

Politecnico di Torino

Politecnico di Torino

Master's Degree in Cybersecurity

Academic Year 2025/2026

Graduation Session October 2026

CVE Intelligence Extraction

Fine-Tuning Local Large Language Models for Structured Vulnerability Data

Supervisors:

Prof. Fulvio Valenza

Prof. Daniele Bringhenti

Prof. Riccardo Sisto

Candidate:

Mohamed Eltoukhi

Abstract

The number of published Common Vulnerabilities and Exposures (CVE) records continues to grow, but much of their useful information is still written in free-form text. People can read these descriptions, whereas automated security tools work better with clearly defined fields. Hand-written parsing rules offer one solution, but they can fail when descriptions differ in wording or level of detail.

An open-weight language model provides an alternative for processing these descriptions. The proposed extractor converts raw CVE records into structured JavaScript Object Notation (JSON) directly on the user's machine. Its target format, VIPER (Vulnerability Intelligence Parsed Extraction Record), organizes affected products, weaknesses, conditions required for exploitation, and possible consequences. A larger teacher model generated 989 CVE-to-VIPER examples, avoiding the need to label hundreds of records by hand. Removing 15 overlong conversations left 974 examples for the fine-tuning pipeline. Llama 3.1 8B was adapted with Low-Rank Adaptation. Applying the same procedure to Llama 3.2 1B showed how the method performed with a smaller model. Both models were then merged, quantized, and deployed locally.

The evaluation compared the two fine-tuned models with their prompted base versions on a separate test set of 200 CVE records. It measured JSON validity, schema conformance, and content accuracy. An independent language model also rated how faithfully each answer represented its source record.

The fine-tuned 8B model followed the schema for 98% of the test records, compared with 79.5% for the prompted 8B model. Its average faithfulness score rose from 3.37 to 4.51 out of 5. Fine-tuning improved schema conformance for the 1B model from 3.5% to 68%, although the smaller model remained less reliable than the 8B model. These results suggest that a compact language model can learn a specialized output format from examples and extract vulnerability information more reliably after fine-tuning. Because inference takes place locally, CVE data do not need to be sent to a cloud service, and the model remains under the user's control.

Acknowledgements

I would like to sincerely thank my supervisors, Professor Fulvio Valenza, Professor Daniele Bringhenti, and Professor Riccardo Sisto, for their guidance, wise feedback, and continued support throughout the development of this thesis.

I am deeply grateful to my parents, whose unconditional love, sacrifices, and encouragement have made this journey possible, and to my siblings for always believing in me.

My warmest thanks go to Rhea for her patience, understanding, and constant support throughout this journey.

Finally, I would like to thank my friends for their companionship, encouragement, and all the moments which made these years memorable.

Table of Contents

List of Tables VII

List of Figures VIII

List of Listings IX

List of Abbreviations XI

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Contributions . . . 3
- 1.4 Thesis Structure . . . 4

2 Vulnerability Intelligence and the CVE Program 5
- 2.1 The CVE Program and Record Publication . . . 5
- 2.2 The CVE Record Format . . . 6
- 2.3 Severity and Classification Standards . . . 8
  - 2.3.1 CVSS . . . 8
  - 2.3.2 CWE . . . 9
  - 2.3.3 Availability in the Test Set . . . 9
- 2.4 The Demand for Structured Threat Information . . . 10
  - 2.4.1 From CVE Records to Structured Fields . . . 10
  - 2.4.2 Limits of Rule-Based Extraction . . . 11
  - 2.4.3 VIPER and Learned Extraction . . . 11

3 Large Language Models, Fine-Tuning, and Local Inference 12
- 3.1 Transformer Language Models . . . 12
  - 3.1.1 Transformer Architecture . . . 12
  - 3.1.2 Autoregressive Generation . . . 13
  - 3.1.3 Context and Prompted Inference . . . 14

3.2 The Llama Model Family . . . 14
3.3 Instruction Tuning and Chat Templates . . . 15
3.3.1 Instruction Tuning . . . 15
3.3.2 Chat Templates . . . 15
3.3.3 Training on Assistant Responses . . . 16
3.4 Parameter-Efficient Fine-Tuning: LoRA and QLoRA . . . 16
3.5 Quantization and Local Inference . . . 18
3.6 Knowledge Distillation with Synthetic Supervision . . . 18

4 Thesis Objectives 20
4.1 Goals . . . 20
4.2 Scope and Constraints . . . 21
4.3 Research Questions . . . 22

5 Methodology 23
5.1 Pipeline Overview . . . 23
5.2 The VIPER Schema . . . 24
5.3 Corpus Sampling . . . 26
5.4 Teacher-Generated Targets . . . 27
5.5 Training-Data Formatting . . . 29
5.6 Fine-Tuning Setup . . . 30
5.7 Quantization and Local Serving . . . 32
5.8 Comparison Across Model Sizes . . . 33

6 Validation 34
6.1 Test Set and Reference Construction . . . 34
6.2 Evaluation Metrics . . . 35
6.2.1 Structural Validity . . . 35
6.2.2 Content Accuracy . . . 36
6.2.3 LLM-as-a-Judge . . . 39
6.3 Comparison Protocol and Failure Handling . . . 39
6.4 Results . . . 40
6.5 Comparison Across Model Sizes . . . 44
6.6 Discussion . . . 45

7 Conclusion and Future Work 47
7.1 Summary of Contributions . . . 47
7.2 Limitations . . . 49
7.3 Future Work . . . 51
7.4 Closing Remarks . . . 52

A Reproducibility Materials 53
A.1 Complete VIPER JSON Schema 53
A.2 Generation and Evaluation Prompts 56
A.2.1 Serialized Llama Conversation 56
A.2.2 DeepSeek Teacher Prompt 57
A.2.3 Prompted Baseline and Evaluation Judge 58
A.3 Training and Inference Configuration 59
A.3.1 Local Inference Environment 60
A.4 Dataset and Prediction Artifacts 60

Bibliography 62

List of Tables

2.1 CVSS 3.1 vector components 9

5.1 Corpus distribution by year 27
5.2 Fine-tuning configuration. 30

6.1 Four-model evaluation results 40
6.2 Responses reaching the generation limit 44
6.3 Fine-tuning gains by model size 45

A.1 Recorded local inference environment 60
A.2 Archived dataset and prediction artifacts 61

List of Figures

3.1 The transformer architecture 13
3.2 Conversation structure used for fine-tuning 16
3.3 The LoRA reparametrization 17
3.4 Distillation with synthetic supervision 19

5.1 Pipeline overview 24

6.1 The three evaluation layers 35
6.2 Headline metrics 41
6.3 Judge score distribution 42
6.4 Per-field scores 43

List of Listings

2.1 Abbreviated CVE record structure 6

5.1 Abbreviated VIPER structure 25
5.2 Abbreviated teacher input 28
5.3 Generated VIPER target 28

A.1 Complete VIPER JSON Schema 56
A.2 Serialized Llama 3 training conversation 56
A.3 DeepSeek teacher prompt 58
A.4 Axolotl configuration 59

List of Abbreviations

API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
CNA	CVE Numbering Authority
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
GGUF	Model file format used by `llama.cpp`
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LLM	Large Language Model
LoRA	Low-Rank Adaptation
NVD	National Vulnerability Database
QLoRA	Quantized Low-Rank Adaptation
VIPER	Vulnerability Intelligence Parsed Extraction Record
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

Publishing information about software vulnerabilities has become a central part of cybersecurity. The scale of this work is considerable. The National Vulnerability Database lists more than 370,000 vulnerabilities [1], and the public CVE List has received over 30,000 new records in each of the last three years [2]. These disclosures help defenders understand their exposure and respond before a vulnerability is exploited.

The Common Vulnerabilities and Exposures (CVE) Program supports this process. Its mission is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities [3]. Each vulnerability is published as a CVE Record with a unique identifier. This shared reference allows different tools and organizations to refer to the same flaw without ambiguity, and much of today's vulnerability management depends on these records.

CVE records, however, are only partly structured. They follow a defined JavaScript Object Notation (JSON) format [4], and fields such as affected products, external references, and severity scores follow established standards [5, 6]. Descriptions are less uniform: many organizations write them as free-form text, using their own styles and levels of detail. Human analysts can read such descriptions, but automated security tools struggle to process them consistently. This leads to the central issue addressed here: how can varied CVE descriptions be converted into a reliable, machine-readable form?

