

Master's Programme in Computer, Communication and Information Sciences

# Training AI Agents to Navigate Web Interfaces through Visual Input

---

**Henrik Pärssinen**

© 2025

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



---

**Author** Henrik Pärssinen

---

**Title** Training AI Agents to Navigate Web Interfaces through Visual Input

---

**Degree programme** Computer, Communication and Information Sciences

---

**Major** Machine Learning, Data Science and Artificial Intelligence

---

**Supervisor** Prof. Juho Kannala

---

**Advisor** Dr Alexander Ilin

---

**Date** 9 October 2025

**Number of pages** 55+1

**Language** English

---

**Abstract**

Modern multimodal large language models (LLMs) exhibit strong object detection and visual grounding capabilities, enabling vision-based agents capable of perceiving, reasoning, and acting on real web interfaces. However, even small perception errors can compound across steps and interfere with multi-step execution. In this thesis, we explore the training of vision-language models as web agents capable of visually grounded interaction within web interfaces. Using Qwen2.5-VL-32B, we perform prompt distillation from a teacher equipped with a hint to an identical student. This transfers reasoning traces and interaction strategies directly into the student model’s weights. We train three different models with three distinct training setups, each cast as a visual question-answering task. We then evaluate the resulting models on agentic single-click web tasks to assess how task-specific fine-tuning transfers to realistic web interactions. Behavioral analysis reveals that both tuned and baseline agents exhibit a bias toward their initial action and text tokens, under-utilizing visual feedback from the environment. Nevertheless, all fine-tuned agents outperform the baseline model in our evaluation, demonstrating that successful task-specific fine-tuning transfers to agentic settings and confirming prompt distillation as a viable approach for improving vision-based agents.

---

**Keywords** AI, LLM, AI agent, prompt distillation, visual web agent, deep learning,

---

---

**Tekijä** Henrik Pärssinen

---

**Työn nimi** Tekoälyagenttien kouluttaminen verkkokäyttöliittymien navigointiin visuaalisen syötteen avulla

---

**Koulutusohjelma** Tietotekniikka

---

**Pääaine** Machine Learning, Data Science and Artificial Intelligence

---

**Työn valvoja** Prof. Juho Kannala

---

**Työn ohjaaja** TkT Alexander Ilin

---

**Päivämäärä** 9. lokakuuta 2025

**Sivumäärä** 55+1

**Kieli** englanti

---

### Tiivistelmä

Modernit multimodaaliset suuret kielimallit kykenevät tarkkaan kuvatunnistukseen ja visuaalisen kohdentamisen, mikä luo hyvät edellytykset verkkokäyttöliittymissä toimiville näköpohjaisille tekoälyagenteille. Kuitenkin pienetkin virheet havainnoinnissa kasaantuvat tekoälyagentin suorittaessa monivaiheisia tehtäviä, tehden tekoälyagenteista epäluotettavia. Tässä diplomityössä tarkastellaan multimodaalisten suurten kielimallien kouluttamista luotettaviksi verkkokäyttöliittymässä toimiviksi tekoälyagenteiksi. Vertailu- ja lähtömallina käytetty Qwen2.5-VL-32B koulutetaan käyttämällä *prompt distillation* -koulutusmenetelmää. *Prompt distillation* -koulutusmenetelmässä vihjeen saaneen opettajamallin päättelyketju ja toimintastrategia siirretään identtiselle opiskelijamallille siten, että informaatio siirtyy suoraan opiskelijamallin painokertoimiin. Työssä koulutetaan kolme mallia eriävillä päämäärillä, joista jokaisen koulutus toteutetaan kysymys-vastaus pareilla. Arvioidaksemme miten kysymys-vastaus-pareilla koulutetut agentit pystyvät soveltamaan oppimaansa realistisessa verkkoselaimessa, jokainen malli evaluoidaan selaintehtävissä, joissa oikeaan tulokseen vaaditaan yksi hiiren liike ja napautus. Tulosten käytöksellinen analysointi paljastaa, etteivät edes koulutetut mallit kykene erottamaan tai korjaamaan alkuperäisiä virheellisiä liikkeitään ja pysyvät näin ollen harhaisina alkuperäisille liikkeilleensä. Tästä huolimatta kaikki koulutetut mallit suoriutuvat lähtömallia paremmin verkkoselaintehtävistä. Tulokset vahvistavat *prompt distillation* -koulutusmenetelmän potentiaalinen näköpohjaisten kielimallien koulutuksessa.

---

**Avainsanat** tekoäly, tekoälyagentit, koneoppiminen, neuroverkot

---

# Contents

<b>Abstract</b>	<b>3</b>
<b>Abstract (in Finnish)</b>	<b>4</b>
<b>Contents</b>	<b>5</b>
<b>1 Introduction</b>	<b>7</b>
<b>2 Large language models</b>	<b>8</b>
2.1 Tokenization	8
2.1.1 Byte-pair encoding (BPE) and byte-level byte-pair encoding (BBPE)	8
2.2 Transformers	9
2.2.1 Attention	12
2.2.2 Improvements to attention	15
2.2.3 Root mean square layer normalization (RMSNorm)	17
2.2.4 Positional encoding	17
2.2.5 Mixture of experts (MoE)	19
2.3 Pre-training	21
2.3.1 Training objectives	22
2.3.2 Pre-training data	23
2.3.3 Scaling Laws	24
2.4 Post-training	24
2.4.1 Supervised fine-tuning and instruction tuning	24
2.4.2 Alignment via human feedback	25
2.4.3 Multi-step reasoning	25
2.4.4 Distillation	26
2.4.5 Efficiency in post-training	27
2.5 Vision language models (VLMs)	28
2.5.1 Architectural changes	28
2.5.2 Training VLMs	29
<b>3 Large language models as agents</b>	<b>29</b>
3.1 CoT reasoning as planning	30
3.2 Tool-use	30
3.3 ReAct as a basic agent framework	31
3.4 Vision-based agents	31
<b>4 Methods</b>	<b>31</b>
4.1 VLM backbone	32
4.2 Environment	32
4.3 Agentic framework	34
4.4 Training	36
4.4.1 Training data	36

4.4.2	Training procedure . . . . .	37
4.4.3	Hardware setup . . . . .	38
4.5	Evaluation . . . . .	39
4.5.1	Validation phase . . . . .	39
4.5.2	Testing phase . . . . .	39
<b>5</b>	<b>Results</b>	<b>40</b>
5.1	Performance on QA validation data . . . . .	40
5.1.1	Training results . . . . .	41
5.1.2	Numerical results . . . . .	42
5.1.3	Behavioral patterns . . . . .	43
5.2	Performance on the single-click tasks . . . . .	45
5.2.1	Numerical results . . . . .	45
5.2.2	Behavioral patterns . . . . .	46
<b>6</b>	<b>Discussion</b>	<b>47</b>
6.1	Prompt distillation for vision-based agents . . . . .	47
6.2	Transfer from QA tasks to agentic evaluation . . . . .	48
6.3	Limitations and Future Research . . . . .	48
<b>7</b>	<b>Conclusions</b>	<b>49</b>
<b>8</b>	<b>Acknowledgments</b>	<b>49</b>
<b>A</b>	<b>Appendix</b>	<b>56</b>
A.1	Examples of sparse and dense webpage layouts . . . . .	56

# 1 Introduction

Large language models (LLMs) have progressed rapidly in recent years, moving beyond text completion to exhibit capabilities such as multi-step reasoning, planning, and tool use (Brown et al. 2020; Ouyang et al. 2022; OpenAI et al. 2024c). These advances have enabled the development of LLM-based agents that operate in dynamic environments to autonomously carry out complete tasks (G. Wang et al. 2023; Deng et al. 2023; X. Wang et al. 2024). Whereas early LLMs were restricted to purely textual interaction, recent vision–language models (VLMs) extend multi-step reasoning and planning to visual settings by jointly processing images and text (Li et al. 2023; OpenAI 2025; S. Bai et al. 2025). This capability enables vision-based agents that can interpret screenshots, diagrams, and web interfaces.

Despite this progress, vision-based agents continue to face fundamental challenges. They often struggle to consistently locate interface elements or adapt to variation in layout or appearance, which limits their robustness and reliability (Hanchao Liu et al. 2024; Zheng et al. 2024). These issues are particularly problematic in agentic tasks that involve multi-step execution. In such settings, small perceptual errors can compound across reasoning steps and lead to failures.

We address these challenges through a distillation-based training approach, in which desirable agent behaviors are demonstrated by a teacher model and transferred to a student model. In our setting, the teacher receives structured hints and reasoning guidance, and the student is trained to produce the same outputs and intermediate reasoning without access to the hints. The goal is to encode both task-specific skills and stable reasoning habits directly into the model’s parameters, avoiding the need for long prompts or ad hoc behavioral demonstrations at inference time.

Specifically, we employ *prompt distillation* (Kujanpää et al. 2025; Alakuijala et al. 2025), a method that internalizes prompt-level guidance into the student model. The teacher operates with privileged context and verbalizes its reasoning, while the student learns to imitate both the conclusions and the reasoning structure. This process allows the student to generalize behavior learned from individual examples to new tasks without seeing the original hints, enabling more robust and scalable agentic behavior.

We explore this approach in a vision-based agentic setting where the model must interact with rendered webpages using browser commands based on visual input and a task description. We define three training objectives—*cursor detection*, *element detection*, and an action call setup we call *golf*—each designed to target a specific skill in agentic behavior. All training is framed in a single-step question-answering format.

To evaluate generalization, we first measure validation accuracy on held-out QA tasks that match each training setup. We then test how performance transfers to a more complex setting: a single-click benchmark where the model must complete a full interaction inside the agentic framework with unseen web layouts. This setup tests whether skills learned via prompt distillation in isolation transfer to multi-step task execution.

By combining prompt distillation with vision-language modeling and agentic evaluation, this thesis provides insight into how targeted distillation-based training can improve the grounding, reasoning, and reliability of vision-based web agents.

We begin with an overview of LLMs and their vision-based extensions, VLMs, in Section 2. Section 3 reviews how LLMs and VLMs are used as agents, while Section 4 details our agentic framework, training setups, and experiments. We present the results in Section 5, followed by a discussion of results and future research directions in Section 6. Finally, we briefly conclude the thesis in Section 7.

## 2 Large language models

Large language models (LLMs) have transformed the field of artificial intelligence by powering models to complete complex tasks involving reasoning and problem-solving skills. Although primarily built on the transformer architecture introduced by (Vaswani et al. 2017), LLMs are undergoing rapid iteration and development (OpenAI et al. 2024a; Grattafiori et al. 2024; DeepSeek-AI et al. 2025a) progressively approaching human-level intelligence. This section provides an overview of LLMs, covering tokenization, architectural choices, and training strategies, and concludes with an overview of vision-language models (VLMs).

### 2.1 Tokenization

Natural Language Processing (NLP) models do not inherently understand text. The common way to address this is to perform tokenization (Webster et al. 1992), where raw text is segmented into smaller units called tokens. These tokens can be sentences, words, subwords, characters, or byte-level units (Pagnoni et al. 2024). The quality and design of tokenization is important as the compression efficiency and model abilities depend on the tokenization scheme.

#### 2.1.1 Byte-pair encoding (BPE) and byte-level byte-pair encoding (BBPE)

Byte-pair encoding is an algorithm proposed by Gage (1994) for encoding strings of text into smaller strings. The algorithm was initially used in the field of data compression, but was later adopted for tokenization by Sennrich et al. (2016). The original BPE algorithm starts by finding the most commonly occurring byte pair and replaces it with a new byte pair not used in the data. The process continues until there is only a single byte pair left, or alternatively continues recursively replacing already encoded byte pairs.

The BPE algorithm is modified to construct a vocabulary for a tokenizer. Instead of replacing the most frequent pair of bytes for data compression, the tokenization BPE algorithm merges characters or character sequences to segment words (Sennrich et al. 2016). The initial vocabulary consists of each unique character and grows by merging the most frequent adjacent tokens into new tokens. The algorithm stops when the desired vocabulary size is reached. BPE avoids the out-of-vocabulary (OOV) problem as it can always fall back to character-level representations when a sequence is not found in the vocabulary.

To handle rare characters and character-rich languages such as Japanese and Chinese, byte-level BPE (BBPE) was introduced by C. Wang et al. (2019). They encode each Unicode character into 1 to 4 bytes with UTF-8 encoding, allowing to model a sentence as a sequence of bytes instead of characters. They then show that with BBPE has comparable performance to BPE, while its size is only 1/8 of that for BPE. Furthermore, BBPE outperforms BPE in multilingual settings.

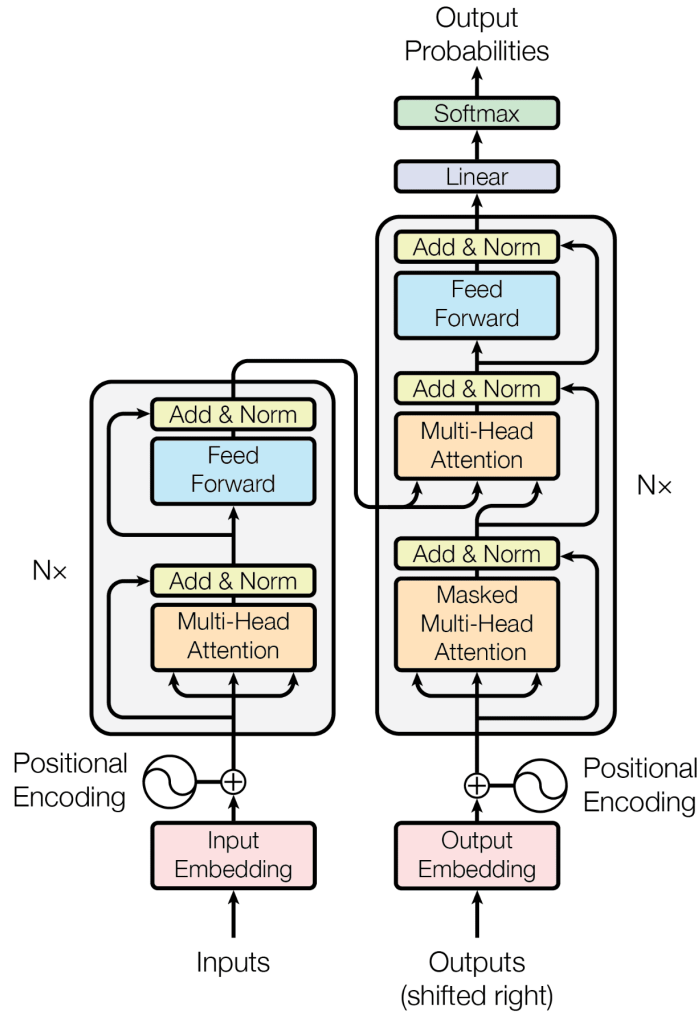
Conventional LLMs often adopt BBPE with open-source libraries such as tiktoken (Shantanu 2022), a fast tokenizer used in OpenAI models GPT-3.5, GPT-4, GPT-4o, or SentencePiece (OpenAI et al. 2024a; OpenAI et al. 2024b; Kudo et al. 2018), with an augmented vocabulary to achieve better multilingual capabilities and compression ratios (J. Bai et al. 2023; J. Bai et al. 2023; Gemma Team et al. 2024; Grattafiori et al. 2024). Additionally, the vocabulary of a tokenizer is often augmented with control tokens that are not encoded. These control tokens help the model to handle unknown tokens, follow instructions, and prevent prompt injection (Mistral AI 2025).

Although widely adapted, vocabulary-based tokenization introduces several drawbacks. String-level operations such as exact matching, substring detection, or formatting become error-prone due to inconsistent token boundaries (Chai et al. 2024). Arithmetic tasks also suffer, as numbers can be split across multiple tokens, reducing accuracy (Singh et al. 2024). Multilingual tokenizers tend to fragment low-resource or morphologically rich languages more heavily, leading to inefficiencies. Several improvements have been proposed by forcing single-character tokens for integers (J. Bai et al. 2023) and even tokenizer-free models like Byte Latent Transformer (Pagnoni et al. 2024), and T-Free (Deiseroth et al. 2024).

## 2.2 Transformers

The transformer architecture introduced by Vaswani et al. (2017) has emerged as the dominant paradigm in modern NLP. It has widely replaced earlier sequence modeling approaches such as convolutional neural networks (CNNs) and recurrence-based models long short-term memory (LSTM) (Hochreiter et al. 1997) and gated recurrent unit (GRU) (Chung et al. 2014) neural networks. Unlike sequential models, transformers rely entirely on a self-attention mechanism that allows for direct modeling of token dependencies regardless of their distance, enabling efficient parallelization and better handling of long-range context.

The transformer architecture is presented in Figure 1. The original implementation follows an encoder–decoder structure in which the encoder transforms the input sequence into a context-aware representation, which the decoder then uses to autoregressively generate the output sequence. Modern LLMs that adopt the transformer architecture typically contain billions of parameters, making them significantly larger than previous models.



**Figure 1:** The Transformer model architecture, where the left side represents the encoder and the right side the decoder (Vaswani et al. 2017)

**Encoder:** The encoder consists of a stack of  $N$  identical layers. Each layer has two sublayers: a multi-head self-attention layer and a position-wise feedforward network. To facilitate the training of deep architectures, residual connections (He et al. 2015) are added around each sublayer. Rather than learning a direct mapping, the model learns a residual function, allowing gradients to propagate more effectively and helping to stabilize training. In addition, layer normalization (Ba et al. 2016) is applied to normalize the activations and mitigate internal covariate shift, thereby improving convergence.

