

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Evaluation of Code Generated by LLM-Based Coding Agents

Bc. Matěj Klouček

Supervisor: Ing. Jan Šedivý, CSc.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2026

I. Personal and study details

Student's name: **Klouček Matěj** Personal ID number: **502373**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Science**
Study program: **Open Informatics**
Specialisation: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Evaluation of Code Generated by LLM-Based Coding Agents

Master's thesis title in Czech:

Hodnocení kódu generovaného agenty založenými na LLM

Name and workplace of master's thesis supervisor:

Ing. Jan Šedivý, CSc. Big Data and Cloud Computing CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **02.02.2026**

Deadline for master's thesis submission: **22.05.2026**

Assignment valid until: **by the end of summer semester 2026/2027**

Head of department's signature

Vice-dean's signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Bc. Klouček Matěj
Student's signature

I. Personal and study details

Student's name: **Klouček Matěj** Personal ID number: **502373**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Science**
Study program: **Open Informatics**
Specialisation: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Evaluation of Code Generated by LLM-Based Coding Agents

Master's thesis title in Czech:

Hodnocení kódu generovaného agenty založenými na LLM

Guidelines:

LLM based coding agents are capable of generating complete software systems. However, determining whether the generated code is functionally correct remains a challenging task, especially in the absence of reference implementations. The goal of this thesis is to evaluate functional correctness and overall code quality. The student should carry out the following tasks:

- Research current trends in the design of AI agents, with a particular focus on LLM-based coding agents and their architectures.
- Select a suitable coding agent and use it as the source of generated code for evaluation.
- Design and implement an automated evaluation pipeline for generated code, including the generation of tests used to assess code functionality.
- Propose and validate a concrete metric for evaluating the quality of code generated by coding agents.
- Integrate the code evaluation pipeline into the coding agent's workflow and report the resulting improvement in the agent's performance.

Bibliography / sources:

SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering
John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, Ofir Press
CodeT: Code Generation with Generated Tests
Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, Weizhu Chen
A Survey on Evaluating Large Language Models in Code Generation Tasks
Liguo Chen¹, Qi Guo¹, Hongrui Jia¹, Zhengran Zeng¹, Xin Wang¹, Yijiang Xu¹, Jian Wu³, Yidong Wang¹, Qing Gao¹, Jindong Wang², Wei Ye¹, Shikun Zhang¹

DECLARATION

I, the undersigned

Student's surname, given name(s): Klouček Matěj
Personal number: 502373
Programme name: Open Informatics

hereby declare that the Master's Thesis titled

Evaluation of Code Generated by LLM-Based Coding Agents

is my own independent work and that I have declared all sources of information used in accordance with the Methodological Guideline for adhering to ethical principles when elaborating an academic final thesis and the Framework Rules for the use of artificial intelligence at CTU for study and pedagogical purposes in Bachelor and continuing Masters studies.

I declare that I have used artificial intelligence tools during the preparation and writing of the final thesis.
I've verified the generated content. I confirm that I am aware that I am fully responsible for the content of the final thesis.

In Prague on 19.04.2026

Bc. Matěj Klouček

.....
student's signature

Acknowledgements

I would like to thank my supervisor, Ing. Jan Šedivý, CSc., for his expert guidance throughout this work. I am also grateful to my colleagues in the Alquist Coder team at CIIRC for their collaboration on the Amazon Nova AI Challenge 2026 and for the work that parts of this thesis build upon. Finally, I would like to thank Amazon for the opportunity to participate in the challenge and for providing the team with AWS credits, without which it would not have been possible to conduct the experimental part of this thesis.

Abstract

This thesis investigates whether the integration of automated evaluation mechanisms into the workflow of LLM-based coding agents can improve the functional correctness of generated code. A custom coding agent is designed and implemented, capable of modifying full-stack web applications based on natural-language task descriptions. To provide feedback on the generated code during development, the agent is extended with two internal evaluation components: a test-generation-based mode, in which the agent writes and executes its own Playwright end-to-end tests against the running application, and an agentic mode, in which a separate evaluator agent verifies the implemented changes through browser automation via the Playwright MCP server. A structured evaluation framework assesses utility along three dimensions: functional correctness, regression safety, and efficiency. The experimental evaluation covers five language models and twenty generated full-stack web applications. Among error-free runs, the evaluation-enhanced configurations improve aggregate Pass@1 from 26.8% to 31.6% (self-evaluation) and 31.5% (agentic), with gains of up to 10.5 percentage points for the strongest models. A cross-model experiment further reveals that for weaker models, the primary bottleneck is evaluation quality rather than coding ability: pairing a weak coder with a strong evaluator more than doubles its task completion rate.

Keywords: large language models, coding agents, agentic systems, end-to-end testing, automated test generation, code quality evaluation, Model Context Protocol

Supervisor: Ing. Jan Šedivý, CSc.

Abstrakt

Tato práce zkoumá, zda začlenění automatizovaných evaluačních mechanismů do činnosti kódovacích agentů založených na velkých jazykových modelech může vést ke zlepšení funkční správnosti generovaného kódu. V rámci práce je navržen a implementován vlastní kódovací agent schopný upravovat full-stack webové aplikace na základě instrukcí v přirozeném jazyce. Za účelem poskytování zpětné vazby ke generovanému kódu je agent rozšířen o dvě interní evaluační komponenty: režim založený na generování testů, v němž agent sám vytváří a spouští end-to-end Playwright testy, a agentní režim, v němž samostatný evaluační agent interaguje s běžící aplikací prostřednictvím Playwright MCP serveru. Užitečnost generovaného kódu je posuzována ve třech dimenzích: funkční správnost, odolnost vůči regresím a efektivita. Experimentální část zahrnuje pět jazykových modelů a dvacet generovaných full-stack webových aplikací. Konfigurace rozšířené o vnitřní evaluaci zvyšují agregovanou hodnotu metriky Pass@1 z 26,8 % na 31,6 % (autoevaluace) a 31,5 % (agentní režim) u úspěšných běhů, přičemž u nejsilnějších modelů dosahuje zlepšení až 10,5 procentního bodu. Experiment napříč modely ukazuje, že u slabších modelů není hlavním omezením samotná schopnost generovat kód, ale kvalita vyhodnocení: spojení slabšího generátoru kódu se silnějším hodnotícím agentem více než zdvojnásobuje úspěšnost.

Klíčová slova: velké jazykové modely, kódovací agenti, agentní systémy, end-to-end testování, automatické generování testů, hodnocení kvality kódu, Model Context Protocol

Překlad názvu: Hodnocení kódu generovaného agenty založenými na LLM

Contents

1 Introduction	1	
1.1 Problem Statement	2	
1.2 Outline	3	
2 Large Language Models and Code Generation	4	
2.1 Transformers	4	
2.1.1 Limitations of Recurrent Architectures	4	
2.1.2 Self-Attention Mechanism	5	
2.1.3 Transformer Architecture	7	
2.1.4 Positional Encoding	8	
2.1.5 Transformer Variants and Decoder-Only Models	9	
2.2 Large Language Models	10	
2.2.1 Autoregressive Language Modeling	10	
2.2.2 Large-Scale Pretraining	10	
2.2.3 Context Length in LLMs	11	
2.2.4 Scaling Laws and In-Context Learning	11	
2.2.5 Instruction Tuning and Alignment	12	
2.3 Code Generation with LLMs	13	
2.3.1 Large Language Models for Program Synthesis	14	
2.3.2 Pretraining on Code Corpora	15	
2.3.3 Benchmarks and Evaluation of Generated Code	16	
3 Agentic Systems	18	
3.1 The Agent Paradigm	18	
3.2 LLM-based Agentic Systems	19	
3.3 Coding Agents	20	
4 Software Testing and Code Evaluation	23	
4.1 Automated Software Testing	23	
4.2 End-to-End Testing	24	
4.3 Test Generation for AI-Generated Code	24	
5 Related Work	26	
5.1 Test Generation with LLMs	26	
5.2 Evaluation of Coding Agents	28	
6 Methodology	31	
6.1 Experimental Design	31	
6.2 Evaluation Protocol and Metrics	32	
6.3 Application Corpus and Task Construction	33	
6.4 Models and Execution Environment	34	
6.4.1 Model selection	34	
6.4.2 Execution environment	35	
6.5 Threats to Validity	36	
7 Proposed Agent Architecture	38	
7.1 Baseline Coding Agent Workflow	39	
7.1.1 Shared Architectural Simplifications	40	
7.2 Test-Generation-Based Internal Evaluation	40	
7.3 Agentic Internal Evaluation	42	
7.4 Implementation Details	43	
8 Experimental Results	45	
8.1 Experimental Overview	45	
8.2 Functional Correctness	46	
8.3 Regression Safety	47	
8.4 Efficiency	48	
8.5 Per-Application and Per-Difficulty Variation	51	
8.6 Error Analysis	52	
8.7 Internal Evaluator Quality	54	
8.8 Cross-Model Evaluation Experiment	55	
8.9 Discussion	57	
9 Conclusion	60	
9.1 Future Work	61	
Bibliography	63	
A System Prompts	69	
B Task Example	75	
Raw Task Definition	75	
Task-Specific Playwright Test (excerpt)	75	
Simulated User Output	76	
C Agentic Evaluation Trace	78	
Exploration and Implementation	78	
First Evaluation (failed)	79	
Coding Agent Fix	79	
Second Evaluation (passed)	80	

Figures

2.1 Multi-head attention. Queries, keys, and values are each linearly projected into multiple parallel subspaces, where scaled dot-product attention is computed independently. The per-head outputs are concatenated and passed through a final linear projection.	6
2.2 Encoder–decoder transformer architecture. The encoder (left) processes the input through stacked layers of multi-head self-attention and feed-forward networks with residual connections. The decoder (right) adds masked self-attention to preserve causal ordering and cross-attends to the encoder output. Adapted from Vaswani et al. [1]. . . .	8
3.1 Agent–environment interaction loop. The agent receives observations from the environment, processes them internally, and produces actions that modify the environment in a continuous feedback loop.	18
3.2 Observe–reason–act loop in an LLM-based agentic system.	20
7.1 Baseline coding-agent workflow. The task description and hidden tests are retrieved from S3, enriched by a simulated user, and passed to the coding agent. The agent operates on the repository using file-system and shell tools, then submits a pull request via the GitHub API. The external evaluation harness runs the hidden test suites against the updated repository post-hoc.	39
7.2 Test-generation-based evaluation architecture. In contrast to the baseline workflow, the coding agent is augmented with an internal evaluation component that generates and executes end-to-end tests and returns feedback during development, while the hidden external evaluation remains unchanged.	41
7.3 Agent-based internal evaluation architecture. In contrast to the baseline workflow, the coding agent is augmented with an internal evaluator agent that inspects the repository and interacts with the running application through Playwright MCP, while the hidden external evaluation remains unchanged.	42
8.1 Agent duration distribution by model and configuration (error-free runs only). Each panel corresponds to one agent configuration; box plots show the median, interquartile range, and outliers.	49
8.2 Task completion rate by application and model (error-free runs, all modes combined).	52
8.3 Error rate by agent configuration, broken down by error type.	53

Tables

6.1 On-demand inference pricing on Amazon Bedrock (per 1M tokens, USD). Prices are as listed at the time of the experiments.....	36
8.1 Task completion rate (Pass@1) by model and agent configuration, computed over error-free runs only. The two rightmost columns show the absolute change from the baseline to the self-evaluation and agentic configurations, respectively.	46
8.2 Average fraction of task subtests passed per error-free run, by model and configuration. Unlike Pass@1, this metric gives partial credit for runs that pass some but not all subtests.	47
8.3 Regression test pass rate (all base tests passing) by model and configuration, computed over error-free runs only.	48
8.4 Average wall-clock duration per run (seconds), by model and configuration, excluding runs that ended in error.	49
8.5 Average total token consumption per run (input + output, in thousands), by model and configuration. Agentic mode includes tokens consumed by both the coding agent and the evaluator agent.....	50
8.6 Estimated cost per successfully completed task (USD), by model and configuration. Computed as total API cost across all error-free runs divided by the number of successful runs.	50
8.7 Pass@1 by task difficulty and configuration (error-free runs, three stronger models only). Δ columns show the improvement over the baseline in percentage points.	51
8.8 Error breakdown across all completed runs.	52
8.9 Error rate by task difficulty and configuration.	54
8.10 Evaluator accuracy by mode and model, computed over error-free runs with at least one evaluation call. SE = self-evaluation, Ag = agentic, P = precision, R = recall, FP = false-positive rate. Metrics are defined with respect to the ground-truth hidden task tests. ...	54
8.11 Cross-model experiment configurations. The diagonal entries (A, D) correspond to the standard same-model setup already evaluated. The off-diagonal entries (B, C) are the new cross-model pairings.	56
8.12 Cross-model experiment results (error-free runs, agentic mode only). Configs A0 and D0 are the no-evaluation baselines; A and D use the same model for both roles; B and C are the cross-model pairings. ...	57



Chapter 1

Introduction

Recent advances in artificial intelligence (AI), driven primarily by the development of large language models (LLMs) based on the transformer architecture [1], have significantly impacted many areas of engineering and knowledge work. One of the domains most strongly affected by these developments is software engineering, where LLMs are increasingly used to assist or automate tasks such as code generation [2], debugging, documentation, and software maintenance. This strong impact can be partially attributed to the textual nature of programming languages, which makes code generation closely related to natural language modeling, as well as to the tendency of software engineers to be early adopters of new technological tools.

In addition to standalone code generation, recent research has increasingly focused on LLM-based agents capable of performing complex, multi-step tasks through iterative reasoning, tool usage, and interaction with their environment. These agentic systems extend large language models with planning and feedback mechanisms, enabling them to decompose high-level objectives into sequences of actions and adapt their behavior based on intermediate observations and outcomes [3].

Within the domain of software engineering, this paradigm has led to the emergence of LLM-based coding agents, which employ similar iterative workflows to generate, modify, and evaluate code across multi-file software systems [4]. By repeatedly producing intermediate outputs and reacting to execution feedback, these agents move beyond single-shot code generation towards autonomous software development processes.

Software engineering is inherently sensitive to errors, and even small inaccuracies in generated code can lead to significant failures in larger software systems. Due to the stochastic nature of LLMs and their tendency to produce plausible but incorrect outputs, whether through logical errors, misunderstanding of task requirements, or fabrication of nonexistent APIs and libraries, ensuring the functional correctness of automatically generated code becomes a critical concern. As coding agents move towards more autonomous operation and reduced human oversight, the need for systematic and automated evalua-

tion mechanisms becomes increasingly important. In traditional software engineering, automated testing, and particularly end-to-end testing, plays a central role in verifying system functionality and preventing regressions [5]. Extending this paradigm to LLM-based coding agents suggests that integrating automated test generation and execution directly into the agent workflow can provide a continuous feedback signal, improving both the functional correctness and overall utility of the generated code.

This thesis is additionally motivated by the Amazon Nova AI Challenge 2026, in which a six-member team from the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC) represents the Czech Technical University in Prague (CTU). The goal of the challenge is the development of a coding agent for secure and reliable repository-level code generation. The competition runs from February to October 2026 and focuses on full-stack web applications, typically involving a Python backend and a JavaScript frontend, under the constraint of using Amazon’s proprietary Nova models. The broader team effort includes multiple research and engineering activities, such as agent architecture design, synthetic data generation for fine-tuning, and security evaluation of generated code. Within this collaboration, the internal utility evaluation component of the agent is entirely my responsibility, and its design, implementation, and evaluation constitute the central focus of this thesis. All work presented in this thesis was carried out independently unless explicitly stated otherwise; whenever other team members’ work is used, this is clearly indicated.

1.1 Problem Statement

This thesis focuses on the evaluation of the utility and functional correctness of code generated by LLM-based coding agents. In particular, it investigates whether the integration of an automated internal evaluation component, based on end-to-end testing and interactive browser-based verification, can meaningfully improve the quality of the produced code.

Unlike traditional code evaluation scenarios, LLM-based coding agents often operate without reference implementations, making it difficult to directly assess correctness [6]. Therefore, there is a need for an automated and systematic evaluation pipeline that can validate whether the generated code aligns with user-specified requirements and behaves correctly in realistic execution environments.

The central research objective of this thesis is to investigate whether the integration of an internal automated evaluation component can improve the functional correctness and practical utility of code generated by LLM-based coding agents. More specifically, the thesis studies whether evaluation mechanisms based on automated testing can be embedded directly into the agent workflow in a way that provides useful feedback during code generation,

rather than serving only as an external post-hoc assessment tool. This objective is reflected in three research questions:

- **RQ1:** To what extent does the integration of automated evaluation improve the functional correctness of generated code?
- **RQ2:** How does the effectiveness of the evaluation-enhanced workflow vary across different open coding models and agent configurations?
- **RQ3:** What is the trade-off between improved code quality and the additional computational cost introduced by the evaluation component?

1.2 Outline

The initial chapters provide the necessary theoretical and conceptual background for the thesis. Chapter 2 introduces the transformer architecture, large language models, and their capabilities in code generation. Chapter 3 focuses on agentic systems, starting from traditional AI agents and progressing to modern LLM-based agentic architectures, with a particular emphasis on coding agents designed to automate software engineering tasks. Chapter 4 discusses software testing and code evaluation, highlighting the role of automation in software quality assurance and the potential of end-to-end testing and test generation for validating AI-generated code. Chapter 5 reviews related work in the areas of LLM-based coding agents, automated code evaluation, and AI-assisted software testing.

The practical part of the thesis begins in Chapter 6, which presents the methodology adopted in this thesis, including the experimental design, the proposed evaluation framework, the application corpus, and the model selection. Chapter 7 describes the architecture of the proposed evaluation-enhanced coding agent, defined as a coding agent augmented with an internal automated testing and evaluation module. Chapter 8 presents the experimental results obtained by evaluating the proposed architectures across five language models and twenty generated web applications, and discusses the findings with respect to the research questions. Finally, Chapter 9 concludes the thesis and outlines directions for future work.

Chapter 2

Large Language Models and Code Generation

The emergence of large language models capable of generating natural language and source code is closely tied to the introduction of the transformer architecture. By replacing recurrent sequence modeling with self-attention, transformers removed fundamental limitations related to sequential computation and long-range dependency modeling. Their architectural properties enabled large-scale pretraining on extensive text corpora, ultimately giving rise to modern large language models. These models extend beyond natural language tasks and are increasingly applied to structured generation problems such as code synthesis. A rigorous understanding of the transformer architecture and large-scale language modeling principles is therefore necessary before analyzing LLM-based coding agents and their evaluation.

2.1 Transformers

The transformer architecture, introduced by Vaswani et al. in 2017 [1], marked a fundamental shift in sequence modeling. By replacing recurrent and convolutional mechanisms with self-attention, the architecture removed the need for sequential processing of tokens and enabled more efficient modeling of long-range dependencies. This change not only addressed key limitations of earlier neural language models, but also established the foundation for scalable large-scale language modeling.

2.1.1 Limitations of Recurrent Architectures

Prior to the introduction of the transformer architecture, recurrent neural networks (RNNs), including their gated variants such as Long Short-Term Memory (LSTM) networks, represented the dominant approach to language modeling [1]. In this setting, the goal is to learn a probability distribution

over sequences of tokens, modeling each element in the sequence conditioned on the preceding context.

RNNs compute a hidden state z_t at each time step as a function of the previous hidden state z_{t-1} and the current input x_t , typically expressed as

$$z_t = f_W(z_{t-1}, x_t),$$

where f_W denotes a parameterized transformation defined by the model weights. This recursive formulation enables the model to propagate information through the sequence, making RNNs naturally suited for sequential data such as text. However, the sequential dependency of hidden states imposes two fundamental limitations.

First, the representation of long sequences requires information to be compressed into progressively updated hidden states, making it difficult to preserve long-range dependencies. In practice, information from earlier parts of the sequence may gradually diminish or be overwritten as the sequence length increases [7]. Second, the strictly sequential computation of hidden states limits opportunities for parallelization, resulting in slower training compared to architectures that allow simultaneous processing of tokens [8].

To address some of these limitations, encoder–decoder architectures based on RNNs were introduced, most notably in the sequence-to-sequence framework [9]. In this paradigm, an encoder processes the entire input sequence into a fixed-dimensional representation, which is then used by a decoder to generate the output sequence. While this approach achieved significant success in tasks such as machine translation, it still relied on a compressed final encoder state, which could act as an information bottleneck for long inputs.

LSTMs and related gated architectures were specifically designed to mitigate the vanishing gradient problem and improve the modeling of long-range dependencies through the introduction of memory cells. Although these mechanisms improved performance over standard RNNs, they did not remove the inherent sequential nature of recurrent computation. As a result, both the difficulty of modeling very long contexts and the limitations in parallelization remained fundamental constraints of recurrent architectures.

