

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 18. 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

Úvod	10
1 Vymezení problému a cílů práce	11
1.1 Motivace	11
1.2 Cíle práce	11
1.3 Rozsah práce	11
2 Teoretická východiska	13
2.1 Softwarové inženýrství	13
2.1.1 Definice a vymezení oboru	13
2.1.2 Historický kontext	15
2.1.3 Komplexita software	16
2.2 Životní cyklus a metodiky	17
2.2.1 Fáze životního cyklu	17
2.2.2 Modely a metodiky	18
2.2.3 Role a komunikace	20
2.2.4 Nástroje a artefakty	22
2.3 Agentic coding	24
2.3.1 Základní pojmy	24
2.3.2 Typy coding agentů	26
2.3.3 Jak agenti mění SDLC	26
2.3.4 Trade-offs a výzvy	27
2.4 Scaffolding pro agenty	29
2.4.1 Problém kontextu	29
2.4.2 Přístupy k memory a kontextu	30
2.4.3 Praktické techniky	35
3 Metodika	37
3.1 Výběr projektu pro case study	37
3.2 Zvolená metodika: Spec-Driven Development	37
3.3 Referenční implementace	38
3.4 Experimenty	41
3.4.1 Experimentální design	41
3.5 Analýza	48
3.5.1 Functional Quality (Funkční kvalita)	49
3.5.2 Compliance (Procesní kvalita)	49
3.5.3 Efficiency (Efektivita)	49
3.5.4 Metody měření	50
3.5.5 Alignment (Soulad se záměrem)	50
3.5.6 Vyhodnocení	50

4 Praktická část	52
5 Vyhodnocení a diskuse	53
Závěr	54
Bibliografie	55
A Formulář v plném znění	60
B Zdrojové kódy výpočetních procedur	61

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/produkту

2. Teoretická východiska

[DRAFT]

Tato kapitola vysvětuje 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, vysvětuje 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 stavby.

→ 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 nereduкуje (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 funkcionality)
 - 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:

→ [liu2024allagents](#):

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:

→ liu202411agents:

- **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:

- **Rychlosť** – 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 více 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 více 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 prompts 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 benchmarkech. 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:

→ **tocoder2024** (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.”

Principle:

- Version-controlled (Git)
 - Iterativní refinement na základě feedback
 - Auditabile 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 escalaci (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** (Piskala, 2026).

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 (Ehsani et al., 2026; Watanabe et al., 2025).
3. **Malé, focused úkoly** – empirická data ukazují že malé agent PRs mají vyšší úspěšnost (Ehsani et al., 2026). SDD spec-first přirozeně vede k dekompozici na testovatelné incremente.
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]

Experiment má tři úrovně:

1. **Referenční implementace** (human-guided) – autor dekomponuje specifikaci do sub-issues, kontroluje průběh, agent implementuje podle předpřipravených issues. Slouží jako ground truth pro metriky (referenční testy, mutation score, kvalita kódu).
2. **Baseline experiment** (full-scale autonomní) – agent dostane pouze specifikaci (issue #1) + plný agents.md + plné prostředí. Sám dekomponuje, orchestrá, implementuje. Maximum instrukcí, žádný lidský zásah. Měří výkon plně autonomního agenta s nejlepším scaffoldingem.
3. **Ablační běhy** – jako baseline, ale s odebranými komponentami scaffoldingu (viz katalog ablatale komponent). Měří dopad jednotlivých komponent.

Porovnání referenční vs. baseline ukazuje vzdálenost autonomního agenta od human-guided vývoje. Porovnání baseline vs. ablace ukazuje dopad jednotlivých scaffolding komponent.

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 redundancy (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

Toto rozdělení slouží jako základ pro experimentální dimenzi “specifikace” (viz sekce Experimenty): referenční implementace používá plnou specifikaci (obě vrstvy), experimenty variaují úroveň detailu – Full (obě vrstvy) vs. Minimal (pouze requirements).

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 **lidé skému 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.

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. (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). 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 (Kruchten, 1995), který sloužil jako validační most mezi

všemi pohledy pro všechny stakeholders, 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).