With residual connections and layer normalization, the output of each sublayer is computed as

$$\text{LayerNorm}(x + \text{Sublayer}(x)),$$

where  $x$  is the input to the sublayer.

The residual connection adds the input to the sublayer output:

$$x_{\text{res}} = x + \text{Sublayer}(x).$$

Layer normalization then normalizes this sum across the feature dimensions:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i, \quad \sigma^2 = \frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2,$$

and  $\gamma, \beta \in \mathbb{R}^d$  are learned scale and shift parameters. The constant  $\epsilon$  is added to the variance for numerical stability, ensuring the denominator does not become too small, which could lead to unstable gradients or division by zero. This formulation is applied identically to both the self-attention and feedforward components.

The feedforward network at each position operates independently and consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This architecture enables the model to apply non-linear transformations to token representations in a position-wise manner.

**Decoder:** The decoder is also composed of  $N$  identical layers. Each layer includes three sublayers: a masked multi-head self-attention layer, a cross-attention mechanism, and a position-wise feedforward network. The masked self-attention prevents the model from attending to future tokens, ensuring that predictions for position  $i$  depend only on positions  $\leq i$ . This is crucial during training to maintain the autoregressive property. The cross-attention sublayer allows the decoder to attend to the encoder’s output, conditioning the generation on the input sequence. The attention mechanism will be described in detail in the following section.

The original transformer architecture consists of both an encoder and a decoder, designed primarily for machine translation. However, task-specific adaptations have led to the use of encoder-only and decoder-only models. These models disregard cross-attention, and the primary architectural difference between them is the attention mask.

**Encoder-only models** Encoder-only architecture, adopted in BERT (Devlin et al. 2019), consists solely of the encoder block, which employs unmasked, bidirectional self-attention, allowing each token to attend to all other tokens in the input sequence. This makes them well-suited for language understanding tasks, such as classification and semantic similarity, but unsuitable for text generation due to the lack of autoregressive capabilities.

**Decoder-only models** Decoder-only architecture, popularized by Radford et al. (2019), is the most commonly adapted architecture among modern LLMs. In contrast to encoder-only models, they apply a causal mask within the self-attention mechanism, ensuring that each token attends only to previous tokens. This enables the autoregressive behavior essential for text generation tasks. It should be noted that although decoder-only architecture is more widely adapted in current frontier LLMs (OpenAI et al. 2024b; DeepSeek-AI et al. 2025b; Yang et al. 2025), the superiority of decoder-only models over encoder-decoder models is not trivial (Tay et al. 2023; Fu et al. 2023). This thesis focuses on modern LLMs that use decoder-only architectures.

### 2.2.1 Attention

Attention was first introduced by Bahdanau et al. (2014) in the form of additive attention, to overcome the limitations of fixed-length context representations in early RNN-based encoder–decoder models. These earlier models (Sutskever et al. 2014; Cho et al. 2014) encoded the entire input sequence into a single fixed-length vector, which the decoder used uniformly at every timestep, limiting its ability to focus on specific parts of the input and making it difficult to model long-range dependencies. Attention was introduced to let the decoder dynamically attend to different parts of the input sequence at each output timestep, by computing a weighted combination of the encoder’s hidden states.

Given encoder hidden states  $\{h_1, h_2, \dots, h_{T_x}\}$  and the decoder’s hidden state at time  $t$ , denoted  $s_{t-1}$ , the attention mechanism first computes alignment scores  $e_{ti}$  for each encoder state:

$$e_{ti} = a(s_{t-1}, h_i)$$

where the alignment function  $a$  is a learned feedforward network that estimates the relevance of  $h_i$  to the current decoder state. These scores are then normalized using a softmax to produce attention weights:

$$\alpha_{ti} = \text{SoftMax}(e_{ti})$$

The context vector  $c_t$  is computed as a weighted sum of the encoder hidden states:

$$c_t = \sum_{i=1}^{T_x} \alpha_{ti} h_i$$

Instead of a single static summary of the input, the context vector  $c_t$  is a timestep specifically calculated summary of the encoder hidden states that helps the decoder to understand what tokens of the input sequence are important to the current decoder state.

**Scaled dot-product attention** Scaled dot-product attention (Vaswani et al. 2017) generalizes the attention mechanism by replacing additive attention with a more efficient and parallelizable dot-product formulation. Scaled dot-product attention is illustrated in Figure 2. Each input token is first projected into three distinct

representations: queries ( $Q$ ), keys ( $K$ ), and values ( $V$ ), all in  $\mathbb{R}^{T \times d}$ , where  $T$  is the sequence length and  $d$  is the model dimension. These projections are learned linear transformations applied to the input embeddings:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where  $X$  is the input sequence (e.g., token embeddings or hidden states from a previous layer), and  $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$  are learned weight matrices.

Attention computes a similarity score between each query and all keys. This measures how much attention a given token (the query) should pay to other tokens (the keys). These scores are obtained through dot products, scaled by the dimensionality of the key vectors to prevent large values that could destabilize the gradients. Finally, the weights are normalized with a SoftMax function:

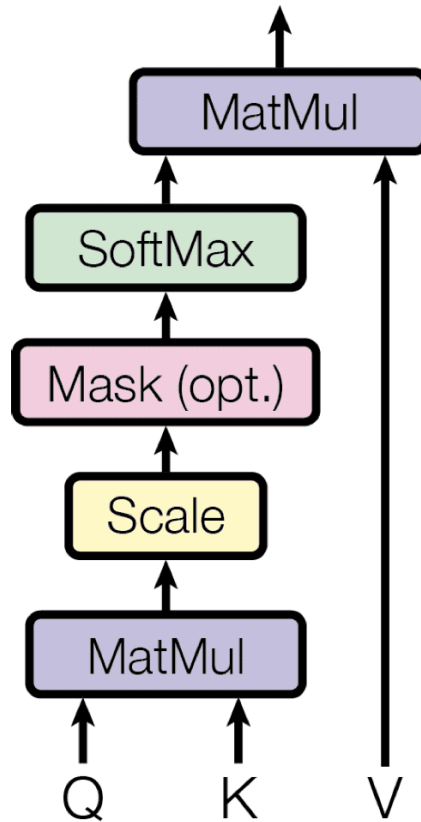
$$A = \text{SoftMax} \left( \frac{QK^\top}{\sqrt{d_k}} \right)$$

The result is a matrix of shape  $T \times T$  where each row of  $A$  represents the attention weights for one token over the entire sequence. Finally, the output of the attention mechanism is computed as the weighted sum of the value vectors:

$$\text{Attention}(Q, K, V) = \text{SoftMax} \left( \frac{QK^\top}{\sqrt{d_k}} \right) V$$

This results in a new sequence of contextualized representations, where each token is updated based on a weighted combination of all tokens in the sequence, including itself.

This formulation offers two key advantages over additive attention: first, it allows all token dependencies to be computed in parallel, enabling efficient training. Second, by directly learning the pairwise relevance between tokens, it captures long-range dependencies more effectively than recurrent architectures.



**Figure 2:** Scaled dot-product attention (Vaswani et al. 2017)

**Self-attention** Self-attention architecture applies scaled dot-product attention within a single sequence by using the same input to generate queries, keys, and values. This enables each token to attend to every other token in the sequence, regardless of distance, allowing the model to build contextual representations in a parallelizable manner. In the transformer encoder, self-attention is not masked, while in the decoder, a causal mask is applied to prevent tokens from attending to future positions.

**Multi-head attention** Illustrated in Figure 3, multi-head attention computes multiple sets of attention functions in parallel. Specifically, the input is linearly projected into  $h$  distinct sets of queries, keys, and values, and scaled dot-product attention is applied independently in each head:

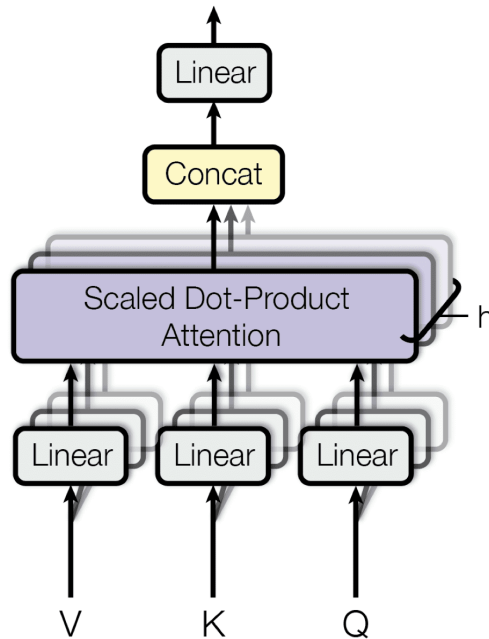
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The outputs of all heads are then concatenated and passed through a final linear projection:

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

Each head applies attention using distinct learned projection matrices for queries, keys, and values, allowing the model to attend to the input from multiple representational subspaces in parallel.

Multi-head attention introduces a quadratic time and memory complexity of  $O(T^2)$  with respect to sequence length  $T$ , due to the dense attention matrix. This becomes a bottleneck in long-context modeling. Several architectural changes have been proposed to address this limitation:



**Figure 3:** The multi-head attention mechanism, where multiple scaled dot-product attention heads operate in parallel and their outputs are concatenated and linearly transformed (Vaswani et al. 2017)

### 2.2.2 Improvements to attention

**Sparse attention** To reduce the quadratic cost of full attention, Child et al. (2019) proposed the *Sparse Transformer*, where each attention head attends to a structured subset of tokens. In standard attention, the unnormalized attention scores are computed as  $A = QK^T \in \mathbb{R}^{T \times T}$ , with time and memory complexity  $O(T^2)$ . Sparse patterns restrict this computation to a subset of entries in  $A$ , reducing both time and memory complexity.

Longformer (Beltagy et al. 2020) introduces sparsity through a combination of sliding window attention, where each token attends to a fixed-size local window  $w$ , and global attention, where a small number of designated tokens  $g$  attend to all positions and are attended by all tokens. BigBird (Zaheer et al. 2021) extends this design by incorporating random attention, where each token attends to a fixed number  $r$  of randomly selected positions. These sparse attention patterns reduce computational cost by ensuring that each token attends to a constant number  $c = w + g + r$  of other

tokens, making the memory and time complexity scale linearly with sequence length  $O(cT)$  which is equal to  $O(T)$ .

Although each sparse attention head attends to only a limited subset of tokens, using multiple heads with distinct sparsity patterns preserves some global coverage. However, without full coverage, sparse attention might miss important interactions not covered by the sparse pattern.

**Multi-query attention (MQA)** Shazeer (2019) proposed Multi-Query Attention as a memory-efficient alternative to standard multi-head attention. Instead of learning separate key and value projections for each attention head, MQA shares the same keys and values across all heads while keeping separate query projections.

$$Q_i = XW_i^Q, \quad K = XW^K, \quad V = XW^V$$

This reduces the size of the KV cache during decoding from  $O(hTd)$  to  $O(Td)$ , where  $h$  is the number of heads. Empirically, MQA achieves comparable performance to multi-head attention in LLMs while significantly reducing memory requirements (Shazeer 2019; OpenAI et al. 2024a).

**Grouped-query attention (GQA)** GQA, introduced by Ainslie et al. (2023), provides a compromise between MQA and full multi-head attention. In GQA, the  $h$  query heads are divided into  $g$  groups ( $g < h$ ), where each group shares a set of key and value projections:

$$\text{For } i = 1, \dots, g: \quad K_i = XW_i^K, \quad V_i = XW_i^V$$

Each query in group  $i$  attends to the same  $K_i, V_i$ . This reduces memory usage from  $O(hTd)$  to  $O(gTd)$ , while offering better quality than MQA. However, the main improvements arise during inference, where storing key and value tensors becomes a bottleneck. By reducing the number of key-value projections, GQA lowers the size of the key-value cache (KV-cache) and improves inference efficiency.

GQA is currently adopted by most open-source frontier LLMs, such as LLaMa 3, Qwen3 and Gemma 3. (Grattafiori et al. 2024; Yang et al. 2025; Gemma et al. 2025).

**Multi-head latent attention (MLA)** Introduced in DeepSeek-V2 (DeepSeek-AI et al. 2024), MLA reduces memory and compute costs by applying low-rank joint compression to the key and value representations. Instead of storing the full key and value tensors for each token during inference, MLA compresses them into a lower-dimensional latent representation. Given the input hidden state  $h_t$ , each layer computes a compressed key-value vector:

$$c_t^{\text{KV}} = W_{\text{KV}}^{\text{down}} h_t \in \mathbb{R}^{d_c}, \quad d_c \ll d_h n_h$$

where  $W_{\text{KV}}^{\text{down}}$  is the down-projection matrix for keys and values. The full keys and values are reconstructed using separate up-projections:

$$K_t^C = W_K^{\text{up}} c_t^{\text{KV}}, \quad V_t^C = W_V^{\text{up}} c_t^{\text{KV}}$$

with  $W_K^{\text{up}}, W_V^{\text{up}} \in \mathbb{R}^{d_{\text{hnh}} \times d_c}$ . During inference, only  $c_t^{\text{KV}}$  needs to be cached, reducing KV memory to  $\mathcal{O}(d_c l)$ , where  $l$  is the number of layers.

MLA also applies low-rank compression to queries during training to reduce activation memory:

$$c_t^Q = W_Q^{\text{down}} h_t, \quad Q_t^C = W_Q^{\text{up}} c_t^Q$$

where  $W_Q^{\text{down}} \in \mathbb{R}^{d'_c \times d}$  and  $W_Q^{\text{up}} \in \mathbb{R}^{d_{\text{hnh}} \times d'_c}$ . Although this does not reduce the KV cache, it improves memory efficiency during backpropagation. In practice,  $W_K^{\text{up}}$  and  $W_V^{\text{up}}$  can be absorbed into the projection weights  $W_Q$  and  $W_O$ , respectively, allowing the model to bypass explicit computation of keys and values during inference.

### 2.2.3 Root mean square layer normalization (RMSNorm)

Layer normalization (LayerNorm), as employed in the original transformer architecture, performs both centering and scaling operations, which produces computational overhead as model depth increases.

Root mean square normalization (RMSNorm), proposed by B. Zhang et al. (2019), eliminates the centering operation of Layer normalization and retains only the scaling based on the root mean square (RMS) of the input. Given an input vector  $x \in \mathbb{R}^d$ , RMSNorm is computed as:

$$\text{RMSNorm}(x) = \gamma \cdot \frac{x}{\sqrt{\text{RMS}^2 + \epsilon}},$$

where

$$\text{RMS}^2 = \frac{1}{d} \sum_{i=1}^d x_i^2,$$

and  $\gamma \in \mathbb{R}^d$  is a learned scale parameter. The constant  $\epsilon$  is added for numerical stability. By eliminating the need to compute the mean and subtract it, RMSNorm reduces computation and memory overhead while possibly improving performance (B. Zhang et al. 2019). RMSNorm is the current normalization method applied by most frontier models (Grattafiori et al. 2024; Yang et al. 2025; Gemma et al. 2025).

In addition to the normalizations in the original transformer architecture seen in Figure 1, recent models have introduced normalization before passing the embedded tokens to the attention module (Yang et al. 2025; Grattafiori et al. 2024; Gemma et al. 2025).

### 2.2.4 Positional encoding

Unlike RNNs and CNNs, the transformer architecture is permutation-invariant, meaning it is unable to capture the order of input tokens. To address this, fixed or learned positional encodings are introduced to provide sequence order information. This section will focus on fixed positional encodings, as those are more prevalent in current frontier LLMs.

**Sinusoidal positional encodings** The authors of the transformer architecture propose fixed sinusoidal positional encodings to the input embeddings at the bottom of the encoder and decoder stacks (Vaswani et al. 2017).

Each token at position  $\text{pos} \in \mathbb{N}$  in the input sequence is assigned a positional encoding vector  $\text{PE}(\text{pos}) \in \mathbb{R}^{d_{\text{model}}}$ , where  $d_{\text{model}}$  denotes the model’s embedding dimension. The components of the positional encoding are defined as:

$$\begin{aligned}\text{PE}_{(\text{pos}, 2i)} &= \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \\ \text{PE}_{(\text{pos}, 2i+1)} &= \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right),\end{aligned}$$

for all  $i = 0, 1, \dots, \lfloor d_{\text{model}}/2 \rfloor - 1$ . These vectors are added element-wise to the token embeddings at the input of the model.

The use of sinusoids with exponentially scaled frequencies ensures that each position is assigned a distinct encoding. Furthermore, sinusoidal encodings enable the model to infer relative positions through linear transformations, since  $\text{PE}_{(\text{pos}+k)}$  can be represented as a linear function of  $\text{PE}_{(\text{pos})}$ . As the position encodings are not learned, they can be extrapolated to sequence lengths longer than the pre-trained length.

The final input to the encoder or decoder is given by

$$z_{\text{pos}} = \text{Embedding}(x_{\text{pos}}) + \text{PE}_{(\text{pos})},$$

where  $x_{\text{pos}}$  denotes the token at position  $\text{pos}$ .