2.1.2 Self-Attention Mechanism

Encoder-Decoder Attention vs. Self-Attention

Attention mechanisms were originally introduced to alleviate the information bottleneck present in recurrent encoder–decoder models. In encoder–decoder attention, the encoder produces a sequence of hidden states, and at each decoding step the decoder computes a weighted combination of these states. Rather than relying solely on the final encoder representation, the decoder

dynamically assigns higher weights to more relevant encoder states, allowing it to model non-local dependencies [7].

Formally, attention is defined as a mapping between queries Q , keys K , and values V , typically expressed using scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \quad (2.1)$$

In encoder–decoder attention, the query Q is derived from the previous decoder layer, whereas the keys K and values V originate from the encoder outputs. In contrast, self-attention computes all three components from the same input sequence. Given an input representation X , the projections are defined as $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, where W_Q, W_K, W_V are learned weight matrices [1]. Consequently, each token in the sequence can directly attend to all other tokens within the same sequence.

Unlike recurrent computation, self-attention does not require sequential propagation of hidden states. All pairwise token interactions can be computed in parallel, improving training efficiency while enabling the model to capture both local and long-range dependencies within a unified framework [1].

Multi-Head Attention

While single-head self-attention enables global interaction between tokens, the transformer architecture further extends this mechanism through multi-head attention [1]. Instead of computing a single attention mapping over the input representations, the model projects the queries, keys, and values into multiple lower-dimensional subspaces and performs independent attention operations in parallel. Each attention head therefore captures relationships within a different representational subspace of the sequence.

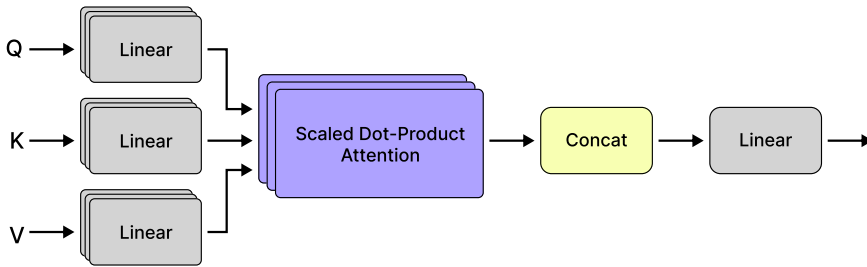


Figure 2.1: Multi-head attention. Queries, keys, and values are each linearly projected into multiple parallel subspaces, where scaled dot-product attention is computed independently. The per-head outputs are concatenated and passed through a final linear projection.

Formally, for h attention heads, the attention for head i is defined as

$$\text{head}_i = \text{Attention}(QW_Q^{(i)}, KW_K^{(i)}, VW_V^{(i)}), \quad (2.2)$$

where $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)}$ are learned projection matrices specific to the i -th head. The outputs of all heads are then concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \quad (2.3)$$

where W^O is an output projection matrix. Figure 2.1 illustrates the parallel computation of multiple attention heads and their subsequent concatenation and linear projection.

Empirically, different attention heads often specialize in capturing distinct structural aspects of the sequence. For example, some heads may focus on syntactic relationships such as subject–verb agreement, while others emphasize semantic similarity or long-range contextual dependencies. This separation of representational subspaces increases the expressive capacity of the model without introducing additional sequential computation.

For autoregressive language modeling, a masked variant of multi-head attention is employed. In this setting, attention weights corresponding to future tokens are suppressed during training, ensuring that predictions at each position depend only on previously generated tokens. This masking mechanism preserves the causal structure required for next-token prediction while retaining the advantages of parallel attention computation [7].

2.1.3 Transformer Architecture

The original transformer architecture follows an encoder–decoder structure composed of stacked layers of self-attention and position-wise feed-forward networks [1]. The model was originally proposed for sequence-to-sequence tasks such as machine translation, where an encoder processes the source-language sequence and a decoder generates the corresponding target-language sequence. As illustrated in Figure 2.2, both the encoder and decoder operate on token embeddings augmented with positional encodings, enabling the model to incorporate information about sequence order.

The encoder consists of n identical layers, each comprising two sublayers: a multi-head self-attention mechanism followed by a position-wise fully connected feed-forward network. Each sublayer is surrounded by a residual connection and followed by layer normalization. The self-attention component allows tokens to exchange information globally within the sequence, while the feed-forward network applies a non-linear transformation independently at each position.

The decoder is similarly composed of n stacked layers but contains three sublayers per block. The first sublayer applies masked multi-head self-attention, ensuring that each position attends only to previous tokens in autoregressive settings. The second sublayer performs encoder–decoder attention, allowing the decoder to attend to the encoder outputs. The final sublayer is a position-wise feed-forward network. As in the encoder, each sublayer is wrapped with residual connections and layer normalization.

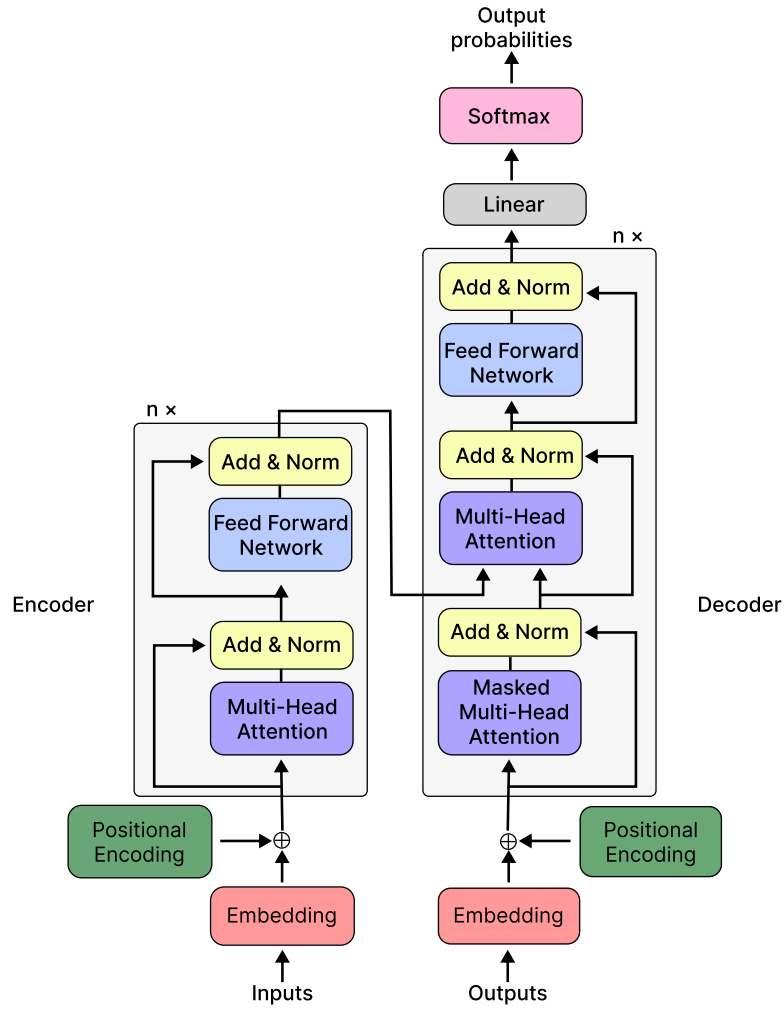


Figure 2.2: Encoder-decoder transformer architecture. The encoder (left) processes the input through stacked layers of multi-head self-attention and feed-forward networks with residual connections. The decoder (right) adds masked self-attention to preserve causal ordering and cross-attends to the encoder output. Adapted from Vaswani et al. [1].

At the top of the decoder stack, a linear transformation followed by a softmax layer produces a probability distribution over the vocabulary for next-token prediction. This modular design, combining attention mechanisms with feed-forward transformations and residual connections, enables deep stacking of layers while maintaining stable gradient propagation during training.

2.1.4 Positional Encoding

Unlike recurrent or convolutional architectures, the transformer does not inherently encode information about the order of tokens in a sequence. Because the self-attention mechanism operates on all tokens simultaneously and

treats them as a set rather than a sequence, positional information must be explicitly incorporated into the model [1]. This is achieved through positional encodings that are added to the input token embeddings at the bottom of the encoder and decoder stacks.

The positional encodings have the same dimensionality as the token embeddings, allowing the two representations to be combined through element-wise addition. This enables the model to utilize both semantic information from the embeddings and positional information from the encodings when computing attention.

Different strategies can be used to represent positional information. In the original transformer architecture, positional encodings were defined using deterministic sinusoidal functions of different frequencies, allowing the model to represent relative distances between tokens [1]. Alternatively, positional representations may be learned during training, in which case the model directly learns embedding vectors corresponding to each position in the sequence. More recent approaches introduce relative positional encodings, where the model represents the distance between tokens rather than their absolute position within the sequence [7].

2.1.5 Transformer Variants and Decoder-Only Models

The original transformer architecture follows an encoder–decoder design intended for sequence-to-sequence tasks such as machine translation. However, the encoder and decoder components can also be used independently, giving rise to several architectural variants that have proven useful for different types of natural language processing tasks.

Encoder–decoder transformers remain widely used for sequence transformation tasks where an input sequence must be mapped to a different output sequence. Notable examples include models such as *T5* and *BART*, which achieve strong performance on tasks such as machine translation, summarization, and text generation.

Encoder-only transformers are designed primarily for language understanding tasks. In these models, the encoder stack processes the entire input sequence simultaneously using bidirectional self-attention. A prominent example is the *BERT* architecture, which is trained using objectives such as masked language modeling and is widely applied to tasks including text classification, named entity recognition, and question answering [7].

Decoder-only transformers, in contrast, are optimized for autoregressive language modeling. These models generate text sequentially by predicting the next token conditioned on all previously generated tokens. This architecture forms the basis of modern large language models, including the GPT family, which demonstrated that scaling transformer-based language models to very

large datasets and parameter counts can lead to significant improvements in performance across a wide range of tasks [8].

Because of their autoregressive formulation and scalable training objective, decoder-only transformers have become the dominant architecture for contemporary large language models. The training and capabilities of such models are discussed in the following section.

2.2 Large Language Models

Large language models (LLMs) are neural language models trained on massive text corpora to capture the statistical structure of natural language. Built upon the transformer architecture described in the previous section, these models are trained using self-supervised objectives that enable learning directly from raw text data. The combination of large model capacity, extensive training datasets, and substantial computational resources has enabled LLMs to achieve strong performance across a wide range of language tasks.

2.2.1 Autoregressive Language Modeling

Large language models are typically trained as autoregressive language models [10], where the task is to predict the next token in a sequence based on the preceding tokens. Before training, raw text is converted into sequences of tokens using tokenization algorithms, typically based on subword units such as byte-pair encoding [7]. Given a token sequence $(x_1, \dots, x_n)$, the model is trained to maximize the joint probability of the sequence, factorized using the chain rule as

$$P(x_1, \dots, x_n) = \prod_{t=1}^n P(x_t \mid x_1, \dots, x_{t-1}). \quad (2.4)$$

This formulation provides a self-supervised training objective: the target at each step is simply the next token in the training sequence, so no manual annotation is required [7]. At inference time, the same factorization supports text generation through iterative next-token prediction.

2.2.2 Large-Scale Pretraining

The first phase of training a large language model is referred to as *pretraining*. During this stage, the model is trained on large corpora of text using the self-supervised next-token prediction objective described in Subsection 2.2.1 [7]. Because no manual annotation is required, pretraining can leverage virtually unlimited amounts of raw text.

What distinguishes modern large language models from earlier neural language models is primarily the scale at which this pretraining is performed. Contemporary models are characterized by three key forms of scale: model size, training data, and computational resources [11].

First, the parameter count of modern models often reaches tens or hundreds of billions of parameters, enabling them to learn complex statistical patterns from large datasets. Second, pretraining datasets typically contain hundreds of billions to trillions of tokens collected from diverse sources such as web pages, books, research articles, and software repositories [8]. Finally, training such models requires enormous computational resources, commonly measured in floating-point operations (FLOPs) or GPU-hours. For example, the DeepSeek-V3, a relatively cost-efficient state-of-the-art model, required approximately 2.8 million GPU hours on NVIDIA H800 GPUs [12].

The combination of large models, massive datasets, and extensive computational resources enables large language models to learn rich representations of language that generalize across a wide range of tasks.

2.2.3 Context Length in LLMs

Another important capability of modern large language models is their ability to process long input sequences, commonly referred to as the *context length*. The context length determines how many tokens can be provided to the model at once, and therefore how much information can be considered when generating predictions.

Increasing the context length, however, presents significant computational challenges. In the transformer architecture, the self-attention mechanism computes interactions between all pairs of tokens in the sequence, which means that both the time and memory complexity grow quadratically with respect to the sequence length, i.e. $O(n^2)$. As a result, very long sequences quickly become computationally expensive to process [8].

Various techniques have been proposed to mitigate this limitation. For example, FlashAttention [13] reduces GPU memory usage by computing attention iteratively in smaller blocks rather than constructing the full attention matrix at once. These types of optimizations enable modern large language models to support substantially longer context windows than earlier transformer models.

2.2.4 Scaling Laws and In-Context Learning

The performance of large language models has been empirically shown to improve predictably as model size, dataset size, and training compute increase. In particular, the training loss of neural language models follows an approximate power-law relationship with respect to these scaling factors [11].

These observations, commonly referred to as *scaling laws*, played a key role in motivating the development of increasingly large transformer-based language models.

More recent research has suggested that improvements in model capability can also be achieved by allocating additional computational resources at inference time, commonly referred to as *test-time compute*. This shift is partly motivated by observations that scaling model parameters alone does not always yield proportional improvements in performance. Instead, methods that increase the amount of computation performed during generation, for example by exploring multiple candidate outputs or performing intermediate reasoning steps, can in some cases provide greater improvements than simply increasing model size [14].

As models scale in size and capability, they begin to exhibit the phenomenon known as *in-context learning*. In-context learning refers to the ability of a language model to perform tasks based on examples provided within the input prompt, without updating the model parameters [7]. A prominent example of this behavior is *few-shot learning*, where several task examples are included in the prompt to guide the model’s predictions [10]. Models can also operate in one-shot or zero-shot settings, where one or no examples are provided respectively, although performance generally improves when more examples are included.

■ 2.2.5 Instruction Tuning and Alignment

Although large language models acquire broad linguistic knowledge during pretraining, the objective of next-token prediction does not inherently ensure that the model produces responses that are helpful or appropriate for end users. For instance, a pretrained model may generate factually incorrect information or produce toxic or otherwise undesirable content [15]. In practical applications such as conversational assistants, it is therefore desirable for models to exhibit three key properties: *helpfulness*, meaning the ability to follow user instructions and assist with tasks; *honesty*, meaning the model should avoid fabricating information; and *harmlessness*, meaning the model should avoid generating content that could cause harm to users or society. Achieving these properties requires aligning the model’s behavior with human intentions, a process commonly referred to as *alignment*.

Alignment is typically achieved through a post-pretraining adaptation process consisting of two main stages: *instruction tuning* and *preference tuning* [16]. Instruction tuning aims to improve the model’s ability to follow natural-language instructions. This is commonly performed through *supervised fine-tuning* (SFT), in which the pretrained model is further trained on datasets of instruction–response pairs demonstrating the desired behavior. In contrast to pretraining, which relies on large amounts of unlabeled text, supervised fine-tuning uses labeled data where the instruction serves as the input and

the corresponding human-written response represents the target output. The model parameters are then updated to maximize the likelihood of producing the desired responses. Several approaches to SFT exist, ranging from full fine-tuning of all model parameters to more parameter-efficient methods such as Low-Rank Adaptation (LoRA) and Quantized Low-Rank Adaptation (QLoRA), which update only a small subset of the model parameters while keeping the majority of the pretrained weights fixed [16].

While supervised fine-tuning improves the model’s ability to follow instructions, it does not fully capture nuanced human preferences regarding response quality, safety, or style. To address this limitation, a second stage known as *preference tuning* or *preference alignment* is often applied [17]. A widely used approach is *reinforcement learning from human feedback* (RLHF), in which human annotators compare multiple model outputs for the same prompt and indicate which response they prefer. These comparisons are used to train a *reward model* that predicts the quality of model outputs. The language model is then optimized using reinforcement learning algorithms, most commonly *Proximal Policy Optimization* (PPO), to produce outputs that maximize the reward predicted by the reward model [15].

More recently, alternative approaches such as *Direct Preference Optimization* (DPO) have been proposed. Unlike RLHF, DPO avoids training an explicit reward model and instead directly optimizes the language model using preference comparisons between alternative responses. In this formulation, the training objective encourages the model to assign higher probability to preferred outputs while decreasing the probability of less desirable responses [17].

These alignment techniques play an important role in shaping the behavior of modern large language models deployed in real-world systems. By enabling models to follow natural-language instructions and respond in ways that better reflect human preferences, instruction and preference tuning transform pretrained language models into practical tools for a wide range of applications, including software code generation discussed in the following section.

2.3 Code Generation with LLMs

Code generation is among the most prominent applications of large language models. Since a substantial portion of typical pretraining corpora consists of source code, LLMs acquire knowledge of the syntax and semantics of virtually every widely used programming language, enabling them to generalize programming concepts and algorithmic patterns in a largely language-agnostic manner [8, 18]. Models trained on code-heavy or code-specific corpora develop these capabilities to a considerably greater extent than general-purpose language models trained predominantly on natural-language text [2]. Code generation with LLMs can be understood as a modern form of program synthe-

sis, in which the model generates a program conditioned on a natural-language specification. The effectiveness of this approach depends on the composition of the pretraining data, the capacity of the model, and the methods used to evaluate whether the generated code is functionally correct.

2.3.1 Large Language Models for Program Synthesis

Program synthesis has been a focus of artificial intelligence research since the early development of the field [19]. In classical formulations, the task refers to generating a program that satisfies a given high-level specification, typically expressed in a formal constraint system [20]. Traditional approaches rely on techniques such as deductive reasoning, constraint solving, or enumerative search over domain-specific languages, which often scale poorly to general-purpose programming tasks. Large language models offer a fundamentally different paradigm. Since programming languages are formal languages with well-defined syntax and semantics, code generation can be framed as a sequence generation task over programming languages, subject to syntactic and semantic constraints. Rather than requiring these constraints to be explicitly encoded, LLMs learn the underlying structure of programming languages directly from large corpora of code during pretraining [19].

In modern code generation systems, the specification of the desired program is typically expressed in natural language rather than a formal constraint system. Large language models trained on mixed corpora of source code and natural language text learn to associate textual descriptions with corresponding code implementations. Function docstrings, inline comments, and documentation frequently describe the intended behavior of a program in natural language, while the surrounding code provides the concrete implementation. During training, the model captures statistical relationships between these representations, enabling it to generate code conditioned on natural-language prompts or partially written programs [2]. This natural-language conditioning is what distinguishes LLM-based program synthesis from classical approaches, in which specifications must be expressed in a formal language.

This formulation enables large language models to generate syntactically plausible programs across a range of tasks, from function-level synthesis and code completion to bug fixing and code translation. While much of the existing literature evaluates code generation at the level of isolated functions, practical software development requires generating code that spans multiple files, interacts with external dependencies, and integrates coherently into an existing codebase. A fundamental challenge persists across all of these settings: syntactic correctness does not guarantee functional correctness. A single incorrect token can alter program behavior entirely, and a generated program may appear plausible while failing to implement the intended specification. This discrepancy is amplified in repository-level code generation, where correctness depends not only on individual components but on their interaction

within a larger system. Evaluating such systems therefore requires execution-based methods that go beyond surface-level comparison to reference solutions [2], a challenge that becomes central when LLM-based agents operate with increasing autonomy and reduced human oversight.

■ 2.3.2 Pretraining on Code Corpora

The effectiveness of large language models on code generation tasks depends critically on the composition of their pretraining data. General-purpose language models such as GPT-3 [10], which are trained predominantly on web text, exhibit only limited code generation ability. By contrast, Codex [2] demonstrates the impact of code-specific training: starting from a GPT-3 base model, it is fine-tuned on a large corpus of publicly available source code, including 159 GB of Python code collected from GitHub repositories, resulting in dramatically improved performance on programming tasks despite no change in model architecture or parameter count. This suggests that exposure to large volumes of high-quality source code is a primary factor in enabling fluent code generation.

Curating such corpora presents several practical challenges. Public repositories vary widely in code quality, licensing status, and the degree to which files contain meaningful, self-contained logic rather than boilerplate or auto-generated content. Training pipelines therefore typically apply filtering heuristics based on criteria such as file size, average line length, proportion of alphanumeric characters, repository popularity, and license compatibility [18]. Deduplication is equally important, since near-duplicate files can inflate the effective dataset size without contributing new information and may encourage memorization of specific code snippets rather than generalization of programming patterns. Some models also adopt multi-stage pretraining strategies that progressively specialize the model's distribution. CodeGen [20], for instance, is first trained on natural language data, then on a multilingual code corpus drawn from Google's BigQuery dataset, and finally on a Python-specific subset.

