Spec-Driven Development (SDD)** na úrovni **spec-first** (**sdd2026**).

Zdůvodnění volby:

1. **Specifikace řídí implementaci** – kvalita specifikace přímo ovlivňuje kvalitu výstupu agenta (viz 2.4.2 bod 7). SDD formalizuje tento princip.

2. **Waterfall per increment** – každý issue = jeden increment s detailní specifikací, ale mezi incrementy iterativní přístup. Odpovídá micro-waterfall hypotéze (2.3.4) podpořené empirickými daty (**watanabe2025agentprs**; **ehsani2026failedprs**).
3. **Malé, focused úkoly** – empirická data ukazují že malé agent PRs mají vyšší úspěšnost (**ehsani2026failedprs**). SDD spec-first přirozeně vede k dekompozici na testovatelné incrementy.
4. **Spec-first stačí** – pro jednorázový experiment není potřeba spec-anchored (udržovat sync spec-kód). Spec se napíše, agent implementuje, vyhodnotí se.

SDD workflow v kontextu BP:

1. **Specify** – napsat GitHub Issue se strukturovanou specifikací (šablona viz níže)
2. **Plan** – (pro agenty: agent sám plánuje; pro referenci: autor plánuje)
3. **Implement** – implementace podle specifikace
4. **Validate** – testy (unit, acceptance criteria), review

3.3 Referenční implementace

[RAW]

Vlastní vývoj systému upomínek faktur se všemi náležitostmi softwarového inženýrství:

- Specifikace a dokumentace
- Implementace (TypeScript)
- Testy (unit, integration)
- Git workflow (issues, commits, PR conventions)
- Quality gates (linting, type checking)

Tato implementace slouží jako "ground truth"pro porovnání.

Formát specifikace: GitHub Issues

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

1. **Akademický standard** – SWE-bench (**swebench2024**), 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 (**cao2008**).
3. **Open source praxe** – issue trackery fungují jako de facto requirements management (**scacchi2002**).
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 (**gotel1994**).

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 (**rope2024**) a že tradiční user stories jsou příliš abstraktní pro přímý vstup do LLM (**ullrich2025**) – 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 (**ticoder2024**). 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 (**domaincodegen2024**)
2. **Architecture** (struktura – JAK je řešení organizované):
   - Type definitions – datové typy, interfaces, enums (**wen2024io**; **specine2025**)
   - Invariants – business pravidla která musí vždy platit (**newcomb2025prepost**)
   - Behavioral model – state diagram, sekvenční logika (Sommerville, 2016, kap. 5.4)
   - Technické constraints – tech stack, patterns, rozhraní

Toto rozdělení slouží i jako základ pro experimentální dimenzi "úroveň detailu specifikace" (viz sekce Experimenty): referenční implementace používá plnou specifikaci (obě vrstvy), experimenty mohou poskytovat agentovi pouze vybrané vrstvy.

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 (**anthropic2025context**) zavádí pojem **context rot** – s rostoucím počtem tokenů klesá schopnost modelu přesně vzpomínat informace. Doporučuje "nejmenší možnou sadu

high-signal tokenů". IEEE 830 (**ieee830**) upozorňuje, že "redundance sama o sobě není chyba, ale snadno k chybám vede". Bockeler (**bockeler2025sdd**) 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 (**kruchten1995**) 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.

Experimenty mohou ukázat optimální kombinaci elementů – ne "čím víc, tím lépe", ale **která minimální sada** reprezentací je efektivní pro obě publika současně.

Zasazení do SASE frameworku:

Hassan et al. (**hassan2025sase**) 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). Experimentální dimenze "úroveň detailu specifikace" přímo testuje to, co Hassan et al. nazývají **duality of control** – kdy dát agentovi strukturu a kdy ho nechat rozhodovat autonomně.

Kruchtenův Scenarios (+1) view (**kruchten1995**), 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 (**specine2025**) 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 (**newcomb2025prepost**), error handling, behavioral model.