1.1 Context and Motivation

CVE identifiers connect many parts of vulnerability management. They appear in scanners, security advisories, patch notes, threat intelligence feeds, and databases such as the National Vulnerability Database (NVD) [7]. Security teams rely on the information linked to these identifiers when they assess an alert, connect an

incident to a known flaw, or decide which systems to patch first.

To use CVE information in automated workflows, tools need separate fields for the weakness, affected vendors and products, conditions required for exploitation, and possible consequences. Producing these fields by hand is not practical at the current rate of disclosure. Rule-based methods, including regular expressions, keyword searches, and custom parsers, also have clear limits. A rule written for one organization's phrasing may fail when another organization describes the same kind of flaw differently [8].

Large Language Models (LLMs) can handle some of this language variation. They learn patterns through different natural-language inputs and generate new text [9, 10]. Using them for security data, however, raises two concerns.

Data control. A cloud-based model requires CVE records to be sent to an external provider. This may introduce recurring costs and privacy concerns while making the system dependent on a service and model version that the organization does not control.

Output reliability. In the early trials conducted for this work, the generated records often appeared convincing but did not always satisfy the required schema, even when the prompt contained detailed instructions. A few examples could not reveal how often these failures occurred, which motivated the controlled evaluation presented later in the thesis.

Together, these concerns motivate the use of a compact, open-weight model that can be fine-tuned for the task and run entirely on local hardware.

1.2 Problem Statement

The central problem can be stated simply: given a raw CVE record, generate a fixed set of fields that captures its relevant threat and vulnerability information. The model must have open weights and must be able to run locally.

Three constraints shape the solution:

1. **Fixed format.** Every output must follow the same JSON schema. VIPER (Vulnerability Intelligence Parsed Extraction Record) defines the required fields and their allowed values for this task.

2. **Local inference.** The model must run on the user's machine without calling a cloud service.

3. **Learning from examples.** The fine-tuned model must learn the schema from its training examples. At inference time, the prompt names VIPER but does not include its definition. The model must generate the structure from what it learned during fine-tuning instead of relying on a detailed prompt regarding each record.

The learning-from-examples constraint leads to an important question about the value of fine-tuning. A general-purpose model can produce convincing output when the full schema is included in its prompt. A few successful examples might suggest that fine-tuning is unnecessary, but they do not show how consistently the model works. A fair answer requires a separate test set and repeatable measurements of both structure and content. Building the extractor addresses only part of the problem; its reliability must also be measured. The experiments compare each fine-tuned model with its prompted base version and determine whether fine-tuning provides a clear improvement.

1.3 Contributions

The main contributions are as follows:

- **The VIPER schema**, a task-specific JSON format that turns information scattered across a CVE record into a consistent extraction target. Its seven top-level groups bring together facts available in structured CVE fields and details that must be recovered from free-form descriptions. The schema defines the required structure, data types, and accepted values for every generated record.
- **A distillation dataset** containing 989 CVE-to-VIPER pairs. A stronger teacher model (DeepSeek) [11] produced the structured target for each raw record. The system prompt used in training names VIPER but does not define it, so the model has to learn the schema from the example outputs. After conversations that exceeded the sequence limit were removed, 974 examples entered the training runs.
- **A fine-tuned extractor** based on Llama 3.1 8B [12]. The model was adapted with Low-Rank Adaptation (LoRA) [13], merged, converted to a quantized GGUF model, and served locally with `llama.cpp` [14]. No cloud service is needed during inference.
- **A comparison across model sizes**. The same training procedure was repeated with the much smaller Llama 3.2 1B model [15]. The comparison shows how the method behaves with a smaller checkpoint, although the two checkpoints come from different Llama releases and do not isolate parameter quantity as the only source of variation.
- **An evaluation method** based on a separate test set of 200 CVE records. It measures structural validity, correctness within individual fields, and faithfulness to the source record. Faithfulness is rated by a language model that is independent of both the teacher and the models under evaluation.

Across both model sizes, fine-tuning improved the main measures of schema conformance and content quality over prompting alone. The complete extraction pipeline can also run locally, keeping inference data on the user's machine and avoiding dependence on an external inference service.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2** provides the domain background by introducing the CVE Program, the format of its records, and the standards used to describe and classify vulnerabilities.
- **Chapter 3** provides the technical background on language models and the fine-tuning, quantization, and distillation methods used in the project.
- **Chapter 4** defines the goals, scope, and research questions.
- **Chapter 5** describes the VIPER schema, training data, fine-tuning process, local deployment, and comparison across model sizes.
- **Chapter 6** explains the evaluation process and reports the results for all four models.
- **Chapter 7** summarizes the findings, discusses the limitations, and proposes directions for future work.

Chapter 2

Vulnerability Intelligence and the CVE Program

CVE records are difficult to process automatically because of how they are created and how they store information: structured fields sit alongside free-form text, and important details may appear in either. The discussion begins with how the CVE Program assigns and publishes records, then moves to the JSON record format and the CVSS and CWE standards. The final section explains why information found only in free-form text needs to be converted into a consistent structure.

2.1 The CVE Program and Record Publication

The Common Vulnerabilities and Exposures (CVE) Program is a community effort whose mission is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities [3].

A CVE identifier gives organizations a shared reference for a vulnerability. Its use in vendor advisories, patches, scanner results, and vulnerability databases links information from these sources to the same flaw, even when their descriptions differ [16].

CVE identifiers follow the format `CVE-YYYY-NNNN...`, where the final sequence contains four or more digits. The year normally indicates when the identifier was reserved. If the vulnerability was already public before reservation, it may instead indicate the year of that disclosure; it does not represent when the flaw was discovered [16]. This distinction matters when records are selected by identifier year, as they are in the corpus used for this thesis.

The CVE Program distributes the work of assigning identifiers among CVE Numbering Authorities (CNAs). Each CNA is authorized to handle security issues within an agreed scope, such as a company's products, an open-source project, or

vulnerabilities coordinated by a security organization. The organization closest to a vulnerability can thus manage its identifier and record [17].

A CNA can reserve an identifier while disclosure is still being coordinated. At this stage, the identifier exists, but the associated record is not yet public. Once the CNA supplies the required information, it publishes the record to the CVE List. Published records include the identifier, a description, the affected products and versions, and at least one public reference. The dataset used in this thesis includes only records in the published state [17].

Under this distributed process, many organizations publish records independently. The resulting records vary with respect to terminology, writing style, and level of detail. Sixty CNAs authored the 200 records in the test set, illustrating the range of sources that an extraction system must handle.

After publication, other organizations may add further analysis. The National Vulnerability Database (NVD), maintained by the National Institute of Standards and Technology (NIST), republishes CVE information and may add severity scores, weakness classifications, and other enrichment [18]. NVD enrichment is excluded from the model input. The models process records from the official CVE List and use only the CNA-provided container, keeping the task focused on information available in the original CVE Record.

2.2 The CVE Record Format

CVE records are published as JSON documents that follow the official CVE Record Format [4]. The records used in this thesis come from the public `cvelistV5` repository [2]. At the top level, every record contains four fields: `dataType`, `dataVersion`, `cveMetadata`, and `containers`.

The first two identify the type and version of the document. The `cveMetadata` object stores administrative information, including the CVE identifier, the record state, and the organization that assigned it. The vulnerability information itself is stored under `containers`. The structure is summarized in Listing 2.1.

```
{
  "dataType": "CVE_RECORD",
  "dataVersion": "5.x",
  "cveMetadata": { "cveId": "CVE-YYYY-NNNN", "state": "PUBLISHED",
                   "assignerOrgId": "..." },
  "containers": { "cna": { "<omitted>": null },
                  "adp": ["<omitted>"] }
}
```

Listing 2.1. Abbreviated structure of a CVE record.

The `cna` container holds the information published by the assigning CNA. The optional `adp` array contains one or more additions from Authorized Data Publishers, which may supply risk scores, references, translations, or other enrichment. Keeping these sources in separate containers preserves their origin instead of combining them into a single record without attribution [4].

The same type of information may appear in both containers, sometimes with different values. The methodology in Chapter 5 resolves this vagueness by using only the CNA container. The model input consequently remains tied to the information supplied by the assigning authority.