3.4 Experimenty

[RAW]

3.4.1 Experimentální design

Experiment používá **meta-instrukční ablaci** – systematické odebírání meta-instrukcí (co + proč, nikoliv jak) za účelem měření jejich individuálního přínosu ke kvalitě výstupu agenta. Místo explicitního scaffoldingu (předpřipravené soubory, konfigurace) dostane agent sadu meta-instrukcí a sám si navrhne konkrétní přístup. Ablace = odebrání jedné dimenze meta-instrukcí. Baseline (R0) obsahuje všechny tři dimenze; referenční implementace (human-guided) slouží jako ground truth pro metriky.

[RAW]

[RAW] **Volba experimentálního přístupu:**

Při návrhu experimentu byly zváženy tři přístupy k ablaci scaffoldingu:

(1) *Vrstvová ablace* (SWE-agent (Yang et al., 2024), Confucius (Y. Wang et al., 2025)): ode-

bírání celých vrstev (CI, agent configs, spec detail). Problém: vrstvy míchají fáze SDLC – odebrání “procesních instrukcí” zasahuje do Requirements (TDD), Design (workflow) i Implementation (git konvence) současně. Nelze izolovat přínos konkrétní komponenty.

(2) *Fázová ablace*: organizace podle SDLC fází (Requirements → Design → Implementation). Čistější rámec, ale naráží na cascade dependencies – Requirements ablace závisí na tom co bylo v Design fázi. Izolace vyžaduje fixaci jedné dimenze a variaci druhé, což exponenciálně zvyšuje počet běhů.

(3) *Meta-instrukční ablace*: místo explicitního scaffoldingu (předpřipravené soubory, konfigurace) dostane agent meta-instrukce které říkají *co* udělat, ale nikoliv *jak*. Ablace = odebrání jedné meta-instrukce. Výhody: (a) binární manipulace (instrukce přítomna nebo ne), (b) výsledky jsou univerzální – meta-instrukce nejsou vázané na konkrétní projekt, (c) žádná studie dosud neabovala jednotlivé meta-instrukce u coding agentů.

Zvolen přístup (3) – meta-instrukční ablace. Přístupy (1) a (2) jsou pokryty existující literaturou (SWE-agent abluje ACI komponenty, Confucius abluje scaffolding features, FeatureBench („FeatureBench: Benchmarking Agentic Coding for Complex Feature Development“, 2026) abluje spec tiers). Přístup (3) přináší nový metodologický příspěvek.

Ablace meta-instrukcí:

Experiment testuje přínos jednotlivých meta-instrukcí pro kvalitu výstupu AI agenta. Místo explicitního scaffoldingu (předpřipravené konfigurace, workflow instrukce, architektura) dostane agent sadu meta-instrukcí – každá říká *co* udělat a *proč*, ale nikoliv *jak*. Agent si na základě meta-instrukce navrhne vlastní přístup.

Tento design je motivován dvěma zjištěními: (1) Suzgun a Kalai (Suzgun & Kalai, 2024) ukázali, že meta-prompting (task-agnostic scaffolding) přináší 17,1% zlepšení oproti standardnímu promptingu; (2) Breunig (Breunig, 2026) empiricky demonstroval, že záměna system promptů mezi coding agenty mění chování agenta více než volba modelu – tentýž model s promptem Codex produkuje metodický workflow, s promptem Claude iterativní. Žádná studie dosud neabovala *jednotlivé* meta-instrukce u coding agentů – toto je hlavní metodologický příspěvek této práce.

Fixní proměnné (stejné pro všechny běhy):

- Prázdné GitHub repo (pouze `package.json`, `tsconfig.json`)
- Specifikace v GitHub Issue #1 – plné acceptance criteria a doménový glossary, bez architecture layer
- Auto-continue plugin (`session.idle` hook s počítadlem restartů)
- Model: GLM-5, výchozí temperature OpenCode
- Fixní části system promptu: role definition, tool instructions, output format, fighting-the-weights instrukce (Breunig, 2026)

Standardizovaná šablona meta-instrukce:

Každá meta-instrukce má identickou strukturu – variuje pouze obsah. Šablona vychází ze systematické analýzy prompt komponent napříč čtyřmi frameworky (Zheng et al., 2025): Directive (87 % prevalence), Context (56 %), Output Format (40 %), Constraints (36 %).