Tato šablona kombinuje přístupy podložené výzkumem: structured natural language (Sommerville, 2016, kap. 4.4), test-driven specifikaci (**ticoder2024**), doménový kontext (**domaincodegen2024**), redukci specification misalignment (**specine2025**), design constraints (**newcomb2025prepost**) a klarifikaci ambiguity (**clarifygpt2024**).

3.4 Experimenty

[RAW]

Příprava různých nastavení scaffoldingu pro agenty:

- Instrukce a procesy (jak má agent postupovat)
- Git automatizace a skripty
- Kontext který má agent k dispozici a který si vytváří
- Nastavení odvozená z literatury a spec-driven development přístupů

Scaffolding jako experimentální dimenze:

Kromě úrovně detailu specifikace je další dimenzí míra scaffoldingu v repozitáři. Instrukce pro agenta jsou definovány v souboru `agents.md` (emerging standard pro AI coding agenty), který obsahuje konvence pro git workflow, testování, code quality a proces vývoje. Experimenty mohou testovat i schopnost agenta nastavit si scaffolding sám (linting, test framework, project structure) vs. dostat ho předpřipravený.

Klíčová experimentální dimenze – úroveň detailu specifikace:

Referenční implementace používá plnou specifikaci (requirements + architecture). Experimenty poskytují agentovi různé podmnožiny:

- **Full** – obě vrstvy (requirements + architecture)
- **Medium** – requirements + partial architecture (typy a invarianty, bez behavioral modelu)
- **Minimal** – pouze requirements (Description, Acceptance criteria, Glossary)

Toto měří dopad úrovně specifikace na kvalitu výstupu agenta – empiricky podpořeno studi-

emi (**specine2025**; **rope2024**; **yang2025underspec**).

Spuštění agentů s různými nastaveními scaffoldingu na stejném zadání. Zaznamenání průběhu a výstupů.

Odvozování testů z acceptance criteria:

Acceptance criteria ve formátu Given/When/Then (BDD) s konkrétními hodnotami jsou přímo mapovatelná na unit testy. Schopnost agenta korektně odvodit test suite z acceptance criteria je měřitelná dimenze experimentu – odpovídá zjištění TiCoder (**ticoder2024**), kde formalizace záměru přes testy vedla k 45.97% zlepšení Pass@1.

3.5 Analýza

[RAW]

[RAW] Tradiční přístupy k měření kvality SW (podpora pro volbu dimenzí):

Sommerville (Ch. 24, s. 705–728):

- Rozlišuje **control metrics** (procesní – sledují proces vývoje) vs. **predictor metrics** (produktové – měří vlastnosti kódu/dokumentů)
- Vztah proces-produkt u SW není přímočarý jako ve výrobě – SW je designován, ne vyráběn, vliv individuálních dovedností je velký (s. 706)
- Produktové metriky (LOC, cyklomatická složitost) nemají jasný a konzistentní vztah ke kvalitativním atributům (s. 721)
- → Naše dimenze Functional Quality = predictor metrics, Compliance = control metrics

McConnell – Code Complete (Ch. 28, s. 715, Table 28-2):

- Kategorie měření: Size (LOC, třídy, komentáře) a Overall Quality (počet defektů, defekty/KLOC, mean time between failures)
- Praktický pohled – co se dá reálně měřit v projektu

SWEBOK v4 (Ch. 12, s. 248–256; Ch. 6, s. 176):

- Software Quality Measurement (s. 253) – kvantifikace atributů pro rozhodování
- Míry údržby (s. 176): complexity, maintainability, testability, supportability, reliability
- Odkaz na ISO 25010 jako standard pro kvalitativní charakteristiky (s. 46, 256)

SWE-bench (Appendix C.7, s. 28):

- Cyklomatická složitost (McCabe) a Halstead measures jako metriky pro hodnocení kódu v benchmarku
- Příklad jak existující benchmarky měří kvalitu kódu agentů – ale jen funkční/strukturální, ne procesní

Jin et al. 2024 – LLM Agents SWE Survey (Table VII, s. 21):

- Přehled evaluačních metrik: Accuracy, Pass@k, Task Completion Time, Task Success, Execution Accuracy, Win-Rate
- Většina existující literatury měří hlavně funkční kvalitu výstupu
- → Naše dimenze Compliance a Alignment jsou méně pokryté v literatuře – vlastní přínos

[RAW]

Hodnocení výstupů agentů probíhá ve čtyřech dimenzích: funkční kvalita, procesní kvalita, efektivita a alignment.