The CVE identifier used in the output comes from `cveMetadata.cveId`. Most of the remaining VIPER content is drawn or inferred from five fields in the CNA container. These are not the only fields passed to the model, but they have the most direct function in constructing the target record [4]:

- `descriptions` contains one or more natural-language descriptions, each marked with its language. Descriptions provide the main source for details that do not appear in dedicated fields, such as the cause of the flaw, its consequences, and how it may be exploited.
- `affected` lists the affected vendors, products, and version ranges. This information is already structured and can often be transferred directly to the corresponding VIPER fields.
- `references` contains links to external sources such as advisories, patches, commits, and exploit reports. VIPER keeps these links so that the generated record remains connected to its supporting material.
- `problemTypes` describes the weakness involved. It may include a CWE identifier, although the presence of the field does not guarantee that a usable CWE value is available.
- `metrics` contains severity assessments. When a supported CVSS vector is present, its version, score, severity label, and selected exploitation conditions can be derived from it.

These source fields do not map to VIPER in the same way. Some values, such as the CVE identifier and reference links, can be copied directly. Others can be recovered by parsing structured data, while interpretive fields must be inferred from the description. Chapter 5 explains how the records are prepared and how the target fields are produced.

Not all of these fields can be assumed to be present. In the 200-record test set, `descriptions`, `affected`, and `references` appear in every record, while `problemTypes` appears in 181 records and `metrics` in 143. The official schema

defines how each field must be structured when present, but it allows records to omit `problemTypes` and `metrics` [4]. Current CNA Rules still require the type of vulnerability to be identified, although it may not appear as a usable CWE identifier in `problemTypes`; a CVSS assessment is not required [19]. The extractor must handle records that provide no dedicated weakness classification or severity assessment.

Even when these fields are present, they may not contain values supported by the extraction pipeline. A `problemTypes` block may lack a valid CWE identifier, and a `metrics` block may contain no supported CVSS vector. The difference between field presence and usable standardized data is quantified after the two standards are introduced in Section 2.3.3.

2.3 Severity and Classification Standards

Two standards are especially important for this thesis. The Common Vulnerability Scoring System (CVSS) describes the severity and exploitation conditions of a vulnerability, while the Common Weakness Enumeration (CWE) classifies the type of weakness involved. Values encoded through these standards can often be recovered directly from the source record. Chapter 6 uses them to evaluate the corresponding output fields through deterministic comparison.

2.3.1 CVSS

The Common Vulnerability Scoring System records vulnerability characteristics in a vector and summarizes their severity with a score from 0.0 to 10.0 [5, 6]. The vector describes factors such as where an attack can originate, the conditions affecting exploitation, and the resulting impact.

Consider the following CVSS 3.1 vector:

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
```

Table 2.1. Meaning of the components in the example CVSS 3.1 vector.

Code	Property	Meaning in this example
`AV:N`	Attack vector	The vulnerability can be exploited over a network.
`AC:L`	Attack complexity	Exploitation does not depend on conditions beyond the attacker's control.
`PR:N`	Privileges required	The attacker needs no prior privileges.
`UI:N`	User interaction	No user action is required.
`S:U`	Scope	The impact remains within the same security authority as the vulnerable component.
`C:H`	Confidentiality	The impact on confidentiality is high.
`I:H`	Integrity	The impact on integrity is high.
`A:H`	Availability	The impact on availability is high.

Together, these values produce a base score of 9.8, which CVSS 3.1 classifies as critical. CVSS 3.1 is the most common version in the corpus, while the newer CVSS 4.0, published in 2023, appears in more recent records.

VIPER retains the vector, version, base score, and severity label. It also uses three vector components as structured exploitation conditions: attack vector, privileges required, and user interaction. Parsing the vector provides these values directly, without natural-language interpretation. Their expected values can be established independently of the model's judgment.

2.3.2 CWE

The Common Weakness Enumeration is a catalog of general software and hardware weakness types [20]. A CVE record refers to a particular vulnerability in a product, whereas a CWE identifier describes the wider kind of weakness that caused it. For example, `CWE-79` denotes cross-site scripting and `CWE-89` denotes SQL injection; many separate CVE records may belong to either category.

CWE provides several views for organizing entries, some of which arrange weaknesses hierarchically from broad classes to more specific types [21]. CWE entries are identified as `CWE-<number>`. In a CVE Record, the identifier normally appears inside `problemTypes`. VIPER preserves the CWE identifier when one is available. Because the identifier can be copied directly from the source, its reproduction is checked by exact comparison during evaluation; the accompanying natural-language weakness name is assessed separately.

2.3.3 Availability in the Test Set

CVSS and CWE provide standardized reference values when they are available, but their inclusion varies across CVE records. The 200-record test set shows how

often the extraction system can rely on these standards. Of the 200 records, 141 carry a CWE identifier, 140 carry usable CVSS data, and 130 carry both. The gaps overlap considerably: 10 records lack only a CWE identifier, 11 lack only usable CVSS data, and 49 lack both. In total, 70 records are missing at least one of the two, while roughly one in four has neither a structured weakness classification nor a severity assessment.

The description may still give clues about the weakness or the conditions of exploitation, but it cannot be assumed to contain every missing value. When the record provides no adequate evidence, the structured output must preserve that absence rather than invent an answer. The frequent overlap between the two gaps also shows that missing CWE and CVSS data tend to occur in the same records, although the test set alone does not establish why an assigning authority omitted them.

2.4 The Demand for Structured Threat Information

2.4.1 From CVE Records to Structured Fields

The JSON format makes CVE records syntactically machine-readable, but it does not make all of their meaning available as structured data. A parser can locate the description, affected products, references, and any available classifications or metrics. It cannot, however, turn a natural-language description into separate values for the cause of the flaw, the conditions required for exploitation, and its possible consequences.

Descriptions become especially important when dedicated fields are missing, but they are not written in a uniform way. In the test set, English descriptions range from approximately 40 to 4,000 characters. A short description may provide little more than a product name and weakness type, while a longer one may include an advisory, version tables, and patch instructions. A condition such as requiring an authenticated attacker may be stated directly, encoded in a CVSS vector, implied by the contextual text, or omitted entirely.

Each VIPER value may come from a different part of the source record. Some values already exist in structured fields and can be copied or parsed. Others require information to be interpreted from the description. The remaining parts of the record may provide background without supplying a usable value. When no adequate evidence is available, the corresponding VIPER field must remain `null`.

2.4.2 Limits of Rule-Based Extraction

Descriptions that express similar facts in different ways make consistent extraction difficult. Regular expressions, keyword searches, and hand-written parsers can recognize known phrases, but each rule depends on wording anticipated by its author. CNAs may use different terminology, arrange the same information in another order, or leave a condition to be understood from context.

Additional rules can cover more patterns, but the rule set becomes harder to maintain as new writing styles appear. Previous research has used regular expressions for identifying important aspects of CVE descriptions, while noting that manually developed rules may contain human prejudices or errors and may not cover emerging description patterns [8].

Rule-based processing remains suitable for deterministic data. It can read a CVE identifier, copy reference links, and decode a supported CVSS vector. Its limitations become apparent when a target value depends on the meaning of free-form text. Fields such as the cause of a vulnerability, its possible consequences, and the way it is exploited require the surrounding context to be interpreted.

2.4.3 VIPER and Learned Extraction

Formal, machine-readable representations already support automation in several areas of security management. For example, formal policies can guide the configuration of security functions and allow the resulting network behavior to be verified automatically [22, 23, 24]. Related work applies structured reasoning to firewall anomalies and configuration changes [25, 26, 27, 28], as well as to VPN configuration, energy-aware security management, and attack mitigation [29, 30, 31]. Although these applications do not process CVE descriptions, they depend on security information being expressed in a form that automated systems can interpret consistently. Vulnerability extraction requires a similarly clear representation.

VIPER provides such a representation for the information needed in this work. Its schema separates details such as the affected product, weakness, exploitation conditions, and possible consequences into fields with consistent types and allowed values. It also preserves missing information using explicit `null` values. Records written by different CNAs can all use the identical framework.

The schema defines the expected output, but producing that output still requires the source record to be interpreted. A learned extractor can recognize relevant information across varied descriptions and use the surrounding context when a value is not expressed through a predictable phrase. Deterministic processing can still handle fields that already follow a known format, while the model addresses the parts that depend on natural-language meaning. The following chapters describe the language-model techniques used for this purpose and the design of the VIPER extraction pipeline.

Chapter 3

Large Language Models, Fine-Tuning, and Local Inference

For the extractor developed in this thesis, a language model must interpret varied CVE descriptions, learn a fixed output format, and remain practical to run on local hardware. The discussion begins with transformer language models and the Llama family, then introduces instruction tuning and chat templates. It continues with the fine-tuning and quantization methods used to adapt and deploy the models, and ends with the teacher–student process used to create the training data.

3.1 Transformer Language Models

3.1.1 Transformer Architecture

Modern language models are based on the transformer architecture [9]. Before text enters the model, a tokenizer divides it into units called tokens. A token may represent a word, part of a word, punctuation, or another recurring text portion. Each token is mapped to a numerical vector and passed through a stack of transformer layers. The model must also represent the position of each token because attention alone does not preserve sequence order. Llama incorporates this information through rotary positional embeddings within its attention layers [12]. Within every layer, attention combines information from relevant positions in the sequence, and a feed-forward network processes the resulting representation. Residual connections preserve information across the layers, and normalization keeps the intermediate values stable.

Figure 3.1 shows the encoder–decoder architecture introduced in the original transformer paper. The encoder builds a representation of the input sequence, and the decoder uses that representation to generate an output. This design was developed for tasks such as machine translation.

The Llama models used in this thesis instead follow a decoder-only architecture. The prompt and generated text pass through one stack of transformer layers, with no separate encoder or encoder–decoder attention. Causal masking prevents each position from accessing tokens that occur later in the sequence. Each prediction can use only the text available up to that point.

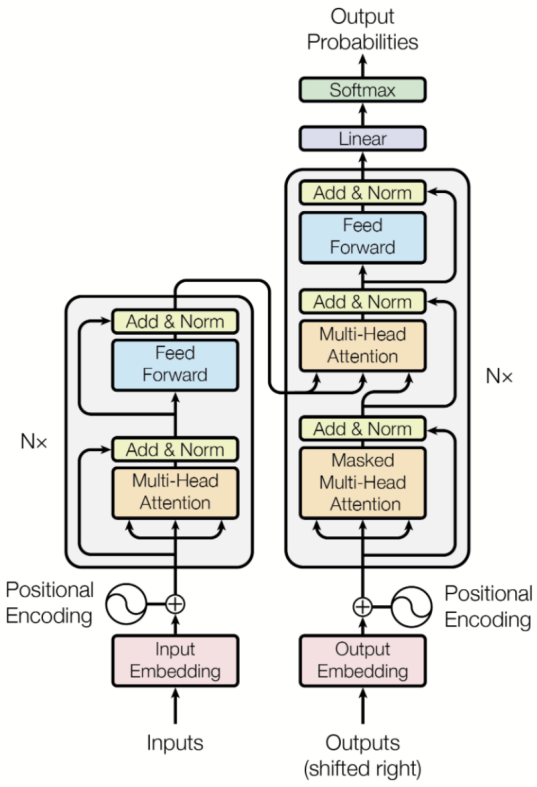


Figure 3.1. The transformer architecture as originally proposed. Figure from [9].

3.1.2 Autoregressive Generation

A decoder-only language model is trained to predict the next token from the tokens that precede it [12]. At each position, the final layer produces a probability distribution over the model's vocabulary. During generation, one token is selected from that distribution and appended to the sequence. The extended sequence is

then used to predict the next token. Generation ends when the model produces a designated stop token or reaches the configured length limit.

The next token can be selected in different ways. Greedy decoding always selects the most probable token, while sampling permits less probable alternatives. Temperature controls how concentrated the sampling distribution is: lower values favor the most likely tokens, while higher values introduce more variation.

Because tokens are generated one at a time, a valid VIPER structure is not guaranteed. The model produces text that is intended to represent a JSON object, but the unconstrained generation used in this thesis does not enforce JSON syntax. It does not guarantee balanced braces, valid field names, or allowed values. The model must learn these patterns from its instructions and training examples, and structural errors remain possible. Chapter 6 evaluates output validity separately from the content of individual fields.

3.1.3 Context and Prompted Inference

The context window sets the maximum number of tokens that the model can process in one sequence. For this task, the input CVE record, the instructions, and the generated VIPER output must all fit within that limit. Version lists and reference URLs can make raw records particularly long.

Language models can also perform some tasks from instructions and examples supplied only in the prompt, without additional task-specific training [10]. A complete VIPER schema can be placed in the prompt and used to guide extraction, provided that the combined input remains within the context window. This prompted approach forms the baseline against which the fine-tuned models are evaluated.

3.2 The Llama Model Family

Llama is a family of decoder-only language models released by Meta [12]. Meta makes weights for both pretrained and instruction-tuned Llama checkpoints available for download and local adaptation under the terms of the accompanying license. Models distributed in this way are commonly described as *open-weight*. The term indicates that the learned parameters are available, but it does not necessarily mean that the complete training data, training process, and source materials have been released under an open-source license.

Access to the weights allows the models in this thesis to be fine-tuned, merged with their adapters, quantized, and deployed on local hardware. Hosted services may allow some customization, but users usually cannot export the underlying model and deploy it independently. Local deployment also keeps the selected model

and its configuration under the user's control. This level of control supports the data-control and deployment requirements introduced in Chapter 1.

Model sizes are commonly described by their number of learned parameters. This thesis uses Llama 3.1 8B as its main model and Llama 3.2 1B for the scale comparison in Chapter 5 [12, 15]. The 1B checkpoint contains approximately 1.23 billion parameters, far fewer than the 8B model. This reduces its memory and computational requirements. Their evaluation shows how schema learning and extraction quality differ between the two checkpoints, without treating parameter count as the only source of variation.

3.3 Instruction Tuning and Chat Templates

3.3.1 Instruction Tuning

Pretraining teaches a language model to predict tokens across a broad collection of text, but it does not by itself teach the model how to respond to user requests. Instruction tuning continues training on examples that pair instructions with appropriate responses [32]. The prediction objective remains the same, but the examples teach the model to interpret its input as a request and produce a suitable answer.

Preference-based alignment may follow instruction tuning. One established approach collects human preferences between candidate responses, uses them to train a reward model, and then optimizes the language model against that reward [33]. This additional stage aims to make responses more helpful and aligned with user expectations.

Both Llama checkpoints used in this thesis are instruction-tuned models. Their existing instruction-following behavior is retained, and the project adds another stage of supervised fine-tuning using CVE-to-VIPER examples. This task-specific training teaches the models the extraction schema and the expected answer format. It does not involve a reward model, preference data, or reinforcement learning.

3.3.2 Chat Templates

Instruction-tuned models expect conversations to follow the format used during their training. A chat template serializes the conversation into a token sequence, marking both the role and the boundary of every turn. The template may include a *system* message that sets general behavior, a *user* message containing the request, and an *assistant* message containing the response. Figure 3.2 shows the conversation before it is serialized into model-specific tokens.