[CÍL] : <co udělat>

[KONTEXT] : <proč je to důležité pro kvalitu výstupu>

[VÝSTUP] : <očekávaný artefakt>

[OMEZENÍ] : <co nedělat>

Standardizace eliminuje formulační bias – všechny instrukce mají stejnou délku, specifičnost a strukturu. Prompt sensitivity (Razavi & Fard, 2025) je přiznána jako limitation; mitigace: fixní šablona, deterministická temperature, behavioral trace.

Tři dimenze meta-instrukcí:

Původní návrh obsahoval pět granulárních meta-instrukcí (sub-agents, architektura, dekompozice, workflow, quality tooling). Analýza nezávislosti ukázala, že dekompozice, role separation, workflow a testovací strategie jsou facety téhož procesu (What/Who/How/When organizace práce) – nejsou vzájemně nezávislé a jejich izolovaná ablace by vytvářela nesmyslné konfigurace (např. sub-agenti bez issues k delegaci).

Výsledný design používá tři ortogonální dimenze, odvozené systematicky ze tří zdrojů: (1) SASE framework (Hassan et al., 2025) rozlišuje BriefingScript (fixní specifikace), LoopScript → O, MentorScript/AEE → Q a architektonické rozhodnutí → P; (2) Wang et al. (J. Wang et al., 2024) klasifikují agentní aktivity na Team Organization → O, Decision Making → P, Testing/QA → Q; (3) RepairAgent (Bouzenia et al., 2025) abluje funkčně nezávislé komponenty – náš design toto respektuje.

Dim.	Název	Meta-instrukce (CÍL)	Očekávané artefakty
P	Planning/Design	Navrhni architekturu řešení před implementací – typy, moduly, rozhraní	Design decisions v issues, type definitions
O	Organization	Zorganizuj si práci – rozděl na části, definuj role, nastav testovací strategii (testy ze specifikace před implementací)	Sub-issues, agent configs, workflow v AGENTS.md
Q	Quality	Nastav si quality feedback mechanismy – linting, testy, CI	Nakonfigurované nástroje, CI pipeline

Každá dimenze operuje na jiné úrovni: P rozhoduje *co* stavět (technická struktura), O rozhoduje *jak* pracovat (organizace procesu), Q rozhoduje *čím* kontrolovat kvalitu (feedback

mechanismy). Tato ortogonalita zajišťuje, že odebrání jedné dimenze neovlivní logiku ostatních.

Ablační běhy:

Run	P	O	Q	Co měříme
R0	✓	✓	✓	Baseline – plný meta-prompt
R1	–	✓	✓	Přínos upfront designu (Perry & Wolf, 1992; Sommerville, 2016)
R2	✓	–	✓	Přínos organizace práce („FeatureBench: Benchmarking Agentic Coding for Complex Feature Development“, 2026; Huang et al., 2024; Mathews & Nagappan, 2024)
R3	✓	✓	–	Přínos quality infrastruktury (Harman et al., 2025; Lulla et al., 2026; Papadakis et al., 2019)

Čtyři běhy, single run per podmínu. Každý běh probíhá v separátním GitHub repozitáři (čistý stav). Artefakty (commit history, issues, PR) zachovány pro post-hoc analýzu. Variace obsahu meta-instrukcí (meta vs. explicit formulace) je navržena jako future work – explicit instrukce jsou vysoce citlivé na konkrétní obsah (Razavi & Fard, 2025), zatímco meta-instrukce jsou robustnější vůči formulaci.

Metriky:

- **Primární:** mutation score (Stryker), test pass rate (referenční testy), počet restartů (auto-continue)
- **Sekundární:** spotřeba tokenů, čas do dokončení, compliance a alignment (LLM-as-a-judge)
- **Behavioral trace:** co agent reálně udělal – navrhl architekturu před kódem (P)? rozložil práci na sub-issues (O)? nastavil CI a linting (Q)? Ověruje, že ablace meta-instrukce měla skutečný efekt na chování agenta.

Pilot studie: Před experimentálními běhy proběhne pilotní validace setupu na R0. Pilot identifikuje technické problémy (timeouty, bugy v orchestraci, nedostatky v šabloně) a kalibruje metriky. Iterace na pilotu se nepočítají do výsledků studie.

Deployment a Maintenance mimo scope: experiment měří co agent vytvoří; npm package nemá reálné uživatele ani provoz.