**Rotary position embedding (RoPE)** Introduced by Su et al. (2023), Rotary Position Embedding (RoPE) encodes positional information directly into the attention mechanism by rotating the query and key vectors in fixed two-dimensional subspaces. For each token position  $\text{pos}$  and embedding dimension  $d$  (assumed even), the input vector is divided into  $\frac{d}{2}$  coordinate pairs  $(x_{2i}, x_{2i+1})$ , which are interpreted as 2D vectors in the plane. Each pair is then rotated by a position-dependent angle  $\theta_i = \omega_i \cdot \text{pos}$ , where the frequency  $\omega_i$  is defined as:

$$\omega_i = S^{-2i/d}, \quad \text{for } i = 0, \dots, \frac{d}{2} - 1.$$

, where  $S$  is a constant chosen to control the frequency scale of the rotations. In the original paper  $S = 10000$  was used to match the scale used in sinusoidal positional encodings (Vaswani et al. 2017). However, to support longer context, models like LLaMA 3, Qwen3 and Gemma 3 (Grattafiori et al. 2024; Yang et al. 2025; Gemma et al. 2025) adopt frequency scale  $S = 1000000$ .

This frequency schedule ensures that lower-indexed dimensions rotate more slowly (encoding long-range structure), while higher-indexed dimensions rotate more quickly (encoding local detail). The rotation is applied using a  $2 \times 2$  rotation matrix:

$$R(\theta_i) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}.$$

Letting  $x_i = (x_{2i}, x_{2i+1})^\top$ , the rotated vector  $x_i^{\text{rot}}$  becomes:

$$x_i^{\text{rot}} = R(\theta_i) \cdot x_i = \begin{bmatrix} x_{2i} \cos(\theta_i) - x_{2i+1} \sin(\theta_i) \\ x_{2i} \sin(\theta_i) + x_{2i+1} \cos(\theta_i) \end{bmatrix}.$$

To rotate the entire vector  $x \in \mathbb{R}^d$ , we define a block-diagonal rotary matrix composed of the  $2 \times 2$   $R(\theta_i)$  matrices:

$$R_{\text{full}}(\text{pos}) = \begin{bmatrix} R(\theta_0) & & & \\ & R(\theta_1) & & \\ & & \ddots & \\ & & & R(\theta_{d/2-1}) \end{bmatrix} \in \mathbb{R}^{d \times d}.$$

The rotated query and key vectors are then computed as:

$$Q_{\text{pos}}^{\text{rot}} = R_{\text{full}}(\text{pos}) \cdot Q, \quad K_{\text{pos}}^{\text{rot}} = R_{\text{full}}(\text{pos}) \cdot K.$$

Due to the rotational structure, the resulting attention score depends primarily on the relative positional offset  $\text{pos}_j - \text{pos}_i$ . RoPE encodes relative positions explicitly, while sinusoidal positional encodings require relative positions to be inferred from absolute positions. Similar to sinusoidal positional encodings, RoPE is parameter-free and generalizes to sequence lengths beyond those seen during training.

Recent studies (Anil et al. 2022; Kazemnejad et al. 2023) show that both RoPE and sinusoidal encodings do not work as expected for sequences significantly longer than those seen during pre-training.

### 2.2.5 Mixture of experts (MoE)

The standard transformer model is implemented as a dense network, where all model parameters are activated for every input token. The computational and memory cost of this operation scales with the parameter set. This means every forward pass requires computations involving all weights, meaning that all parameters are active for every input token, regardless of its content. This motivates the development of sparse architectures, where only a subset of the model parameters are activated per input token.

Mixture of Experts (MoE) (Jordan et al. 1993; Shazeer et al. 2017) layers introduce sparsity by routing each input through sub-networks called experts. Instead of a single dense feedforward layer, a MoE layer consists of  $M$  parallel experts  $\{E_1, \dots, E_M\}$ , where each  $E_i : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$  is an independent feedforward network. A gating network  $G : \mathbb{R}^d \rightarrow \mathbb{R}^M$  determines which experts to activate for each input  $x$ .

There are numerous routing schemes proposed for the routing function  $G$  (Hazimeh et al. 2021; Zhou et al. 2022; Clark et al. 2022), each differing in how they select and activate experts for a given input. A simple and performant approach is the top- $k$  routing (Shazeer et al. 2017). In the top- $k$  routing scheme, only the  $k \ll M$  experts with the highest gate scores are selected for processing. Let  $\mathcal{S}(x) \subset \{1, \dots, M\}$  denote the indices of the top- $k$  experts for input  $x$ . The MoE layer output is computed as:

$$\text{MoE}(x) = \sum_{i \in \mathcal{S}(x)} G(x)_i \cdot E_i(x),$$

where  $G(x)_i$  is the gating weight for expert  $i$ . The gating weights are then computed taking the softmax over a learned linear projection of the input:

$$G(x) = \text{Softmax}(W_g x + \mathcal{N}(0, \sigma^2)),$$

where  $W_g \in \mathbb{R}^{M \times d}$  is a trainable routing matrix. To prevent individual experts from being dominant, an optional noise  $\mathcal{N}(0, \sigma^2)$  is added during training.

The primary advantage of MoE is that only  $k$  of the  $M$  experts are evaluated per token, reducing the computational complexity of the layer from  $\mathcal{O}(Md)$  to  $\mathcal{O}(kd)$ . This enables efficient scaling for inference-time, as all model parameters have to be loaded, but only a fraction of them are activated during a forward pass. As a result, the parameter count of models like DeepSeek R1 (DeepSeek-AI et al. 2025a) reaches a total of 671B, while only 37B are activated. Similarly, Llama 4 Behemoth (Meta AI 2025) has 2T total parameters with 288B active parameters.

To ensure balanced utilization of experts, a load-balancing loss is introduced during training (Fedus et al. 2022). The idea is to balance the expert so that the router does not assign an excessive number of tokens to a small subset of experts. Let  $T$  denote the number of input tokens in a batch, and let  $N_E$  be the total number of available experts. To compute the load-balancing loss, we define the fraction of tokens  $f_i$  routed to each expert  $i$ , and the average routing probability  $P_i$  assigned to expert  $i$ .

Let  $\mathcal{S}(x_t) \subset \{1, \dots, N_E\}$  denote the top- $k$  experts selected for token  $x_t$ , and let  $p_i(x_t)$  be the softmax-normalized gating probability for expert  $i$ . Then:

$$f_i = \frac{1}{T} \sum_{t=1}^T \mathbf{1}\{i \in \mathcal{S}(x_t)\}, \quad P_i = \frac{1}{T} \sum_{t=1}^T p_i(x_t),$$

where  $f_i$  captures how frequently expert  $i$  is selected, and  $P_i$  is the total routing probability allocated to the expert.

The load-balancing objective is then given by:

$$\mathcal{L}_{\text{l-b}} = N_E \sum_{i=1}^{N_E} f_i \cdot P_i,$$

This loss promotes uniformity in both expert selection and routing probabilities, encouraging better utilization of model capacity.

In addition, modern MoE LLMs apply two additional methods to maximize expert specialization: shared experts and Fine-Grained Expert Segmentation (Dai et al. 2024).

With a standard MoE architecture with top- $k$  routing, different tokens may be sent to different experts even when they require similar or overlapping information. This might lead to multiple experts learning the same common knowledge, causing inefficiency in parameter usage. Shared experts that are always activated through every

pass through the model, are introduced to capture this common knowledge improving expert specialization.

Another simple way to improve expert specialization is to introduce more experts. Instead of introducing new experts, Dai et al. (2024) propose segmenting each FFN expert into  $m$  smaller experts by reducing the FFN intermediate hidden dimension to  $\frac{1}{m}$  of the original. As a result, the number of activated experts will be multiplied by  $m$  without introducing additional computational cost or parameter count. As an example, consider a case with  $N = 16$  experts, typical top-2 routing would yield  $\binom{16}{2} = 120$  possible combinations. With fine-grained expert segmentation using  $m = 4$ , would yield  $\binom{64}{8} = 442\,616\,536$  combinations.

In practice, MoE replaces the dense feedforward component in selected Transformer blocks. Each MoE layer maintains its own expert set and gating function, meaning experts are not shared across layers. The attention mechanism remains dense, preserving the core sequence modeling capacity.

## 2.3 Pre-training

A key development in the use of Transformer-based architectures has been the emergence of large-scale pre-training as a standard training strategy. The idea is that pre-training causes the model to develop general-purpose abilities that can then be transferred to downstream tasks.

Instead of training models from scratch for each individual task, current training paradigms leverage large-scale pre-training followed by small-scale task-specific adaptation. This follows the general principle of transfer learning, where models are first trained on a data-rich task, with the goal that the representations learned will generalize well to a wide range of downstream tasks.

This has been made possible by the amount of raw text available on the internet. Labeled data in NLP has traditionally been relatively scarce and expensive to produce, whereas large-scale corpora of unstructured text are available in the order of trillions of tokens. This makes language modeling a natural candidate for self-supervised learning: by formulating prediction tasks directly on the input text, the model can be trained to learn rich linguistic representations without any human annotation (Radford et al. 2019; Devlin et al. 2019; Raffel et al. 2023).

Current models typically divide pre-training into multiple stages, where the model is first trained with a large-scale dataset to learn multilingual language proficiency and general world knowledge and a short context window. Then with an increased learning rate decay, STEM, coding, and reasoning data are introduced. Finally, the model’s context length is increased by training with a smaller dataset containing longer context length data. (Yang et al. 2025)

Empirical results from a wide range of benchmarks imply the effectiveness of this approach (Zhao et al. 2025; Minaee et al. 2025). Pre-trained models achieve state-of-the-art performance on numerous NLP tasks with little or no fine-tuning, demonstrating strong zero-shot and few-shot capabilities (Brown et al. 2020; Wei et al. 2022a).

### 2.3.1 Training objectives

Due to the extensive availability of unlabeled text data, modern language models are typically trained using self-supervised learning objectives that do not rely on manual annotation. These objectives allow the model to learn semantic representations directly from raw text. Naturally, the choice of training objective plays a central role in shaping the model’s abilities and biases.

**Masked language modeling (MLM)** Early Transformer-based models, such as BERT (Devlin et al. 2019), employed the masked language modeling (MLM) objective to train deep bidirectional representations. Due to the ability to capture context from past and future tokens, bidirectional representations are particularly effective for tasks such as classification and question answering.

In masked language modeling, a random subset of input tokens is masked, and the model is trained to predict the original tokens given their surrounding (unmasked) context. Let  $x = (x_1, x_2, \dots, x_T)$  denote the input sequence, and let  $\mathcal{M} \subset \{1, \dots, T\}$  be the set of masked token positions. The model receives a corrupted version of the input sequence, denoted  $\tilde{x}$ , in which tokens at positions in  $\mathcal{M}$  are modified. The loss function is defined as

$$\mathcal{L}_{\text{MLM}}(\theta) = - \sum_{t \in \mathcal{M}} \log P(x_t | \tilde{x}; \theta), \quad (1)$$

where  $\tilde{x}$  is the corrupted input and  $\theta$  are the model parameters.

To enable robust learning while reducing pre-train–fine-tune discrepancy, BERT uses a probabilistic masking strategy: 15% of tokens are selected for prediction. Of these, 80% are replaced with the special [MASK] token, 10% are replaced with a random token, and 10% are left unchanged. This prevents the model from overfitting to the [MASK] symbol, which does not appear after pre-training. Unlike previous approaches like denoising autoencoders (Vincent et al. 2008), MLM models like BERT only predicts the masked tokens rather than reconstructing the entire input.

As a result, most MLM-based models adopt an encoder-only Transformer architecture, which fully attends to all input positions. This formulation is well-suited to understanding text, but it is not trained to generate text in an autoregressive way.

**Autoregressive next-token prediction** Large-scale decoder-only models such as GPT-4, LLama 3, Qwen3 (OpenAI et al. 2024a; Grattafiori et al. 2024; Yang et al. 2025) adopt an autoregressive next-token prediction objective. The model is trained to predict the next token in a sequence given all tokens that precede it. Let  $x = (x_1, x_2, \dots, x_T)$  denotes an input sequence of length  $T$ . The objective is to maximize the likelihood of the sequence under the model’s parameters  $\theta$ , which corresponds to minimizing the negative log-likelihood:

$$\mathcal{L}_{\text{NTP}}(\theta) = - \sum_{t=1}^T \log P(x_t | x_{<t}; \theta), \quad (2)$$

where  $x_{<t} = (x_1, \dots, x_{t-1})$  represents the preceding tokens of the input sequence up to position  $t - 1$ , and  $P(x_t | x_{<t}; \theta)$  is the model's estimated probability of the next token  $x_t$  given the preceding context. The autoregressive approach trains the model to generate coherent text by predicting the next token only on the preceding context, without relying on future information. This reflects the natural, sequential structure of language.

### 2.3.2 Pre-training data

State-of-the-art LLMs are typically trained on massive corpora of text. For example, GPT-3 was trained on 500 billion tokens (Brown et al. 2020), while more recent models like Qwen3 (Yang et al. 2025) was trained on a total of 36 trillion tokens. These corpora are diverse by design, drawing from a wide range of sources to ensure linguistic diversity and domain coverage.

The training datasets include several key source types:

- **Web-scale crawls:** Filtered web data from sources such as Common Crawl (Common Crawl 2025) provide large volumes of general-domain text (Brown et al. 2020).
- **Curated text corpora:** High-quality sources such as books, Wikipedia and academic articles content is included to ensure linguistic richness and coherent structure (Raffel et al. 2023).
- **Domain-specific and technical texts:** Text from domains like programming (e.g., GitHub), STEM fields, and forums enhance robustness in specialized tasks.
- **Multilingual corpora:** Text in many languages enables cross-lingual generalization, with recent models covering over 100 languages.
- **Synthetic data:** A growing portion of training corpora consists of model-generated text, including textbooks, instruction-following data, Q&A, and code. This synthetic content is often generated using other LLMs.

Data preprocessing includes deduplication, language identification, and de-noising steps to enhance quality and reduce redundancy. Sampling strategies are used to balance domain representation. Because of the massive dataset size, models are typically trained for only one epoch over the corpus using token sampling with replacement.

To enhance reasoning capabilities and score high on common benchmarks, the data composition is often constructed to optimize performance on coding, mathematics and reasoning tasks.

### 2.3.3 Scaling Laws

Work by Kaplan et al. (2020) has shown that model performance scales with increases in parameter count  $N$ , dataset size  $D$ , and training compute  $C$ , while architectural design choices like depth versus width have minimal impact on performance. In particular, the cross-entropy loss was found to follow a power law decay with respect to increases in  $N$ ,  $D$ , and  $C$ . These power laws hold at all scales, although loss can never reach zero.

The authors further propose a universal overfitting law showing that scaling  $N$  and  $D$  together yields predictable performance improvements, whereas fixing one while increasing the other leads to diminishing returns with the ratio  $N^{0.74}/D$ . This means that increasing the model size by a factor of eight requires increasing the data by a factor of approximately five to maintain efficiency. Training dynamics follow predictable power law trends that are largely independent of model size, allowing final performance to be estimated from early stages of training (Kaplan et al. 2020). This property is used to first train smaller models to identify hyperparameters, learning schedules, or compute allocation strategies, enabling efficient training of larger models without the risk and cost of training a large model blindly (Grattafiori et al. 2024).

## 2.4 Post-training

Post-training refers to the class of techniques applied after pre-training to improve LLM behavior and performance for deployment. These techniques include supervised fine-tuning, alignment via human feedback, distillation, and reasoning-focused adaptations (Tie et al. 2025). This section summarizes recent developments in each of these areas as well as efficient implementations.

### 2.4.1 Supervised fine-tuning and instruction tuning

Supervised fine-tuning (SFT) (Ouyang et al. 2022) continues pre-training-like gradient-based training on labeled input-output pairs  $(x, y)$  to specialize the model for particular downstream tasks. The standard objective minimizes the negative log-likelihood:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(x,y)}[\log P_{\theta}(y | x)], \quad (3)$$

where  $P_{\theta}$  denotes the model distribution parameterized by  $\theta$ . This technique helps the model learn to produce responses that match the ones found in the training data.

As LLMs are initially only trained to predict the next word, they do not by default excel in instruction following. Instruction tuning is introduced to fix this issue. Instruction tuning is a variant of SFT, where the fine-tuning dataset consists of (instruction, response) pairs across multiple domains. Unlike single-task fine-tuning, instruction tuning exposes the model to various ways users may formulate goals or requests, thus improving general instruction-following and zero-shot task performance (Wei et al. 2022b). Instruction-tuned models demonstrate superior generalization on unseen tasks compared to models trained only on domain-specific objectives. Instruction tuning is the backbone of deployed LLMs like chatbots and agents.

### 2.4.2 Alignment via human feedback

Instruction tuning or task-specific training alone does not guarantee that a model’s behavior aligns with human preferences. Reinforcement learning from human feedback (RLHF) (Y. Bai et al. 2022) addresses this by introducing a reward-based optimization stage. The core idea of RLHF is to train a language model to produce outputs that align with human preferences by introducing a reward model  $R_\phi$  that predicts human preferences. The model is trained on human feedback data. The reward model is used to fine-tune the base LLM to prefer outputs that are evaluated as more desirable.

The most commonly used optimization method in RLHF is Proximal Policy Optimization (PPO) (Brown et al. 2020; Ouyang et al. 2022; Grattafiori et al. 2024). PPO is a policy gradient method that improves stability by restricting the updated policy to remain close to a reference policy. In the context of RLHF, the policy  $\pi_\theta(y | x)$  defines a probability distribution over output sequences  $y$ , given an input prompt  $x$ , corresponding to the language model’s generation.

The standard RLHF objective is given by:

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \mathbb{E}_{y \sim \pi_\theta(\cdot | x)} [R_\phi(x, y)] - \beta \text{KL}(\pi_\theta(\cdot | x) || \pi_{\theta_0}(\cdot | x)), \quad (4)$$

where  $\mathcal{D}$  is a dataset of prompts,  $\pi_\theta$  is the trainable policy,  $\pi_{\theta_0}$  is the reference (pre-trained) policy,  $R_\phi(x, y)$  is the reward model parameterized by  $\phi$ , and  $\beta$  is a hyperparameter that controls the strength of the KL penalty to limit deviation from the reference policy.