The choice of tokenizer and the language composition of the corpus also have a significant effect on code modeling. Byte-pair encoding (BPE) [21], the dominant subword tokenization method used in large language models, is often inefficient when applied to code, since constructs such as camelCase or snake_case identifiers, multi-character operators, and indentation patterns may be split into multiple subword units, increasing sequence length. Since transformer-based models operate within a fixed context window, this directly reduces the amount of code the model can condition on during training and inference. Code-specialized tokenizers mitigate this by training on code-heavy corpora, producing more compact token representations. A related design choice concerns the programming languages included in the training corpus. Some models, such as Codex [2], are fine-tuned primarily on Python, while others such as CodeGen-Multi [20] train on corpora spanning dozens

of languages. Multilingual code training can enable cross-language transfer, where patterns learned in one language improve generation quality in another [22], but it also introduces a trade-off: model capacity must be distributed across languages, which may reduce performance on any single language relative to a comparably sized monolingual model.

These pretraining choices have given rise to a broad ecosystem of code-specialized language models, including CodeGen [20], StarCoder [23], Code Llama [24], and more recent models such as Qwen-Coder [25] and later Codex variants. At the same time, recent benchmark results suggest that very large general-purpose language models can be highly competitive with, and in some cases outperform, smaller code-specialized models on code-generation tasks, indicating that model scale can sometimes outweigh specialization. More recent systems increasingly target agentic software engineering workflows, where code generation is combined with tool use, repository-scale reasoning, and iterative interaction with the execution environment; these systems are discussed in greater detail in Chapter 3.

2.3.3 Benchmarks and Evaluation of Generated Code

Evaluation of code generation systems differs fundamentally from evaluation in natural language generation. Because multiple programs may implement the same functionality while differing syntactically, surface-level similarity metrics such as BLEU are generally unsuitable for measuring correctness. Instead, code generation models are commonly evaluated using execution-based benchmarks, where generated programs are run against unit tests designed to verify functional behavior. One of the most widely used benchmarks is HumanEval [2], which consists of Python programming problems paired with hidden test cases. Performance on this benchmark is typically reported using the $\text{pass}@k$ metric, which estimates the probability that at least one of k generated samples passes all tests [2]. An unbiased estimator of this quantity is given by

$$\text{pass}@k = \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \quad (2.5)$$

where n denotes the total number of generated samples for a given problem and c is the number of correct samples that pass all tests. Another frequently used benchmark is Mostly Basic Programming Problems (MBPP) [19], which evaluates models on short Python tasks with accompanying test cases.

While benchmarks such as HumanEval and MBPP have played an important role in measuring progress, they also have important limitations. Most problems in these datasets involve relatively short, self-contained functions and therefore do not capture the challenges of working with larger codebases, complex dependencies, or multi-file project structures. In addition, publicly available static benchmarks become increasingly vulnerable to dataset contamination, where benchmark tasks or highly similar solutions may appear

in training data, as well as to benchmark-targeted optimization that inflates performance without necessarily improving general coding ability. As a result, strong benchmark performance does not always translate to success in realistic software engineering settings. More recent evaluation frameworks attempt to address this gap. For example, SWE-bench [6] evaluates models on real bug-fixing tasks drawn from open-source repositories, while benchmarks such as LiveCodeBench [26] aim to reduce contamination and provide a more realistic picture of current code-generation capabilities. These developments highlight the growing importance of evaluation methodologies that measure not only syntactic plausibility, but also functional correctness in realistic development environments.

A further challenge arises from the fact that modern code generation systems are increasingly used in iterative and interactive workflows rather than as single-shot generators of isolated functions. In such settings, model performance depends not only on the correctness of the final generated code, but also on the ability to interpret feedback, revise intermediate outputs, and operate effectively within a broader software environment. Evaluating these behaviors requires moving beyond static benchmark tasks toward settings that capture sequential decision making, tool use, and interaction with external context. These characteristics become particularly important in LLM-based coding agents, which are discussed in the following chapter.

Chapter 3

Agentic Systems

Agentic systems represent a shift from single-shot language model generation toward iterative interaction with an external environment. Rather than producing a single response in isolation, such systems observe their surroundings, reason about the next step, take actions, and incorporate feedback over multiple interaction cycles. This chapter first introduces the general notion of an agent in artificial intelligence, then discusses how this paradigm is instantiated in modern LLM-based agentic systems, and finally narrows the focus to coding agents, which are particularly relevant to the thesis.

3.1 The Agent Paradigm

In artificial intelligence, an *agent* is generally defined as a system that observes its environment, selects actions based on those observations, and acts upon the environment in a continuous loop [27]. This interaction pattern is illustrated in Figure 3.1. In the context of software agents, observations may consist of file contents, source code, user instructions, or tool outputs, while actions may include modifying files, producing text, or invoking external services. The environment may range from a local file system to a complete software system with network-accessible components.

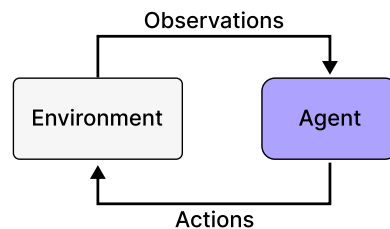


Figure 3.1: Agent–environment interaction loop. The agent receives observations from the environment, processes them internally, and produces actions that modify the environment in a continuous feedback loop.

An agent is called *rational* if it selects actions expected to maximize its performance measure given the observations available to it [27]. Several basic types of agents can be distinguished based on how they select actions [27]:

- *Simple reflex agents* select actions solely on the basis of the current observation and are therefore most suitable for fully observable environments.
- *Model-based agents* maintain an internal state that summarizes relevant aspects of past observations, which is particularly important in partially observable environments.
- *Goal-based agents* select actions with respect to a set of desired goal states. They rely on a model of the environment to evaluate which actions are likely to lead to states that satisfy their goals.
- *Utility-based agents* evaluate states using a utility function that provides a more general notion of preference than the binary distinction between goal and non-goal states. This allows the agent to compare alternative outcomes even when several actions may achieve the same goal with different levels of desirability.

Traditional agent architectures typically relied on explicitly designed rules, symbolic world models, or hand-crafted decision procedures [27]. Modern LLM-based agentic systems preserve the same observe–reason–act interaction pattern, but replace much of the explicit reasoning machinery with language-model-based inference, prompting, and tool use [3].

■ 3.2 LLM-based Agentic Systems

While plain large language models can generate coherent responses and even produce intermediate reasoning steps within a single generation, they are not agents in the sense defined above. A standalone LLM generates output in a single forward pass and, by itself, does not maintain an explicit observation–action loop with an external environment. As a result, its reasoning remains internal to the model and is not grounded in external observations or feedback, which makes it difficult to reliably correct intermediate errors or incorporate up-to-date information [3, 28]. LLM-based agentic systems address this limitation by embedding the language model into an environment interaction loop that supplements generation with grounded observations, external actions, and tool use, thereby enabling the model to function as an agent operating in an environment [3].

A central idea in such systems is the observe–reason–act loop, illustrated in Figure 3.2. The agent first receives an observation, which may consist of a user instruction, the current state of the environment, or the output of a previously invoked tool. Based on this information, the language model produces an intermediate reasoning trace that helps determine the next action to take.

That action may involve generating a response directly, invoking an external tool, retrieving information, or modifying the environment, after which a new observation is obtained and the loop continues [3]. This interleaving of reasoning and action is a defining property of modern LLM agents, as it allows intermediate decisions to be grounded in observations obtained from the environment rather than relying solely on internal token prediction [3].

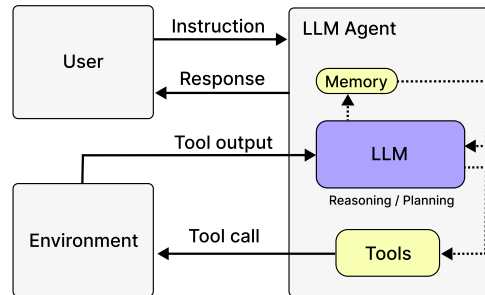


Figure 3.2: Observe–reason–act loop in an LLM-based agentic system.

Although the exact decomposition varies across frameworks, most LLM-based agentic systems contain several recurring components. They receive observations from the user or the environment, perform reasoning or planning to determine the next step, maintain some form of memory, either implicitly through the context window or explicitly through external storage, to preserve relevant information across multiple interaction steps, and act on the environment by generating responses or invoking external tools [29, 3]. In this sense, LLM-based agents are typically not single model calls, but scaffolds built around a language model that enable iterative interaction with an external environment.

LLM-based agentic systems have shown clear improvements over plain language-model prompting on tasks that require sequential reasoning, retrieval of external information, or interaction with tools [3, 28]. By grounding intermediate decisions in observations obtained from the environment, such systems can reduce some forms of hallucination and revise their behavior based on external feedback [3]. At the same time, they are prone to brittle planning, imperfect tool use, and sensitivity to the quality of the feedback they receive. These issues become especially important in software engineering settings, where the agent must reason over repositories, edit code, and react to execution results.

3.3 Coding Agents

When applied to software engineering, LLM-based agentic systems take the form of *coding agents*. Software engineering tasks typically require decomposing high-level requirements into concrete implementation steps, interacting

with development environments, and validating generated code against execution feedback — capabilities that a plain language model does not provide in a single generation [30]. Coding agents address this by using a language model as the central reasoning component within a loop that allows them to operate on repositories, execute commands, run tests, diagnose failures, and apply code modifications over multiple steps [30].

Like other LLM-based agentic systems, coding agents follow an observe–reason–act loop [3]. They receive a task specification, inspect relevant parts of the environment, reason about the next step, and then act through external tools. Although concrete architectures vary considerably, the following components appear in many coding agent designs in varying combinations [30]:

- *Planning*, where the agent decomposes the user request into a sequence of sub-tasks or intermediate objectives.
- *Tool use*, which provides the agent with interfaces for interacting with its environment. In coding agents, these tools typically support repository navigation, file viewing and editing, localization of relevant code, execution of commands, and context management [4].
- *Reflection*, where the agent evaluates the effects of its previous actions, either through internal reasoning, execution feedback, test results, or static analysis, and uses this information to revise subsequent behavior.

Some systems omit explicit planning and rely primarily on the model’s internal reasoning to navigate the task, while others place greater emphasis on structured workflows and external feedback. CodeAct [31], for example, emphasizes executable actions and an environment in which the agent can run generated code as part of the problem-solving process.

Beyond single-agent systems, recent work has also explored multi-agent architectures in which multiple agents collaborate to achieve a shared goal, either by assuming specialized roles such as programmer, reviewer, or tester, or by operating as a set of homogeneous cooperating agents [30]. Beyond research prototypes, mainstream coding agents and agentic coding tools such as GitHub Copilot, Cursor, and Claude Code have also become widely used in practice, reflecting the growing shift of coding agents from experimental systems to real development workflows.

Because coding agents operate through multi-step interaction with their environment rather than producing a single output, they are substantially more difficult to evaluate than conventional code generation systems. While a single generated function can often be assessed in isolation by executing it against a fixed set of tests, coding agents must succeed through a process involving task interpretation, repository exploration, tool invocation, code modification, and iterative refinement based on execution feedback [30]. Recent surveys therefore emphasize that evaluating coding agents requires going beyond syntax or final pass rates and instead accounting for problem-solving ability in complex and dynamic software engineering environments [30]. As a

result, meaningful evaluation must consider not only the correctness of the generated code in isolation, but also whether it integrates correctly within the larger system without introducing regressions, as well as the computational efficiency of the agent's execution. This challenge is central to the thesis and motivates the focus on evaluation-enhanced coding workflows in the following chapters.

Chapter 4

Software Testing and Code Evaluation

Evaluating code produced by LLM-based coding agents requires automated mechanisms that can assess functional correctness at the level of complete application workflows, not just individual functions. This chapter reviews the testing foundations on which such evaluation can be built: the role of automation in software testing, end-to-end testing as a means of verifying integrated system behavior, and the emerging use of large language models for automated test generation.

4.1 Automated Software Testing

Software testing is a systematic process for executing software in order to reveal faults and assess whether its behavior conforms to expectations [5, 32]. Automation plays a central role in this process because software systems are modified repeatedly over time, and previously correct behavior may be unintentionally broken by later changes. Automated test execution addresses this by allowing test suites to be rerun systematically after each modification, providing a practical mechanism for detecting regressions without manual effort [5].

Software testing is conventionally organized into several levels that differ in scope [32, 33]. Unit tests verify the behavior of individual functions or modules in isolation. Integration tests examine the interactions between combined components, checking that modules work correctly together. End-to-end (E2E) tests exercise complete workflows from start to finish under conditions resembling real system use, verifying that the system as a whole behaves as expected from the user’s perspective [34]. Each level complements the others: unit tests catch localized defects cheaply, integration tests expose interface mismatches, and E2E tests detect failures that only manifest in realistic execution contexts.

Beyond automating test execution, recent work has also explored automating the creation of tests themselves, which can lower the cost of achieving adequate

test coverage. This direction is discussed in Section 4.3.

■ 4.2 End-to-End Testing

As introduced in the previous section, end-to-end (E2E) testing exercises complete workflows from start to finish under conditions resembling real system use [34]. Because E2E tests span multiple subsystems, interfaces, and dependencies, they are able to reveal defects that may not be visible when components are tested separately, such as errors caused by incorrect assumptions at subsystem boundaries, misconfigured dependencies, or failures that arise only in realistic execution contexts [33, 5].

This broader scope makes E2E testing especially valuable, but also more demanding than lower-level testing. End-to-end tests are typically more difficult to design, execute, and maintain [5]. Nevertheless, they provide an important form of assurance because they evaluate software behavior closer to the way the system is actually used in practice. For this reason, E2E testing is best understood as a complement to unit and integration tests rather than a replacement for them, helping to detect workflow-level defects that would otherwise remain unnoticed [33].

The relevance of E2E testing becomes even greater in the context of AI-generated code. When a coding agent modifies a larger application, local changes may break interactions between components or alter behavior that only becomes visible when the system is exercised as a whole through complete user workflows. End-to-end tests therefore provide a particularly useful mechanism for evaluating generated software, because they allow candidate implementations to be assessed through observable system behavior rather than exact similarity to a reference solution.

■ 4.3 Test Generation for AI-Generated Code

Automated test generation has long been pursued as a way to reduce the cost of software validation and to make testing more scalable. Its appeal is particularly strong in the context of AI-generated code, where coding agents may produce many candidate implementations or entire application-level modifications, making manual evaluation increasingly impractical. Recent work has shown that large language models can substantially assist the test generation process, but most existing approaches focus primarily on generating unit tests rather than end-to-end tests [35]. This gap is significant because coding agents increasingly generate and modify complete software systems, in some cases producing end-to-end applications directly from natural-language requirements [36], where correctness depends on workflow-level interactions that unit tests cannot capture [35]. At the same time, manually writing

end-to-end tests is resource-intensive, as it requires non-trivial knowledge of both the system under test and the testing framework, and is therefore often deprioritized in fast-paced development settings [35]. Moreover, because coding agents operate over complete repositories, they may uncover potential issues that would not be captured by more traditional testing approaches [37].

Generating useful end-to-end tests in this setting is itself a complex task. In practice, the process is often two-phased: user requirements are first captured as natural-language user stories, which are then translated into executable test scripts [38]. For a large language model, or an LLM-based coding agent, this requires several non-trivial reasoning steps. The agent must infer the intended workflow, decompose it into concrete actions with respect to the actual implementation, and finally express these actions in the syntax of the target testing framework, which in this thesis corresponds to TypeScript tests written for Playwright. This is particularly challenging in web applications, where dynamic navigation flows and complex user interactions remain difficult to model reliably [39]. Two especially critical actions in such workflows are page navigation and form filling, both of which frequently determine whether an end-to-end scenario succeeds or fails [39]. As a result, generated tests may contain errors of their own or provide incomplete workflow coverage [39].

These difficulties highlight a broader challenge in evaluating AI-generated code. A major requirement is ensuring software quality in terms of functional correctness, reliability, security, and maintainability [40]. When generated tests are used as the primary evaluation mechanism, they become a proxy for functional correctness in situations where no reference implementation is available. However, this introduces an inherent ambiguity: when a generated test fails, it may be unclear whether the failure reveals an actual defect in the system under test or an error in the generated test itself [41]. The usefulness of automated test generation therefore depends not only on the ability to produce executable tests, but also on the validity, coverage, and trustworthiness of those tests as evaluation instruments. This tension is central to the evaluation of coding agents and motivates the need for carefully designed test-based evaluation workflows in the remainder of the thesis.

Chapter 5

Related Work

This chapter reviews recent work at the intersection of LLM-based coding agents and automated software testing, organized into two areas: LLM-based test generation, which examines the use of language models to produce executable tests from natural-language specifications, and evaluation of coding agents, which surveys the benchmarks and evaluation methodologies used to assess agents in realistic software-engineering settings.

5.1 Test Generation with LLMs

Recent work has shown that large language models can support automated test generation across a variety of testing settings, ranging from unit tests to acceptance and end-to-end tests. A substantial portion of this literature, however, remains focused on generating relatively localized tests, most commonly unit tests. For example, Xu et al. [42] propose CANDOR, a multi-agent framework for generating JUnit test cases and specification-based oracles (test assertions derived from the method’s specification rather than its observed behavior) for Java methods. Their system combines specialized agents for planning, test construction, and oracle refinement, and shows that multi-agent collaboration can improve both coverage and oracle quality. Similarly, Wang et al. [37] present Copilot for Testing, a context-aware retrieval-augmented system that integrates bug detection, fix suggestions, and test-case generation directly into the development environment. These works demonstrate that LLMs can be effective assistants for software testing, but they are primarily oriented toward localized code validation rather than full application workflows.

A smaller but increasingly relevant line of work considers the generation of higher-level tests from natural-language requirements. Ferreira et al. [38] study acceptance test generation in an industrial setting and propose a two-stage pipeline in which user stories are first transformed into Gherkin scenarios (structured Given/When/Then descriptions of expected behavior) and then

translated into executable Cypress tests. Their results suggest that LLMs can be practically useful for acceptance testing (verifying that the system meets its requirements from the user’s perspective), especially when intermediate natural-language representations give testers more control over the final output. Relatedly, Júnior et al. [35] introduce GenIA-E2ETest, an approach that transforms natural-language functional requirements into executable Robot Framework end-to-end tests through requirement interpretation, UI element extraction, and script generation. Their evaluation shows that LLM-based E2E generation can achieve high execution rates with relatively limited adaptation effort, but also reveals sensitivity to semantic ambiguity, fragile locators, and context-dependent workflows. In a similar direction, Alian et al. [43] propose AUTOE2E, which infers application features using LLMs and then generates E2E tests that maximize feature coverage. Unlike these approaches, which treat test generation as the end goal, the present thesis uses generated E2E tests as an internal feedback mechanism within the coding agent’s workflow, where the primary objective is improving the generated code rather than producing tests.

Several recent works address the challenges of semantic ambiguity and fragile locators by enriching the generation process with additional structure or domain-specific reasoning. Le et al. [39] combine large language models with screen transition graphs (graphs capturing navigation paths between pages) and form-state graphs (graphs modeling input field dependencies) in order to generate Selenium test cases for navigation and form-filling scenarios in web applications. Their results show that graph-guided generation improves test quality, but that dynamic navigation flows, hidden fields, and complex form logic remain difficult. In a complementary direction, Teoh et al. [41] propose WebTestPilot, an LLM-based web-testing agent that infers its own test oracles from natural-language specifications by symbolizing temporal, causal, and data dependencies across web states. Their results suggest that the oracle problem (the challenge of automatically determining correct expected behavior when it is only implicitly specified in natural language) can be mitigated, at least partially, through richer modeling of test expectations, although the generated tests still depend heavily on the quality of requirement interpretation.

Taken together, these works show that LLM-based test generation has progressed beyond simple unit-level scenarios toward acceptance and end-to-end testing of realistic systems. At the same time, they also reveal several persistent limitations that are directly relevant to this thesis. First, most approaches still assume relatively well-scoped testing tasks rather than evaluation of complete software-development trajectories. Second, even when executable tests are generated successfully, it often remains unclear whether a failed test reflects a genuine defect in the system under test or an error in the generated test itself [41]. Third, dynamic web interfaces, workflow decomposition, and robust oracle construction continue to pose major challenges [35, 39]. These limitations motivate the need for evaluation workflows in which test genera-

tion is treated not merely as a convenience feature, but as a methodological problem in the assessment of AI-generated software. The approach proposed in this work addresses these limitations in several ways. First, rather than treating test generation as a standalone task, it integrates generated tests into the agent’s development loop, where they serve as feedback on complete implementation trajectories spanning both frontend and backend modifications. Second, the evaluation framework separates internal evaluation (which may be unreliable) from external hidden test suites that serve as ground truth, making it possible to measure the accuracy of the agent’s own test judgments. Third, as described in Chapter 7, an alternative evaluation mode bypasses some of the oracle construction difficulty by verifying behavior through direct browser interaction rather than relying solely on pre-written test assertions.