3.5.1 Functional Quality (Funkční kvalita)

Měření funkčních vlastností výstupu dle ISO 25010:

Completeness (Úplnost):

- Míra pokrytí požadované funkcionality
- Měření: Checklist požadavků ze specifikace → procento implementovaných

Correctness (Správnost):

- Správnost implementace - funguje to jak má?
- Měření: Spuštění referenčních testů na kód agenta (pass rate)
- Kvalita testů agenta: Mutation testing (Stryker) - mutation score určuje jak dobře testy detekují chyby

3.5.2 Compliance (Procesní kvalita)

Dodržování softwarově-inženýrských praktik:

Workflow:

- Dodržení flow: issues → branch → commits → PR
- Měření: Automatická kontrola git historie a GitHub artefaktů

Conventions (Konvence):

- Kvalita commit messages (formát, atomicita, srozumitelnost)
- Kvalita issues (popis, acceptance criteria)
- Kvalita dokumentace a PR description
- Měření: LLM-as-a-judge s definovaným rubrikem (Gu, Xu et al., 2024)

Transparency (Transparentnost):

- Vysvětluje agent svá rozhodnutí?
- Dokumentuje postup a důvody?
- Měření: LLM-as-a-judge + manuální review

3.5.3 Efficiency (Efektivita)

Náklady na dosažení výsledku:

- **Token usage** - spotřeba tokenů (náklady na API)
- **Iterations** - počet pokusů a oprav potřebných k dokončení
- **Time** - celkový čas do dokončení
- **Human intervention** - míra nutných lidských zásahů a korekcí

Měření: Logování z agenta a konverzačních sessions.

3.5.4 Metody měření

Kombinace tří přístupů:

- **Automatické** - testy, mutation testing, git log analýza, token counting
- **LLM-as-a-judge** - hodnocení subjektivních aspektů (kvalita commit messages, dokumentace) pomocí LLM s definovaným rubrikem
- **Manuální review** - kvalitativní zhodnocení celku autorem

LLM-as-a-judge přístup využívá strukturované hodnocení kde LLM dostane kritéria a škálu, a konzistentně hodnotí všechny běhy. Validace tohoto přístupu probíhá porovnáním s manuálním hodnocením na vzorku (Gu, Xu et al., 2024).

3.5.5 Alignment (Soulad se záměrem)

Alignment měří, zda agent pochopil skutečný záměr zadání - ne jen doslovnou instrukci, ale co uživatel skutečně chtěl (Gu, Xu et al., 2024).

Agent může mít 100% Correctness a Completeness, ale být misaligned - technicky splnil zadání, ale výsledek neodpovídá záměru.

Co se hodnotí:

- **Over-engineering** - přidal agent funkcionalitu která nebyla požadována?
- **Under-delivering** - vynechal agent implicitní požadavky které byly zřejmé z kontextu?
- **Misinterpretation** - pochopil agent zadání špatně?

- **Scope adherence** - držel se agent vymezeného rozsahu?

Měření: Manuální review autorem + LLM-as-a-judge porovnávající zadání vs. skutečný výstup.

3.5.6 Vyhodnocení

Identifikace vzorů - která nastavení scaffoldingu vedla k lepším výsledkům v jednotlivých dimenzích. Porovnání trade-offs (např. vyšší kvalita vs. vyšší náklady).

4. Praktická část

5. Vyhodnocení a diskuse

Závěr

Bibliografie

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

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

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

[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 Development [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 vysvět-

lující, 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