### 2.4.3 Multi-step reasoning

Reasoning in the context of large language models refers to the ability to solve tasks that require decomposing a problem into intermediate steps before reaching a final solution. Such multi-step reasoning is essential for domains like mathematics, commonsense inference, and symbolic manipulation.

A widely adopted approach for implementing reasoning behavior is *Chain-of-Thought* (CoT) prompting (Wei et al. 2023). In CoT, the model is explicitly encouraged to generate intermediate reasoning steps before producing the final answer. Wei et al. (2023) demonstrated that providing only a small number of CoT examples was sufficient for reasoning to emerge in very large models: for example, a 540B-parameter PaLM model trained with just eight examples reached strong accuracy on the GSM8K mathematical word-problem benchmark, outperforming fine-tuned GPT-3 baselines.

Kojima et al. (2023) further extended these findings by introducing zero-shot CoT. They showed that simply appending a short instruction such as "Let’s think step by step" to the prompt could substantially improve reasoning performance, boosting accuracy on GSM8K from roughly 10% to over 40%. These results show that prompting strategies can dramatically enhance reasoning capability and model performance without additional parameter updates.

Building on these insights and motivated by the development of the proprietary model OpenAI o1 (OpenAI et al. 2024c), more recent research has explored whether reasoning can be induced without relying on supervised annotations. DeepSeek-R1

(DeepSeek-AI et al. 2025a) demonstrates that reinforcement learning alone can give rise to strong reasoning capabilities. The central innovation is Group Relative Policy Optimization (GRPO) (Shao et al. 2024), a variant of PPO that eliminates the need for a learned value critic by standardizing rewards relative to a group of sampled outputs.

For each input state  $s$ , the policy  $\pi_{\theta_t}$  samples a group of candidate outputs  $\{a_i\}_{i=1}^G$ . Raw verifiable rewards  $r(s, a_i)$ , such as the correctness of a math problem, are computed for each task, and a group-relative advantage is defined as

$$A^{\pi_{\theta_t}}(s, a_i) = \frac{r(s, a_i) - \mu}{\sigma},$$

where  $\mu$  and  $\sigma$  denote the mean and standard deviation of the group rewards. The policy is then updated according to a clipped PPO objective:

$$\frac{1}{G} \sum_{i=1}^G \min\left(\frac{\pi_{\theta}(a_i|s)}{\pi_{\theta_t}(a_i|s)}, 1 + \epsilon\right) A^{\pi_{\theta_t}}(s, a_i),$$

where the ratio  $\frac{\pi_{\theta}(a_i|s)}{\pi_{\theta_t}(a_i|s)}$  measures how the probability of each response changes under the new policy. The clipping term prevents overly large updates by restricting how much the policy can increase or decrease the likelihood of an action in a single step. Symmetric clipping is applied for negative advantages to avoid amplifying poor responses. In addition, a KL penalty is included to keep the updated policy close to a reference model.

By rewarding outputs that perform better than a model’s own alternatives rather than optimizing absolute scores, GRPO mitigates reward scaling issues, avoids the need for an explicit value network, and reduces both computational and memory overhead.

#### 2.4.4 Distillation

The idea of distillation is to distill the information of a teacher model into a student model. There are two common approaches to distillation: knowledge distillation and context distillation. Although both approaches share the same underlying principle of transfer learning from a teacher to a student, their intentions are different.

**Knowledge distillation** (Hinton et al. 2015; Sanh et al. 2020) In knowledge distillation a superior teacher model is distilled into the weights of a smaller model. In the context of LLMs, knowledge distillation is often used to transfer the knowledge and reasoning capabilities of a larger teacher model into a smaller student model (Mukherjee et al. 2023). This helps the student capture the correct answers as well as the teacher’s uncertainty and reasoning structure. Knowledge distillation is often used to improve the overall performance of the student instead of fine-tuning it for a specific downstream task.

**Context distillation** (Hinton et al. 2015) Context distillation is a method where a language model is fine-tuned on its own outputs, typically sampled from prompts that contain useful information such as behavioral demonstrations, reasoning steps, or factual knowledge. The same model serves as both teacher and student, allowing new capabilities to be internalized without requiring additional model components.

Early work focused on behavioral alignment through distillation. Askill et al. (2021) demonstrated that desirable human-aligned conversational traits—such as politeness, helpfulness, and factual accuracy—could be distilled by fine-tuning on multi-turn conversations in which the assistant always followed a preferred policy. Choi et al. (2022) extended this to short prompts that defined personas or injected lightweight task instructions, showing that prompt-based behavior is learnable.

Later research applied context distillation to structured reasoning and task generalization. Snell et al. (2022) showed that more complex prompts containing task instructions, in-context examples, and chain-of-thought reasoning could be distilled into the model, allowing it to perform structured reasoning without requiring explicit prompting at inference time.

**Prompt distillation** A natural extension of context distillation is prompt distillation, in which factual knowledge is injected into model weights by fine-tuning on outputs conditioned on additional information or hints. In this setting, the teacher model is given access to additional context and asked to answer questions or summarize information based on the hints or information, while the student learns to replicate these outputs without access to the additional context. Kujanpää et al. (2025) and Alakuijala et al. (2025) show that prompt distillation can match or even exceed the performance of retrieval-augmented generation (RAG) and supervised fine-tuning (SFT).

Prompt distillation with an open-source model is advantageous because the teacher and student logits are directly comparable. This makes loss calculation particularly simple. As shown by (Kujanpää et al. 2025), the loss can be calculated as the Kullback–Leibler divergence between the teacher and student:

$$\mathcal{L}_{\text{distill}} = \frac{1}{N} \sum_{i=1}^N \text{KL} \left( \pi_{\text{teacher}}(a_i \mid c_T, q, a_{<i}) \parallel \pi_{\text{student}}(a_i \mid q, a_{<i}) \right)$$

Where  $N$  is the number of token positions in the answer sequence,  $\pi_{\text{teacher}}(a_i \mid c_T, q, a_{<i})$  is the probability distribution over the  $i$ -th answer token produced by the teacher, conditioned on privileged context  $c_T$ , the prompt  $q$ , and previous tokens  $a_{<i}$ ,  $\pi_{\text{student}}(a_i \mid q, a_{<i})$  is the student’s corresponding distribution, given only the base prompt  $q$  and  $\text{KL}(\cdot \parallel \cdot)$  denotes Kullback–Leibler divergence.

#### 2.4.5 Efficiency in post-training

Full fine-tuning of a LLM can be very expensive because of the parameter count reaching billions. Parameter-efficient fine-tuning (PEFT) techniques address this by updating only a small subset of model parameters. The most commonly adopted

strategy is using Low-Rank Adaptation (LoRA) (Hu et al. 2021), where the pre-trained model weights are frozen during training and trainable rank decomposition matrices are added into layers of the transformer architecture.

Let  $W_0 \in \mathbb{R}^{d \times k}$  denote a frozen weight matrix in the transformer. Instead of updating  $W_0$  directly, LoRA introduces trainable low-rank update matrices  $\Delta W$  parameterized as a product of two low-rank matrices:

$$W = W_0 + \Delta W, \quad \Delta W = BA,$$

where  $A \in \mathbb{R}^{r \times k}$  and  $B \in \mathbb{R}^{d \times r}$  with  $\text{rank } r \ll \min(d, k)$ . Only  $A$  and  $B$  are modified during training, while  $W_0$  remains fixed. This significantly reduces the memory requirement and number of trainable parameters during training. LoRA is shown to achieve comparable results to full fine-tuning, where none of the model parameters are frozen (Hu et al. 2021). In practice, multiple LoRA adapters are added throughout the model to maximize the efficiency of learning. LoRA is used with most of the previously mentioned fine-tuning and distillation methods.

## 2.5 Vision language models (VLMs)

Vision language models (VLMs) are LLMs that combine visual and textual inputs to gain proficiency in multi-modal comprehension and generation. These models excel in tasks like image captioning, visual question answering, grounding, and zero-shot image classification. Similarly to LLMs, VLMs have been driven by large-scale pretraining with the addition of image-text datasets, enabling the models to learn rich multi-modal representations. Architecturally, VLMs typically integrate a visual encoder to the language model backbone to seamlessly integrate visual perception with natural language.

### 2.5.1 Architectural changes

Most contemporary vision-language models (VLMs) adopt a modular *ViT-MLP-LLM* architecture. The visual encoder is typically a Vision Transformer (ViT) (Dosovitskiy et al. 2021), which applies the transformer architecture to image data. The ViT is then connected to the backbone LLM using a connector, most often a simple multilayer perceptron (MLP).

The ViT processes an input image  $x \in \mathbb{R}^{H \times W \times C}$ , where  $H$ ,  $W$ , and  $C$  denote the height, width, and number of channels. The image is first divided into non-overlapping square patches of size  $P \times P$  pixels. Each image patch is first flattened into a 1-dimensional vector  $p_i \in \mathbb{R}^{P^2 C}$  and then passed through a learnable linear layer  $W_p \in \mathbb{R}^{(P^2 C) \times D}$  to obtain a smaller embedding  $z_i = p_i W_p \in \mathbb{R}^D$ .

This produces a sequence of  $N = \frac{HW}{P^2}$  patch embeddings  $\{z_1, z_2, \dots, z_N\}$ , which are treated similarly to token embeddings in language models. A positional encoding is then added to each  $z_i$  to retain spatial structure, and the resulting sequence is passed through a series of transformer layers, producing a contextualized sequence of visual tokens.

To interface with a language model, the output of the ViT is passed through a connector module, typically a small MLP. This module projects the visual token embeddings into the embedding space of the LLM. The transformed visual tokens are then integrated with the text tokens, allowing the LLM to process both modalities within a shared token space for unified multi-modal processing.

### 2.5.2 Training VLMs

There are multiple training strategies for integrating a ViT, a connector module (MLP), and a LLM. These vary in which components are frozen or jointly trained. Some approaches include training only the connector while keeping the pre-trained ViT and LLM frozen (Li et al. 2023; Zhu et al. 2023), tuning the LLM while keeping the ViT frozen (Haotian Liu et al. 2023) and hybrid approaches where different parts of the model are frozen at different stages of training (Lu et al. 2024; S. Bai et al. 2025).

Similarly to LLM training, the objective of training a VLM is autoregressive next-token prediction. For VLMs, the training sequence typically includes both visual and textual tokens. The model is trained to predict the next token in this multi-modal sequence, enabling it to generate coherent and grounded text responses conditioned on visual context.

This formulation allows multiple supervision signals, such as image captioning, visual question answering, object detection, and instruction following, under the same autoregressive loss. As a result, the model benefits from the scalability and generality of large-scale language modeling as well as acquires the capacity for open-ended generation over visual inputs.

After such training, the model can follow natural-language instructions about images, describe scenes, answer visual questions, localize objects and regions, do optical character recognition, analyze charts and tables, and retrieve or ground objects. Recent models (OpenAI 2025; X. Zhang et al. 2025) have also experimented with visual reasoning, where the model performs zooming, rotation, and cropping of images by calling external tools within its own reasoning monologue to improve accuracy on hard visual tasks.

## 3 Large language models as agents

While LLMs are models trained to generate text based on input prompts, agents are autonomous entities that interact with and navigate their environment dynamically. This means that rather than just responding to static inputs, agents operate in an environment that provides feedback in response to the agent’s actions. The environment could be a web browser, a simulated game, or a Python interpreter, as long as it serves as a dynamic source of information that the agent can interpret and act upon in order to progress toward its goals.

To function autonomously in an environment, an agent must integrate four capabilities: a foundation model (LLM) for general-purpose reasoning, a planning mechanism

to structure goal-directed behavior, a memory system to retain and retrieve relevant context, and a set of tools or action interfaces to interact with the environment.

### 3.1 CoT reasoning as planning

In traditional LLM applications, responses are generated in a single forward pass, often without explicit intermediate reasoning. While this suffices for simple tasks, more complex, multi-step tasks require the model to decompose the task into subgoals and reason through them sequentially. As mentioned in 2.4.3, Chain-of-Thought (CoT) reasoning (Wei et al. 2023) offers a simple yet effective approach for encouraging models to produce intermediate reasoning steps by “thinking aloud” before arriving at a final answer. Instead of generating an immediate response, the model is guided to lay out a sequence of logical or procedural steps, resembling the way humans solve complex problems.

When dealing with LLM-based agents, the agent can be encouraged to outline a clear plan within its reasoning. This reasoning trace then functions as a basic form of planning, forming a structured path toward solving the main task. When combined with tool-use or environmental feedback, CoT enables the model to guide its actions more deliberately, adjusting its plan based on intermediate outcomes. Importantly, this method requires no changes to the model architecture and can be applied to any sufficiently capable language model through prompt engineering.

### 3.2 Tool-use

As previously discussed, modern LLMs acquire key capabilities such as instruction following, deliberate reasoning, and multi-modal understanding through a combination of large-scale pre-training and post-training techniques. Despite their impressive capabilities, language models remain fundamentally limited by their training data and architecture. Errors commonly arise due to several factors, including the model’s knowledge cut-off, which prevents access to recent or real-time information (Brown et al. 2020) and lack of reliable arithmetic or symbolic reasoning abilities (Parisi et al. 2022). These limitations are particularly problematic in agentic settings, where completing tasks often requires accurate computation, dynamic information retrieval, or interaction with external environments.

A natural step toward enabling agents to overcome these limitations is to equip them with tool-use capabilities via simple APIs (Qin et al. 2023; Schick et al. 2023). These tools are tailored to help with previously mentioned limitations and include calculators (Qin et al. 2023), code interpreters (Gao et al. 2023) and web browsers (Nakano et al. 2022). The LLM issues tool calls by producing special textual outputs that specify the tool name and its arguments. Once the tool is executed, the resulting output is appended to the model’s context, allowing following responses to be conditioned on the new information. To enable efficient tool use, the prompt is augmented with textual descriptions of the available tools, accompanied by a small number of usage examples.

To enable effective tool use, the agent is typically provided with textual descriptions of the available tools, along with a small number of usage examples within the prompt.

The model can also be fine-tuned to better follow tool-related instructions or output formats.

### 3.3 ReAct as a basic agent framework

While simple tool-use extends the capabilities of LLMs, more complex tasks often require structured decision-making, where reasoning and interaction with tools occur over multiple steps. This is especially true in agentic tasks, where each action depends on prior reasoning steps that guide future decisions. ReAct (Reason + Act) (Yao et al. 2023) addresses this need by introducing a prompting framework that unifies CoT reasoning with action execution and feedback from the environment. Rather than issuing tool calls in isolation, a ReAct-augmented model alternates between internal reasoning and external actions, creating an interaction loop. In each iteration, it generates a reasoning trace, decides on a tool-based action, and receives the change in its environment as feedback. This feedback becomes part of the next input, allowing the model to update its beliefs and plan the next step.

ReAct augments the foundation language model with three core capabilities to be a complete agentic framework: a simple memory mechanism through the feedback loop, planning through chain-of-thought reasoning, and action execution via tool use. This framework enables the model to gain agency by being able to autonomously complete tasks within its environment.

### 3.4 Vision-based agents

While most LLM-based agents operate purely on textual inputs, recent advances have extended their capabilities to visual domains. Vision-based agents are systems that incorporate images or visual observations into their reasoning process, allowing them to act based on both text and visual input. This enables a broader range of feasible tasks, such as interpreting diagrams, analyzing screenshots, navigating interfaces, or making decisions based on camera input.

Despite these benefits, vision-based agents face considerable limitations. One key issue is visual consistency. Vision-based agents reference objects not present in the image, fail to remember visual details across steps, or rely too heavily on language priors, leading to hallucinations (Li et al. 2023; Hanchao Liu et al. 2024). This makes multi-step interaction particularly fragile, as each round of perception and reasoning risks drifting from the original input. This problem is not as prominent in text-only agents, as locating tokens from a text document is far more robust compared to visual object detection. As a result, vision-based agents often struggle with reliability in practice (Zheng et al. 2024).

## 4 Methods

The goal of this work is to improve the capabilities of a LLM-based web agent operating in a visual interaction framework. While untrained VLM-agents can reason, act,

and observe in an interactive setting, they often fail to complete simple web-based tasks reliably. We hypothesize that the agent’s performance is bottlenecked by its inconsistent ability to recognize and reason about visual interface elements, such as the cursor and interactive elements on the webpage. By improving the agent’s capacity to interpret these elements from visual input, we expect to enable more accurate action selection and task completion.

We begin by describing the agent and the agentic framework. Following this, we explain the data and training setup, in particular the prompt distillation process adapted for vision-based agentic tasks, which is the core contribution of this thesis. Two distinct training objectives are described, each designed to reinforce different aspects of generalization and performance. Finally, we evaluate the agent using our single-click test dataset designed to measure real-world task competence.

## 4.1 VLM backbone

Our agent is built upon the Qwen2.5-VL-32B (S. Bai et al. 2025) model — a 32-billion-parameter multimodal vision-language model developed by Alibaba’s Qwen team. Qwen2.5-VL-32B provides a good balance between performance and efficiency. It is powerful enough to be fine-tuned for our task, yet compact enough to allow for faster training and easier deployment compared to larger variants such as Qwen2.5-VL-72B. We use the base model Qwen2.5-VL-32B as our baseline in our experiments.