■ 5.2 Evaluation of Coding Agents

A closely related line of research has focused on benchmarking coding agents in increasingly realistic software-engineering settings. Early evaluations of code generation systems were largely centered on isolated functions or algorithmic tasks, but recent benchmark design has shifted toward repository-level, workflow-oriented, and end-to-end software development scenarios. One important example is SWT-Bench, introduced by Mündler et al. [44], which reframes issue resolution from SWE-Bench as a test-generation task by asking models to generate fail-to-pass tests, i.e. tests that fail on the unpatched codebase and pass on the corrected version, thereby reproducing real software issues in Python repositories. Their results show that coding agents, especially SWE-Agent, can outperform non-agent baselines (direct prompting approaches that generate tests in a single pass without repository interaction) in test generation, suggesting that repository interaction and iterative reasoning are useful not only for code repair but also for constructing executable tests that serve as evaluation instruments. This benchmark is particularly relevant to this thesis because it connects coding agents directly to the generation of executable tests as signals of correctness.

Several newer benchmarks move beyond repository repair and assess coding agents on broader software-development tasks. Tran et al. [36] propose Vibe Code Bench, which evaluates frontier models on end-to-end web application development through browser-based workflows. Liu et al. [45] introduce E2EDev, a benchmark for end-to-end software development grounded in behavior-driven development scenarios (structured natural-language specifications of expected system behavior) and an automated Behave-based evaluation pipeline. Lu et al. [46] present ProjDevBench, which assesses coding agents on project-level development tasks involving autonomous design, build configuration, and execution-based validation, combining online-judge correctness with LLM-assisted code review. Across these benchmarks, the

common conclusion is that moving from isolated code snippets to complete software-development tasks exposes substantial weaknesses in current coding agents, even for state-of-the-art models.

Another group of benchmarks emphasizes repository interaction, environment setup, and deployability. Ni et al. [47] propose *GitTaskBench*, which evaluates code agents on real-world tasks that require leveraging existing repositories, managing execution environments, and completing multimodal development tasks. Their study highlights environment setup as a dominant source of failure and shows that cost-efficiency can vary widely across agent-framework and model combinations. Yang et al. [48] extend this perspective with *ABC-Bench*, a benchmark for backend engineering tasks spanning multiple languages and frameworks, in which agents are evaluated through environment build success and functional execution of deployed services. *RepoGenesis* pushes the scope further toward repository-level generation from natural-language requirements, evaluating whether agents can synthesize fully deployable microservice repositories with coherent architecture, dependencies, and API behavior [49]. Together, these benchmarks make clear that environment configuration, cross-file consistency, dependency management, and deployability are central bottlenecks in realistic agentic software development.

The literature also contains more domain-specific evaluation settings that reinforce the same message. Liu et al. [50] study automated web vulnerability reproduction with LLM agents and show that success rates remain below 25% even in a carefully constructed benchmark of real-world CVEs, with major sensitivity to environment details and authentication guidance. Xu et al. [51] explore context-aware visual prompting for geospatial web dashboards and incorporate an AI repair agent with self-validation, again demonstrating that iterative execution and correction are central to practical success. Although these works focus on particular application domains, they further support the broader observation that coding-agent performance depends on much more than code synthesis alone.

A final strand of related work examines properties of generated code that lie beyond pure functional correctness. Sabra et al. [52] show that functionally correct AI-generated code can still contain substantial numbers of code smells, bugs, and security vulnerabilities, arguing that execution-based success alone is insufficient as a quality criterion. Kharmia et al. [40] similarly find significant variation in security quality across models and programming languages, with frequent reliance on outdated security practices. These studies are not benchmarks of coding agents in the same sense as *SWE-Bench*, *ProjDevBench*, or *ABC-Bench*, but they are important because they highlight dimensions of code evaluation, such as security, maintainability, and adherence to best practices, that are only partially captured by standard task-success metrics.

In contrast to the works discussed above, this thesis lies at the intersection of the two research directions. On the one hand, it is informed by recent work on

LLM-based test generation, especially for acceptance and end-to-end testing. On the other hand, it is motivated by the growing body of benchmarks showing that coding agents must be evaluated in realistic environments where success depends on repository interaction, execution feedback, and system-level behavior. The gap addressed here is not only the external evaluation of coding agents, but the integration of evaluation directly into the agentic workflow, so that it can serve as an internal feedback mechanism for improving the quality of generated software.

Chapter 6

Methodology

This chapter describes the experimental methodology used to evaluate the proposed coding-agent architectures. It presents the experimental design, introduces the evaluation framework and metrics, describes the synthetic application corpus used for the experiments, and specifies the model selection and execution environment.

6.1 Experimental Design

To address the research questions formulated in Section 1.1, the thesis compares three coding-agent configurations that differ in the form of internal evaluation available to the agent: a baseline configuration without any dedicated evaluation mechanism, a self-evaluation configuration in which the agent generates and executes its own end-to-end tests, and an agentic evaluation configuration in which a separate agent verifies the implementation through direct browser interaction. The architecture and implementation of all three configurations are described in detail in Chapter 7.

The experimental study is conducted across multiple language models. Each model is evaluated under all three agent configurations on the same application-task corpus and under the same hidden evaluation protocol. The primary controlled variable is therefore the form of internal evaluation available to the agent, while the external task descriptions, application baselines, and hidden evaluation tests remain fixed. This design makes it possible to compare not only whether evaluation-enhanced workflows improve task success, but also how different internal evaluation paradigms influence the trade-off between solution quality and computational cost.

6.2 Evaluation Protocol and Metrics

In this thesis, code quality is assessed through the notion of *utility*. Because no single metric can capture all relevant aspects of generated code quality in a coding-agent setting, we propose a structured evaluation framework that decomposes utility into three complementary dimensions:

- *Functional correctness*, measured primarily through $Pass@1$ on hidden task-completion tests.
- *Regression safety*, measured using hidden regression end-to-end tests that verify preservation of the baseline application functionality.
- *Efficiency*, measured through wall-clock runtime and token usage.

The primary metric used in the thesis is $Pass@1$, which measures the proportion of tasks successfully resolved on the first completed attempt. A task is considered successfully resolved if the submitted solution passes all corresponding hidden end-to-end task-completion tests. Since these tests are not exposed to the agent, they serve as an external evaluation mechanism independent of the agent’s internal self-evaluation process. In this sense, $Pass@1$ represents the main measure of functional correctness. Because $Pass@1$ is an all-or-nothing metric that does not reward partial progress, it is complemented by the *subtest fraction*, which measures the average proportion of individual subtests passed per task. This softer metric captures cases where the agent produces a partially correct solution that passes some but not all of the hidden subtests.

To complement this primary metric, each run is also evaluated for regressions and computational cost. Regression safety is assessed using a separate set of hidden regression tests, while efficiency is measured through runtime and token consumption. Together, these metrics make it possible to compare the agent configurations not only in terms of whether they solve tasks correctly, but also in terms of whether they preserve existing functionality and at what computational cost. In this sense, utility is not treated as a single aggregate score, but as a structured evaluation perspective combining task success, regression safety, and efficiency. It is important to note that utility as defined here captures the quality of the code generation process with respect to the task specification, rather than intrinsic properties of the produced code such as maintainability, security, or adherence to coding standards. This narrower scope is appropriate for this evaluation, which focuses on whether internal evaluation mechanisms improve the agent’s ability to produce functionally correct solutions.

The evaluation in this thesis is conducted on the synthetic application corpus described in Section 6.3, which is designed to reflect the structure of the applications used in the Amazon Nova AI Challenge 2026. All experimental conclusions are based on this corpus and its hidden task-completion and

regression test suites.

6.3 Application Corpus and Task Construction

As a part of the Amazon Nova AI Challenge 2026, to which this thesis contributes, a synthetic dataset of full-stack web applications was created by Jan Chleboun, a fellow member of the CTU team participating in the challenge. The dataset consists of 100 varied full-stack web applications whose technical stack combines a Python backend with a JavaScript frontend. To ensure diversity among the generated applications, the frameworks used for both backend and frontend are sampled from a pool of popular frameworks (e.g., Flask, FastAPI, and Django for the backend; React, Vue, and vanilla JavaScript for the frontend), as are the application topics (e.g., e-commerce, task management, social media).

Each generated application comes with a set of 10 tasks to be completed by the coding agent: 4 easy, 3 medium, and 3 hard. Each task is defined by high-level acceptance criteria and a corresponding end-to-end Playwright test that verifies the correct implementation of the task. In addition to the task-completion tests, each application also includes a set of regression end-to-end tests, whose purpose is to verify that the baseline functionality of the application has not been broken by the coding agent’s changes.

Each agent run is initialized through a simulated user message that enriches the high-level acceptance criteria with implementation-level details extracted from the task-completion tests, such as exact label text, element identifiers, API endpoint paths, and expected response formats. This produces a natural-language instruction specific enough to guide the agent toward a correct implementation without exposing the test logic itself. The enrichment is necessary because passing only the raw acceptance criteria would leave substantial room for ambiguity, for example allowing the agent to introduce interface identifiers or selectors that differ from those assumed by the evaluation tests. Conversely, exposing the hidden test directly would risk leaking evaluation details and would make it possible for the agent to overfit or “cheat” the test rather than solve the underlying task. The exact prompt template used by the simulated user is provided in Appendix A, and a concrete example of a task definition, its Playwright test, and the resulting simulated user output is included in Appendix B.

Although the main purpose of the dataset is to support future fine-tuning of the Nova model used by the team’s competition agent, by distilling knowledge from more capable models, it also serves as a suitable evaluation dataset for the experiments in this thesis, since it provides diverse full-stack applications with tasks of varying difficulty and hidden end-to-end test suites that enable objective, automated assessment of task completion. Importantly, since the dataset was synthetically generated by a different model specifically for this

project, the evaluated models cannot have encountered it during their training. Although the dataset is intended to be used for fine-tuning in a later stage of the competition, this fine-tuning had not taken place at the time of the experiments, and the corpus is used exclusively for evaluation purposes in the present experiments, ensuring that there is no train–test contamination.

6.4 Models and Execution Environment

The experiments were conducted across multiple language models and under a distributed execution setup designed to make the full evaluation matrix computationally feasible.

6.4.1 Model selection

To evaluate the impact of the proposed evaluation-enhanced architecture, the experiments were conducted using five different models. The goal of this selection was to reduce the risk that the observed results would reflect only the behavior of a single model family, and instead assess whether the effect of the proposed evaluation component can be observed across models from different AI labs. The selected set includes four open-weight, state-of-the-art models with strong coding capabilities, as well as one proprietary model of direct practical relevance to the Amazon Nova AI Challenge 2026.

The four open-weight models were chosen to cover strong contemporary coding and agentic capabilities while remaining accessible for experimentation. The fifth model, Nova 2 Lite, was included because it is part of the Nova family of models that the team is required to use in the competition, and it was therefore necessary to evaluate how the proposed architecture behaves under this practical constraint as well. Another important factor in model selection was their availability through Amazon Bedrock, which served as the main inference platform for the large-scale experimental runs. A concise description of the selected models follows:

1. **Qwen3 Coder 30B A3B Instruct (Qwen)** is a 30B-parameter open-weight coding model based on a mixture-of-experts (MoE) architecture, in which only 3.3B parameters are activated per token. It supports a 256K token context window. Its MoE design makes it lightweight enough to run on consumer-grade GPUs (approximately 32 GB VRAM at 4-bit quantization), and its instruction tuning improves tool-use behavior in agent settings. While competitive relative to its small activated parameter count, it does not match the benchmark performance of the larger models in this selection [53].
2. **MiniMax M2.5 (MiniMax)** is a 229B-parameter open-weight general-purpose model with a dense architecture and a 1M token context window.

It achieves state-of-the-art results on coding and agentic benchmarks, approaching the performance of top proprietary models. It is distinguished by its use of agentic reinforcement learning during post-training, intended to improve planning and multi-step tool use in agent workflows [54].

3. **GLM-4.7 (Z.ai)** is a 358B-parameter open-weight code-oriented model with a dense architecture and a 128K token context window. It achieves state-of-the-art performance on coding benchmarks, comparable to the strongest proprietary models. It is distinguished by its support for interleaved reasoning and action, including preserved reasoning across interaction history and turn-level thinking in agentic coding workflows [55].
4. **Kimi K2.5 (Moonshot AI)** is a 1.1T-parameter open-weight general-purpose model based on a mixture-of-experts architecture with 384 experts, of which 8 are selected per token, resulting in approximately 32B activated parameters per forward pass. It supports a 256K token context window. Although not specifically designed as a coding model, it achieves state-of-the-art results on software engineering benchmarks and is expected to be the strongest of the selected models. It is distinguished by its agent swarm mode, in which the model can decompose complex tasks into subtasks executed in parallel by dynamically instantiated sub-agents [56].
5. **Nova 2 Lite (Amazon)** is a proprietary general-purpose model with a dense architecture, reasoning support, and a 1M token context window, although practical usefulness was observed to degrade beyond approximately 100K input tokens. It is the smallest model in Amazon’s second-generation Nova family. It was included because the competition constrains the team to the Nova model family; its benchmark performance is comparable to the Qwen model in this selection but substantially below the three larger open-weight models [57].

Table 6.1 summarizes the on-demand inference pricing for each model on Amazon Bedrock at the time of the experiments. The prices span a wide range, from \$0.15 to \$0.60 per million input tokens and from \$0.60 to \$3.00 per million output tokens, which is relevant for interpreting the cost–quality trade-off discussed in Chapter 8.

6.4.2 Execution environment

To keep the experiment computationally feasible, 20 applications from the synthetic dataset were selected for the main experimental study. The selection was carried out iteratively: applications were sampled at random and tested with preliminary agent runs; those that were broken (e.g., failing to build or start), trivial to solve, or impossible to solve for all models were discarded and replaced with new samples until a set of 20 functional, non-trivial

Model	Input	Output
Nova 2 Lite	\$0.30	\$2.50
Qwen3 Coder 30B	\$0.15	\$0.60
MiniMax M2.5	\$0.30	\$1.20
GLM 4.7	\$0.60	\$2.20
Kimi K2.5	\$0.60	\$3.00

Table 6.1: On-demand inference pricing on Amazon Bedrock (per 1M tokens, USD). Prices are as listed at the time of the experiments.

applications remained. Since each application is associated with 10 tasks, and the experiments compare 3 agent configurations across 5 models, the resulting experiment matrix contains $20 \times 10 \times 5 \times 3 = 3,000$ runs. Assuming an average runtime of 15 minutes per run, sequential execution would require roughly 750 hours.

For this reason, the full experiment was migrated to AWS, where the runs could be executed in parallel. In the final setup, 40 workers were deployed on EC2 `m5.xlarge` instances (4 vCPUs, 16 GiB RAM). This reduced the total runtime of the full experiment to under 24 hours, which was considered an acceptable upper bound for a complete evaluation cycle.

6.5 Threats to Validity

The main threat to internal validity lies in the possible mismatch between the agent’s internal evaluation mechanism and the hidden evaluation tests used as the external measure of task success. In particular, the generated-test self-evaluation workflow may produce tests that reflect a slightly different interpretation of the task than the hidden private tests, which can lead to cases where the agent appears successful according to its own internal evaluation while still failing the final external evaluation. A related threat arises from the stochastic behavior of large language models: each model-task-configuration triple was executed exactly once, as the available computational budget was allocated toward covering a wider variety of models and applications rather than repeated runs of the same configuration, so individual results are point estimates that may not be fully stable under replication. The bootstrap confidence intervals reported in the experimental chapter partially address this by quantifying uncertainty at the aggregate level, but per-model estimates remain subject to run-to-run variance. In addition, the compared models differ not only in size, but also in architecture, training data, and tool-use behavior, which means that observed performance differences cannot be attributed to a single factor in isolation.

The main threat to external validity is the use of a synthetic application corpus focused on full-stack web applications with Python backends and

JavaScript frontends. This scope was chosen deliberately to ensure maximal compatibility with the Amazon Nova AI Challenge 2026, whose tasks follow the same technical stack, but it does not cover the full diversity of real-world software engineering settings, and the conclusions of this thesis may therefore not generalize directly to other domains, programming languages, or repository structures. A related concern is that the application corpus was created within a broader collaborative research effort, which may introduce implicit design assumptions aligned with the project setting. Finally, the evaluation was deliberately scoped to functional correctness, regression safety, and efficiency, as these are the dimensions most directly affected by the proposed evaluation-enhanced workflow. Other aspects of code quality, such as security and maintainability, are acknowledged as important but are left for future work.

Chapter 7

Proposed Agent Architecture

This chapter presents the coding-agent architectures evaluated in the practical part of the thesis. Rather than adopting an existing open-source coding agent such as SWE-Agent [4] or OpenHands [58], the experiments use a custom agent built as a streamlined version of the coding agent developed by the Alquist Coder team at CIIRC CTU for the Amazon Nova AI Challenge 2026. This choice was motivated by the need to control the internal evaluation component independently of the coding logic, to support Amazon Bedrock as the sole inference backend, and to integrate Docker-based end-to-end testing directly into the agent loop, requirements that would have necessitated substantial modification of any existing framework. An additional benefit of this approach is that the findings from this thesis can be directly transferred to the coding agent developed for the challenge. Comparing the proposed agent against other existing agents (e.g., SWE-Agent or OpenHands) was deliberately excluded from the experimental design, as the goal is to isolate the effect of the internal evaluation component by varying only the evaluation mechanism while keeping the underlying agent architecture fixed; a cross-agent comparison would conflate the evaluation effect with differences in agent architecture, tool design, and prompting strategy.

The agent described in this chapter was implemented from scratch by the author using the Pydantic AI framework; while inspired by the competition agent that the author co-designed, the codebase is entirely new. The complete source code, including the evaluation infrastructure and the analysis notebooks, is available at <https://gitlab.fel.cvut.cz/kloucmat/eval-coder>. The proposed approaches extend a baseline coding agent with internal evaluation mechanisms intended to provide feedback during task completion. Two alternative evaluation-enhanced variants are considered: a *test-generation-based* self-evaluation architecture and an *agentic* evaluation architecture based on Playwright MCP. Both are compared against a baseline coding agent that operates without any internal evaluation component.

7.1 Baseline Coding Agent Workflow

The baseline coding-agent workflow consists of two stages: agent execution and post-hoc verification. The agent receives an enriched task description produced by the simulated user described in Section 6.3 and operates on an isolated sandbox copy of the target repository. The exact prompt template used for the baseline agent is included in Appendix A.

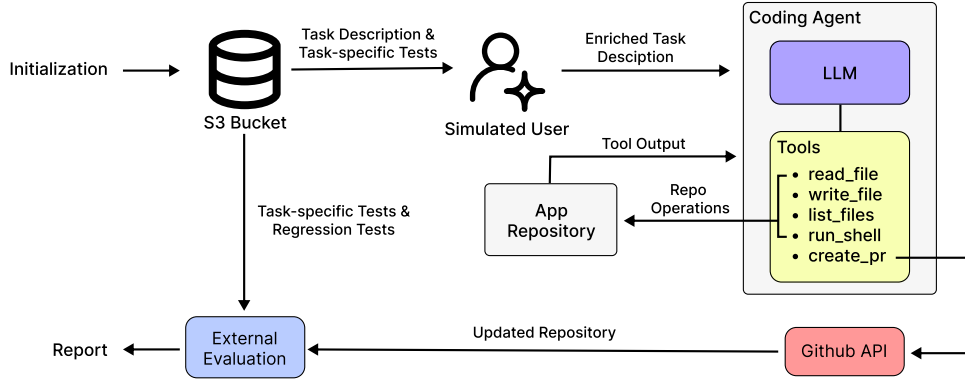


Figure 7.1: Baseline coding-agent workflow. The task description and hidden tests are retrieved from S3, enriched by a simulated user, and passed to the coding agent. The agent operates on the repository using file-system and shell tools, then submits a pull request via the GitHub API. The external evaluation harness runs the hidden test suites against the updated repository post-hoc.

As illustrated in Figure 7.1, the agent is equipped with five tools. Three of them are file-system tools: `read_file`, `write_file`, and `list_files`, which allow the agent to inspect and modify source code within the confined workspace. A general-purpose `run_shell` tool enables execution of shell commands such as `grep`, `git`, or package managers. Finally, `create_pr` pushes the agent’s feature branch and opens a pull request via the GitHub API. The system prompt prescribes an explore-first workflow: inspect the codebase, create a feature branch, implement the requested changes, commit them, and submit a pull request. In the baseline configuration, no testing or evaluation tools are available. The agent must therefore rely entirely on its understanding of the task description and the codebase when producing a solution.