Proč tři SDLC fáze (bez Testing): Testing fáze se v kontextu TDD rozpadá na dvě aktivity: psaní testů = interpretace requirements (spustitelná specifikace (IEEE Computer

Society, 2024; Sommerville, 2016)), spouštění testů = verifikační feedback loop implementace. Test oracle problem (Mathews & Nagappan, 2024) potvrzuje, že testy odvozené z kódu (nikoliv ze specifikace) validují chybné chování v 68,1 % případů.

Katalog ablutable komponent (brainstorm):

Následující seznam identifikuje komponenty, které lze jednotlivě odebrat a měřit jejich dopad. Vychází z empirických ablačních studií v literatuře – SWE-agent (Yang et al., 2024) abluje ACI komponenty, CCA (Y. Wang et al., 2025) abluje scaffolding features, Anthropic (Anthropic, 2025b) identifikuje failure-specific komponenty.

Specifikace (BriefingScript):

- Architecture vrstva (types, invariants, behavioral model) (Tian & Chen, 2026)
- Doménový glossary (X. Gu et al., 2024)
- Detail acceptance criteria (konkrétní hodnoty vs. obecný popis) (Fakhoury et al., 2024)
- Behavioral model / state diagram (Sommerville, 2016)

Instrukce (LoopScript + MentorScript):

- `agents.md` jako celek (Hassan et al., 2025)
- TDD instrukce (testy před implementací) (Mathews & Nagappan, 2024)
- Git workflow konvence (branching, commits, PR)
- Code quality standardy (strict mode, no-any)
- Code review instrukce

Prostředí (Agent Execution Environment):

- CI pipeline (automatická kontrola)
- Pre-konfigurovaný tooling (`tsconfig`, `eslint`, `vitest`, `stryker`)
- Project structure (adresářová struktura, `package.json`)

Orchestrace a session management:

- Session-per-issue granularita – empiricky validovaná jako optimální scope (Spotify Engineering, 2025); agenti selhávají při feature-level scope („FeatureBench: Benchmarking Agentic Coding for Complex Feature Development“, 2026)
- Issue-based task decomposition – dekompozice úkolů do sub-issues s dependencies; task decomposition jako context management strategie (Kang & Kim, 2024)
- Sub-agent delegace – orchestrátor deleguje issues na sub-agenty s čistým kontextem; hierarchická orchestrace dosahuje SOTA (Zhang et al., 2025); trust calibration mezi delegátorem a delegátem (Google DeepMind, 2026)
- Role separation – oddělení analytických a exekučních rolí. MASAI (Arora et al., 2024) definuje 5 specializovaných sub-agentů se structured artifact passing (ne konverzací); AgentCoder (Huang et al., 2024) empiricky validuje separaci test designu od implementace (96,3 % HumanEval); Agyn (Benkovich & Valkov, 2026) replikuje inženýrský tým

s dedikovanými rolemi a dosahuje 72,2 % SWE-bench 500. Studie ukazují, že organizační design je stejně důležitý jako schopnosti modelu.

- Single vs. multi-agent trade-off – přínos multi-agent architektury klesá s rostoucí schopností modelů („Single-agent or Multi-agent Systems? Why Not Both?“, 2025); hybridní přístup (multi-agent jen tam kde je potřeba) přináší +1–12 % accuracy při –88 % cost. Multi-agent systémy mají 14 identifikovaných failure modes ve 3 kategoriích (system design, inter-agent misalignment, task verification) (Cemri et al., 2025). Minimální efektivní rozdělení = separace testování od implementace.
- Meta-prompting – místo explicitních instrukcí agent dostane meta-instrukci navrhnutou si vlastní workflow; task-agnostic scaffolding pattern (Suzgun & Kalai, 2024); automatizovaný design agentních systémů (Hu et al., 2025)
- Completion verification loop – mechanismus zajišťující, že agent dokončí celý projekt, ne jen jeden krok. Tři varianty: (1) Ralph Loop – vnější cyklus s čistým kontextem per iteraci, stav přes soubory/issues, implementován v Claude Code i OpenCode; (2) session.idle hook – plugin uvnitř agentního frameworku, při ukončení agenta ověří podmínu dokončení (např. počet otevřených issues přes GitHub API) a pokud není splněna, re-injektuje prompt s aktuálním stavem; počet restartů slouží jako metrika autonomie agenta; (3) instruktážní – agent dostane instrukci “neskončí dokud všechny issues nejsou uzavřeny”, bez vnější kontroly.
- Retry strategie – single-shot vs. fresh-context retry vs. child fix issues; scaffolding a retry jsou nezávisle hodnotné a multiplikativní („Agentic Program Repair with Neuro-Symbolic Approach“, 2025)
- Strukturovaná dokumentace v issues – záznam pro následující agent session (co bylo zkušeno, co selhalo, aktuální stav) (Anthropic, 2025b). Každá inter-agent zpráva je potenciální failure point; chain-style error propagation kaskáduje malé nepřesnosti do systémových selhání („AgentAsk: Multi-Agent Systems Need to Ask“, 2025). Tři validované mechanismy předávání stavu: (1) progress file + git commity (Anthropic, 2025b), (2) GitHub issues jako komunikační kanál (issue → PR → review), (3) structured artifact passing s formální output specifikací (Arora et al., 2024)