Notably, Qwen2.5-VL-32B has received specialized training in object detection and visual grounding, allowing it to identify and localize interface components such as cursors and buttons. Moreover, it is capable of producing and following structured outputs and engaging in step-by-step reasoning about visual scenes. The model also exhibits robust performance on purely textual tasks, underscoring its versatility beyond multimodal contexts. For instance, Qwen2.5-VL-32B achieves competitive scores of 78.4 on MMLU, 82.2 on the MATH benchmark, and an impressive 91.5 on HumanEval, outperforming competing small models like GPT-4o-Mini (S. Bai et al. 2025). In terms of vision–language tasks, it surpasses GPT 4o-0513 on several benchmarks—scoring 70.0 on MMMU, 74.7 on MathVista, and 94.8 on DocVQA.

## 4.2 Environment

The agent operates within an interactive Python environment implemented using an iPython kernel. This workspace allows the language model to generate and execute Python code, with the resulting outputs such as return values, printed messages, or error messages captured in the session history. The environment maintains state across steps, enabling the agent to build on previous actions as it progresses through the task.

Using a general-purpose programming environment instead of a fixed tool interface provides greater flexibility (X. Wang et al. 2024). The agent is not restricted to a predefined set of actions but can issue any Python code to interact with the environment. While this introduces more variability in behavior, the ability to observe execution results and adjust accordingly helps improve robustness.

We chose Python due to its popularity, ease of use, extensive library coverage, and widespread adoption. Since language models are typically trained on large volumes of Python code, the model can produce syntactically correct code without additional adaptation. In particular, we use the iPython kernel that is convenient due to the cell-based execution model and workspace memory that allows for stepwise execution without recompilation, which is essential for agents.

Inside Python, we implemented a vision-enabled web browser as an extension of an environment developed by System 2 AI. Using an actual web browser interface instead of plain images allows the agent to get feedback from its actions. We built the web browser environment on Playwright, which by itself provides core interaction possibilities such as clicking and scrolling. Importantly, the system can capture screenshots of the current page layout. We employ mouse movement by overlaying a cursor icon on the captured screenshot in the location specified by the agent. To enhance visual reasoning (OpenAI 2025; Shen et al. 2024; X. Zhang et al. 2025), we employ a zooming functionality that lets the agent zoom into regions of the webpage with a specified command. The full action space of our vision web browser is shown in table 1. Additionally, when the agent is finished with the task it calls a specified `complete_task()` function.

**Table 1:** Vision Web Browser control functions used by the agent

Function	Description
<code>async def goto(url: str)</code>	Open the page at the given URL.
<code>async def mouse_move(x: int, y: int)</code>	Move the mouse to the specified coordinates.
<code>async def mouse_click()</code>	Click the mouse at the current cursor position.
<code>async def scroll(heights: float = 1.0)</code>	Scroll up if <code>heights &lt; 0</code> or down if <code>heights &gt; 0</code> by the given number of window heights.
<code>async def keyboard_press(key_name: str)</code>	Trigger the action for the key (e.g., Enter submits, PageDown scrolls).
<code>async def keyboard_type(query: str)</code>	Type the string at the current cursor location.
<code>async def back()</code>	Return to the previous page.
<code>async def zoom(bounding_box: list)</code>	Zoom to <code>[x1, y1, x2, y2]</code> ; useful for focusing on details or obtaining precise coordinates for <code>mouse_move</code> .

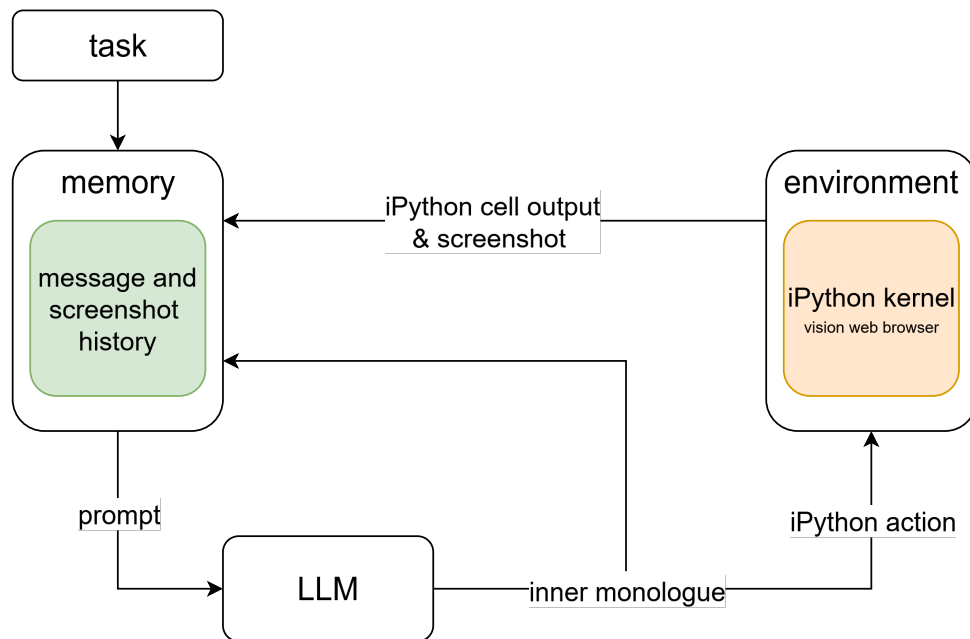
Technically the agent is free to execute any code inside the Python environment. One could imagine this would cause problems with the behavior of the agent, but the feedback loop that allows the agent to inspect the outputs of its code makes it self-correcting. We further constrain behavior by instructing the agent to only interact with the vision web browser. Since Qwen2.5-VL-32B is a strong baseline model that

follows instructions reliably, the ambiguity of the environment is reduced.

Even though the agent itself only has access to the visual interface, the vision web browser enables access to the HTML structure to enable data generation and task verification. Data generation is further discussed in the Sections 4.4.1, 4.5.2.

### 4.3 Agentic framework

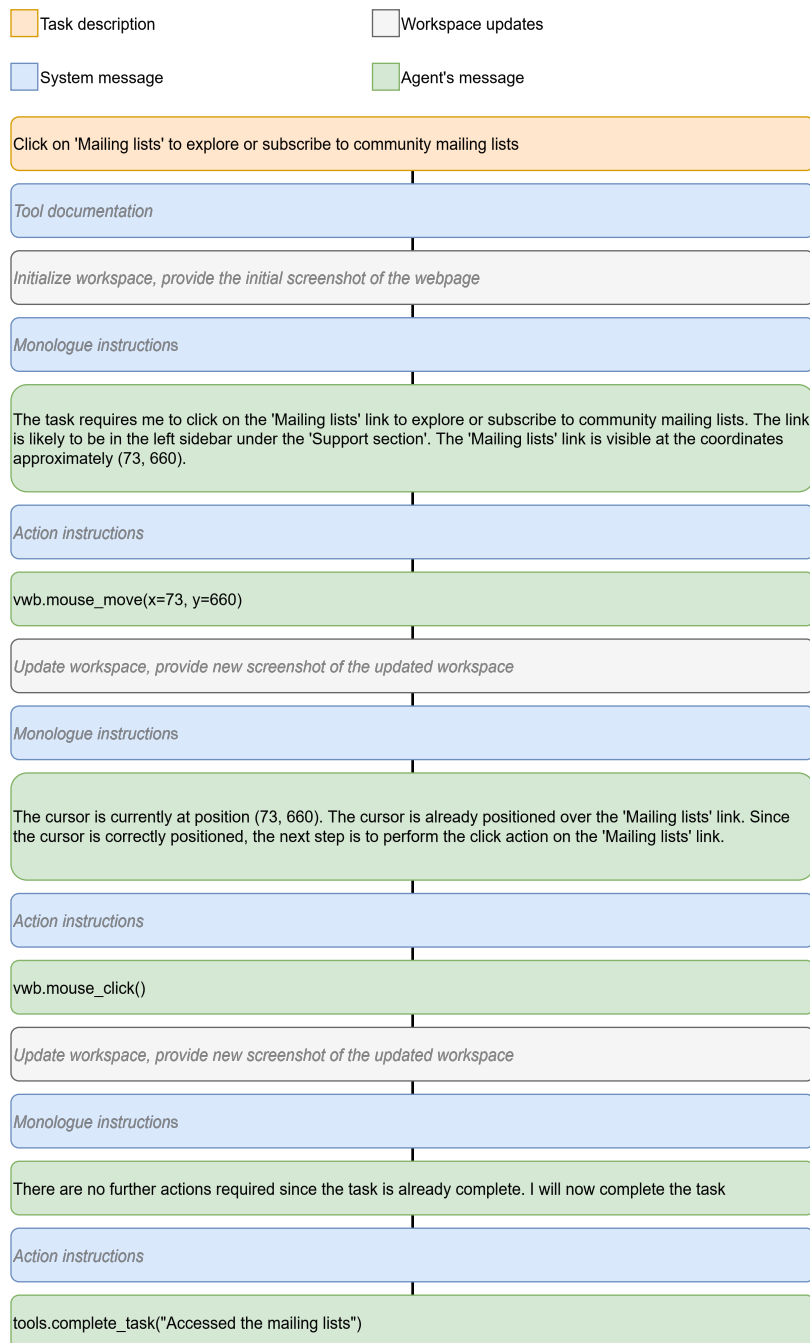
Our agent is based on a simple ReAct architecture. Given a task, a system prompt, and a screenshot of the current state of the webpage, the agent is encouraged to generate a reasoning trace as its planning and executable code as its action. The action is then executed, and the modified environment state is appended to the context of the agent as a screenshot of the current state of the webpage. Consequently, as the whole history is visible during each agent call, it acts as the memory for the agent. This is then continued in a loop until the agent decides the task is finished and calls `complete_task()` function to close the loop. The loop is illustrated in Figure 4.



**Figure 4:** Diagram of the agentic framework. Memory includes the task, prior inner monologue, screenshots, and code outputs. The LLM consumes this memory as a prompt to first produce an inner monologue and then a Python action, which is executed by an iPython kernel. Resulting outputs and screenshots are fed back into memory, closing the loop.

To ensure that the agent’s memory and output is both interpretable and structured, all information passed to and generated by the LLM is segmented using explicit XML-style tags. Each type of content is enclosed within dedicated tags (e.g., `<think>...</think>`, `<ipython_cell>...</ipython_cell>`). This formatting enables the model to distinguish between internal thoughts, executable actions, and environmental feedback, which is essential for coherent decision-making in a multi-step reasoning loop. By clearly

separating sections in the prompt, the agent is less likely to confuse thought with action or outputs with inputs. To enforce the correct response format, each request to the agent specifies the expected format of the next response. An example workflow of the agent is illustrated in Figure 5.



**Figure 5:** A simplified example of the agent's workflow

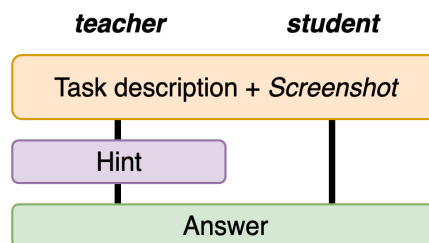
## 4.4 Training

Parameter-driven optimization of LLM-based agents is commonly done through reinforcement learning or fine-tuning (Du et al. 2025). In this thesis, we will be focusing on direct fine-tuning. The current limitations of vision-language models (Sec 3.4) lead us to hypothesize that improving the agent’s visual localization and grounding capabilities within the web browser context will improve overall task performance. The agent should not be restricted by its technical capabilities of localizing the current cursor position and target elements to complete tasks. We aim to mitigate this problem by specifically fine-tuning the model to localize user interface elements and the cursor position to support accurate mouse actions.

### 4.4.1 Training data

To support grounded visual learning in the browser environment, we construct our own training datasets using a collection of 1138 real-world webpages. Instead of training the agent on the full trajectory presented in Section 5, we train the agent on single-step tasks that mimic the inner monologue and action of the agent. All training datasets are based on this shared URL set but differ in how the cursor and target elements are positioned and annotated. We define three training setups: *cursor detection*, *element detection*, and *golf*.

All of the training setups follow the same single-step question-answering format that is visualized in Figure 6. Each student and teacher trajectory is created from a screenshot of the rendered webpage and a structured task description with the hint that is given to the teacher and hidden from the student. To generate the tasks and screenshots, we extract three visible and clickable HTML elements from each collected URL. To improve data quality, we automatically filter out elements with excessively large bounding boxes (e.g., banners, overlays), followed by manual filtering to remove items that are semantically irrelevant, visually ambiguous, or redundant. This ensures that the training signals focus on meaningful user-interactive components.



**Figure 6:** All student and teacher trajectories are created using a simple single-step question-answering format. The task descriptions and hints vary across training setups.

In the *cursor detection* training setup, the cursor is positioned directly on top of the target element. The model is asked to localize the cursor and identify the underlying element. To improve robustness, the dataset also includes examples where the cursor is placed in a corner or is completely absent from the image. Additionally, we added a

description of the appearance of the cursor to the task description to make the model understand what the cursor looks like. This setup is designed to improve the model’s capabilities on locating the cursor and describing different elements on a webpage. Because the cursor position within the target element’s bounding box is uniformly sampled, it also trains the model to infer bounding boxes of elements. This task is evaluated using our baseline model Qwen2.5-vl-32B as a judge.

In the *element detection* training setup, the goal is to find the coordinates of the bounding box of a target element. To simulate a random step and not just the initial step, the cursor is placed at a random position uniformly sampled from the screen. The model is then asked to localize the target element and provide visual reasoning for how it found the coordinates of the element. Any coordinates that are inside the target element’s bounding box are valid. This training objective trains the model to directly localize target elements without paying attention to the cursor position. We evaluate the task programmatically by checking if the provided coordinates are inside the target element’s bounding box.

In the *golf* setup, the objective is to localize the cursor and target element and perform a `mouse_move(x, y)` or `mouse_click()` action. To accommodate this, we extend the task description to mimic the system prompt in our agentic environment. We include a text that informs the agent it is supposed to interact with the vision web browser by using its methods `mouse_move(x, y)` or `mouse_click()`.

For each task, the cursor is initialized at an offset from the target center, with horizontal and vertical displacements sampled independently from normal distributions:

$$dx \sim \mathcal{N}(0, \sigma_x^2), \quad dy \sim \mathcal{N}(0, \sigma_y^2),$$

where the standard deviations are set by the element dimensions,

$$\sigma_x = 2 \cdot \text{elem\_width}, \quad \sigma_y = 2 \cdot \text{elem\_height}.$$

By exposing the model to a wide range of starting distances, the model should learn to move the cursor from any distance to the target element. Consequently, the model would learn to iteratively move the cursor closer and closer in case a misclick happens. This resembles how golf is played. Additionally, the setup closely mimics the agentic framework, where the model needs to execute actions to interact with its environment.

Similarly to element detection, the golf setup is evaluated programmatically by verifying the action type and the predicted coordinates.

#### 4.4.2 Training procedure

We apply a fine-tuning technique called prompt distillation (Kujanpää et al. 2025). Originally developed for knowledge injection, prompt distillation uses a teacher–student setup in which the teacher’s guidance is distilled into a student model. This approach has been successfully adopted to train agents through hints internalization (Alakuijala et al. 2025) by guiding the agent’s behavior.

In our training setup, we provide the teacher model with a hint that contains the ground-truth answer and formatting instructions. This enables the teacher to generate

high-quality, informative outputs that include the correct answer and follow expected formatting. Additionally, we instruct the teacher to verbalize its reasoning before answering our question. This structure ensures that the answer is logically grounded in the inner monologue of the teacher. The student model’s answer is generated without seeing the hint. Although the student does not see the hint itself, it learns from examples in which the intended behavior is clearly motivated by the inner monologue. We use Kullback–Leibler divergence between the student and teacher predictive distributions as a simple loss function. Using the full predictive distributions is advantageous, because the student learns the teacher’s intended behavior and also learns to steer clear of improbable actions (Kim et al. 2021). In practice, the KL-divergence is computed for each generated token and back-propagated to a LoRA adapter of the student model. To accelerate learning, we employ student dropout, in which the student is occasionally given access to the hint during training. Concretely, we use a student dropout probability of  $p = 0.90$ .

In our distillation setup, an important training constraint is that the teacher model must not reveal, either directly or indirectly, that it has access to a structured hint. We do not want to train the student model to think it has access to a hint when in reality it does not. To ensure this, we form the hint to explicitly state that it should not be mentioned or referenced in the generation. Additionally, we noticed that formatting the hint as a "Short answer" instead of "Hint" significantly reduced the generations where the hint was mentioned by the teacher.

To ensure variability in teacher answers, previous work (Kujanpää et al. 2025; Alakuijala et al. 2025) suggests using a high temperature ( $\tau > 1$ ) for teacher answer generation. However, our baseline model Qwen2.5-VL-32B does not perform as expected with high temperatures, as generated answers often have artifacts of Chinese language. Thus, we ended up using a relatively low temperature ( $\tau = 1.1$ ), whereas Kujanpää et al. (2025) used a higher number ( $\tau = 1.5$ ). Additionally, we add hints that are optional for the model to mention in its answer, such as "cursor is far from the element" or "cursor should be moved to the upper right". This results in a wider range of reasoning styles and answer structures, which in turn encourages the student model to generalize beyond specific phrasings. Rather than learning a fixed mapping, the student is exposed to a distribution of valid behaviors.