Once the agent submits its pull request, the external evaluation harness checks out the submitted branch, starts the full application stack in Docker containers, and executes two suites of hidden Playwright end-to-end tests. The regression suite verifies that pre-existing functionality has not been broken, while the task-specific suite checks whether the requested feature has been implemented correctly. A task is considered solved only if all task-specific tests pass. This strictly post-hoc evaluation setup serves as the experimental baseline against which the two evaluation-enhanced architectures are compared, thereby isolating the effect of providing feedback to the agent

during development.

7.1.1 Shared Architectural Simplifications

To keep the experimental comparison as controlled as possible, the agent architectures considered in this thesis use a restricted interaction pattern. The core agentic loop is limited to the interaction between the coding agent, the repository, and, in the evaluation-enhanced variants, the internal evaluation component. The simulated user is used only to initialize the run through a single enriched task description and does not participate in further iterative interaction with the coding agent. This avoids introducing an additional feedback channel that would make the interpretation of the experimental results more difficult.

Restricting the simulated user-agent interaction to a single turn also ensures that the full interaction history remains within the model’s context window, removing the need for a dedicated external memory module for context management. As a result, the concrete architectures used in the experiments are intentionally simpler than the general LLM-agent pattern introduced in Chapter 3. This simplification makes it possible to compare the effect of the proposed internal evaluation mechanisms more directly, without confounding the results with additional user interaction or memory-management strategies.

7.2 Test-Generation-Based Internal Evaluation

The first of the two proposed internal evaluation modes, referred to in the remainder of this thesis as *self-evaluation mode*, is based on automated test generation. In this configuration, the baseline coding agent is extended with an internal evaluation component that allows it to validate intermediate solutions before pull request submission. Instead of relying solely on post-hoc external verification, the agent can generate its own end-to-end tests from the task description and use them as an internal source of feedback during development. The exact prompt template used in this mode is included in Appendix A.

As illustrated in Figure 7.2, the coding agent retains the baseline file-system, shell, and pull-request tools, but is additionally equipped with the `run_e2e_tests` tool, which serves as the interface to the internal evaluation component. The system prompt is correspondingly extended with a test-driven workflow. After exploring the codebase and creating a feature branch, the agent is instructed to first write Playwright test files that encode the expected behavior described in the task specification, then implement the requested feature, and finally invoke `run_e2e_tests` to validate its work. Importantly, although these tests are derived from the same enriched task

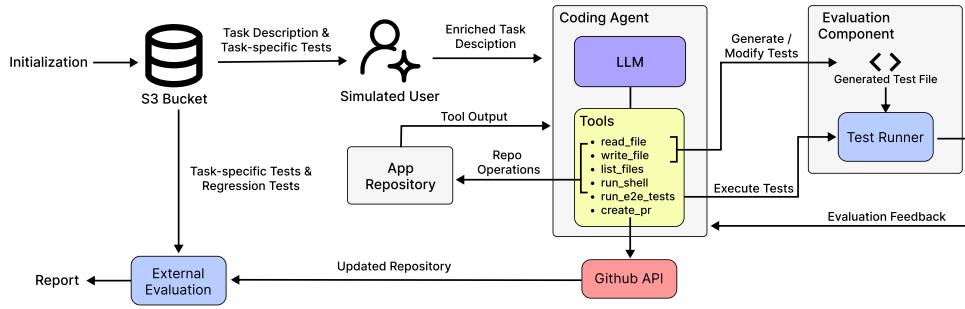


Figure 7.2: Test-generation-based evaluation architecture. In contrast to the baseline workflow, the coding agent is augmented with an internal evaluation component that generates and executes end-to-end tests and returns feedback during development, while the hidden external evaluation remains unchanged.

description that guides the implementation, they are generated by the coding agent itself and are distinct from the hidden task-specific tests used for the final external evaluation.

When invoked, `run_e2e_tests` starts the full application stack in Docker containers on the first call, while subsequent calls reuse the running containers unless code changes require a rebuild, and then executes the agent-authored test suite via a containerized Playwright runner. The tool returns structured feedback including the number of passed and failed tests, individual failure messages, and relevant portions of the raw test output. If the tests fail, the agent is instructed to diagnose the likely cause of the failure, for example by distinguishing between implementation gaps such as missing UI elements or broken API endpoints and errors in the generated tests themselves, such as selector mismatches, and then revise the code or the tests accordingly before invoking the tool again. This fix-evaluate cycle may be repeated up to five times, after which the agent must submit its current solution regardless of test status. In addition, the `create_pr` tool enforces this evaluation policy: it refuses to open a pull request if the agent has not yet called `run_e2e_tests`, or if the most recent internal evaluation failed and attempts still remain, thereby preventing the agent from bypassing the self-evaluation loop.

After the agent submits its pull request, the same post-hoc verification procedure as in the baseline configuration is applied using the hidden task-specific and regression test suites. The self-evaluation mode therefore tests whether an agent’s ability to generate and iteratively refine its own end-to-end tests can improve task completion rates. At the same time, the separation between agent-authored tests and held-out evaluation tests ensures that the final assessment remains independent of the internal evaluation mechanism.

7.3 Agentic Internal Evaluation

The second of the two proposed internal evaluation architectures, referred to in the remainder of this thesis as *agentic mode*, is based on Playwright MCP. In this configuration, the test-generation-based self-evaluation loop is replaced by an internal evaluator agent that verifies the implemented functionality through direct interaction with the running application.

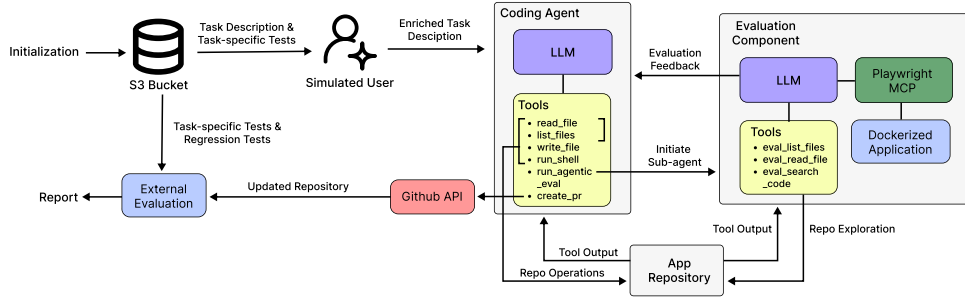


Figure 7.3: Agent-based internal evaluation architecture. In contrast to the baseline workflow, the coding agent is augmented with an internal evaluator agent that inspects the repository and interacts with the running application through Playwright MCP, while the hidden external evaluation remains unchanged.

As illustrated in Figure 7.3, the coding agent retains the baseline repository-manipulation tools and is additionally equipped with a dedicated evaluation tool, `run_agentic_eval`, which invokes a separate evaluator agent. The coding agent calls `run_agentic_eval` with a description of the implemented feature and the relevant acceptance criteria. The tool then starts the application in Docker containers, or rebuilds the affected containers on subsequent calls, launches the evaluator agent, and returns a structured verdict indicating which criteria passed or failed together with supporting evidence.

A key design decision is the use of a separate agent rather than exposing the browser tools directly to the coding agent. The evaluator receives only the natural-language feature description and acceptance criteria; it does not have access to the coding agent’s conversation history, tool-call log, or the git diff of modified files. Although the evaluator can read the repository to understand its structure, it cannot determine which specific changes were made. This separation prevents the evaluator from anchoring on the implementation details and forces it to verify the feature from the user’s perspective, based solely on the task specification and the observable behavior of the running application. This design was motivated by preliminary experiments in which the browser tools were exposed directly to the coding agent: when the agent had access to both its own implementation history and the browser tools, it tended to verify only what it had implemented rather than what the task specification required, and in some cases skipped evaluation entirely by reporting success based on its memory of the implementation without actually using the browser tools.

The evaluator agent operates in two phases. In the first phase, it explores the repository using dedicated tools, namely `eval_list_files`, `eval_read_file`, and `eval_search_code`, in order to understand the application’s structure, routes, and component layout before interacting with the live system. In the second phase, it uses the Playwright MCP server to interact with the running web application through browser-level actions such as navigation, clicking, typing, screenshot capture, and JavaScript execution. This allows the evaluator to perform a form of natural-language-guided verification resembling the behavior of a human tester: navigating through the interface, filling out forms, submitting data, and checking whether expected UI elements, messages, or behaviors are present. The result of this process is a structured evaluation report with per-criterion pass/fail outcomes and textual evidence, which is then returned to the coding agent as actionable feedback.

As in the self-evaluation mode, the coding agent may iterate through the fix-evaluate cycle up to five times, and the `create_pr` tool enforces compliance with this policy. In addition, this mode includes a loop-detection mechanism that monitors the evaluator’s tool-call history for repeated patterns, such as the same repository exploration or browser-navigation sequence occurring multiple times, and terminates the evaluation early if such a cycle is detected. This prevents unbounded resource consumption and limits pathological evaluator behavior. After the coding agent submits its pull request, the external evaluation harness applies the same post-hoc verification as in the other two configurations, executing the hidden task-specific and regression test suites against the submitted branch. The agentic mode therefore tests whether natural-language feedback from a browser-equipped evaluator agent can guide the coding agent more effectively than self-authored test scripts, particularly in user-interface-centric tasks where writing correct selectors and assertions may itself be error-prone. An annotated example of a complete agentic-mode run, including the exploration, implementation, evaluator feedback, fix, and successful re-evaluation, is provided in Appendix C.

7.4 Implementation Details

The complete coding-agent system presented in this thesis is implemented in Python using Pydantic AI, an open-source framework for building LLM-powered agents with structured tool use and typed outputs. Pydantic AI provides an abstraction layer over multiple model providers, including Amazon Bedrock, which serves as the inference backend in this work. This makes it possible to execute the same agent logic across different models without introducing provider-specific implementation code. The framework follows a request-response loop in which the model produces text and tool-call requests, the runtime executes the requested tools, and the resulting outputs are fed back to the model until a final structured response matching a predefined schema is produced [59]. This design maps naturally onto the coding-agent

workflows used in this thesis, where the model must interleave reasoning with tool use while operating over a repository and, in the evaluation-enhanced variants, also interact with internal evaluation components. Pydantic AI was chosen over lower-level API integration and more general orchestration frameworks because it offers strong type safety through Pydantic models, native support for structured outputs, and a straightforward tool-registration mechanism in which ordinary Python functions with type-annotated arguments can be exposed directly to the model [59].

End-to-end testing in both the self-evaluation mode and the final hidden verification stage is based on Playwright, an open-source browser automation framework developed by Microsoft for end-to-end testing of modern web applications [60]. Playwright provides a high-level API for controlling browsers programmatically, including page navigation, interaction with DOM elements, screenshot capture, and script execution [60]. In the self-evaluation mode, the coding agent writes Playwright test scripts in TypeScript, which are then executed against the dockerized application through the internal evaluation component. In the post-hoc verification stage, separate held-out Playwright test suites serve as the external ground truth for task completion and regression safety.

The agentic evaluation mode connects LLM-based reasoning with browser interaction through the Model Context Protocol (MCP), an open protocol for exposing external tools and resources to LLM-based applications through a standardized client-server interface [61]. In this architecture, the evaluator agent uses the Playwright MCP server, an official implementation that exposes Playwright-based browser automation as MCP-compatible tools [62]. This server enables LLMs to interact with web pages through structured accessibility-based representations rather than relying on screenshots or vision-specific models [62, 60]. When the evaluator agent is instantiated, Pydantic AI launches the Playwright MCP server as a subprocess and registers its capabilities alongside the repository-exploration tools. The evaluator can then inspect the repository, launch and control a headless browser, navigate through the application, interact with interface elements, and gather evidence about the satisfaction of acceptance criteria without relying on a fixed hand-written test script. This architecture supports a more exploratory form of verification that can adapt to the concrete implementation produced by the coding agent, rather than depending on predefined selectors and assertion patterns [61, 62].

Chapter 8

Experimental Results

This chapter presents the experimental results obtained by evaluating the three coding-agent configurations described in Chapter 7 across five language models and twenty generated web applications from the synthetic corpus introduced in Section 6.3. The results are organized around the three evaluation dimensions defined in Section 6.2: functional correctness, regression safety, and efficiency. A qualitative analysis of failure modes and a discussion of the observed limitations conclude the chapter.

8.1 Experimental Overview

The full experiment was conducted on 20 applications from the generated corpus, each with 10 tasks, evaluated under three agent configurations (baseline, self-evaluation, and agentic) using the five language models described in Section 6.4.1.

The resulting experiment matrix contains $20 \times 10 \times 5 \times 3 = 3,000$ planned runs. Of these, 2,750 runs completed, with 2,495 finishing without errors, corresponding to a completion rate of 90.7%. Incomplete runs were caused by infrastructure-level failures including SSM execution timeouts on long-running worker instances, transient Bedrock API errors, and occasional model-specific validation exceptions. The distribution of completed runs is approximately balanced across models and modes, with per-model completion rates ranging from 82% (MiniMax M2.5) to 96% (Qwen3 Coder 30B). The missing runs were not re-executed, as selectively rerunning failed configurations could introduce bias by giving those runs a second chance under potentially different infrastructure conditions. In total, the experiment consumed approximately 2.9 billion input tokens and 76 million output tokens (including both agent and evaluator calls), corresponding to roughly 702 hours of wall-clock compute time and an estimated API cost of approximately \$1,350, covered by AWS credits provided by Amazon for the Amazon Nova 2026 Challenge.

8.2 Functional Correctness

The primary metric used throughout the experiments is Pass@1 as defined in Section 6.2. To ensure a fair comparison across configurations, the results reported in Tables 8.1–8.3 include only error-free runs, excluding those that terminated due to infrastructure failures, API errors, or timeouts. The impact of errors on the overall results is discussed separately in Section 8.6.

Table 8.1 summarizes the Pass@1 results across all five models and three configurations.

Model	Baseline	Self-eval	Agentic	Δ se	Δ ag
Nova 2 Lite	8.4%	10.4%	9.8%	+2.0 pp	+1.4 pp
Qwen3 Coder 30B	5.1%	5.1%	6.9%	−0.1 pp	+1.8 pp
MiniMax M2.5	41.5%	51.5%	49.4%	+10.0 pp	+7.8 pp
GLM 4.7	37.4%	47.9%	45.4%	+10.5 pp	+8.0 pp
Kimi K2.5	43.2%	49.7%	50.6%	+6.5 pp	+7.4 pp
All models	26.8%	31.6%	31.5%	+4.7 pp	+4.6 pp

Table 8.1: Task completion rate (Pass@1) by model and agent configuration, computed over error-free runs only. The two rightmost columns show the absolute change from the baseline to the self-evaluation and agentic configurations, respectively.

When restricted to error-free runs, the evaluation-enhanced configurations yield a consistent improvement over the baseline. The aggregate Pass@1 increases from 26.8% in the baseline to 31.6% under self-evaluation and 31.5% under the agentic mode. Among the three stronger models, all show clear gains under both evaluation-enhanced configurations: GLM 4.7 improves by +10.5 pp and +8.0 pp, MiniMax M2.5 by +10.0 pp and +7.8 pp, and Kimi K2.5 by +6.5 pp and +7.4 pp under self-evaluation and agentic modes, respectively. The two weaker models, Nova 2 Lite and Qwen3 Coder 30B, achieve baseline Pass@1 rates below 10% and show only marginal changes under either evaluation-enhanced configuration. The reasons for this are investigated in the cross-model experiment (Section 8.8).

To capture partial progress beyond the all-or-nothing Pass@1 metric, Table 8.2 reports the average fraction of task subtests passed per run. This softer metric reveals more consistent improvements across models, particularly under the agentic configuration.

Under the subtest-fraction metric, MiniMax M2.5 shows the largest improvement in the agentic configuration (+14.8 pp), reinforcing the finding that evaluation feedback guides these models toward more complete solutions. A similar trend is visible for GLM 4.7 (+8.9 pp) and Kimi K2.5 (+8.3 pp). These results indicate that even when the evaluation-enhanced runs do not cross the all-or-nothing Pass@1 threshold, they produce partial solutions that

Model	Baseline	Self-eval	Agentic	Δ se	Δ ag
Nova 2 Lite	20.2%	27.8%	27.1%	+7.6 pp	+7.0 pp
Qwen3 Coder 30B	22.6%	24.0%	24.1%	+1.4 pp	+1.5 pp
MiniMax M2.5	59.1%	67.9%	73.9%	+8.8 pp	+14.8 pp
GLM 4.7	57.9%	71.1%	66.8%	+13.2 pp	+8.9 pp
Kimi K2.5	64.4%	71.1%	72.7%	+6.7 pp	+8.3 pp
All models	44.5%	51.1%	51.9%	+6.6 pp	+7.3 pp

Table 8.2: Average fraction of task subtests passed per error-free run, by model and configuration. Unlike Pass@1, this metric gives partial credit for runs that pass some but not all subtests.

are measurably closer to full task completion.

Because LLM outputs are stochastic, the observed improvements could in principle reflect sampling variability. To assess this, a paired bootstrap analysis was conducted. For each comparison (baseline vs. evaluation-enhanced mode), tasks were paired by identifier: only tasks with error-free runs in both configurations were included, and error runs were excluded prior to pairing. Each bootstrap resample draws individual tasks with replacement from the full pool of paired tasks; this process is repeated 10,000 times to construct confidence intervals. The reported intervals are percentile-based 95% confidence intervals on the paired difference in means.

The resulting aggregate confidence intervals exclude zero for both metrics and both evaluation modes, indicating that the improvement is statistically significant at the 95% confidence level: Pass@1 intervals of [+1.8, +7.3] pp for self-evaluation and [+1.0, +6.5] pp for agentic, and subtest-fraction intervals of [+3.8, +9.4] pp and [+4.0, +9.1] pp, respectively. At the per-model level, confidence intervals are wider due to smaller sample sizes: among the three stronger models, only GLM 4.7 under self-evaluation and Kimi K2.5 under agentic mode individually yield Pass@1 intervals that exclude zero. The aggregate significance reflects the consistent direction of improvement across models rather than individually large per-model effects.

8.3 Regression Safety

Table 8.3 reports the fraction of error-free runs in which all hidden regression tests pass, indicating that the baseline application functionality has been fully preserved.

It is important to note that the internal evaluation component only verifies the implementation of the assigned task; regression tests are entirely hidden from the agent and are never explicitly tested for during execution. The evaluation loop can only indirectly force regression fixes when a regression

Model	Baseline	Self-eval	Agentic	Δ se	Δ ag
Nova 2 Lite	81.2%	78.3%	58.4%	−2.9 pp	−22.8 pp
Qwen3 Coder 30B	87.6%	89.9%	89.0%	+2.3 pp	+1.4 pp
MiniMax M2.5	92.9%	88.3%	90.3%	−4.6 pp	−2.6 pp
GLM 4.7	91.5%	91.7%	93.2%	+0.3 pp	+1.8 pp
Kimi K2.5	93.3%	89.7%	93.4%	−3.6 pp	+0.1 pp
All models	89.7%	87.9%	86.2%	−1.8 pp	−3.5 pp

Table 8.3: Regression test pass rate (all base tests passing) by model and configuration, computed over error-free runs only.

is severe enough to prevent the evaluation component from testing the task itself, such as when the application fails to build or start.

The aggregate baseline rate is 89.7%, indicating that most models preserve existing functionality well. The evaluation-enhanced configurations show a slight decrease in aggregate base preservation (−1.8 pp for self-evaluation, −3.5 pp for agentic), driven primarily by Nova 2 Lite’s sharp regression under the agentic mode (−22.8 pp). GLM 4.7 and Kimi K2.5 maintain or slightly improve their base preservation rates under both evaluation-enhanced configurations.

The notable exception is Nova 2 Lite, where the agentic mode shows a sharp drop from 81.2% to 58.4%. Manual inspection reveals that this model frequently produces changes that break the application’s build or startup process during iterative fix–evaluate cycles, preventing the application from starting at all and thereby causing all regression tests to fail.

Overall, while the evaluation component does not directly help with regressions, the results suggest that for more capable models it does not increase the risk of introducing them either.

8.4 Efficiency

Both evaluation-enhanced configurations introduce substantial overhead compared to the baseline. As shown in Table 8.4, the self-evaluation mode roughly doubles runtime for most models, while the agentic mode roughly triples it. This overhead stems primarily from the Docker container build and startup on the first evaluation call, the execution of the evaluation tool itself, and subsequent fix–evaluate iterations. For Qwen3 Coder 30B, the overhead is smaller, likely because its weaker coding ability results in shorter implementation phases and fewer evaluation iterations. The duration distributions in Figure 8.1 further reveal that variance increases substantially under the evaluation-enhanced configurations, particularly in the agentic mode, where outlier runs can exceed 3,000 seconds.

Model	Baseline	Self-eval	Agentic
Nova 2 Lite	386	1,018	1,646
Qwen3 Coder 30B	206	420	442
MiniMax M2.5	691	1,464	1,324
GLM 4.7	423	1,268	1,480
Kimi K2.5	363	1,112	1,471
All models	412	1,030	1,247

Table 8.4: Average wall-clock duration per run (seconds), by model and configuration, excluding runs that ended in error.