Context window management:

- Context window degraduje s délkou, ne jen “dojde” – coding performance klesá z 29 % na 3 % při long-context úlohách („LongCodeBench: Evaluating Coding LLMs at 1M Context Windows“, 2025); U-shaped křivka, informace uprostřed se ztrácí (Liu et al., 2024); nerovnoměrná degradace napříč 18 modely (Hong et al., 2025)
- Observation masking (skrytí detailů tool outputu) je stejně efektivní jako LLM summarization za poloviční cenu (Lindenbauer et al., 2025)

Tento katalog byl při návrhu experimentu seskupen do tří ortogonálních dimenzií: **P** (Planning/Design) pokrývá architektonické rozhodnutí, **O** (Organization) pokrývá orchestraci, dekompozici, role separation, workflow a testovací strategii, **Q** (Quality) pokrývá tooling a CI. Položky z katalogu, které se týkají specifikace (BriefingScript), jsou fixní proměnnou – specifikace je stejná pro všechny běhy. Položky z context window management jsou infrastrukturální

a řeší je agentní framework (OpenCode).

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.

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 (Fakhoury et al., 2024), kde formalizace záměru přes testy vedla k 45.97% zlepšení Pass@1.

Strategie testování a mutation testing:

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.
- **Mutation testing jako metrika kvality testů** – strukturální coverage (branch, statement) je nutná, ale nedostatečná podmínka. Papadakis et al. (Papadakis et al., 2019) v přehledové studii ukázali, že mutation score je silnější prediktor detekce reálných chyb než strukturální coverage – 36 % chyb je odhalitelných pouze mutation testingem. Harman et al. (Harman et al., 2025) potvrdili v produkčním nasazení na Meta (10 795 tříd), že 70 % mutantů zůstává neodhalených i při plném coverage.

Pro referenční implementaci volíme Stryker (mutation testing framework pro TypeScript) jako objektivní metriku kvality testů agenta. Mutation score měří, jak dobře testy detekují simulované chyby – na rozdíl od coverage, která měří pouze dosažitelnost kódu.

Doplňková dimenze – formát/reprezentace specifikace:

Nezávisle na úrovni detailu ovlivňuje kvalitu výstupu i způsob strukturování informace. Volba kombinovaného formátu (structured text + tabulky + behavioral model) vychází z teorie vizuálních notací (Moody, 2009) a výzkumu informační hustoty diagramů (Larkin & Simon, 1987). Systematické porovnání formátů je navrženo jako future work.

3.5 Analýza

[RAW]

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

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 rubrikiem (J. 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 (J. Gu, Xu et al., 2024).

3.5.5 Alignment (Soulad se záměrem)

Alignment měří, zda agent pochopil skutečný záměr zadání - ne jen doslovou instrukci, ale co uživatel skutečně chtěl (J. 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 funkcionality která nebyla požadována?
- **Under-delivering** - vynechal agent implicitní požadavky které byly zřejmě 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 – které dimenze meta-instrukcí (P, O, Q) měly největší dopad na kvalitu výstupu agenta. Porovnání trade-offs (např. vyšší kvalita vs. vyšší spotřeba tokenů).

[RAW]

[RAW] Rozšíření: meta vs. explicit instrukce

Ablační série měří přínos jednotlivých dimenzií meta-instrukcí. Přirozenou navazující otázkou je: záleží na *formulaci* instrukce? Meta-instrukce říkají co a proč (“navrhni architekturu před implementací”); explicit instrukce říkají přesně jak (“vytvoř type definitions v `src/types.ts`, rozděl moduly na `transition.ts` a `business-days.ts`”).