### 4.4.3 Hardware setup

For all stages of our training pipeline, including data generation, teacher and student trajectory sampling, and model fine-tuning, we utilized the LUMI supercomputer operated by CSC – IT Center for Science. The experiments were conducted on a single node equipped with four AMD MI250x GPUs, each providing 128 GB of memory. This setup offered the necessary computational throughput and GPU memory to handle the demands of multi-modal model fine-tuning and large-scale trajectory generation. In particular, the VRAM capacity was essential for running a 32B model like Qwen2.5-VL-32B efficiently during both inference and training.

## 4.5 Evaluation

Our evaluation protocol is divided into two phases: validation on held-out data from the training distribution and final testing in a distinct agentic framework setting. Each phase provides complementary insights into the capabilities of the trained model, ranging from its internalization of task structure to its robustness in more realistic deployment scenarios.

### 4.5.1 Validation phase

To evaluate how well the student model has learned the desired behavior from the prompt distillation setup, we run it on validation datasets corresponding to each training configuration described in Section 4.4.1. These validation sets follow the same data generation pipeline as their training counterparts but are strictly disjoint in terms of URL origin and element content. Importantly, during validation, the model is evaluated without access to the teacher-provided hint. This setup mirrors the intended deployment conditions and allows us to assess how well the student has internalized the reasoning patterns demonstrated during training.

Each validation dataset is task-aligned with its corresponding training run. For example, the validation set for the *cursor detection* training run contains examples where the model must infer the cursor location from image and layout context alone. Similarly, the validation set for the *element detection* run requires the model to identify the bounding box of a semantically specified target element. This per-setup evaluation allows us to isolate performance improvements stemming from individual training objectives.

The results on these validation sets offer insight into generalization within the training distribution. Since the hints are removed during inference, successful predictions indicate that the model is not merely copying teacher outputs but has in fact learned transferable decision-making behavior. We consider this an essential step in confirming that prompt distillation has led to meaningful behavior acquisition.

### 4.5.2 Testing phase

In the second phase, we evaluate the model’s performance in a real-world setting using our full agentic framework (see Figure 5). Here, the model is deployed as an agent that observes a static HTML snapshot of a webpage and is asked to perform a single-click task specified in natural language. This setting simulates a lightweight interactive agent capable of executing simple browser actions based on textual instructions.

The test dataset contains 178 single-click tasks sampled from a pool of URLs that are disjoint from both training and validation, ensuring that performance reflects generalization to unseen websites and interface designs. Modern webpages often contain layered and overlapping elements. To handle these cases and to verify actions reliably, we rely on two complementary references to the target element: an absolute XPath, which provides a deterministic handle to the DOM node in the static HTML, and a bounding box, which captures the location of the target element even when several elements are stacked.

Each task consists of a rendered static HTML page, a natural language task description, and a ground-truth target annotated with both the absolute XPath and the bounding box. A click is deemed successful if either of the following holds:

- The clicked element’s absolute XPath matches the target element’s XPath.
- The click lands within the bounding box of the target element.

This dual criterion improves robustness to minor rendering differences and DOM ambiguities while ensuring that the agent’s action corresponds to the intended outcome. The final evaluation is binary; the agent either completes the task (score 1) or fails to do so (score 0).

To bound compute and keep the evaluation protocol consistent, we set a fixed budget of at most 20 LLM calls per task. A call is any model invocation within the agentic loop, including planning and action steps. If the budget is exhausted before the agent finishes the task, we terminate the task and immediately score the task based on the current state. This prevents the agent from spending too long on a single task and ensures that all tasks are evaluated within the same computational resources.

Ground-truth annotations are produced automatically with vision-web-browser over static HTML snapshots, which identifies clickable targets and records both their absolute XPaths and bounding boxes. Task descriptions are generated with the Qwen2.5-VL-32B. Because modern websites often provide multiple clickable elements that lead to the same outcome, automated task generation introduces ambiguities. To address this, we manually review each task and the description to check that they are unambiguous, feasible, and aligned with the visible page content. Tasks that do not meet these criteria are deleted.

## 5 Results

In this section, we present the main results of the thesis. We first report training results and performance on the validation data, including loss curves and task performance across the three training setups: cursor detection, element detection, and golf. We then evaluate the model in the agentic framework by measuring its performance on the single-click tasks introduced in Section 4.5.2. In addition to numeric results, we analyze recurring behavioral patterns and common failure cases. Together, these results provide a structured view of what the model learns during training, how learned capabilities translate to interactive evaluation, and where further improvements are needed to develop fully autonomous vision-based web agents.

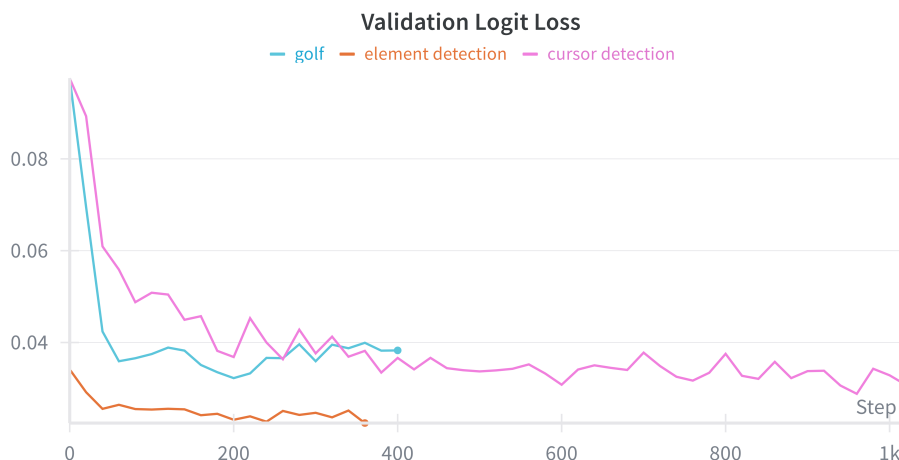
### 5.1 Performance on QA validation data

We start by examining the learning dynamics of each model by analyzing the validation loss curves. Following this, we evaluate the tuned model’s performance on held-out validation data using task-level accuracy in a single-turn QA setting. This allows us to assess whether the student model has internalized the teacher’s reasoning patterns and

whether prompt distillation leads to measurable improvements across training setups. Finally, we analyze adopted behavioral patterns in the single-turn QA settings.

### 5.1.1 Training results

To monitor the learning dynamics of each training configuration, we tracked the KL-divergence loss on held-out validation data during training. The results are shown in Figure 7, which plots the validation logit loss for the *cursor detection*, *element detection*, and *golf* training setups.



**Figure 7:** Validation logit loss across training steps for the three training setups. The element detection setup achieves the lowest and most stable loss, followed by the golf and cursor detection setups.

Due to concerns of overfitting, we employed early stopping for all training runs based on validation loss. Among the three setups, *element detection* exhibited the fastest convergence, reaching a stable loss within the first 200 steps. However, because the initial loss was already low, the overall improvement during training was relatively small. This suggests that the element detection task is well-aligned with the model’s pretraining and provides a clean learning signal.

The *golf* setup demonstrates a sharper initial loss reduction but plateaus rather quickly, similarly to the *element detection* setup. The model reaches a stable loss with a small upward drift, suggesting slight overfitting. Compared to the other runs, the trajectory is less stable. Greater task diversity or additional data would likely improve stability and reduce variance.

In contrast, *cursor detection* converged much more slowly, with loss decreasing steadily throughout the full training window, showing no signs of overfitting even after 1000 steps. The steady downward trend suggests that extended training with more data could yield further gains.

### 5.1.2 Numerical results

We evaluate each trained model on held-out validation data using a structured question-answering (QA) format. In this setting, the model is presented with a natural language question and a screenshot of a webpage and must produce a structured response—such as bounding box coordinates or an action—without access to the teacher-provided hint. This is the exact setup used during training with student dropout probability set to 1. To illustrate the power of hints and potential maximum performance, we also provide the teacher scores. Each question is scored as 1 if correct and 0 otherwise. Table 2 reports the validation accuracy for each setup, comparing the baseline model, the fine-tuned student model, and the teacher.

**Table 2:** Validation accuracy for each training setup evaluated in a single-turn QA format illustrated in Figure 5. We report the accuracy of the untuned baseline model, the fine-tuned student model, and the teacher model.

Task	Baseline	Tuned	Teacher
Cursor detection	0.49	<b>0.73</b>	0.93
Element detection	0.84	<b>0.87</b>	0.99
Golf	0.79	<b>0.94</b>	0.99

As shown in Table 2, all three training setups result in clear improvements over the baseline. The most substantial gain is observed in the *cursor detection* task, where accuracy increases from 0.49 to 0.73. This suggests that the model significantly benefits from explicit supervision on cursor localization, a task where vision-language models typically struggle due to the small size and ambiguous visual appearance of the cursor. Although the tuned model does not reach the teacher’s performance (0.93), the improvement indicates that the student is able to internalize aspects of cursor-target reasoning from teacher demonstrations. The relatively low teacher score is largely attributable to the LLM-as-judge evaluation. Manual inspection of incorrect answers shows that most predictions labeled as incorrect are in fact correct and include the appropriate target element. Due to the LLM-as-judge protocol, the reported scores are not directly comparable to other QA scores that are evaluated programmatically, yet the fine-tuning gains are still significant, and comparisons within this task remain valid.

In contrast, the *element detection* setup shows only a small performance gain—from 0.84 to 0.87. The baseline model already performs strongly on this task, which likely reflects its alignment with the model’s pretrained capabilities on visual grounding of UI elements. It is likely that the baseline model was specifically trained on a similar task. This result is consistent with the earlier loss curves: the element detection setup starts with a low validation loss and shows little room for further improvement. The near-perfect teacher performance of 0.99 confirms that the task itself is solvable under full supervision, but the marginal gain from fine-tuning suggests the baseline model is already trained on this task setup and gains only diminishing returns.

The *golf* configuration yields the highest overall tuned accuracy at 0.94, up from a strong baseline of 0.79. The relative improvement is strong, especially considering the golf task’s increased complexity due to the extended task description and requirement to provide actions as function calls.

Overall, the results confirm that prompt distillation is a suitable fine-tuning method for single-step visual QA tasks for varying setups, with the greatest relative impact in cursor detection. In the following section, we further analyze the behavioral patterns of each tuned model, focusing on qualitative differences in reasoning, recurring failure cases, and generalization behavior that is not captured by accuracy alone.

### 5.1.3 Behavioral patterns

While accuracy provides a coarse evaluation of model performance, it does not capture the underlying behavioral patterns. Analyzing the output patterns of baseline and fine-tuned models allows us to better understand how models reason over visual and text inputs and where further improvements are needed. In this section, we analyze behavioral patterns on QA tasks across the three training setups.

**Cursor detection** The baseline model exhibits significant limitations in cursor localization. In many cases, it fails to identify the cursor even when clearly visible in the screenshot. Conversely, when the cursor is entirely absent, the baseline often hallucinates its presence, indicating that the model lacks a grounded understanding of the visual signal. These behaviors align with its low task accuracy (0.49).

Fine-tuning substantially improves the performance. The tuned model performs cursor localization much more consistently and rarely hallucinates cursor presence in its absence. However, this improvement comes at the cost of reduced inner monologue diversity. The model follows a strict format, and nearly all responses begin with the phrase “*The cursor ...*”, mirroring the teacher outputs. This pattern likely results from using low temperature when generating teacher trajectories during prompt distillation. Increasing the temperature would encourage more vibrant reasoning, though using a high temperature with Qwen2.5-vl-32B tends to introduce Chinese language artifacts into English output (see Section 4.4.2).

**Element detection** The baseline model performs strongly on the element detection task and exhibits well-structured reasoning. Many responses include multi-step explanations in a numbered list format (e.g., “1., 2., 3., 4.”), even though the model was not prompted to follow any particular reasoning structure. This suggests that the model might leverage prior training on similar object detection or reasoning tasks. Most failure cases are genuine errors rather than formatting issues. Overall, the baseline outputs are well-structured and informative, and the high accuracy score (0.84) reflects both visual understanding and task familiarity.

The tuned model improves slightly on the already strong baseline, producing more accurate predictions while maintaining a similar reasoning style. This indicates that fine-tuning did not overfit and instead led to refinements in decision-making. These

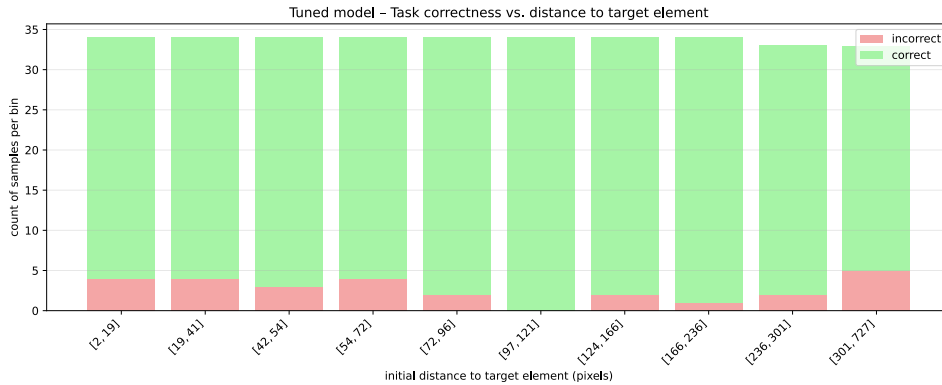
findings reflect the training behavior noted in Figure 7, as the element detection setup converged the fastest and required only minor loss reductions, implying strong initial familiarity with the task.

Teacher responses fail very rarely on this task. When errors occur, they are exclusively minor formatting issues, such as missing commas between coordinate values.

**Golf** In the golf setup, the baseline model demonstrates relatively strong performance (0.79), but inspecting the answers reveals a systematic error pattern. The model consistently produces actions and adheres to the correct output formatting, yet it frequently issues `mouse_click()` actions when the correct action would be to move the mouse. We hypothesize this follows from our task formulation, where we simulate a multi-turn environment by giving the agent information about a previous move that was issued to move the cursor to its current location. The model then thinks a correct movement has already been made and proceeds to click without inspecting the image. This tendency suggests that the model has bias towards previous language tokens rather than verifying the cursor’s position in the image. Despite these limitations, the baseline model’s ability to consistently follow the expected function calls format is a promising indicator of its potential for agent-like behavior with appropriate fine-tuning.

To examine the golf-like behavior this training setup was designed for, we also investigated whether the spatial distance between the cursor and the target element had an effect on model accuracy. During data generation, the cursor was placed at varying distances from the target using a Gaussian distribution. However, as shown in Figure 8, we found no clear correlation between distance and the correctness of the model’s response. This suggests that neither the baseline nor the fine-tuned model internalized a systematic golf-like spatial reasoning strategy. This suggests that, at least without extended fine-tuning, the model does not find small mouse movements easier than large ones and does not use relative distances between objects in this task.

Remaining failure cases mostly involved incorrect cursor movements, where the model failed to move toward the target element. These errors reinforce the need for stronger visual grounding, as well as mechanisms for the model to reason over sequential visual steps.



**Figure 8:** Correctness of tuned model predictions at different distances between the initial cursor and the target element in the golf setting. There is no consistent trend, indicating that spatial distance between cursor and target element has little impact on model behavior.

## 5.2 Performance on the single-click tasks

### 5.2.1 Numerical results

Table 3 reports success rates for the baseline and all three fine-tuned models. The agent is deemed successful if the predicted click either lands within the bounding box of the target element or selects the correct DOM node via XPath. Each task is scored as 1 if correct and 0 otherwise.

To better understand performance across different types of web interfaces, we segmented the results by layout density. Layouts were manually labeled as either *sparse* or *dense*, based on the number of visible elements and text intensity of the webpage. Sparse layouts are typically cleanly structured landing or product pages, while dense layouts refer to documentation or encyclopedia pages that contain many closely packed, visually similar elements. Examples of a sparse and a dense layout are presented in Appendix A.1.

**Table 3:** Single-click task success rates across different webpage layout types. We report performance for the untuned baseline model and each fine-tuned variant. Layout labels (sparse/dense) were annotated manually based on webpage layout density and structure.

Model	Sparse Layout	Dense Layout	Overall
Baseline	0.92	0.73	0.83
Tuned on cursor detection	0.90	0.83	<b>0.86</b>
Tuned on element detection	0.90	0.82	<b>0.86</b>
Tuned on golf	0.87	0.84	<b>0.86</b>

All three fine-tuned models outperform the baseline, improving overall task

performance from 0.83 to 0.86. This suggests that prompt distillation contributes to more effective action selection, particularly in the initial step.

To evaluate how well the validation task performance transfers to real agentic use, we compare the results from the single-turn QA setting (Table 2) to those in Table 3. In an ideal case, we would expect a perfect transfer, i.e., models that perform better on QA tasks should show corresponding gains in interactive settings. This holds partially, but there are deviations across training setups.

The element detection task exhibits the most consistent transfer. The tuned model achieves 0.87 accuracy in the QA setting and 0.86 in the agentic setting, with similar performance across both sparse and dense layouts. This suggests that fine-tuning on element detection generalizes well from single-turn QA to grounded action-taking in unseen environments.

For cursor detection, transfer is less direct. The tuned model improves from 0.49 to 0.73 on validation QA but reaches only 0.86 on single-click tasks. This discrepancy is likely due to imperfect evaluation in the QA setting, where LLM-as-a-judge evaluation lowers the number of valid answers.