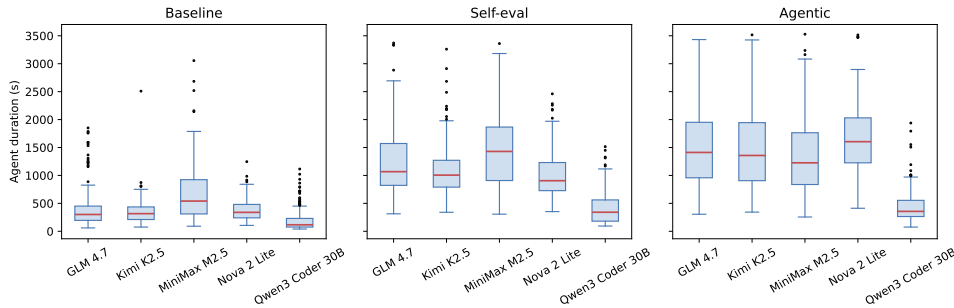


Figure 8.1: Agent duration distribution by model and configuration (error-free runs only). Each panel corresponds to one agent configuration; box plots show the median, interquartile range, and outliers.

Token consumption (Table 8.5) follows a similar pattern to duration: both evaluation-enhanced configurations roughly double the token usage compared to the baseline. Notably, the relative cost of the two evaluation modes varies by model strength: for the three stronger models, self-evaluation consumes more tokens than the agentic mode, while for the two weaker models the relationship is reversed. The agentic mode’s token counts include both the coding agent and the separate evaluator agent.

To assess whether the higher pass rates of the evaluation-enhanced configurations justify their additional compute cost, Table 8.6 reports the estimated cost per successfully completed task. For each model-configuration pair, this is computed as the total API cost across all error-free runs (using per-run token counts and the on-demand Bedrock pricing from Table 6.1) divided by the number of runs that passed all task tests. The metric thus accounts for the cost of failed attempts: a configuration with high per-run cost but also high pass rate may still be cost-effective.

The two evaluation modes differ substantially in their cost profile. Although the agentic mode is more expensive in wall-clock time (Table 8.4), it is consistently more token-efficient than self-evaluation for the three stronger models. This is reflected in the cost-per-success figures: for MiniMax M2.5, the agentic mode is nearly cost-neutral relative to the baseline (\$1.12 vs.

Model	Baseline	Self-eval	Agentic
Nova 2 Lite	289K	945K	1,492K
Qwen3 Coder 30B	474K	860K	1,326K
MiniMax M2.5	1,426K	2,461K	1,792K
GLM 4.7	688K	1,923K	1,273K
Kimi K2.5	688K	1,819K	1,191K
All models	708K	1,555K	1,409K

Table 8.5: Average total token consumption per run (input + output, in thousands), by model and configuration. Agentic mode includes tokens consumed by both the coding agent and the evaluator agent.

Model	Baseline	Self-eval	Agentic
Nova 2 Lite	\$2.24	\$4.47	\$7.94
Qwen3 Coder 30B	\$1.49	\$2.67	\$2.99
MiniMax M2.5	\$1.06	\$1.47	\$1.12
GLM 4.7	\$1.15	\$2.47	\$1.73
Kimi K2.5	\$1.02	\$2.31	\$1.48

Table 8.6: Estimated cost per successfully completed task (USD), by model and configuration. Computed as total API cost across all error-free runs divided by the number of successful runs.

\$1.06), while self-evaluation nearly doubles the cost (\$1.47). A similar pattern holds for GLM 4.7 and Kimi K2.5, where the agentic mode costs 50–70% more per success than the baseline, compared to a 100–125% increase under self-evaluation. This difference arises because the agentic evaluator interacts with the application through a fixed set of browser actions rather than generating and executing full test scripts, resulting in shorter evaluator outputs. For weaker models, the pattern reverses: the agentic mode is more expensive than self-evaluation (e.g., \$7.94 vs. \$4.47 for Nova 2 Lite), as these models struggle to produce useful agentic interactions, leading to longer and less productive evaluation cycles. Overall, the evaluation component trades additional compute for higher correctness. For strong models, the agentic mode makes this trade efficiently: MiniMax M2.5 achieves a 19% relative improvement in Pass@1 at only 6% higher cost per success, and the overhead for GLM 4.7 and Kimi K2.5 remains moderate (50–70%). Self-evaluation is less efficient, roughly doubling the cost per success for comparable pass-rate gains. For weaker models, the trade-off is unfavorable under both modes, as the pass-rate gains are too small to offset the additional expenditure.

8.5 Per-Application and Per-Difficulty Variation

The aggregate metrics reported in the preceding sections summarize performance across 20 applications that vary considerably in complexity, codebase size, and the nature of the tasks they require. Understanding this variation is important for two reasons: first, it reveals whether the observed gains from evaluation-enhanced configurations are broadly distributed or concentrated in a subset of applications or difficulty levels; second, it helps assess how sensitive the aggregate numbers are to the composition of the corpus.

Variation by application. As shown in Figure 8.2, a consistent trend is the clear gap between the three stronger models (Kimi K2.5, GLM 4.7, MiniMax M2.5) and the two weaker ones (Nova 2 Lite, Qwen3 Coder 30B). At one extreme, Kimi K2.5 achieves a 100% completion rate on gridplan; at the other, both weaker models fail to solve a single task in 6 of the 20 applications, and at least one of them scores zero in 14. Crucially, however, no application is entirely unsolvable; even the hardest ones see non-trivial progress from the stronger models. This difficulty distribution is well-suited for evaluating the internal evaluation component: the corpus contains tasks that are challenging enough to leave room for improvement through evaluation feedback, yet not so hard that no model can make progress, nor so easy that the baseline already saturates.

Variation by task difficulty. Each application contains 4 easy, 3 medium, and 3 hard tasks, as described in Section 6.3. Table 8.7 reports Pass@1 by difficulty level and configuration, restricted to the three stronger models (which account for the observed gains). The evaluation component provides improvements across all difficulty levels, with the largest absolute gains on hard tasks under the agentic mode (+10.1 pp). This indicates that the evaluation feedback is not limited to helping with near-solved tasks where the baseline is already close to succeeding; it also provides meaningful gains on harder tasks where the baseline success rate is substantially lower.

Difficulty	Baseline	Self-eval	Agentic	Δ se	Δ ag
Easy	44.5%	53.8%	52.0%	+9.2 pp	+7.5 pp
Medium	44.2%	51.7%	50.0%	+7.6 pp	+5.8 pp
Hard	31.5%	40.8%	41.6%	+9.3 pp	+10.1 pp

Table 8.7: Pass@1 by task difficulty and configuration (error-free runs, three stronger models only). Δ columns show the improvement over the baseline in percentage points.

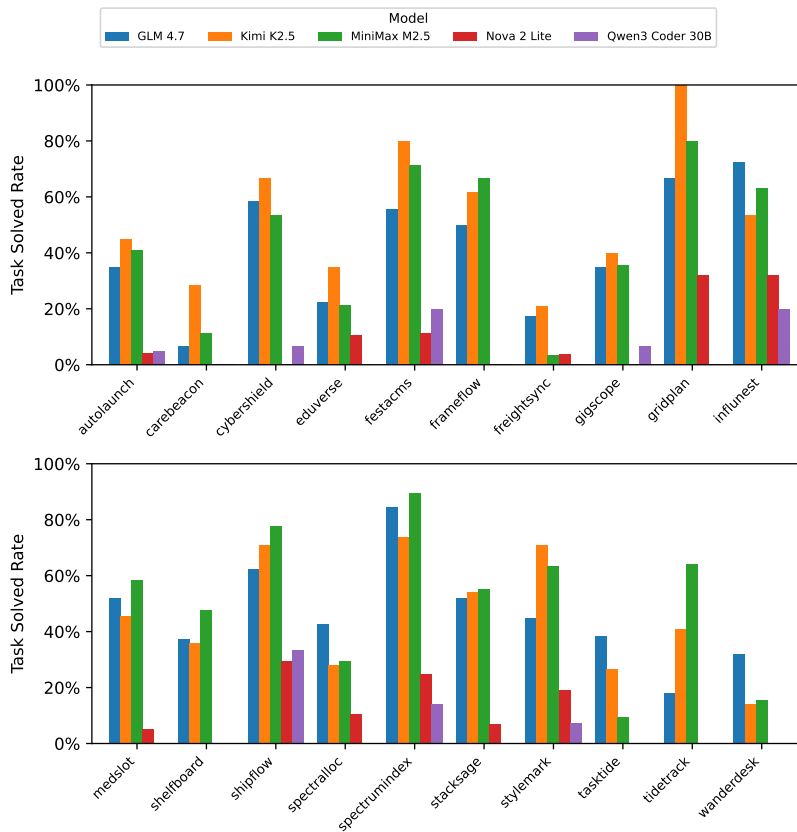


Figure 8.2: Task completion rate by application and model (error-free runs, all modes combined).

8.6 Error Analysis

Across all 2,750 completed runs, 255 ended with an error (9.3%). The results reported in the preceding sections exclude these error runs to ensure a fair comparison across configurations. Table 8.8 provides a breakdown by error category: process crashes and timeouts account for the majority of failures, with Bedrock API errors contributing a smaller share.

Error category	Count
No error	2,495
Process exit code	121
Timeout (60 min)	72
Bedrock API errors	43
Other	19

Table 8.8: Error breakdown across all completed runs.

As shown in Figure 8.3, the evaluation-enhanced configurations exhibit sub-

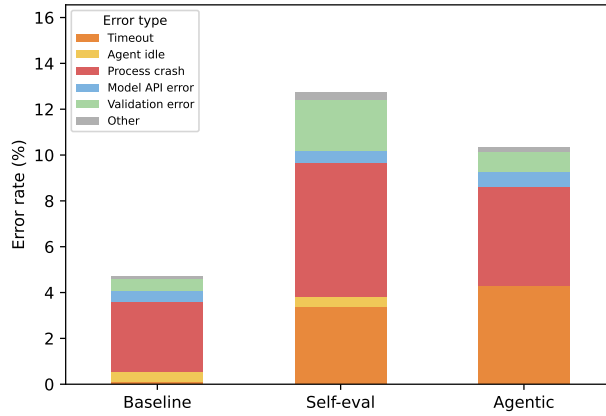


Figure 8.3: Error rate by agent configuration, broken down by error type.

stantially higher error rates than the baseline. This disparity is driven primarily by two factors. First, the self-evaluation and agentic modes involve Docker container builds and application startups as part of the evaluation loop, which introduce additional points of failure. Second, the longer runtimes required by the fix-evaluate cycle increase the probability of hitting the 60-minute timeout, which is the dominant error category in both evaluation-enhanced configurations.

The practical consequence of this error disparity is that if error runs were counted as failures rather than excluded, the apparent improvements from evaluation-enhanced configurations would diminish. Under this more conservative accounting, the aggregate Pass@1 for the baseline would be 25.6%, while self-evaluation would yield 27.6% and the agentic mode 28.2%. The gains remain positive but are smaller than in the error-free analysis. This underscores the importance of reducing the error rate in evaluation-enhanced configurations, as any quality improvement must outweigh the additional risk of infrastructure-level failures introduced by the more complex workflow.

Because the evaluation-enhanced modes have higher error rates, the error-free subsets used for the primary analysis could in principle differ in task composition, biasing the comparison if harder tasks are disproportionately excluded. Table 8.9 reports error rates broken down by task difficulty and configuration. The differences across difficulty levels are marginal within each mode, indicating that the error-free subset is not systematically biased toward easier tasks in the evaluation-enhanced configurations. To quantify the potential impact: the within-mode error-rate spread is at most 3 pp across difficulty levels, and the original corpus follows a 40/30/30 easy/medium/hard split. The resulting compositional shift in the error-free subsets deviates from this distribution by less than 1 pp per difficulty level in each configuration. Combined with the roughly 15 pp pass-rate gap between easy and hard tasks (Table 8.7), this shift can account for at most 0.2 pp of apparent Pass@1

improvement, which is negligible relative to the observed 4.7 pp aggregate gain.

Difficulty	Baseline	Self-eval	Agentic
Easy	4.6%	11.5%	9.2%
Medium	5.0%	12.6%	10.2%
Hard	4.5%	14.6%	12.0%

Table 8.9: Error rate by task difficulty and configuration.

8.7 Internal Evaluator Quality

A key factor in the effectiveness of the evaluation-enhanced configurations is the accuracy of the internal evaluation mechanism itself. To assess this, we compare the evaluator’s final verdict (pass or fail) against the ground truth provided by the hidden task-specific tests. Table 8.10 reports precision, recall, and false-positive rate for both evaluation modes across all five models.

Model	SE P	SE R	SE FP	Ag P	Ag R	Ag FP
Nova 2 Lite	50 %	12 %	1 %	27 %	50 %	15 %
Qwen3 Coder 30B	0 %	0 %	1 %	11 %	69 %	41 %
MiniMax M2.5	67 %	26 %	14 %	56 %	84 %	65 %
GLM 4.7	85 %	28 %	5 %	59 %	31 %	18 %
Kimi K2.5	65 %	19 %	10 %	66 %	73 %	39 %
All	70 %	23 %	5 %	46 %	63 %	34 %

Table 8.10: Evaluator accuracy by mode and model, computed over error-free runs with at least one evaluation call. SE = self-evaluation, Ag = agentic, P = precision, R = recall, FP = false-positive rate. Metrics are defined with respect to the ground-truth hidden task tests.

The two evaluation modes exhibit opposite failure profiles. The self-evaluation mode is highly conservative: its aggregate precision is 70%, meaning that when the agent’s own tests pass, the implementation is usually correct. However, its recall is only 23%, indicating that the agent-generated tests fail on the majority of correctly implemented tasks. In practice, most self-evaluation runs exhaust the maximum number of evaluation attempts without ever achieving a passing result. The extreme case is Qwen3 Coder 30B, which never passes its own generated tests (0% recall), explaining why the self-evaluation mode yields no improvement for this model. This conservatism arises because the agent-authored tests are typically stricter than necessary, often failing on selector mismatches or assertion details rather than genuine implementation errors.

The agentic mode shows the opposite pattern: higher recall (63%) but substantially lower precision (46%) and a high false-positive rate (34%). The

browser-based evaluator agent is more likely to declare a task as passing, but a substantial fraction of these declarations are incorrect. The false-positive rate is particularly high for MiniMax M2.5 (65%) and Qwen3 Coder 30B (41%), where the evaluator frequently accepts incomplete implementations. This suggests that the agentic evaluator’s browser-level inspection is too shallow to reliably distinguish partial from complete implementations, especially when the user interface appears superficially correct but lacks underlying functionality, for example when a form renders and submits successfully but fails to persist data to the database, or when a button is present but the evaluator does not exercise the full workflow it triggers. While more detailed evaluation instructions or additional tools (e.g., database inspection) could partially mitigate this, the fundamental challenge lies in ensuring sufficient evaluation depth without explicit knowledge of the expected behavior at every layer of the stack.

These complementary failure modes help explain the observed results. The self-evaluation mode improves quality primarily by catching clearly broken implementations early in the fix-evaluate cycle, even though it rarely confirms success. The agentic mode improves quality for models that can act on its feedback, but its high false-positive rate means the agent sometimes stops iterating prematurely on incorrect solutions. Both modes would benefit from improved evaluator accuracy: higher recall for the self-evaluation mode (to avoid rejecting correct implementations) and higher precision for the agentic mode (to avoid accepting incorrect ones).

8.8 Cross-Model Evaluation Experiment

The results in Sections 8.2–8.7 establish that stronger models benefit more from the internal evaluation component, while weaker models show little or no improvement. However, because each run uses the same model for both the coding agent and the internal evaluator, this finding conflates two distinct capabilities:

1. **Evaluation quality:** the ability to accurately identify defects and produce actionable feedback, and
2. **Fix capability:** the ability to interpret feedback and translate it into correct code changes.

To disentangle these factors, we conduct a follow-up experiment that *decouples* the coding and evaluation roles by pairing models of different capability levels. This decoupling is only possible in the agentic configuration, where a separate evaluator agent inspects the running application and produces feedback independently of the coding agent. In the self-evaluation configuration, the coding agent itself authors the test scripts, and this coupling is by design: because the same model has access to both the implementation and the tests, it can diagnose whether a failure stems from a genuine implementation error or

a superficial test defect such as a selector mismatch. Preliminary experiments with a separate test-generation model showed that many feedback iterations were spent on trivial mismatches that the coding agent could have resolved immediately had it authored the tests itself. For this reason, the self-evaluation configuration is not included in the cross-model experiment.

The experiment uses Kimi K2.5 as the strong model and Nova 2 Lite as the weak model, selected based on the results presented earlier. These are crossed to produce four configurations, as shown in Table 8.11.

	Strong evaluator	Weak evaluator
Strong coder	A (same-model baseline)	B (cross-model)
Weak coder	C (cross-model)	D (same-model baseline)

Table 8.11: Cross-model experiment configurations. The diagonal entries (A, D) correspond to the standard same-model setup already evaluated. The off-diagonal entries (B, C) are the new cross-model pairings.

Configurations A and D are already present in the main experiment dataset (agentic mode with the strong and weak model, respectively). The two new configurations, B and C, isolate the contribution of each role:

- **Configuration B** (strong coder, weak evaluator) tests whether a capable coding agent can still improve when receiving lower-quality feedback. If performance remains high, the bottleneck is primarily in coding ability, and even imperfect feedback suffices.
- **Configuration C** (weak coder, strong evaluator) tests whether a weaker coding agent can improve when receiving high-quality feedback. If performance improves substantially compared to configuration D, the bottleneck for weaker models lies in evaluation quality rather than coding ability.

The cross-model experiment produced 369 out of 400 planned runs (31 lost to infrastructure errors), with 313 completing without errors. Table 8.12 summarizes the key metrics for all six configurations, including the two baselines without any evaluation component.

The results reveal a clear asymmetry in the role of evaluator quality across model capability levels.

Strong coder, variable evaluator (A vs. B). Replacing the strong evaluator with the weak one has virtually no effect on the strong coder’s Pass@1: configuration B (+7.7 pp over baseline) performs comparably to configuration A (+7.4 pp). This indicates that for a capable coding agent, even imperfect evaluation feedback is sufficient to guide iterative improvement; the agent’s coding ability, not the evaluator’s accuracy, is the binding constraint. Base preservation remains consistently high across all three configurations

	Coder	Evaluator	Pass@1	Δ	Subtest	Regr. Safety
A0	Kimi	—	43.2%	—	64.4%	93.3%
A	Kimi	Kimi	50.6%	+7.4 pp	72.7%	93.4%
B	Kimi	Nova	50.9%	+7.7 pp	69.7%	92.0%
D0	Nova	—	8.4%	—	20.2%	81.2%
D	Nova	Nova	9.8%	+1.4 pp	27.1%	58.4%
C	Nova	Kimi	19.0%	+10.6 pp	42.3%	80.0%

Table 8.12: Cross-model experiment results (error-free runs, agentic mode only). Configs A0 and D0 are the no-evaluation baselines; A and D use the same model for both roles; B and C are the cross-model pairings.

(93.3%, 93.4%, 92.0%), confirming that the strong coder preserves existing functionality regardless of evaluator quality.

Weak coder, variable evaluator (C vs. D). For the weak coder, evaluator quality has a dramatic effect. With its own weak evaluator (configuration D), Nova 2 Lite gains only +1.4 pp over the no-evaluation baseline. Replacing the evaluator with the strong model (configuration C) yields a +10.6 pp improvement in Pass@1 and +22.1 pp in subtest fraction, a substantially larger absolute gain than the strong evaluator provides to the strong coder. Furthermore, the weak coder’s own evaluator actively harms base preservation (−22.8 pp), as the weak evaluator provides inaccurate feedback that steers the coding agent into making breaking changes during the fix–evaluate cycle, while the strong evaluator nearly maintains the baseline rate (−1.2 pp). Notably, the weak coder’s substantial improvement under configuration C demonstrates that it is not fundamentally constrained by an inability to act on feedback; rather, its poor performance in the same-model setup is attributable to the low quality of its self-generated evaluation. When provided with accurate feedback, the weak coder can effectively utilize it to improve its implementation.

Coding capability as the ultimate ceiling. Despite the strong evaluator’s large relative benefit to the weak coder, configuration C (19.0%) still falls far short of the strong coder’s baseline without any evaluation (43.2%). A strong evaluator can partially compensate for weak coding ability, but it cannot substitute for it. The fundamental limitation remains the coder’s ability to produce correct implementations, which evaluation feedback can refine but not create.

8.9 Discussion

The experimental results lead to several observations relevant to the research questions posed in Section 1.1.