System	You are a cybersecurity analyst who converts raw CVE records into the VIPER JSON format.
User	Convert this raw CVE record into the VIPER schema: `CVE-2024-1234`.
Assistant	A VIPER object whose `cve_id` is `CVE-2024-1234`.

Figure 3.2. The three-message conversation used to form a training example.

The Llama 3 chat template converts these messages into one token sequence. Reserved control tokens identify each role and mark where its turn ends [34]. During inference, the sequence stops at the beginning of the assistant turn, prompting the model to generate the VIPER object.

Keeping the template consistent between fine-tuning and inference prevents the model from receiving an unfamiliar arrangement of roles and special tokens during evaluation. The prompted baseline must also use the template expected by its original checkpoint, so differences in formatting do not distort the comparison [35].

Listing A.2 in Appendix A.2 gives the serialized form used by the project, including the Llama control tokens and the boundaries between roles.

3.3.3 Training on Assistant Responses

The training procedure can also choose which tokens contribute to the loss. In this work, the user request remains part of the input context, but loss is calculated only for the assistant response. Prompt tokens are masked from the objective, directing the update toward generating the expected VIPER output.

3.4 Parameter-Efficient Fine-Tuning: LoRA and QLoRA

Full fine-tuning updates every parameter in the model. Training must store the model weights, their gradients, and the additional states maintained by the optimizer. Together, the weights, gradients, and optimizer states require considerably more memory than the stored model alone. For a model with eight billion parameters, full fine-tuning is generally impractical on a single consumer graphics processing unit (GPU).

Low-Rank Adaptation (LoRA) reduces this cost by keeping the original weight matrix W frozen and learning a separate update [13]. The update is represented as the product of two smaller matrices:

$$\Delta W = \frac{\alpha}{r} BA, \tag{3.1}$$

where r is the adapter rank and α controls the scale of the update. During the forward pass, the layer uses $W + \Delta W$, but only the parameters in A and B are trained. When the rank is small, the two narrow matrices contain far fewer trainable parameters than a full update to W.

Figure 3.3 shows the arrangement.

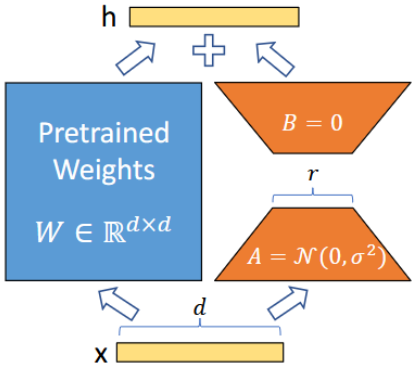


Figure 3.3. The LoRA reparametrization: the pretrained weights are frozen and only A and B are trained. Figure from [13].

The rank r determines both the width of the adapter matrices and the number of trainable parameters. The scaling factor α controls the strength of the learned update, while dropout can be applied during training as a form of regularization. LoRA adapters must also be assigned to specific model components. Attention projections are common targets, and adapters may also be added to the feed-forward projections.

After training, the adapter may remain separate from the base model or be merged into its original weights. Merging incorporates the learned update into a single model, which can then be converted and served without loading a separate adapter. The deployment pipeline in this thesis uses the merged form.

Quantized Low-Rank Adaptation (QLoRA) further reduces training memory by storing the frozen base model in a quantized 4-bit representation while keeping the trainable adapters and necessary computations at higher precision [36]. This enables larger models to be adapted with limited GPU memory. The experiments in this thesis use standard LoRA and keep the base model in 16-bit format during training. Chapter 5 explains the implementation choice and training configuration.

3.5 Quantization and Local Inference

Model weights are commonly stored in 16-bit floating-point formats during fine-tuning. Post-training quantization converts them to a lower-precision representation for deployment, often using approximately four bits per weight. The weights are processed in groups. Each group is stored using lower-precision values together with the scale needed to interpret them. This reduces storage and memory use, although the approximation may affect model output.

In this project, quantization is applied after fine-tuning and adapter merging. The merged model is converted to GGUF, a format used by the `llama.cpp` ecosystem to package model tensors, tokenizer information, and metadata needed for inference [14]. It is then quantized with the `Q4_K_M` scheme. This scheme uses a K-quant representation centered on approximately 4-bit weights and may retain higher precision for selected tensors, depending on the model and conversion settings [37].

At 16 bits, eight billion parameters require approximately 16 GB for the weights alone. The `Q4_K_M` model produced in this project occupies roughly 4.92 GB, while the quantized 1B model occupies 0.81 GB. Inference requires additional memory for runtime state and the key–value cache, but these model sizes remain practical for consumer hardware.

For inference, `llama-server`, provided by the `llama.cpp` project, loads the quantized GGUF model and provides an OpenAI-compatible application programming interface (API) over the Hypertext Transfer Protocol (HTTP) [14]. The evaluation code can submit requests to every local model through the same interface. Chapter 5 describes the server configuration and generation settings used in the experiments.

3.6 Knowledge Distillation with Synthetic Supervision

Supervised fine-tuning requires an expected VIPER output for every input record. Producing these targets manually would require detailed annotation of hundreds of CVE records, so this thesis uses a stronger language model to generate them. That model acts as the teacher, and Llama learns as the student. This process is related to knowledge distillation.

Traditional knowledge distillation transfers information from a larger teacher model to a smaller student model [38]. The student is trained using the teacher's output distribution, whose probabilities convey more information than a single hard label.

In this work, the student never receives the teacher's internal probabilities.

DeepSeek receives each raw CVE record together with the VIPER instructions and generates a structured response [11]. That response is paired with the original record and used as the supervised target for Llama. Since another model generates the target responses, they provide synthetic supervision rather than classical distillation targets.

Figure 3.4 summarizes the process: the teacher converts raw records into structured targets, and the student learns from the resulting input–output pairs.

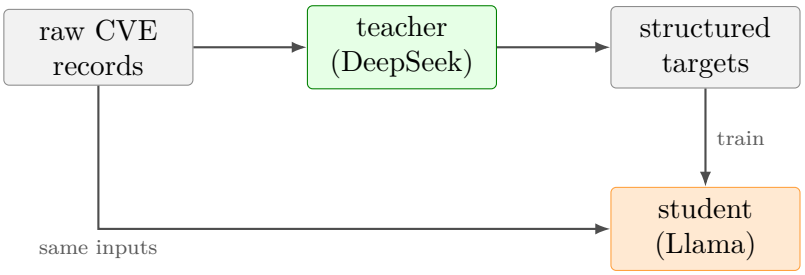


Figure 3.4. The teacher–student setup used to build the training data.

Teacher responses may contain errors or unsupported interpretations. Once included in the training data, these problems may also be learned by the student. For this reason, the evaluation distinguishes fields that can be checked directly against the source record from those that require interpretation. Some fields cannot be evaluated reliably through exact comparison because two correct answers may use different wording. A language model can instead receive the source record and a generated extraction and assess whether the extraction is complete and faithful. This method, commonly called LLM-as-a-judge, provides a reference-free assessment of qualities that are difficult to capture with fixed metrics [39]. Its scores remain model-based judgments and may contain biases, so they complement structured and field-level metrics instead of replacing them. The judge used in this work is separate from both the teacher that generated the training targets and the Llama models being evaluated.

With the data source and modeling methods established, Chapter 4 defines the goals, scope, constraints, and research questions that guide the implementation and evaluation.

Chapter 4

Thesis Objectives

Chapters 2 and 3 established the background for the problem addressed in this thesis. CVE records contain useful security information, but much of it appears in prose written by numerous organizations. Language models can interpret this prose, and open-weight models can be adapted to the task and deployed locally. These observations lead to the objectives and research questions defined below.

4.1 Goals

The main goal is to build and evaluate a system that converts a raw CVE record into the fixed VIPER format using an open-weight language model that runs locally. This goal is divided into five specific objectives:

1. **Define a fixed format.** Design the VIPER schema and specify its fields, types, and allowed values.

2. **Construct the training data.** Create CVE-to-VIPER pairs using targets generated by a stronger model.

3. **Learn the schema from examples.** Fine-tune an open-weight model to produce VIPER records without receiving the complete schema at inference time.

4. **Support local inference.** Merge, quantize, and serve the fine-tuned model on consumer hardware.

5. **Evaluate the approach.** Compare fine-tuned and prompted models on a test set kept separate from training, and analyze how the results change with model size.

A small number of convincing outputs cannot establish how consistently the extractor performs, so evaluation is included as one of the project's objectives. An early look at the base model suggested that it already handled the task when given the schema in its prompt, but that impression came from reading only a few outputs. Testing it requires a separate dataset, a reference for comparison, and metrics that distinguish a well-formed record from one whose contents are correct.

4.2 Scope and Constraints

Data. The work uses CVE records from the public `cvelistV5` repository [2], restricted to identifiers from 2021 to 2026 and to records that are published and carry an English description. Older records and records in other languages are left out.

Models. Only open-weight models are fine-tuned or deployed. Llama 3.1 8B is the main model and Llama 3.2 1B is used for the comparison across scale. Hosted models appear in two supporting roles only: DeepSeek produces the training targets, and Claude acts as the judge during evaluation. Neither model forms part of the delivered extractor. Assigning target generation and evaluation to separate models prevents one system from filling both roles, as discussed in Chapter 6.

Fixed format. The target format is fixed before training and does not change per record. The aim is not to propose a new standard for the community, but to show that a model can be taught to produce one reliably. An organization with a different internal format could repeat the method with its own schema.

Learning from examples. The fine-tuned model is not given the schema at inference time. Its system prompt names the format but does not describe it. This constraint makes the comparison with the prompted baseline meaningful.

Hardware and local inference. Training is limited to a single GPU, which motivates the use of LoRA instead of full fine-tuning. Quantization reduces the memory required for local inference on consumer hardware.

Out of scope. The thesis does not cover multilingual extraction, comparison with commercial extraction products, or integration into an operational security-management pipeline. An example of such a downstream system is the autonomous cyberattack-mitigation process proposed by Bringhenti et al. [31]. Prompt engineering is used only to construct a fair baseline; optimizing prompts as an alternative solution falls outside the scope of the work.

4.3 Research Questions

The evaluation addresses three research questions, which are answered in Chapter 6.

RQ1: Schema learning. Can an open-weight model learn the VIPER schema from examples and produce conforming records without receiving the schema definition at inference time?

The fine-tuned models never receive a description of the schema when they run. Structural measurements show how often their outputs are valid JSON and how often they conform to the formal schema.

RQ2: Effect of fine-tuning. Does fine-tuning improve structural validity, field-level accuracy, and overall faithfulness compared with prompting the corresponding base model?

Within each model size, the base and fine-tuned checkpoints receive the same 200 test records. The base models are given the full schema in their prompts, while the fine-tuned models are not. The evaluation covers structure, field-level accuracy, and a separate judgment of how faithfully each output represents its source.

RQ3: Comparison across model sizes. How do schema learning and the improvement obtained through fine-tuning differ between Llama 3.1 8B and Llama 3.2 1B?

Both checkpoints follow the same procedure. The results indicate whether the schema can be learned at the smaller scale and how closely a fine-tuned 1B model approaches the performance of the larger models. Since the checkpoints belong to different Llama releases, the results do not isolate parameter count from every other model difference.

Chapter 5

Methodology

The methodology defines the VIPER schema and draws separate training and test sets from `cvelistV5`. A teacher model supplies the training targets, and the training pipeline fine-tunes the 8B and 1B Llama checkpoints with LoRA. The deployment pipeline merges and quantizes the adapters before serving the models locally. Prompted versions of the corresponding base models provide the baselines, and the evaluation processes all four models on the same test records.

5.1 Pipeline Overview

The workflow has two connected paths. The first turns selected CVE records into CVE–VIPER pairs, uses them to fine-tune the 8B and 1B checkpoints, and prepares the resulting models for local deployment. The second sends the independent test set to both the prompted base models and their fine-tuned counterparts, producing four sets of predictions for evaluation. Figure 5.1 summarizes these paths.

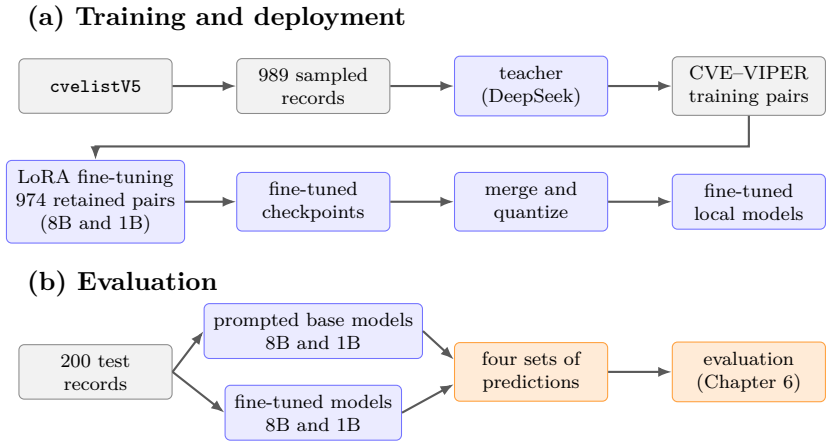


Figure 5.1. The stages of the pipeline. The test records are separated before target generation. The evaluation compares prompted base models with their fine-tuned counterparts at both model sizes.

Three controls make the model comparisons consistent. **Data separation:** the sampling procedure separates the test records before target generation and keeps them outside the training path. **Teacher isolation:** DeepSeek produces the training targets but takes no part in inference. **Controlled comparison:** the base and fine-tuned models process the same test records through the same local runtime. Their prompts differ because the base models receive the complete schema, while the fine-tuned models receive the minimal prompt used during training.

5.2 The VIPER Schema

The target format is called VIPER, for Vulnerability Intelligence Parsed Extraction Record. It is a single JSON object with seven top-level fields, shown in abbreviated form in Listing 5.1. The complete schema used by the validator is reproduced in Appendix A.1.