Porovnání meta vs. explicit by přímo testovalo SASE *duality of control* (Hassan et al., 2025) – kdy dát agentovi strukturu a kdy ho nechat rozhodovat autonomně. Explicit instrukce jsou však vysoko citlivé na konkrétní obsah (Razavi & Fard, 2025) – tentýž záměr lze formulovat mnoha způsoby s odlišnými výsledky. Meta-instrukce jsou robustnější vůči formulaci, protože abstrahují nad konkrétním provedením.

Toto rozšíření je mimo scope této BP – vyžaduje vlastní sérii běhů s kontrolou formulačního biasu. Je navrženo jako přímé pokračování.

4. Praktická část

5. Vyhodnocení a diskuse

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>
- AgentAsk: Multi-Agent Systems Need to Ask [Every inter-agent message is a potential failure point; edge-aware clarification policy reduces chain-style error propagation]. (2025). *arXiv preprint arXiv:2510.07593*. <https://arxiv.org/abs/2510.07593>
- Agentic Program Repair with Neuro-Symbolic Approach [Agent alone 28.5%; + scaffolding 42.3%; + retry 61.0%; scaffolding and retry are multiplicative]. (2025). *arXiv preprint arXiv:2507.18755*.
- 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>
- Arora, D., Sonwane, A., Wadhwa, N., Mehrotra, A., Utpala, S., Bairi, R., Kanade, A., & Nararajan, N. (2024). MASAI: Modular Architecture for Software-engineering AI Agents [5 specialized sub-agents, structured artifact passing; 28.33% SWE-bench Lite; modular wins via different strategies per sub-agent]. *arXiv preprint arXiv:2406.11638*. <https://arxiv.org/abs/2406.11638>
- Benkovich, N., & Valkov, V. (2026). Agyn: Team-Based Autonomous Software Engineering [72.2% SWE-bench 500; replicates engineering team structure; organizational design as important as model improvements]. *arXiv preprint arXiv:2602.01465*. <https://arxiv.org/abs/2602.01465>
- 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>
- Bouzenia, I., Devanbu, P., & Pradel, M. (2025). RepairAgent: An Autonomous, LLM-Based Agent for Program Repair [Component ablation study: tools, state machine, memory, localization; each removed independently]. *Proceedings of the ACM on Software Engineering, 2(FSE)*. <https://arxiv.org/abs/2403.17134>
- Breunig, D. (2026). How System Prompts Define Agent Behavior [Analyzed 6 coding agent prompts; prompt swaps change behavior more than model choice; fighting-the-weights pattern]. <https://www.dbreunig.com/2026/02/10/system-prompts-define-the-agent-as-much-as-the-model.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>
- Cemri, M., Pan, M. Z., Yang, S., et al. (2025). Why Do Multi-Agent LLM Systems Fail? [MAST taxonomy: 14 failure modes in 3 categories; 1600+ annotated traces across

- 7 MAS frameworks]. *arXiv preprint arXiv:2503.13657*. <https://arxiv.org/abs/2503.13657>
- 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*.
- Google DeepMind. (2026). Intelligent AI Delegation [Trust calibration framework for AI-to-AI delegation; authority gradient issues]. *arXiv preprint arXiv:2602.11865*. <https://arxiv.org/abs/2602.11865>
- 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>
- Hong, K., Troynikov, A., & Huber, J. (2025). Context Rot: How Increasing Input Tokens Impacts LLM Performance [18 LLMs tested; performance increasingly unreliable with input length; distractors amplify degradation].
- Hu, S., Lu, C., & Clune, J. (2025). Automated Design of Agentic Systems [Meta Agent Search iteratively programs new agents; discovered agents transfer across domains and models]. *Proceedings of the International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2408.08435>
- Huang, D., Bu, J., Zhang, J., Luck, M., & Wen, H. (2024). AgentCoder: Multi-Agent-Based Code Generation with Iterative Testing and Optimisation [Programmer + Test De-