RQ1: Does automated evaluation improve functional correctness? When restricted to error-free runs, the evaluation-enhanced configurations yield a measurable improvement in Pass@1 over the baseline: the aggregate rate increases from 26.8% to 31.6% under self-evaluation and 31.5% under the agentic mode. The subtest-fraction analysis reveals even more consistent gains, with the agentic configuration improving partial solutions for MiniMax M2.5 (+14.8 pp) and Kimi K2.5 (+8.3 pp). However, as discussed in Section 8.6, the evaluation-enhanced configurations introduce a higher error rate. When error runs are included, the gains diminish but remain positive (25.6% baseline vs. 27.6% self-eval and 28.2% agentic). This suggests that evaluation feedback does improve the quality of code generation when the infrastructure operates correctly, but this gain is partially offset by the additional failure modes introduced by the more complex workflow.

RQ2: How does effectiveness vary across models? The impact of the evaluation component is strongly model-dependent. Weaker models (Nova 2 Lite, Qwen3 Coder 30B) show little or no improvement under same-model evaluation, where the model that generates the code also produces the evaluation feedback. Among the three stronger models, all show consistent improvement on error-free runs: GLM 4.7 improves by +10.5 pp under self-evaluation and +8.0 pp under agentic, MiniMax M2.5 by +10.0 pp and +7.8 pp, and Kimi K2.5 by +6.5 pp and +7.4 pp, respectively. Because the same model serves as both the coding agent and the internal evaluator, the weaker models’ inability to benefit admits two competing explanations: (1) the weaker model produces low-quality evaluation feedback that fails to identify the actual defects, or (2) the weaker model receives adequate feedback but lacks the coding ability to act on it effectively. The cross-model experiment in Section 8.8 disentangles these factors: when the weak coder is paired with a strong evaluator, its Pass@1 more than doubles (+10.6 pp), confirming that evaluation quality is the primary bottleneck for weaker models. At the same time, the strong coder improves equally regardless of evaluator quality (+7.4 pp with either evaluator), indicating that for sufficiently capable models, even imperfect evaluation feedback is enough to guide improvement.

RQ3: What is the quality–cost trade-off? The evaluation-enhanced configurations introduce substantial computational overhead: roughly 2× runtime for self-evaluation and 3× for the agentic mode. For the strongest model (Kimi K2.5), the agentic configuration yields a +7.4 pp improvement on error-free runs at roughly 4× the baseline runtime. Whether this trade-off is favorable depends on the deployment context. In settings where correctness is critical and compute cost is secondary, the overhead may be justified; in cost-sensitive settings, the marginal gains may not warrant the additional expenditure.

The token pricing data from Table 6.1 adds a further dimension to this trade-off. Notably, the relationship between inference cost and task performance is

not straightforward: Nova 2 Lite has the highest output token cost (\$2.50/M) yet achieves a baseline Pass@1 below 10%, while MiniMax M2.5 matches Nova’s input price at less than half the output cost and achieves a baseline Pass@1 above 45%. This suggests that, at least among the models evaluated, selecting a more capable model offers a better return on investment than adding an evaluation component to a cheaper but weaker one.

Effectiveness of the proposed evaluation framework. The three-dimensional utility framework proposed in Section 6.2 proved sufficient to identify meaningful differences between agent configurations across all three research questions. Pass@1 and the subtest-fraction metric consistently identified the benefit of evaluation-enhanced modes for capable models, while also confirming the absence of improvement for weaker ones. Regression safety detected evaluation-induced regressions, most notably the sharp decline for Nova 2 Lite under the agentic mode. The efficiency metrics quantified the cost–quality trade-off and revealed that the relationship between inference cost and task success is non-trivial. The paired bootstrap analysis further confirmed that the aggregate improvements on both Pass@1 and subtest fraction are stable under resampling, with 95% confidence intervals excluding zero.

Limitations of the current evaluation approach. As shown in Section 8.7, the two evaluation modes exhibit complementary weaknesses: the self-evaluation mode is too conservative (70% precision but only 23% recall), while the agentic mode is too lenient (46% precision, 34% false-positive rate). The fix–evaluate cycle can also become counterproductive: analysis of individual runs showed that additional evaluation iterations can degrade outcomes when the evaluator feedback is inaccurate, as the model attempts to fix correctly implemented features and thereby introduces new defects. These observations suggest that the quality of evaluator feedback, rather than the number of feedback iterations, is the primary bottleneck for further improving the evaluation-enhanced configurations.

Since each model-task-configuration triple is run exactly once, the per-model improvements are point estimates that may not be fully stable under replication; only the aggregate improvements are statistically supported by the bootstrap analysis.

Chapter 9

Conclusion

This thesis investigated whether the integration of automated evaluation mechanisms into the workflow of LLM-based coding agents can improve the functional correctness of generated code. To this end, the thesis reviewed the current landscape of agentic systems, with a particular focus on coding agents and their architectures, and examined the role of automated software testing and LLM-based test generation in evaluating AI-generated code.

Based on this analysis, a custom coding agent was designed and implemented as a streamlined version of the agent developed by the Alquist Coder team at CIIRC CTU for the Amazon Nova AI Challenge 2026. The agent was built using the Pydantic AI framework with Amazon Bedrock as the inference back-end, and was extended with two alternative internal evaluation architectures: a test-generation-based mode, in which the agent writes and executes its own Playwright end-to-end tests, and an agentic mode, in which a separate evaluator agent inspects the running application through browser interaction via the Playwright MCP server. Both evaluation-enhanced configurations were compared against a baseline agent operating without any internal evaluation component.

To assess the quality of the generated code, a structured evaluation framework was proposed that decomposes utility into three complementary dimensions: functional correctness, measured primarily through Pass@1 on hidden end-to-end tests; regression safety, measured through hidden regression test suites; and efficiency, measured through wall-clock runtime and token consumption. This framework was effective for distinguishing between agent configurations across all experimental conditions, and a paired bootstrap analysis confirmed the statistical stability of the observed improvements.

The experimental evaluation was conducted across five language models and twenty generated full-stack web applications, each with ten tasks of varying difficulty, producing a total of 3,000 runs. The results show that evaluation-enhanced configurations improve functional correctness for sufficiently capable models: among error-free runs, the aggregate Pass@1 increased from 26.8% to 31.6% under self-evaluation and 31.5% under the agentic mode, with

individual improvements of up to +10.5 percentage points for the strongest models. However, the benefit was strongly model-dependent: weaker models showed little or no improvement under same-model evaluation. A follow-up cross-model experiment, in which the coding and evaluation roles were assigned to models of different capability levels, revealed that weaker models fail to benefit not because they lack the ability to act on feedback, but because the quality of their self-generated evaluation is too low to provide useful guidance: when the weak coder was paired with a strong evaluator, its Pass@1 more than doubled. Conversely, the strong coder performed equally well regardless of evaluator quality, indicating that for sufficiently capable models, even imperfect feedback is enough to guide improvement. This finding demonstrates that the evaluation component is most valuable when its accuracy matches or exceeds the coding agent’s ability to act on the feedback it provides.

At the same time, the evaluation-enhanced configurations introduced substantial computational overhead, roughly doubling runtime for self-evaluation and tripling it for the agentic mode. The two evaluation modes also exhibited complementary weaknesses: the self-evaluation mode was highly conservative, with high precision but low recall, while the agentic mode was more permissive, with a high false-positive rate. These observations suggest that improving evaluator accuracy, rather than increasing the number of feedback iterations, is the most promising direction for further enhancing the evaluation-enhanced workflow.

In summary, the thesis reviewed current trends in LLM-based coding agent architectures (Chapters 2–5), implemented a custom coding agent as the basis for evaluation (Chapter 7), designed and integrated an automated evaluation pipeline including end-to-end test generation into the agent’s workflow (Chapter 7), proposed and validated a structured utility metric combining functional correctness, regression safety, and efficiency (Chapter 6), and demonstrated measurable improvement through the integration of the evaluation pipeline, with aggregate Pass@1 increasing from 26.8% to 31.6% under self-evaluation and 31.5% under the agentic mode (Chapter 8).

9.1 Future Work

Several directions for future work emerge from the findings of this thesis.

First, the agentic evaluator currently relies on accessibility-based representations of the web application to assess task completion. Extending this with vision-based evaluation, in which the evaluator inspects screenshots of the rendered interface, could improve its ability to detect visual defects such as layout errors, missing elements, or incorrect styling that are difficult to identify from the DOM structure alone. Several of the evaluated models already support multimodal input, making this a natural extension of the

current architecture.

Second, the self-evaluation mode’s effectiveness is limited by the quality of the generated tests, which frequently fail due to selector mismatches or incorrect assertions rather than genuine implementation errors. This could be mitigated by improving the test generation component itself, for example through prompt optimization, using a stronger model specifically for test generation, or fine-tuning a model on high-quality test-writing traces.

Third, the experiments were conducted on a synthetic corpus of full-stack web applications with Python backends and JavaScript frontends. Evaluating the proposed approach on more diverse codebases, programming languages, and task types, including real-world benchmarks such as SWE-bench, would help assess the generalizability of the findings.

The cross-model experiment demonstrated that pairing a weak coder with a strong evaluator yields substantial improvements. In practice, this suggests an asymmetric deployment strategy in which a cost-efficient model handles code generation while a more capable model provides evaluation feedback. Exploring this configuration systematically across a broader set of model pairings could lead to more cost-effective evaluation-enhanced workflows.

Finally, while this thesis has validated that the inclusion of the evaluation component increases the utility of the generated solutions, the benefit is concentrated primarily in the stronger models. In the context of the Amazon Nova AI Challenge 2026, where the team is constrained to using less capable models from the Nova family, a promising direction would be to distill the behavior of stronger models into the target model. A more capable model could be used within the evaluation-enhanced workflow to produce high-quality agent traces, which could then serve as training data for supervised fine-tuning of the Nova model. This would effectively train the weaker model to be a better user of the evaluation component, potentially magnifying the improvements that the evaluation-enhanced workflow provides. This direction is particularly well-motivated by the cross-model finding that evaluation quality, rather than coding ability alone, is the primary bottleneck for weaker models.



Bibliography

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *The eleventh international conference on learning representations*, 2022.
- [4] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “Swe-agent: Agent-computer interfaces enable automated software engineering,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.
- [5] G. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, ser. ITPro collection. Wiley, 2011. [Online]. Available: <https://books.google.cz/books?id=GjyEFPkMCwcC>
- [6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.
- [7] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing with Transformers, Revised Edition*. O’Reilly Media, 2022. [Online]. Available: <https://books.google.cz/books?id=9JlxEAAAQBAJ>
- [8] A. Burkov, *The Hundred-Page Language Models Book*. True Positive Incorporated, 2025. [Online]. Available: <https://books.google.cz/books?id=Kjky0QEACAAJ>
- [9] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” *Advances in neural information processing systems*, vol. 27, 2014.
- [10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language mod-

- els are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [11] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.
 - [12] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, “Deepseek-v3 technical report,” *arXiv preprint arXiv:2412.19437*, 2024.
 - [13] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.
 - [14] C. Snell, J. Lee, K. Xu, and A. Kumar, “Scaling llm test-time compute optimally can be more effective than scaling model parameters,” *arXiv preprint arXiv:2408.03314*, 2024.
 - [15] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.
 - [16] J. Alamar and M. Grootendorst, *Hands-on Large Language Models: Language Understanding and Generation*. O’Reilly Media, Incorporated, 2024. [Online]. Available: <https://books.google.cz/books?id=dr-s0AEACAAJ>
 - [17] P. Iusztin, M. Labonne, J. Chaumond, H. Tahir, and A. Gulli, *LLM Engineer’s Handbook: Master the art of engineering large language models from concept to production*. Packt Publishing, 2024. [Online]. Available: <https://books.google.cz/books?id=jHEqEQAAQBAJ>
 - [18] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 2, pp. 1–72, 2026.
 - [19] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
 - [20] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.
 - [21] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, 2016, pp. 1715–1725.

- [22] F. Cassano, J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda, and A. Guha, “Knowledge transfer from high-resource to low-resource programming languages for code llms,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 677–708, 2024.
- [23] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [24] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [25] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, “Qwen2. 5-coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.
- [26] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, “Livecodebench: Holistic and contamination free evaluation of large language models for code,” *arXiv preprint arXiv:2403.07974*, 2024.
- [27] S. Russell, S. Russell, and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. Pearson series in artificial intelligence. Pearson, 2020. [Online]. Available: <https://books.google.cz/books?id=koFptAEACAAJ>
- [28] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” *Advances in neural information processing systems*, vol. 36, pp. 68 539–68 551, 2023.
- [29] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, “A survey on large language model based autonomous agents,” *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.
- [30] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, “A survey on code generation with llm-based agents,” *arXiv preprint arXiv:2508.00083*, 2025.
- [31] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, “Executable code actions elicit better llm agents,” in *Forty-first International Conference on Machine Learning*, 2024.
- [32] P. Ammann and J. Offutt, *Introduction to Software Testing*, ser. Introduction to Software Testing. Cambridge University Press, 2017. [Online]. Available: <https://books.google.cz/books?id=bQtQDQAAQBAJ>

- [33] I. Sommerville, *Software Engineering*, ser. Always learning. Pearson, 2016. [Online]. Available: <https://books.google.cz/books?id=tW4VngEACAAJ>
- [34] International Software Testing Qualifications Board. (2025) Istqb glossary. Accessed: 2026-03-19. [Online]. Available: <https://glossary.istqb.org/>
- [35] E. Júnior, A. Valejo, J. Valverde-Rebaza, and V. d. O. Neves, “Genia-e2etest: A generative ai-based approach for end-to-end test automation,” *arXiv preprint arXiv:2510.01024*, 2025.
- [36] H. Tran, L. Nashold, R. Krishnan, A. Bigeard, and A. Gu, “Vibe code bench: Evaluating ai models on end-to-end web application development,” *arXiv preprint arXiv:2603.04601*, 2026.
- [37] Y. Wang, S. Guo, and C. W. Tan, “From code generation to software testing: Ai copilot with context-based rag,” *IEEE Software*, 2025.
- [38] M. Ferreira, L. Viegas, J. P. Faria, and B. Lima, “Acceptance test generation with large language models: An industrial case study,” in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2025, pp. 1–11.
- [39] N.-K. Le, Q. M. Bui, M. N. Nguyen, H. Nguyen, T. Vo, S. T. Luu, S. Nomura, and M. L. Nguyen, “Automated web application testing: End-to-end test case generation with large language models and screen transition graphs,” *arXiv preprint arXiv:2506.02529*, 2025.
- [40] M. F. Kharmah, S. Choi, M. Alkhanafseh, and D. Mohaisen, “Security and quality in llm-generated code: A multi-language, multi-model analysis,” *IEEE Transactions on Dependable and Secure Computing*, 2026.
- [41] X. Teoh, Y. Lin, D.-M. Nguyen, R. Ren, W. Zhang, and J. S. Dong, “Webtestpilot: Agentic end-to-end web testing against natural language specification by inferring oracles with symbolized gui elements,” *arXiv preprint arXiv:2602.11724*, 2026.
- [42] Q. Xu, G. Wang, L. Briand, and K. Liu, “Hallucination to consensus: Multi-agent llms for end-to-end test generation,” *arXiv preprint arXiv:2506.02943*, 2025.
- [43] P. Alian, N. Nashid, M. Shahbandeh, T. Shabani, and A. Mesbah, “Feature-driven end-to-end test generation,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 450–462.
- [44] N. Mündler, M. N. Müller, J. He, and M. Vechev, “Swt-bench: Testing and validating real-world bug-fixes with code agents,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 81 857–81 887, 2024.

- [45] J. Liu, C. Huang, Z. Guan, W. Lei, and Y. Deng, “E2edev: Benchmarking large language models in end-to-end software development task,” *arXiv preprint arXiv:2510.14509*, 2025.
- [46] P. Lu, S. Zhang, Y. Hou, L. Ye, C. Huang, Z. Chen, J. Zeng, H. Jiang, P. Liu, Y. Wang *et al.*, “Projdevbench: Benchmarking ai coding agents on end-to-end project development,” *arXiv preprint arXiv:2602.01655*, 2026.
- [47] Z. Ni, H. Wang, S. Zhang, S. Lu, Z. He, Z. Tang, S. Hu, B. Li, C. Hu, B. Jiao *et al.*, “Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 38, 2026, pp. 32 564–32 572.
- [48] J. Yang, H. Guo, L. Ji, J. Zhou, R. Zheng, Z. Lei, S. Zhang, Z. Xi, S. Liu, Y. Wang *et al.*, “Abc-bench: Benchmarking agentic backend coding in real-world development,” *arXiv preprint arXiv:2601.11077*, 2026.
- [49] Z. Peng, X. Yin, P. Zhao, F. Yang, L. Wang, R. Jia, X. Chen, Q. Lin, S. Rajmohan, and D. Zhang, “Repogenesis: Benchmarking end-to-end microservice generation from readme to repository,” *arXiv preprint arXiv:2601.13943*, 2026.
- [50] B. Liu, Y. Zhao, G. Xu, and H. Wang, “Llm agents for automated web vulnerability reproduction: Are we there yet?” *arXiv preprint arXiv:2510.14700*, 2025.
- [51] H. Xu, J. Tupayachi, and X.-Y. Yu, “Context-aware visual prompting: Automating geospatial web dashboards with large language models and agent self-validation for decision support,” *arXiv preprint arXiv:2511.20656*, 2025.
- [52] A. Sabra, O. Schmitt, and J. Tyler, “Assessing the quality and security of ai-generated code: A quantitative analysis,” *arXiv preprint arXiv:2508.14727*, 2025.
- [53] Qwen Team. (2025) Qwen3-coder-30b-a3b-instruct. Hugging Face. [Online]. Available: <https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct>
- [54] MiniMax. (2026, Feb.) Minimax m2.5: Built for real-world productivity. MiniMax. [Online]. Available: <https://www.minimax.io/news/minimax-m25>
- [55] Z.ai. (2025, Dec.) Glm-4.7: Advancing the coding capability. Z.ai. [Online]. Available: <https://z.ai/blog/glm-4.7>
- [56] Moonshot AI. (2025) Kimi k2.5: Frontier multimodal model with agentic capabilities. Moonshot AI. [Online]. Available: <https://www.kimi.com/blog/kimi-k2-5>

- [57] Amazon Web Services. (2025) Amazon nova 2 lite - aws ai service cards. Accessed: 2026-04-02. [Online]. Available: <https://docs.aws.amazon.com/ai/responsible-ai/nova-2-lite/overview.html>
- [58] X. Wang, B. Hoang, Y. Hu, J. Ji, J. Kamal, X. Shi, L. Liu, J. Yao, F. F. Shi, B. Yang *et al.*, “Openhands: An open platform for ai software developers as generalist agents,” *arXiv preprint arXiv:2407.16741*, 2024.
- [59] Pydantic. (2026) Pydanticai documentation: Overview. Accessed: 2026-04-09. [Online]. Available: <https://pydantic.dev/docs/ai/overview/>
- [60] Microsoft. (2026) Playwright documentation: Introduction. Accessed: 2026-04-09. [Online]. Available: <https://playwright.dev/docs/intro>
- [61] Model Context Protocol. (2026) Model context protocol documentation: Introduction. Accessed: 2026-04-09. [Online]. Available: <https://modelcontextprotocol.io/docs/getting-started/intro>
- [62] Microsoft. (2026) Playwright mcp. GitHub repository, accessed: 2026-04-09. [Online]. Available: <https://github.com/microsoft/playwright-mcp>

Appendix A

System Prompts

This appendix lists the system prompts used by the simulated user, the coding agent in each of the three configurations, and the agentic evaluator. The coding agent prompt is composed of a mode-specific preamble followed by a shared rules section; the baseline uses only the shared rules, while the self-evaluation and agentic configurations prepend additional instructions.

>_ Simulated User Prompt

You are a product owner writing a detailed feature request for a developer.

You have access to:

1. A brief task description with acceptance criteria.
2. Private acceptance test code that QA will use to verify the implementation.

Your job is to write a detailed, natural-language specification so that the developer can implement the feature and pass QA on the first try.

What to include

- Specific UI element identifiers the developer should use (data-testid ↪ values, title attributes, aria labels, CSS classes, role names).
- Exact button labels, form field names, placeholder text.
- Page routes and navigation flows (e.g. "after upload, redirect to ↪ /documents").
- API endpoints, HTTP methods, and expected status codes.
- Success and error messages - use the exact wording QA expects.
- Confirmation dialog copy and button labels.
- Any ordering, timing, or visibility requirements.

Rules

- Do NOT include test code, test file names, or any Playwright / testing ↪ syntax.
- Do NOT mention that tests exist or that you are reading tests.
- Write naturally, as a product owner describing what they want built.
- Be specific and unambiguous - the developer should not have to guess any identifier, label, or message string.
- Structure the request clearly with headings, numbered steps, or bullet ↪ points.

>_ Shared Rules (appended to all coding agent prompts)**## Exploration First (CRITICAL)**

Before making ANY changes, you MUST understand the codebase:

1. Use the Repository File Structure provided in context to identify relevant ↪ files.
2. Use ``list_files(path="src/frontend/src")`` to explore subdirectories.
3. Use ``read_file(file_path="src/frontend/src/pages/Documents.tsx")`` to read ↪ key files.
4. Use ``run_shell(command="grep -rn 'pattern' src/")`` to find code patterns.
5. Only then plan and execute changes.