```
{
  "cve_id": "string",
  "vulnerability_type": { "name": "string|null", "cwe_id": "string|null" },
  "affected_system": {
    "vendor": "string|null", "product": "string|null",
    "versions": ["string", "<additional values>"], "component": "string|null"
  },
  "threat_information": {
    "caused_by": ["string", "<additional values>"],
    "can_lead_to": ["string", "<additional values>"],
    "exploited_by": "string|null",
    "preconditions": {
      "privileges_required": "none|low|high|null",
      "user_interaction":    "none|required|null",
      "attack_vector":       "network|adjacent|local|physical|null",
      "other_conditions":    ["string", "<additional values>"]
    }
  },
  "severity": {
    "cvss_version": "string|null", "cvss_vector": "string|null",
    "base_score": "number|null",
    "severity_label": "LOW|MEDIUM|HIGH|CRITICAL|null"
  },
  "references": ["string", "<additional values>"],
  "description": "string"
}
```

Listing 5.1. Abbreviated structure of a VIPER record.

The seven top-level fields contain twenty leaf values. `cve_id` identifies the source record. `vulnerability_type` names and classifies the weakness, while `affected_system` identifies the vendor, product, affected versions, and component. `threat_information` records the cause, possible consequences, exploitation method, and required preconditions. `severity` contains the CVSS information. Finally, `references` and `description` preserve supporting material from the source record.

Four schema rules directly affect the extraction task:

1. **The required structure remains present.** All seven top-level fields are required, and nested objects must retain their object shape. When a leaf value cannot be established, the missing value is represented inside its parent object. A required object cannot be replaced with `null`.

2. **Missing scalar and list values are represented differently.** As discussed in Section 2.3, CVE records frequently lack a CWE identifier or CVSS data. Many scalar fields may be `null` when a value cannot be established from the source. The identifier and description remain strings, while required objects and list-valued fields remain present so that every VIPER record retains the same overall structure. The schema sets no minimum length for an array, so an empty list represents a multi-valued field for which the source supports no

values.

3. **The preconditions use a closed vocabulary.** The three precondition fields, covering the privileges an attacker needs, whether a user must act, and where the attack comes from, are each restricted to a small vocabulary based on the corresponding CVSS metrics [5, 6]. A CVSS vector encodes these three properties, so VIPER can obtain their values by parsing and, where necessary, normalizing the vector. CVSS 3.x values `UI:N` and `UI:R` map to `none` and `required`. CVSS 4.0 distinguishes passive and active user interaction; the reference-construction code maps both `UI:P` and `UI:A` to VIPER's broader `required` value. Chapter 6 uses this property to build a reference that does not depend on any model's judgment.

4. **No extra fields are allowed.** The schema is formalized as a JSON Schema document [40], with `additionalProperties` set to false at every level. A record containing an unexpected field is invalid, even when the rest of its content is correct. This rule gives schema conformance a precise meaning for records intended to be consumed by software.

The fields also differ in how their values are obtained. Some are copied from the source record, while others can be derived deterministically, particularly from a CVSS vector. Fields describing causes, consequences, and exploitation may require interpretation of free text. Chapter 6 uses these differences when constructing the reference and selecting the evaluation metrics.

5.3 Corpus Sampling

Sampling used a snapshot of the `cvelistV5` repository cloned on 4 April 2026 [2]; Appendix A.4 records the corresponding Git commit. It considered records from 2021 to 2026, focusing on recent disclosures and limiting the variation found in the formatting of older entries. Eligible records had to be in the `PUBLISHED` state and include an English description. Reserved and rejected entries were excluded because they do not provide a published vulnerability description suitable for extraction.

The candidate pool was shuffled before sampling. CVE identifiers are grouped by year and may also cluster by assigning authority, so following the repository order could overrepresent the years or authorities listed first. Table 5.1 gives the resulting distribution.

The resulting sample contained 989 source records. A teacher-generated target was added to each one, forming the dataset supplied to both training runs. Section 5.6 describes the subsequent preprocessing.

Table 5.1. Year distribution of the training and test records.

Set	2021	2022	2023	2024	2025	2026	Total
Training	102	160	157	206	218	146	989
Test	22	22	34	36	50	36	200

Training records were sampled with seed 42 by `extract_raw_cves.py`. The test set was assembled separately: an initial set of 60 records was later extended by 140 records sampled with seed 7. The test-generation script excluded every identifier in the training manifest, and a direct comparison of the completed manifests confirms that they have no identifiers in common. None of the test records entered target generation or fine-tuning. DeepSeek was used later when constructing the test references, as described in Chapter 6.

Knowing the seed is not enough to reproduce either draw. The result also depends on the Python random-number generator and the order in which the filesystem returns files. The files `raw_cves_manifest.json` and `test_cves_manifest.json` list the exact identifiers used in the two sets. The scripts can rebuild each set from its manifest, so these explicit lists serve as the authoritative definitions of the experimental data.

5.4 Teacher-Generated Targets

Manually writing 989 VIPER targets was impractical, so DeepSeek generated them using the distillation procedure described in Section 3.6 [11].

The compaction function prepared each record before target generation. It removed the `x_legacyV4Record` block because that block duplicates the record in the older CVE format. It also copied unique reference URLs from the `adp` containers into the CNA reference list before removing those containers. This preserved the links used by VIPER while omitting the remaining ADP enrichment. The dataset builder applied the same function when assembling the training conversations, so the student received the representation used to generate its target.

The teacher received a long system prompt containing the schema represented in Listing 5.1, together with instructions for every field. The prompt requested one JSON object. JSON mode constrained the response to valid JSON [41], and a temperature of 0.1 limited sampling variation. The script `generate_target_cves.py` sent requests directly to DeepSeek's OpenAI-compatible chat-completions endpoint using the Python `requests` library. It used four concurrent workers by default and a 120-second timeout. Rate-limit responses, server errors, network failures, and responses that could not be decoded as JSON were retried up to five times with exponential backoff capped at 30 seconds. A response that lacked one of the seven

top-level fields was rejected but not retried. Records with an existing saved target were skipped, allowing generation to resume after an interruption.

The generation script produced the targets in April 2026 through the `deepseek-chat` endpoint. It did not store a more specific model identifier.

The complete teacher prompt and the compact schema representation embedded in it are reproduced in Appendix A.2.

Listings 5.2 and 5.3 present an abbreviated example for CVE-2022-45482: the compacted record supplied to the teacher and the VIPER object returned for it. Both are taken from the saved dataset; administrative fields, long text, and URLs are shortened only for display.