- signer + Test Executor; 96.3% HumanEval; separating test design from code prevents test oracle bias]. *arXiv preprint arXiv:2312.13010*. <https://arxiv.org/abs/2312.13010>
- 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>
- Kang, Y., & Kim, J. (2024). Chain of Agents: Large Language Models Collaborating on Long-Context Tasks [Task decomposition as context management strategy; +10% over RAG and multi-agent baselines]. *NeurIPS*.
- 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>
- Larkin, J. H., & Simon, H. A. (1987). Why a Diagram is (Sometimes) Worth Ten Thousand Words [Diagrams enable perceptual inference – spatial arrangement makes implicit relationships explicit]. *Cognitive Science*, 11(1), 65–100. <https://doi.org/10.1111/j.1551-6708.1987.tb00863.x>
- Lindenbauer, T., et al. (2025). The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management [Observation masking halves cost, matches LLM summarization; hybrid reduces cost 7–11% further]. *NeurIPS*.
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts [U-shaped performance curve; info in middle performs worse than no-context baseline]. *Transactions of the Association for Computational Linguistics*, 12.
- LongCodeBench: Evaluating Coding LLMs at 1M Context Windows [Claude drops from 29% to 3% on long-context coding; all models degrade]. (2025). *arXiv preprint arXiv:2505.07897*.
- 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>
- 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>
- Moody, D. L. (2009). The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering [9 principles for effective visual notations: semiotic clarity, perceptual discriminability, semantic transparency, etc.]. *IEEE Transactions on Software Engineering*, 35(6), 756–779. <https://doi.org/10.1109/TSE.2009.67>
- 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>
- 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>
- Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 40–52.
- 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]. *IEEE Proceedings – Software*, 149(1), 24–39. <https://doi.org/10.1049/ip-sen:20020202>
- Single-agent or Multi-agent Systems? Why Not Both? [Multi-agent benefits diminish with better models; hybrid: +1–12% accuracy, -88% cost]. (2025). *arXiv preprint arXiv:2505.18286*. <https://arxiv.org/abs/2505.18286>
- Sommerville, I. (2016). *Software Engineering* (10th). Pearson.
- Spotify Engineering. (2025). *Context Engineering: Background Coding Agents (Part 2)* [1500+ merged PRs; “do one change at a time”; 10 turns per session, 3 retries]. <https://>

- //engineering.atspotify.com/2025/11/context-engineering-background-coding-agents-part-2
- Suzgun, M., & Kalai, A. T. (2024). Meta-Prompting: Enhancing Language Models with Task-Agnostic Scaffolding [LLM as conductor managing multiple expert instances; task-agnostic scaffolding pattern]. *arXiv preprint arXiv:2401.12954*. <https://arxiv.org/abs/2401.12954>
- 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>
- Wang, J., Wang, K., Chen, Y., Peng, X., & Zheng, Z. (2024). LLM-based Agents for Software Engineering: A Survey [Survey of 124 papers on LLM agents for SE; 59.7% are multi-agent; taxonomy of agent roles (manager, assistant, specialist)]. *arXiv preprint arXiv:2409.02977*. <https://arxiv.org/abs/2409.02977>
- Wang, Y., et al. (2025). Confucius Code Agent: Scalable Agentic Scaffolding for SWE Tasks [4 scaffolding features ablated; “scaffolding, not model capability, is primary determinant of agent performance”]. *arXiv preprint arXiv:2512.10398*.
- Watanabe, M., Li, H., Kashiwa, Y., Reid, B., Iida, H., & Hassan, A. E. (2025). On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub [Under review. 567 Claude Code PRs, 157 projects, 83.8% acceptance rate]. *ACM Transactions on Software Engineering and Methodology*.
- 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>
- Yang, J., Jimenez, C. E., Wettig, A., Liber, K., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering [ACI ablation: +10.7pp over baseline shell; context management and error guardrails largest gains]. *NeurIPS*.
- Zhang, J., et al. (2025). AgentOrchestra: Hierarchical Multi-Agent Framework [Central planning agent delegates to specialized sub-agents; TEA protocol; 89.04% on GAIA benchmark]. *arXiv preprint arXiv:2506.12508*. <https://arxiv.org/abs/2506.12508>
- Zheng, Y., Rao, B., Yang, S., Chen, S., & Jiang, H. (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>

[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/2071-1050/15/11/876> – Praktická aplikace - dívají se na vývoj jako systém (developeri, 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/is-0-25000-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 – cyklometrická 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 SOFTWAREOVÉHO 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 softwareového 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