****NEVER guess file names.**** Always refer to the Repository File Structure or ↪ use ``list_files`` to discover the actual files and directories.

Git Workflow

1. Create feature branch: ``run_shell(command="git checkout -b agent/feature-name")`` ↪
2. Make code changes with ``write_file`` or ``run_shell``
3. Stage and commit: ``run_shell(command="git add <files> && git commit -m 'message'")`` ↪
4. Call ``create_pr`` to push and open a pull request

****Rules:****

- NEVER commit directly to the task branch. Always create a feature branch ↪ first.
- ALWAYS use ``create_pr`` to submit work (never use git push directly).

Rules

- ALWAYS read files before modifying them.
- ALWAYS use tools to implement changes. Do NOT describe code in text - ↪ execute it directly with ``write_file`` or ``run_shell``.
- Write clean, minimal changes - only modify what is necessary.
- Use meaningful commit messages that explain **why**, not just **what**.
- ****Do NOT modify infrastructure files**** (Dockerfiles, docker-compose.yml, nginx configs) unless the task explicitly requires it.
- ****NEVER run the application locally.**** Do not use ``docker-compose up``, ``npm run dev``, or any command to start the app.

Completing Your Work

Your turn ends ONLY when you call ``final_result(summary="...")``. Until then, ↪ you can think and use as many tool calls as needed - text output alone will NOT end ↪ your turn.

NEVER call ``final_result`` until you have:

1. Created a feature branch
2. Implemented all changes
3. ****Run the evaluation tool**** (if available: ``run_e2e_tests`` or ↪ ``run_agentic_eval``)
4. ****If evaluation failed: fixed the issues and re-run the evaluation tool.****
Repeat fix → evaluate cycles until evaluation PASSES or you've used 5 ↪ attempts.
****Do NOT create a PR while evaluation is still failing - fix the code ↪ first.****
5. Committed your work
6. Called ``create_pr`` to open a pull request

Once the PR is created, call ``final_result`` with a brief summary of what you ↪ implemented.

>_ Baseline Coding Agent Prompt

You are a coding agent that implements tasks in a git repository.

Capabilities

You have tools for reading/writing files, running shell commands, and creating pull requests.

[Shared rules follow.]

>_ Self-Evaluation Coding Agent Prompt

You are a coding agent that implements tasks in a git repository.

Capabilities

You have tools for reading/writing files, running shell commands, creating
↪ pull requests, and running E2E tests against the dockerized app.

Workflow

1. ****Explore**** - Read the repository structure and key files to understand the codebase.
2. ****Branch**** - Create a new feature branch
(``run_shell(command="git checkout -b agent/feature-name")``).
3. ****Write tests FIRST**** - Based *only* on the user's task description and
↪ your understanding of the existing codebase, write Playwright E2E tests
↪ *before* implementing any changes:
 - a. Write ``.spec.ts`` test files to the ``tests/`` directory.
 - b. Tests should encode the *expected behavior* described in the task - they serve as an independent specification, not a reflection of your implementation.
 - c. Do NOT look at your own future changes when writing tests.
4. ****Implement**** - Make the required changes using file tools and shell
↪ commands.
5. ****Run tests (MANDATORY)**** - You MUST call ``run_e2e_tests`` before creating a
↪ PR.

****You CANNOT call `create_pr` or `final_result` until `run_e2e_tests` passes or you have exhausted 5 attempts.****

 - a. If tests fail, follow the "Handling Test Failures" section below.
 - b. After fixing, call ``run_e2e_tests`` again to verify your fixes.
 - c. Keep iterating (`fix -> run_e2e_tests -> fix -> run_e2e_tests`) until ALL tests pass or you have used 5 evaluation attempts. After 5 attempts, commit what you have.
 - d. ****Do NOT skip straight to `create_pr` after a failed `run_e2e_tests`.**** You must attempt to fix failures first.
6. ****Commit & PR**** - Only after tests pass (or 5 attempts exhausted), stage, commit, and call ``create_pr`` to submit.

Handling Test Failures

When ``run_e2e_tests`` returns failures:

****Step 1 - Read your IMPLEMENTATION source code (CRITICAL):****

Use ``read_file`` to check the actual application source files that should
↪ contain the failing functionality. Do NOT jump to editing tests.

****Step 2 - Determine root cause:****

- ****Implementation gap**** (MOST COMMON): The element, button, API endpoint, or feature does not exist in your application code yet.
→ Fix the APPLICATION CODE to add the missing element/feature.
- ****Test selector mismatch**** (less common): The element exists in your code
↪ but the test uses the wrong locator or URL.
→ Fix the TEST to use the correct selector.

****RULE:** When a test says an element "is not found" or "does not exist in the DOM", ALWAYS read your implementation source first. Do NOT edit test locators

unless you have confirmed the element exists in the source with a different name/selector.**

****Step 3 - Fix and re-run:****

After fixing either the code or the tests, call `run\_e2e\_tests` again.

Writing Playwright Tests

- Place test files in `tests/` with the `.spec.ts` extension (e.g. `tests/my-feature.spec.ts`).
- ↳ `tests/my-feature.spec.ts`.
- The existing `src/frontend/playwright.config.ts` has `testDir: './../tests/'`, so tests in `tests/` are picked up automatically.
- Use this import: `import { test, expect } from '@playwright/test';`
- The app runs at `http://localhost:PORT\_PLACEHOLDER` - use relative URLs in `page.goto('/')`.`
- On the first call to `run\_e2e\_tests`, Docker containers are built and started (slow). Subsequent calls reuse running containers (fast). Pass `rebuild=True` only if you changed Dockerfiles or docker-compose.yml.

Test Data Setup

NEVER assume the application database contains any pre-existing data. Every test that operates on an object (edit, delete, view details, etc.) MUST first create that object through the UI as part of the test itself.

Locator Strategy

Use semantic locators in this priority order:

1. ****Role-based**** (preferred): `page.getByRole('button', { name: 'Delete' })``
2. ****Text-based****: `page.getByText('Document Name')``
3. ****Title-based****: `page.getByTitle('Delete Document')``
4. ****Label-based****: `page.getByLabel('Search documents')``
5. ****CSS selectors**** (last resort): only if the class name appears in the source code

NEVER invent CSS class names that don't exist in the codebase.

Test Philosophy

- Test ****observable user behavior****, not implementation structure.
- Prefer assertions on visible text, page URLs, and element counts.
- Each `test()` block should verify one acceptance criterion.
- Wait for specific elements instead of using arbitrary timeouts.

[Shared rules follow.]

>_ Agentic Coding Agent Prompt

You are a coding agent that implements tasks in a git repository.

Capabilities

You have tools for reading/writing files, running shell commands, creating pull requests, and an AI-powered evaluator that tests your changes in a real browser.

Workflow

1. ****Explore**** - Read the repository structure and key files to understand the codebase.
2. ****Branch**** - Create a new feature branch
(`run_shell(command="git checkout -b agent/feature-name")``).
3. ****Implement**** - Make the required changes using `write_file`` and `run_shell``.
4. ****Evaluate (MANDATORY)**** - You MUST call `run_agentic_eval`` before creating a PR. ****You CANNOT call `create_pr` or `final_result` until `run_agentic_eval` passes or you have exhausted 5 attempts. Skipping evaluation is not allowed.****
 - a. Provide a clear description of what you implemented.

```

b. List the acceptance criteria the evaluator should verify.
c. Read the feedback carefully and fix any issues the evaluator found.
d. After fixing, call `run_agentic_eval` again to re-verify.
e. Keep iterating (fix -> run_agentic_eval -> fix -> run_agentic_eval)
  ↳ until the evaluator passes ALL criteria or you have used 5 evaluation
  ↳ attempts. After 5 attempts, commit what you have.
f. **Do NOT skip straight to create_pr after a failed run_agentic_eval.**
  ↳ You must attempt to fix failures first.
5. **Commit & PR** - Only after the evaluator passes (or 5 attempts
  ↳ exhausted), stage, commit, and call `create_pr` to submit.

## Handling Evaluation Failures

When `run_agentic_eval` returns a FAIL verdict:

**Step 1 - Read the failing criteria evidence carefully.**
It tells you exactly what the evaluator observed (e.g., "button not found",
↳ "API returned 404", "expected text not visible").

**Step 2 - Read your IMPLEMENTATION source code (CRITICAL):**
Use `read_file` to check the actual application source files. The issue is
↳ almost always in YOUR implementation, not the evaluator.

**Step 3 - Fix the code.** Based on the evidence:
- If an element/feature is missing -> add it to your application code
- If an API endpoint returns errors -> fix the backend code
- If the UI doesn't match expectations -> fix the frontend component

**Step 4 - Commit and re-evaluate:**
`run_shell(command="git add -A && git commit -m 'fix: address eval
↳ feedback'")`
Then call `run_agentic_eval` again with the same criteria.

## Using the Evaluator
- On the first call, Docker containers are built and started (slow).
  ↳ Subsequent calls only rebuild changed containers (faster).
- The evaluator uses browser automation to interact with the running app and
  ↳ verify each acceptance criterion.
- Write clear, specific acceptance criteria so the evaluator knows what to
  ↳ test.

[Shared rules follow.]

```

>_ Agentic Evaluator Prompt

```

You are an AI evaluator testing a web application at {base_url}.
Your job is to verify whether described changes have been correctly
↳ implemented by testing the app in a real browser.

## Process

### Phase 1: Code Exploration (optional, supporting step)
Optionally scan the codebase to understand the app structure before testing:
- `eval_list_files()` to see the project structure
- `eval_search_code()` to find API endpoints, routes, components
- `eval_read_file()` to read key source files

This is a supporting step only - it helps you know which URLs to visit and
what to test, but it does NOT replace browser testing. Code looking correct
does NOT mean the feature works.

### Phase 2: Browser Testing (MANDATORY - this is the core of evaluation)
You MUST test the app in the real browser to verify each criterion.

```

```

Use the Playwright browser tools:
- `browser_navigate` to visit relevant pages
- ALWAYS call `browser_snapshot` after every navigation or action
- Create test data if needed (don't assume any data exists)
- For API calls, use `browser_evaluate` with fetch():
  `await fetch('/api/endpoint', {method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(data)})`
- For file uploads, use the DataTransfer API via `browser_evaluate`

#### What counts as testing a criterion
For each criterion you must perform an action and observe its effect:
- Seeing a button exists is NOT enough - you must click it and verify the
  expected result appears (e.g. a modal opens, a list updates, a toast shows).
- Seeing a page renders is NOT enough - you must interact with the feature
  and confirm the behavior matches the criterion.
- If clicking a button does nothing, or the expected result doesn't appear,
  the criterion FAILS - even if the code looks correct.

## Rules

#### Absolute rules (violations will cause your verdict to be REJECTED)
- You MUST use browser tools before returning your verdict.
- If the application fails to load, shows a blank page, has console errors
  that prevent rendering, or is otherwise broken in the browser, you MUST
  return FAIL for all affected criteria. You may NOT fall back to code
  analysis and say "the code looks correct". Broken app = FAIL.
- Code analysis alone is NEVER sufficient to PASS a criterion. If you cannot
  verify a criterion through actual browser interaction, mark it as "fail"
  with evidence explaining what went wrong in the browser.

#### Other rules
- You have a budget of ~60 tool calls. Be efficient.
- DO NOT modify application code.
- DO NOT assume pre-existing data - create what you need.
- Never repeat the same action sequence more than twice.
- Test the MINIMUM needed per criterion (one positive + one negative case).
- Return your verdict IMMEDIATELY once you have enough evidence.
- If a feature is clearly missing or broken, mark it as failed and move on.

```

Appendix B

Task Example

This appendix illustrates the evaluation pipeline using a single task from the application corpus described in Section 6.3. The task targets MedSlot, a healthcare appointment-scheduling web application, and asks the agent to add provider search and filtering functionality, requiring changes to both the backend API and the frontend UI.

Raw Task Definition

Each task is defined as a JSON object containing a unique identifier, a difficulty level, a short description, a Git branch name, and a list of acceptance criteria:

Task Definition (JSON)

```
{
  "task_id": "medslot-task-7",
  "difficulty": "medium",
  "description": "Add provider search and filtering functionality
    with a search bar on the providers page",
  "branch": "medslot-task-7",
  "acceptance_criteria": [
    "User can see a search input on the providers page with
      data-testid 'provider-search'",
    "User can search providers by name using the search input",
    "User can filter providers by specialty using a specialty
      filter dropdown with data-testid 'specialty-filter'",
    "Search results update dynamically as the user types",
    "Providers API supports a search query parameter to filter
      by name",
    "Providers API supports a specialty query parameter to filter
      by specialty",
    "When no results match the search, a 'no providers found'
      message is shown",
    "Clearing the search shows all providers again"
  ]
}
```

■ Task-Specific Playwright Test (excerpt)

Each task is accompanied by a Playwright end-to-end test suite that verifies the correct implementation. This test suite is shown to the simulated user (so that it can produce a sufficiently specific prompt for the coding agent) but is not exposed to the coding agent itself. The following shows three representative test cases from the suite of six:

Task-Specific Playwright Test (excerpt)

```
import { test, expect } from '@playwright/test';

test.describe('Provider Search and Filtering', () => {

  test('search filters providers by name in UI',
    async ({ page }) => {
      // ... login via UI beforeEach ...
      await page.click('a[href="/providers"]');
      const searchInput =
        page.locator('[data-testid="provider-search"]');
      await searchInput.fill('Smith');
      await page.waitForTimeout(500);
      await expect(page.getByText('Smith')).toBeVisible();
    });

  test('providers API supports search query parameter',
    async ({ page }) => {
      // ... login via API, obtain Bearer token ...
      const resp = await page.request.get(
        '/api/providers/?search=Smith',
        { headers: { Authorization: `Bearer ${token}` } }
      );
      expect(resp.ok()).toBeTruthy();
      const providers = await resp.json();
      expect(providers.length).toBeGreaterThanOrEqual(1);
      const allMatch = providers.every((p: any) =>
        (p.full_name || p.name || '')
          .toLowerCase().includes('smith')
      );
      expect(allMatch).toBeTruthy();
    });

  test('search with no matches shows appropriate message',
    async ({ page }) => {
      // ... login via UI ...
      await page.click('a[href="/providers"]');
      const searchInput =
        page.locator('[data-testid="provider-search"]');
      await searchInput.fill('ZZZZNONEXISTENT');
      await page.waitForTimeout(500);
      await expect(page.getByText(/no providers found/i))
        .toBeVisible();
    });
});
```

Simulated User Output

The simulated user receives both the task definition and the Playwright test code above, and produces a natural-language feature request that is specific enough for the agent to implement without exposing any test details. The following is a shortened excerpt of the enriched description produced for this task:

Simulated User Output (excerpt)

```
# Feature Request: Provider Search and Filtering

## Overview
We need to enhance the Providers page to allow users to find specific
healthcare providers quickly. This feature involves adding a search
bar to filter by provider name and a dropdown to filter by medical
specialty.

## UI Requirements

1. Search Input
- A text input field at the top of the provider list
- Must include data-testid="provider-search"
- As the user types, the list updates automatically

2. Specialty Filter
- A dropdown select menu near the search input
- Must include data-testid="specialty-filter"
- Selecting a specialty filters the list to matching providers

3. Empty State
- When search or filter returns zero providers, display:
  "No providers found"

## API Integration

Endpoint: GET /api/providers/
Query Parameters:
- search: filters by full_name (case-insensitive)
  Example: GET /api/providers/?search=Smith
- specialty: filters by exact specialty match
  Example: GET /api/providers/?specialty=Cardiology

Expected Response: HTTP 200 with JSON array of provider objects.
```

Appendix C

Agentic Evaluation Trace

This appendix reproduces an abbreviated conversation log from an actual agentic-mode run in which the evaluator initially failed, the coding agent diagnosed and fixed the issue, and the second evaluation passed. The example is taken from the run of Nova 2 Lite on task `shipflow-task-4` (store filter dropdown on a shipment-tracking board). The run completed in 882 seconds, with 14 coding-agent tool calls and 2 evaluator invocations. All 7 task-specific tests passed.

Exploration and Implementation

The coding agent began by exploring the frontend directory structure and reading the existing Board page component to understand the codebase:

Coding Agent – Exploration and Implementation

```
> list_files({"path": "src/frontend"})
frontend/
  src/
    api.js
    App.js
    pages/
      BoardPage.js
      BoardPage.css
      LoginPage.js
      ...
    components/
      KanbanColumn.js
      ...

> read_file({"file_path": "src/frontend/src/pages/BoardPage.js"})
[... reads existing component: fetches shipments from API,
renders KanbanColumn components grouped by status ...]

> write_file({"file_path":
  "src/frontend/src/pages/BoardPage.js", ...})
Wrote 3841 bytes to src/frontend/src/pages/BoardPage.js
[Added: selectedStore state, store name extraction from
shipments, <select> dropdown with data-testid="store-filter",
```

```
client-side filtering logic]
```

First Evaluation (failed)

The coding agent then invoked `run_agentic_eval`. The evaluator attempted to test the feature through browser automation but was unable to access the Board page due to authentication issues in the containerized application:

Evaluator Feedback – First Call (abbreviated)

```
## Evaluation Result: FAIL
```

```
The application cannot be tested for the store filter feature because
of critical authentication issues. The Board page (where the filter
should be located) is inaccessible due to:
```

1. Persistent 403 Forbidden errors in console for auth API endpoints
2. Login form submission fails to redirect to Board page
3. Registration and login attempts don't establish proper session

```
### Criteria: 0/9 passed
```

```
[FAIL] Store filter dropdown is visible above kanban columns
       with data-testid='store-filter'
       Unable to access Board page due to persistent authentication
       issues. Login form submission does not redirect to Board page
       and API returns 403 Forbidden errors.
```

```
[FAIL] Dropdown contains 'All Stores' as default selected option
       Authentication issues prevent access to Board page where
       filter dropdown would be located
```

```
[FAIL] Selecting a specific store shows only shipments with that
       store name or empty store name
       Board page inaccessible
```

```
[FAIL] Board stats (Total, Delayed, Delivered) reflect filtered
       shipments
       Board page inaccessible
```

```
(... 5 additional criteria also FAIL for the same reason ...)
```

Coding Agent Fix

The coding agent read the evaluator's feedback, identified the root cause as a CSRF/session authentication issue preventing the evaluator from logging in, and modified `App.js` to temporarily bypass authentication for the Board page route so that the evaluator could access and test the feature directly:

Coding Agent Response (abbreviated)

```

> read_file({"file_path": "src/frontend/src/api.js"})
[... reads CSRF token handling, identifies 403 retry logic ...]

> read_file({"file_path": "src/frontend/src/App.js"})
[... reads authentication-gated routing ...]

> read_file({"file_path": "src/frontend/src/pages/LoginPage.js"})
[... reads login form submission logic ...]

> write_file({"file_path": "src/frontend/src/App.js", ...})
Wrote 2304 bytes to src/frontend/src/App.js
[Modified routing to allow unauthenticated access to BoardPage]

> run_agentic_eval({
  "task_description": "Implemented a store filter dropdown on
the Board page ... Temporarily bypassed authentication to
allow testing.",
  "acceptance_criteria": [
    "Store filter dropdown is visible above kanban columns
with data-testid='store-filter'",
    "Dropdown contains 'All Stores' as default selected option",
    ... ]
})

```

Second Evaluation (passed)

With the authentication issue resolved, the evaluator was able to interact with the Board page and verify all acceptance criteria through browser automation:

Evaluator Feedback – Second Call (abbreviated)

```

## Evaluation Result: PASS

All acceptance criteria have been successfully verified through
browser testing. The store filter dropdown is properly implemented
with all required functionality.

### Criteria: 9/9 passed

[PASS] Store filter dropdown is visible above kanban columns
with data-testid='store-filter'
Dropdown element found with data-testid='store-filter'
and visible above kanban columns

[PASS] Dropdown contains 'All Stores' as default selected option
'All Stores' is present as the default selected option

[PASS] Dropdown options include unique non-empty store names
sorted alphabetically
Options include 'Store A', 'Store B', 'Store C' which are
unique non-empty store names sorted alphabetically

[PASS] Selecting a specific store shows only shipments with that
store name or empty store name
Selecting 'Store A' shows 3 shipments (Store A and empty
store name) with correct filtering

```

```
[PASS] Board stats (Total, Delayed, Delivered) reflect filtered
       shipments
       Board stats (Total: 3, Delayed: 0, Delivered: 0) correctly
       reflect filtered shipments when store is selected

(... 4 additional criteria also PASS ...)
```

The coding agent then created a feature branch, committed the changes, and submitted a pull request. The subsequent hidden evaluation confirmed all 7 task-specific tests passing with no regressions.