The golf setup also shows a gap between the QA and agentic performance. While the tuned model achieves 0.94 accuracy on the validation QA task, it does not translate into agentic behavior: performance in the single-click is similar to other training setups. One likely explanation is a mismatch in initialization assumptions. In QA, some examples include a correctly positioned cursor, allowing the model to issue a `mouse_click()` action directly. The single-click agentic task uses random cursor initialization, which makes initial correct cursor positioning very unlikely. This leads to no easy tasks where instant `mouse_click()` yields a correct result. The final score of single-click agentic tasks is thus lower than that of the QA tasks.

As shown in Table 3, all models perform substantially better on sparse layouts. The baseline achieves 0.92 accuracy on sparse layouts compared to only 0.73 on dense layouts. Fine-tuned models improve this dense layout performance to 0.82–0.84, suggesting that training enhances robustness in cluttered environments, even though the absolute gains are moderate.

Interestingly, all tuned models obtain slightly lower accuracy on sparse layouts than the baseline, while yielding clear gains on dense layouts. Since all tuned variants converge to the same overall score of 0.86, the improvement is driven by performance gains on dense pages. This suggests that the benefits of fine-tuning are concentrated in dense layouts, whereas performance on easier sparse layouts may be more sensitive to small policy changes or overfitting.

### 5.2.2 Behavioral patterns

Similarly to the QA tasks, we analyze the behavioral tendencies of our models in the agentic single-click setting. This analysis is particularly important in interactive multi-step environments, where correct outcomes depend on the model’s initial prediction as well as its ability to interpret visual feedback and adapt across reasoning steps.

All models exhibit a strong commitment to their initial action: when the first `mouse_move()` places the cursor off target, the agent rarely verifies the placement against

the image before calling `mouse_click()`. This suggests a bias toward prior textual context (previous thoughts or actions) over visual input, leading to task termination despite an incorrect cursor location.

When the target is not identified immediately, agents often enter a scrolling loop, issuing repeated `scroll_down()` actions until the call budget is exhausted. Other options are available—such as `zoom()`—and the model does call them occasionally. However, when zooming occurs, it typically follows a reasoning trace in which the agent has already stated that it located the target, rather than being used as an exploratory step to narrow the search region. The agent is thus unable to perform real exploration outside of scrolling the page.

Finally, all models can identify non-textual clickable targets such as buttons and icons, although these remain more difficult than text-bearing elements. This capability nonetheless illustrates an advantage of vision–language models over text-only systems in web-based agentic tasks.

## 6 Discussion

In this section, we discuss what our results imply for training vision–language agents, compare them to related approaches, and highlight practical constraints of our setup. Finally, we outline new research directions for improving exploration, vision-driven decision making, and robustness to prompt and tool design.

### 6.1 Prompt distillation for vision-based agents

The numerical results indicate that prompt distillation is a suitable fine-tuning method for vision-based agentic fine-tuning. Across all three validation tasks in Table 2, the tuned models outperform the baseline. These improvements suggest that prompt distillation can reliably transfer task-specific skills and reasoning patterns from a teacher with privileged hints into the student’s parameters in a vision-based setting.

At the same time, our analysis reveals a systematic limitation that is not reflected by accuracy alone: the model’s behavior is largely insensitive to the distance between the cursor and the target element. As illustrated in Figure 8, correctness does not improve when the cursor starts nearer to the target, indicating that the model does not use relative distance when deciding how to move the cursor. In practice, the policy commits once a semantic match is formed and then acts as if small and large corrections were equally difficult. This gap between semantic grounding (identifying *what* to click) and visuomotor precision (deciding *where* to move) explains why high QA scores do not automatically translate into spatially calibrated actions.

These findings connect to prior work on prompt distillation for agents, which reports effective transfer of procedural and factual know-how from guidance text into model parameters, alongside reduced reliance on long prompts. Our results extend that picture to vision–language agents: we confirm the efficiency and stability benefits, but we also show that purely semantic imitation does not endow the student with a metric notion of space. Methodologically, this clarifies how to use distillation in practice. It

is well-suited as a first stage for installing task templates, visual grounding heuristics, and formatting discipline, thereby cutting token costs and simplifying deployment. To obtain precise cursor control, however, it should be paired with objectives that expose spatial error—e.g., distance-aware coordinate or heatmap heads, counterfactual negatives at controlled offsets, or short-horizon reinforcement/reward shaping that directly penalizes cursor-to-target deviation. In short, distillation efficiently teaches *what* to look for; additional signals are required to teach *where* to act.

## 6.2 Transfer from QA tasks to agentic evaluation

The single-click evaluation shows that distilled knowledge transfers to agentic use, but the resulting agents are still brittle. All tuned variants improve overall success from 0.83 to 0.86 (Table 3). Yet even in this single-step setting, failures are frequent and follow two recurring patterns.

First, the agents exhibit strong commitment to the initial action. After issuing an early `mouse_move()`, they often click without verifying the cursor’s placement against the screenshot. This behavior aligns with prior reports that VLM agents lean on language tokens and recent textual context over visual feedback (Li et al. 2023; Hanchao Liu et al. 2024). Concurrent to this work, a newer foundation model Qwen3-vl (Yang et al. 2025) claims mitigation of language-dominant biases. However, the foundation model Qwen2.5-vl-32B studied in this thesis remains to have weak vision-driven corrective behavior.

Second, exploration is limited. When the target is not found immediately, the default chain of actions is to scroll repeatedly until the budget is exhausted. Alternative exploratory tools such as `zoom()` are used rarely and typically only after the agent has already found the target element rather than as a strategy to localize it. This behavior likely follows from the baseline model’s training bias toward scrolling, as zooming actions were likely underrepresented or absent in its pretraining data. As a result, the model learns to rely on scrolling as its only exploratory maneuver. To increase variability and adaptability in tool selection, future training should explicitly expose the agent to zoom-based exploration.

Finally, we observe that agentic performance is highly sensitive to system prompts and task formulation. In our experiments, constraining the action set to just `mouse_move` and `mouse_click` improved performance significantly, whereas the full action space seems to distract the model from the task. Practically, this means that the agentic framework should favor minimal action spaces and concise task descriptions that force visual verification between all actions.

## 6.3 Limitations and Future Research

While this work demonstrates the promise of prompt distillation for training vision-based agents, several limitations constrain the scope and generality of the results.

First, the experiments were limited to single-click tasks. Although these tasks isolate key visual and reasoning components of agentic behavior, they do not fully capture the sequential decision-making required for multi-step agentic tasks. Extending

the evaluation to multi-step workflows would provide a more complete measure of agentic competence.

Second, the setup employs a single-agent framework with static screenshots as the sole visual modality. While this simplifies evaluation and clearly bounds the scope of this thesis, it prevents the model from leveraging structured HTML signals. This choice is reasonable for Qwen2.5-VL, but concurrent foundation models such as Qwen3-VL (Yang et al. 2025) claim reduced modality bias where multimodal environmental feedback is likely to be more informative.

Finally, while prompt distillation improves alignment between perception and action, it does not address the problem of integrating visual feedback from the environment into the reasoning process. The observed tendency to ignore visual feedback and follow the initial reasoning plan indicates that the model does not operate as a reactive autonomous agent. Incorporating reinforcement learning objectives could strengthen the utilization of visual environmental feedback and lead to more autonomous, self-correcting behavior.

## 7 Conclusions

This thesis explored how prompt distillation can be applied to improve the reliability and visual grounding of vision–language models (VLMs) acting as web-based agents. By integrating a prompt-distillation fine-tuning approach into an agentic evaluation framework, we investigated how task-specific skills and reasoning learned from question–answering (QA) tasks transfer to interactive vision-based agentic tasks. The experiments demonstrate that prompt distillation enables measurable improvements across all evaluated setups. The student models trained through distillation internalized reasoning and grounding behaviors from the teacher, achieving higher task accuracy. Evaluation in the agentic framework confirmed that skills learned in single-turn QA tasks partially transfer to interactive behavior. While fine-tuned agents performed more robustly than the baseline, they remained sensitive to initial conditions and often failed to fully use visual feedback from the environment. These findings suggest that prompt distillation strengthens perception and decision consistency but by itself does not yield autonomous, self-correcting vision-based agents. Overall, the results highlight prompt distillation as a lightweight and effective post-training method for improving visual grounding and reasoning in vision-based agents.

## 8 Acknowledgments

We acknowledge Aalto University (Finland) and CSC – IT Center for Science (Finland) for awarding this project access to the LUMI supercomputer owned by the EuroHPC Joint Undertaking.

## References

- Ainslie, Joshua et al. (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. DOI: [10.48550/arXiv.2305.13245](https://doi.org/10.48550/arXiv.2305.13245). URL: <http://arxiv.org/abs/2305.13245>.
- Alakuijala, Minttu et al. (2025). *Memento No More: Coaching AI Agents to Master Multiple Tasks via Hints Internalization*. DOI: [10.48550/arXiv.2502.01562](https://doi.org/10.48550/arXiv.2502.01562). URL: <http://arxiv.org/abs/2502.01562>.
- Anil, Cem et al. (2022). *Exploring Length Generalization in Large Language Models*. DOI: [10.48550/arXiv.2207.04901](https://doi.org/10.48550/arXiv.2207.04901). URL: <http://arxiv.org/abs/2207.04901>.
- Askell, Amanda et al. (2021). *A General Language Assistant as a Laboratory for Alignment*. DOI: [10.48550/arXiv.2112.00861](https://doi.org/10.48550/arXiv.2112.00861). URL: <http://arxiv.org/abs/2112.00861>.
- Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. DOI: [10.48550/arXiv.1607.06450](https://doi.org/10.48550/arXiv.1607.06450). URL: <http://arxiv.org/abs/1607.06450>.
- Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). *Neural Machine Translation by Jointly Learning to Align and Translate*. DOI: [10.48550/arXiv.1409.0473](https://doi.org/10.48550/arXiv.1409.0473). URL: <http://arxiv.org/abs/1409.0473>.
- Bai, Jinze et al. (2023). *Qwen Technical Report*. DOI: [10.48550/arXiv.2309.16609](https://doi.org/10.48550/arXiv.2309.16609). URL: <http://arxiv.org/abs/2309.16609>.
- Bai, Shuai et al. (2025). *Qwen2.5-VL Technical Report*. DOI: [10.48550/arXiv.2502.13923](https://doi.org/10.48550/arXiv.2502.13923). URL: <http://arxiv.org/abs/2502.13923>.
- Bai, Yuntao et al. (2022). *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. DOI: [10.48550/arXiv.2204.05862](https://doi.org/10.48550/arXiv.2204.05862). URL: <http://arxiv.org/abs/2204.05862>.
- Beltagy, Iz, Matthew E. Peters, and Arman Cohan (2020). *Longformer: The Long-Document Transformer*. DOI: [10.48550/arXiv.2004.05150](https://doi.org/10.48550/arXiv.2004.05150). URL: <http://arxiv.org/abs/2004.05150>.
- Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners*. DOI: [10.48550/arXiv.2005.14165](https://doi.org/10.48550/arXiv.2005.14165). URL: <http://arxiv.org/abs/2005.14165>.
- Chai, Yekun et al. (2024). *Tokenization Falling Short: On Subword Robustness in Large Language Models*. DOI: [10.48550/arXiv.2406.11687](https://doi.org/10.48550/arXiv.2406.11687). URL: <http://arxiv.org/abs/2406.11687>.
- Child, Rewon et al. (2019). *Generating Long Sequences with Sparse Transformers*. DOI: [10.48550/arXiv.1904.10509](https://doi.org/10.48550/arXiv.1904.10509). URL: <http://arxiv.org/abs/1904.10509>.
- Cho, Kyunghyun et al. (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. DOI: [10.48550/arXiv.1406.1078](https://doi.org/10.48550/arXiv.1406.1078). URL: <http://arxiv.org/abs/1406.1078>.
- Choi, Eunbi et al. (2022). *Prompt Injection: Parameterization of Fixed Inputs*. DOI: [10.48550/arXiv.2206.11349](https://doi.org/10.48550/arXiv.2206.11349). URL: <http://arxiv.org/abs/2206.11349>.
- Chung, Junyoung et al. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. DOI: [10.48550/arXiv.1412.3555](https://doi.org/10.48550/arXiv.1412.3555). URL: <http://arxiv.org/abs/1412.3555>.
- Clark, Aidan et al. (2022). *Unified Scaling Laws for Routed Language Models*. DOI: [10.48550/arXiv.2202.01169](https://doi.org/10.48550/arXiv.2202.01169). URL: <http://arxiv.org/abs/2202.01169>.

- Common Crawl (2025). *Common Crawl - Open Repository of Web Crawl Data*. URL: <https://commoncrawl.org> (visited on 08/08/2025).
- Dai, Damai et al. (2024). *DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models*. DOI: [10.48550/arXiv.2401.06066](https://doi.org/10.48550/arXiv.2401.06066). URL: <http://arxiv.org/abs/2401.06066>.
- DeepSeek-AI et al. (2024). *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*. DOI: [10.48550/arXiv.2405.04434](https://doi.org/10.48550/arXiv.2405.04434). URL: <http://arxiv.org/abs/2405.04434>.
- DeepSeek-AI et al. (2025a). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. DOI: [10.48550/arXiv.2501.12948](https://doi.org/10.48550/arXiv.2501.12948). URL: <http://arxiv.org/abs/2501.12948>.
- DeepSeek-AI et al. (2025b). *DeepSeek-V3 Technical Report*. DOI: [10.48550/arXiv.2412.19437](https://doi.org/10.48550/arXiv.2412.19437). URL: <http://arxiv.org/abs/2412.19437>.
- Deiseroth, Björn et al. (2024). *T-FREE: Tokenizer-Free Generative LLMs via Sparse Representations for Memory-Efficient Embeddings*. DOI: [10.48550/arXiv.2406.19223](https://doi.org/10.48550/arXiv.2406.19223). URL: <http://arxiv.org/abs/2406.19223>.
- Deng, Xiang et al. (2023). *Mind2Web: Towards a Generalist Agent for the Web*. DOI: [10.48550/arXiv.2306.06070](https://doi.org/10.48550/arXiv.2306.06070). URL: <http://arxiv.org/abs/2306.06070>.
- Devlin, Jacob et al. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. DOI: [10.48550/arXiv.1810.04805](https://doi.org/10.48550/arXiv.1810.04805). URL: <http://arxiv.org/abs/1810.04805>.
- Dosovitskiy, Alexey et al. (2021). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. DOI: [10.48550/arXiv.2010.11929](https://doi.org/10.48550/arXiv.2010.11929). URL: <http://arxiv.org/abs/2010.11929>.
- Du, Shangheng et al. (2025). *A Survey on the Optimization of Large Language Model-based Agents*. DOI: [10.48550/arXiv.2503.12434](https://doi.org/10.48550/arXiv.2503.12434). URL: <http://arxiv.org/abs/2503.12434>.
- Fedus, William, Barret Zoph, and Noam Shazeer (2022). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. DOI: [10.48550/arXiv.2101.03961](https://doi.org/10.48550/arXiv.2101.03961). URL: <http://arxiv.org/abs/2101.03961>.
- Fu, Zihao et al. (2023). *Decoder-Only or Encoder-Decoder? Interpreting Language Model as a Regularized Encoder-Decoder*. DOI: [10.48550/arXiv.2304.04052](https://doi.org/10.48550/arXiv.2304.04052). URL: <http://arxiv.org/abs/2304.04052>.
- Gage, Philip (1994). “A new algorithm for data compression”. In: *C Users J*. 12.2, pp. 23–38. DOI: [10.5555/177910.177914](https://doi.org/10.5555/177910.177914).
- Gao, Luyu et al. (2023). *PAL: Program-aided Language Models*. DOI: [10.48550/arXiv.2211.10435](https://doi.org/10.48550/arXiv.2211.10435). URL: <http://arxiv.org/abs/2211.10435>.
- Gemma, Team et al. (2025). *Gemma 3 Technical Report*. DOI: [10.48550/arXiv.2503.19786](https://doi.org/10.48550/arXiv.2503.19786). URL: <http://arxiv.org/abs/2503.19786>.
- Gemma Team et al. (2024). *Gemma: Open Models Based on Gemini Research and Technology*. DOI: [10.48550/arXiv.2403.08295](https://doi.org/10.48550/arXiv.2403.08295). URL: <http://arxiv.org/abs/2403.08295>.
- Grattafiori, Aaron et al. (2024). *The Llama 3 Herd of Models*. DOI: [10.48550/arXiv.2407.21783](https://doi.org/10.48550/arXiv.2407.21783). URL: <http://arxiv.org/abs/2407.21783>.

- Hazimeh, Hussein et al. (2021). *DSelect-k: Differentiable Selection in the Mixture of Experts with Applications to Multi-Task Learning*. DOI: [10.48550/arXiv.2106.03760](https://doi.org/10.48550/arXiv.2106.03760). URL: <http://arxiv.org/abs/2106.03760>.
- He, Kaiming et al. (2015). *Deep Residual Learning for Image Recognition*. DOI: [10.48550/arXiv.1512.03385](https://doi.org/10.48550/arXiv.1512.03385). URL: <http://arxiv.org/abs/1512.03385>.
- Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean (2015). *Distilling the Knowledge in a Neural Network*. DOI: [10.48550/arXiv.1503.02531](https://doi.org/10.48550/arXiv.1503.02531). URL: <http://arxiv.org/abs/1503.02531>.
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8, pp. 1735–1780. DOI: [10.1162](https://doi.org/10.1162).
- Hu, Edward J. et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. DOI: [10.48550/arXiv.2106.09685](https://doi.org/10.48550/arXiv.2106.09685). URL: <http://arxiv.org/abs/2106.09685>.
- Jordan, M.I. and R.A. Jacobs (1993). “Hierarchical mixtures of experts and the EM algorithm”. In: 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan). Vol. 2, 1339–1344 vol.2. DOI: [10.1109/IJCNN.1993.716791](https://doi.org/10.1109/IJCNN.1993.716791). URL: <https://ieeexplore.ieee.org/document/716791>.
- Kaplan, Jared et al. (2020). *Scaling Laws for Neural Language Models*. DOI: [10.48550/arXiv.2001.08361](https://doi.org/10.48550/arXiv.2001.08361). URL: <http://arxiv.org/abs/2001.08361>.
- Kazemnejad, Amirhossein et al. (2023). *The Impact of Positional Encoding on Length Generalization in Transformers*. DOI: [10.48550/arXiv.2305.19466](https://doi.org/10.48550/arXiv.2305.19466). URL: <http://arxiv.org/abs/2305.19466>.
- Kim, Taehyeon et al. (2021). *Comparing Kullback-Leibler Divergence and Mean Squared Error Loss in Knowledge Distillation*. DOI: [10.48550/arXiv.2105.08919](https://doi.org/10.48550/arXiv.2105.08919). URL: <http://arxiv.org/abs/2105.08919>.
- Kojima, Takeshi et al. (2023). *Large Language Models are Zero-Shot Reasoners*. DOI: [10.48550/arXiv.2205.11916](https://doi.org/10.48550/arXiv.2205.11916). URL: <http://arxiv.org/abs/2205.11916>.
- Kudo, Taku and John Richardson (2018). *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. DOI: [10.48550/arXiv.1808.06226](https://doi.org/10.48550/arXiv.1808.06226). URL: <http://arxiv.org/abs/1808.06226>.
- Kujanpää, Kalle et al. (2025). *Efficient Knowledge Injection in LLMs via Self-Distillation*. DOI: [10.48550/arXiv.2412.14964](https://doi.org/10.48550/arXiv.2412.14964). URL: <http://arxiv.org/abs/2412.14964>.
- Li, Junnan et al. (2023). *BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models*. DOI: [10.48550/arXiv.2301.12597](https://doi.org/10.48550/arXiv.2301.12597). URL: <http://arxiv.org/abs/2301.12597>.
- Liu, Hanchao et al. (2024). *A Survey on Hallucination in Large Vision-Language Models*. DOI: [10.48550/arXiv.2402.00253](https://doi.org/10.48550/arXiv.2402.00253). URL: <http://arxiv.org/abs/2402.00253>.
- Liu, Haotian et al. (2023). *Visual Instruction Tuning*. DOI: [10.48550/arXiv.2304.08485](https://doi.org/10.48550/arXiv.2304.08485). URL: <http://arxiv.org/abs/2304.08485>.
- Lu, Haoyu et al. (2024). *DeepSeek-VL: Towards Real-World Vision-Language Understanding*. DOI: [10.48550/arXiv.2403.05525](https://doi.org/10.48550/arXiv.2403.05525). URL: <http://arxiv.org/abs/2403.05525>.

- Meta AI (2025). *The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation*. Meta AI. URL: <https://ai.meta.com/blog/llama-4-multimodal-intelligence/> (visited on 08/27/2025).
- Minaee, Shervin et al. (2025). *Large Language Models: A Survey*. DOI: 10.48550/arXiv.2402.06196. URL: <http://arxiv.org/abs/2402.06196>.
- Mistral AI (2025). *Tokenization | Mistral AI Large Language Models*. URL: <https://docs.mistral.ai/guides/tokenization/> (visited on 08/12/2025).
- Mukherjee, Subhabrata et al. (2023). *Orca: Progressive Learning from Complex Explanation Traces of GPT-4*. DOI: 10.48550/arXiv.2306.02707. URL: <http://arxiv.org/abs/2306.02707>.
- Nakano, Reiichiro et al. (2022). *WebGPT: Browser-assisted question-answering with human feedback*. DOI: 10.48550/arXiv.2112.09332. URL: <http://arxiv.org/abs/2112.09332>.
- OpenAI (2025). *Thinking with images*. URL: <https://openai.com/index/thinking-with-images/> (visited on 09/02/2025).
- OpenAI et al. (2024a). *GPT-4 Technical Report*. DOI: 10.48550/arXiv.2303.08774. URL: <http://arxiv.org/abs/2303.08774>.
- OpenAI et al. (2024b). *GPT-4o System Card*. DOI: 10.48550/arXiv.2410.21276. URL: <http://arxiv.org/abs/2410.21276>.
- OpenAI et al. (2024c). *OpenAI o1 System Card*. DOI: 10.48550/arXiv.2412.16720. URL: <http://arxiv.org/abs/2412.16720>.
- Ouyang, Long et al. (2022). *Training language models to follow instructions with human feedback*. DOI: 10.48550/arXiv.2203.02155. URL: <http://arxiv.org/abs/2203.02155>.
- Pagnoni, Artidoro et al. (2024). *Byte Latent Transformer: Patches Scale Better Than Tokens*. DOI: 10.48550/arXiv.2412.09871. URL: <http://arxiv.org/abs/2412.09871>.
- Parisi, Aaron, Yao Zhao, and Noah Fiedel (2022). *TALM: Tool Augmented Language Models*. DOI: 10.48550/arXiv.2205.12255. URL: <http://arxiv.org/abs/2205.12255>.
- Qin, Yujia et al. (2023). *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. DOI: 10.48550/arXiv.2307.16789. URL: <http://arxiv.org/abs/2307.16789>.
- Radford, Alec et al. (2019). “Language Models are Unsupervised Multitask Learners”. In: *OpenAI*. URL: [https://cdn.openai.com/better-language-models/language\\_models\\_are\\_unsupervised\\_multitask\\_learners.pdf](https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf).
- Raffel, Colin et al. (2023). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. DOI: 10.48550/arXiv.1910.10683. URL: <http://arxiv.org/abs/1910.10683>.
- Sanh, Victor et al. (2020). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. DOI: 10.48550/arXiv.1910.01108. URL: <http://arxiv.org/abs/1910.01108>.
- Schick, Timo et al. (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. DOI: 10.48550/arXiv.2302.04761. URL: <http://arxiv.org/abs/2302.04761>.

- Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016). *Neural Machine Translation of Rare Words with Subword Units*. DOI: [10.48550/arXiv.1508.07909](https://doi.org/10.48550/arXiv.1508.07909). URL: <http://arxiv.org/abs/1508.07909>.
- Shantanu, Jain (2022). *tiktoken: tiktoken is a fast BPE tokeniser for use with OpenAI's models*. URL: <https://github.com/openai/tiktoken> (visited on 08/10/2025).
- Shao, Zhihong et al. (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. DOI: [10.48550/arXiv.2402.03300](https://doi.org/10.48550/arXiv.2402.03300). URL: <http://arxiv.org/abs/2402.03300>.
- Shazeer, Noam (2019). *Fast Transformer Decoding: One Write-Head is All You Need*. DOI: [10.48550/arXiv.1911.02150](https://doi.org/10.48550/arXiv.1911.02150). URL: <http://arxiv.org/abs/1911.02150>.
- Shazeer, Noam et al. (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. DOI: [10.48550/arXiv.1701.06538](https://doi.org/10.48550/arXiv.1701.06538). URL: <http://arxiv.org/abs/1701.06538>.
- Shen, Haozhan et al. (2024). *ZoomEye: Enhancing Multimodal LLMs with Human-Like Zooming Capabilities through Tree-Based Image Exploration*. DOI: [10.48550/arXiv.2411.16044](https://doi.org/10.48550/arXiv.2411.16044). URL: <http://arxiv.org/abs/2411.16044>.
- Singh, Aaditya K. and D. J. Strouse (2024). *Tokenization counts: the impact of tokenization on arithmetic in frontier LLMs*. DOI: [10.48550/arXiv.2402.14903](https://doi.org/10.48550/arXiv.2402.14903). URL: <http://arxiv.org/abs/2402.14903>.
- Snell, Charlie, Dan Klein, and Ruiqi Zhong (2022). *Learning by Distilling Context*. DOI: [10.48550/arXiv.2209.15189](https://doi.org/10.48550/arXiv.2209.15189). URL: <http://arxiv.org/abs/2209.15189>.
- Su, Jianlin et al. (2023). *RoFormer: Enhanced Transformer with Rotary Position Embedding*. DOI: [10.48550/arXiv.2104.09864](https://doi.org/10.48550/arXiv.2104.09864). URL: <http://arxiv.org/abs/2104.09864>.
- Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). *Sequence to Sequence Learning with Neural Networks*. DOI: [10.48550/arXiv.1409.3215](https://doi.org/10.48550/arXiv.1409.3215). URL: <http://arxiv.org/abs/1409.3215>.
- Tay, Yi et al. (2023). *UL2: Unifying Language Learning Paradigms*. DOI: [10.48550/arXiv.2205.05131](https://doi.org/10.48550/arXiv.2205.05131). URL: <http://arxiv.org/abs/2205.05131>.
- Tie, Guiyao et al. (2025). *A Survey on Post-training of Large Language Models*. DOI: [10.48550/arXiv.2503.06072](https://doi.org/10.48550/arXiv.2503.06072). URL: <http://arxiv.org/abs/2503.06072>.
- Vaswani, Ashish et al. (2017). *Attention Is All You Need*. DOI: [10.48550/arXiv.1706.03762](https://doi.org/10.48550/arXiv.1706.03762). URL: <http://arxiv.org/abs/1706.03762>.
- Vincent, Pascal et al. (2008). “Extracting and composing robust features with denoising autoencoders”. In: *Proceedings of the 25th international conference on Machine learning - ICML '08*. Helsinki, Finland: ACM Press, pp. 1096–1103. DOI: [10.1145/1390156.1390294](https://doi.org/10.1145/1390156.1390294).
- Wang, Changan, Kyunghyun Cho, and Jiatao Gu (2019). *Neural Machine Translation with Byte-Level Subwords*. DOI: [10.48550/arXiv.1909.03341](https://doi.org/10.48550/arXiv.1909.03341). URL: <http://arxiv.org/abs/1909.03341>.
- Wang, Guanzhi et al. (2023). *Voyager: An Open-Ended Embodied Agent with Large Language Models*. DOI: [10.48550/arXiv.2305.16291](https://doi.org/10.48550/arXiv.2305.16291). URL: <http://arxiv.org/abs/2305.16291>.
- Wang, Xingyao et al. (2024). *Executable Code Actions Elicit Better LLM Agents*. DOI: [10.48550/arXiv.2402.01030](https://doi.org/10.48550/arXiv.2402.01030). URL: <http://arxiv.org/abs/2402.01030>.

- Webster, Jonathan J. and Chunyu Kit (1992). “Tokenization as the initial phase in NLP”. In: *Proceedings of the 14th conference on Computational linguistics - Volume 4*. COLING '92. USA: Association for Computational Linguistics, pp. 1106–1110. DOI: [10.3115/992424.992434](https://doi.org/10.3115/992424.992434).
- Wei, Jason et al. (2022a). *Emergent Abilities of Large Language Models*. DOI: [10.48550/arXiv.2206.07682](https://doi.org/10.48550/arXiv.2206.07682). URL: <http://arxiv.org/abs/2206.07682>.
- Wei, Jason et al. (2022b). *Finetuned Language Models Are Zero-Shot Learners*. DOI: [10.48550/arXiv.2109.01652](https://doi.org/10.48550/arXiv.2109.01652). URL: <http://arxiv.org/abs/2109.01652>.
- Wei, Jason et al. (2023). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. DOI: [10.48550/arXiv.2201.11903](https://doi.org/10.48550/arXiv.2201.11903). URL: <http://arxiv.org/abs/2201.11903>.
- Yang, An et al. (2025). *Qwen3 Technical Report*. DOI: [10.48550/arXiv.2505.09388](https://doi.org/10.48550/arXiv.2505.09388). URL: <http://arxiv.org/abs/2505.09388>.
- Yao, Shunyu et al. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. DOI: [10.48550/arXiv.2210.03629](https://doi.org/10.48550/arXiv.2210.03629). URL: <http://arxiv.org/abs/2210.03629>.
- Zaheer, Manzil et al. (2021). *Big Bird: Transformers for Longer Sequences*. DOI: [10.48550/arXiv.2007.14062](https://doi.org/10.48550/arXiv.2007.14062). URL: <http://arxiv.org/abs/2007.14062>.
- Zhang, Biao and Rico Sennrich (2019). *Root Mean Square Layer Normalization*. DOI: [10.48550/arXiv.1910.07467](https://doi.org/10.48550/arXiv.1910.07467). URL: <http://arxiv.org/abs/1910.07467>.
- Zhang, Xintong et al. (2025). *Chain-of-Focus: Adaptive Visual Search and Zooming for Multimodal Reasoning via RL*. DOI: [10.48550/arXiv.2505.15436](https://doi.org/10.48550/arXiv.2505.15436). URL: <http://arxiv.org/abs/2505.15436>.
- Zhao, Wayne Xin et al. (2025). *A Survey of Large Language Models*. DOI: [10.48550/arXiv.2303.18223](https://doi.org/10.48550/arXiv.2303.18223). URL: <http://arxiv.org/abs/2303.18223>.
- Zheng, Boyuan et al. (2024). *GPT-4V(ision) is a Generalist Web Agent, if Grounded*. DOI: [10.48550/arXiv.2401.01614](https://doi.org/10.48550/arXiv.2401.01614). URL: <http://arxiv.org/abs/2401.01614>.
- Zhou, Yanqi et al. (2022). *Mixture-of-Experts with Expert Choice Routing*. DOI: [10.48550/arXiv.2202.09368](https://doi.org/10.48550/arXiv.2202.09368). URL: <http://arxiv.org/abs/2202.09368>.
- Zhu, Deyao et al. (2023). *MiniGPT-4: Enhancing Vision-Language Understanding with Advanced Large Language Models*. DOI: [10.48550/arXiv.2304.10592](https://doi.org/10.48550/arXiv.2304.10592). URL: <http://arxiv.org/abs/2304.10592>.

# A Appendix

## A.1 Examples of sparse and dense webpage layouts

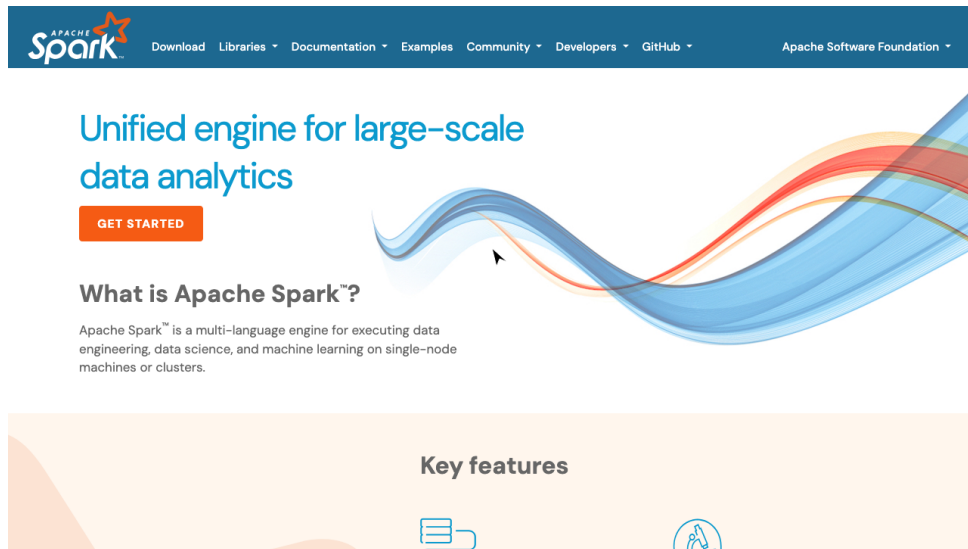


Figure A1: Example of a sparse layout.

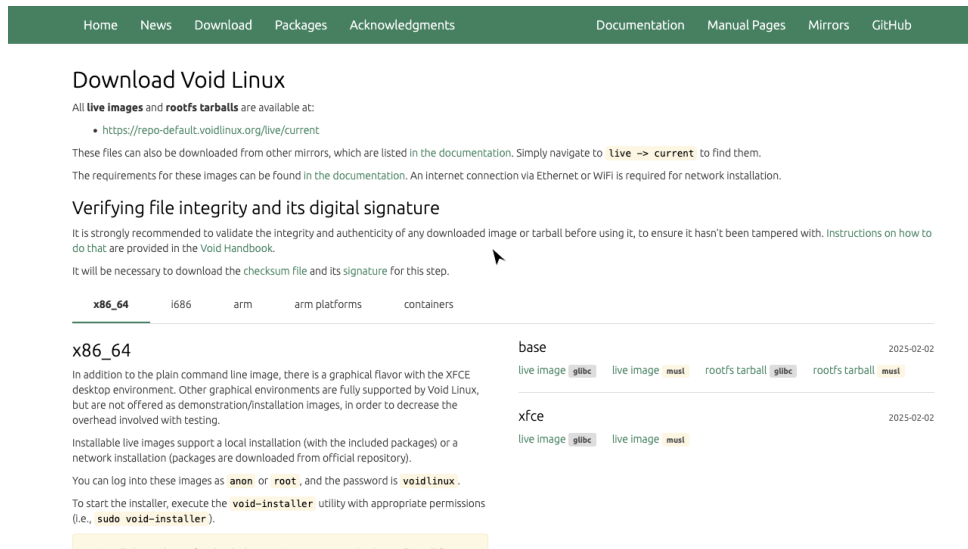


Figure A2: Example of a dense layout.