```
{"cveMetadata": {"state": "PUBLISHED", "cveId": "CVE-2022-45482",
                 "<omitted>": null},
 "containers": {"cna": {
   "descriptions": [{"lang": "en", "value":
      "Lazy Mouse server enforces weak password requirements and doesn't
       implement rate limiting, allowing remote unauthenticated users to
       easily and quickly brute force the PIN and execute arbitrary
       commands. CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H"}],
   "affected": [{"vendor": "thisAAY", "product": "Lazy Mouse",
                 "versions": [{"version": "<= 2.0.1"}]}],
   "references": [{"url": "https://www.synopsys.com/blogs/..."}],
   "problemTypes": [{"descriptions": [{"type": "CWE", "cweId": "CWE-521",
                     "description": "CWE-521: Weak Password Requirements"}]}]
 }}}
```

Listing 5.2. Compacted teacher input for CVE-2022-45482.

```
{"cve_id": "CVE-2022-45482",
 "vulnerability_type": {"name": "Weak Password Requirements",
                        "cwe_id": "CWE-521"},
 "affected_system": {"vendor": "thisAAY", "product": "Lazy Mouse",
                     "versions": ["<= 2.0.1"], "component": null},
 "threat_information": {
   "caused_by": ["weak password requirements", "no rate limiting"],
   "can_lead_to": ["remote code execution", "arbitrary command execution"],
   "exploited_by": "brute force PIN",
   "preconditions": {"privileges_required": "none", "user_interaction": "none",
                     "attack_vector": "network", "other_conditions": []}},
 "severity": {"cvss_version": "3.1",
              "cvss_vector": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H",
              "base_score": 9.8, "severity_label": "CRITICAL"},
 "references": ["https://www.synopsys.com/blogs/..."],
 "description": "Lazy Mouse server enforces weak password requirements ..."}
```

Listing 5.3. Abbreviated VIPER target generated for CVE-2022-45482 and used as the training label.

The output combines values copied from the source with values inferred from its description. The vendor, product, version range, and CWE identifier come from existing structured fields. The `caused_by` list instead separates weak password requirements and missing rate limiting from a single sentence, while `can_lead_to`

expresses the consequences in normalized terms.

This record carries no `metrics` block at all; the CVSS vector appears only as a string at the end of the description, where the reporting organization had appended it. The teacher recovered it from the prose, derived the three preconditions from its components, and assigned the base score and label implied by the vector. The fine-tuned model must learn each of these transformations from the generated examples. The results in Chapter 6 report the largest fine-tuning gains on these derived fields.

The generation script checked each target for the seven required top-level fields before saving it. This light check did not catch everything: a later checking against the formal schema found that two of the 989 generated targets, CVE-2023-30085 and CVE-2023-30297, used `local` as the value of `privileges_required`. These cases illustrate the risk discussed in Section 3.6: synthetic targets can contain errors and should be validated before training.

Exact regeneration of the targets is not guaranteed because sampling at temperature 0.1 is not deterministic and the model behind the `deepseek-chat` name may change over time. For this reason, the saved dataset serves as a more precise record of the experiment than a new run through the current API. The dataset is published on the Hugging Face Hub as `mx003/viper`.

5.5 Training-Data Formatting

The dataset builder stored each raw record and its target as a three-turn conversation: a system turn defines the task, a user turn contains an instruction and the compacted CVE record, and an assistant turn contains the VIPER JSON. It wrote one conversation per line to `dataset.jsonl`. Each line contains a `messages` array with the three roles and their text; the Llama special tokens are not stored in the file. Axolotl applies the Llama 3 chat template when the data are prepared for training [12, 42]. Every user turn begins with the same instruction, "Convert this raw CVE record into the VIPER schema:", followed by a newline and the compacted record.

The minimal system prompt is central to the experimental design. It reads, in full:

```
You are a cybersecurity analyst who converts raw CVE records
into the VIPER (Vulnerability Intelligence Parsed Extraction
Record) JSON format.  Respond with a single valid JSON object
and nothing else.
```

It names the format and asks for JSON. It does not list the fields, describe the nesting, or give the allowed values. The teacher received a detailed prompt, whereas the fine-tuned models had to learn the schema from the 974 retained target

responses without receiving its definition in their prompt. Whether this works is the focus of RQ1 in Section 4.3, and Chapter 6 answers it.

The training objective calculates loss only on the assistant response. Tokens from the system and user turns remain in the input context but do not contribute to the objective [42]. In both Axolotl configurations, the dataset type is `chat_template`, the conversation field is `messages`, and `train_on_inputs: false` applies this masking after the Llama 3 chat template is rendered.

5.6 Fine-Tuning Setup

Axolotl [43] reads the model, dataset, and hyperparameters from a YAML (YAML Ain't Markup Language) configuration file and runs the training procedure. Table 5.2 lists the main settings for the 8B run. The 1B configuration uses the same recipe with a different base checkpoint and output directory, as described in Section 5.8.

Table 5.2. Fine-tuning configuration.

Setting	Value
Base model	`Meta-Llama-3.1-8B-Instruct`
Adapter	LoRA
Rank r / α	32 / 64
LoRA dropout	0.05
Target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj` `gate_proj`, `down_proj`, `up_proj`
Precision	bf16
Epochs	2
Learning rate	2×10^{-4}, cosine schedule
Warmup	3% of steps
Optimizer	`adamw_torch_fused`
Sequence length	4096
Micro-batch / accumulation	2/8 (16 sequences per update)
Validation split	5%
Gradient checkpointing	enabled
Sample packing	enabled
Save / logging interval	10 / 5 steps
Loss on inputs	disabled

The two configuration files identify the checkpoints as `unsloth/Meta-Llama-3.1-8B-Instruct` and `unsloth/Llama-3.2-1B-Instruct`. Table 5.2 reports the

parameters explicitly set for the experiments, while Appendix A.3 reproduces the configuration files and records the available reproducibility details.

The configuration applies LoRA to four attention projections and three feed-forward projections in every transformer block. Both models use rank 32. The experiments did not search over alternative ranks, so the selected configuration does not represent an estimated optimum.

The 4096-token sequence length accommodates a compacted CVE record and its VIPER target in one training sequence. Some records are long, which is why compaction is part of data preparation. Sample packing combines short examples into full windows and reduces padding [42]. With packing, the effective batch of 16 refers to packed sequences per optimizer update, not necessarily to 16 original conversations.

The project documentation records that 15 of the 989 conversations exceeded the 4096-token limit and were discarded instead of truncated. This left 974 conversations before Axolotl applied the 5% validation split. Both model configurations specify the same split fraction, but the files do not preserve the selected validation records or the seed used to choose them.

Four-bit and eight-bit loading are disabled in both configurations, so the method uses LoRA [13] instead of QLoRA [36] and loads the base weights in bfloat16. Flash Attention is likewise disabled. The 8B configuration uses PyTorch's scaled dot-product attention interface [44]. These choices change the memory and runtime requirements without changing the training objective.

Training used an RTX 5090 with 32 GB of GPU memory. The recorded duration was approximately one hour, although the surviving notes do not separate the two run times. They also omit the Axolotl, PyTorch, and CUDA versions.

Axolotl reserved 5% of the available conversations for validation, but the surviving trainer state contains no evaluation entries. Consequently, no validation metric guided model selection. The runs did not use early stopping, and the pipeline merged the adapter left in the output directory at the end of training.

The 1B run completed all 32 configured steps. Its last logged training loss was 0.082 at step 30; the final two steps occurred between logging intervals. The 8B trainer state was lost when the training environment was removed, so its final loss and completion of the configured two epochs cannot be confirmed.

Training and inference impose different hardware requirements. Fine-tuning needs enough accelerator memory for the base weights, adapter states, and training activations, but it is performed only when producing a new model. Inference runs whenever a record is processed and has to fit the user's machine. The next section explains how merging and quantization reduce that deployment cost.

5.7 Quantization and Local Serving

The deployment pipeline merged each adapter into its base weights and converted the resulting checkpoint to GGUF. It then applied the `Q4_K_M` quantization used in the evaluation [14, 37]. The base models used the same quantization as their fine-tuned counterparts. The final `Q4_K_M` files occupy 4.92 GB for the 8B model and 0.81 GB for the 1B model.

The recorded 8B procedure merges the adapter with `axolotl merge-lora config.yaml`, checks that the merged tokenizer configuration contains a chat template, and uploads the merged checkpoint to the Hugging Face Hub. The `ggml-org/gguf-my-repo` Hugging Face Space then performed conversion and quantization. Appendix A.3 records the available commands and conversion details.

LM Studio served the quantized models locally through an OpenAI-compatible HTTP interface [45]. The inference script requested responses at temperature 0 with a maximum of 2048 new tokens. This limit bounds the length of generation and normally leaves room for a complete VIPER object, but it cannot ensure that every response finishes before reaching the cap. The script also supplied four stop strings, including the Llama end-of-turn and end-of-text tokens, and processed one request at a time. Sampling parameters not included in the request, such as `top_p` and `top_k`, were left to the runtime defaults.

The output parser attempted to recover a complete JSON object when generation reached the token limit. Chapter 6 reports how often this occurred and how many responses remained unparseable.

Inference ran locally on Windows through LM Studio and its Vulkan-based `llama.cpp` backend. Within each model size, the base and fine-tuned variants used the same context length: 22,784 tokens for the 8B pair and 131,072 for the 1B pair. Appendix A.3 records the available hardware and runtime details.

At each model size, the base and fine-tuned models processed the same 200 records with the same quantization, runtime, and generation settings. The two approaches differed in both their weights and their prompts. The base models retained their original weights and received the complete schema in their prompt. The fine-tuned models used the adapted weights and the minimal prompt seen during training. Giving the complete schema to the base models creates a strong prompted baseline for the learned extractors.

The inference script uses the same baseline prompt at both model sizes. It consists of the teacher system prompt followed by an instruction not to call a tool or function and to return only the raw VIPER object. The prompt material to be preserved with the experiment is identified in Appendix A.2.

The inference script offers two request paths. Its default path sends the system and user turns to the chat-completions endpoint. An optional `-use-completions` flag instead serializes the same turns with the Llama 3 chat template and sends

them to the raw completions endpoint. This option was added after the chat endpoint caused tool-call syntax to appear in baseline responses. All four final prediction runs used `/v1/completions`, with the script applying the Llama 3 chat template before each request.

5.8 Comparison Across Model Sizes

The model-size comparison repeats the Llama 3.1 8B training and evaluation procedure with Llama 3.2 1B [12, 15].

The checkpoints follow the same training recipe. The 1B configuration changes the base model to `Llama-3.2-1B-Instruct` and assigns a different output directory. They draw from the same 989-conversation source dataset, including the same 4096-token filtering described above, and the settings listed in Table 5.2. The deployment pipeline quantizes both to `Q4_K_M` and serves them through the same runtime with the same generation settings. The evaluation uses the same 200 test records, references, and metrics. The experiments did not run a separate hyperparameter search for the smaller model, so its results show how the unchanged 8B recipe transferred to that checkpoint. They do not estimate the best performance available from a separately tuned 1B configuration.

Both YAML files specify the Llama 3 chat template and differ only in the base checkpoint and output directory. The archived prediction directories are `predictions/base`, `predictions/finetuned`, `predictions_llama3-2/base-1b`, and `predictions_llama3-2/finetuned-1b`. Each directory accounts for all 200 test records through either a parsed JSON prediction or a saved raw response.

The evaluation covers four models: the base and fine-tuned variants at 8B and 1B. The checkpoints differ most visibly in parameter count, but they also belong to different Llama releases. The results show how the method behaves at these two model sizes without isolating parameter count as the sole cause of any difference. Chapter 6 evaluates the four prediction sets for structural validity, field-level accuracy, and overall faithfulness.

Chapter 6

Validation

Four sets of predictions are evaluated: the base and fine-tuned variants of Llama 3.1 8B and Llama 3.2 1B. A 200-record test set is used to measure structural validity, field-level accuracy, and overall faithfulness. Before comparing the four models, the evaluation defines how failed outputs affect each metric. The resulting measurements answer the research questions from Chapter 4.

6.1 Test Set and Reference Construction

The evaluation uses the 200-record test set introduced in Section 5.3. It follows the same eligibility criteria as the training corpus, and the two manifests contain no common identifiers. None of the test records entered target generation or fine-tuning.

Evaluating a prediction requires a reference, but VIPER fields do not all have the same type of reference value. Some can be recovered deterministically from the CVE record, while others require interpretation and may be expressed in more than one valid way.

The reference separates the evaluated fields into two groups. Eleven fields are treated as *objective*: the identifier, the CWE, the four CVSS fields, the three preconditions, the reference list, and the description. Their values come from parsing the source record. The three preconditions require an additional step because they are not stored as separate fields. Their values are recovered by decoding the CVSS vector [5, 6] described in Section 2.3; a vector containing `AV:N` fixes the attack vector as `network`, and so on. Because this is a deterministic transformation of data already in the record, no model-generated judgment is involved.

The remaining nine fields are *subjective*: the weakness name, the vendor, product, versions and component, the causal fields, the exploitation method, and any extra

conditions. After training-data generation, a separate teacher call produced an initial reference for these interpretive fields in the test set. All 200 draft references were compared with their corresponding raw CVE records, and 124 were revised. The changelog records each revised field together with its original and replacement values.

The results report the two groups separately. The objective macro compares predictions with values recovered deterministically from the source record. The subjective macro compares them with the human-reviewed reference, so it retains some uncertainty from interpretive annotation.

6.2 Evaluation Metrics

No single number describes an extraction well. A record can be perfectly formed and full of wrong values, or carry the right information in a shape no program can read. For this reason, the evaluation uses three layers, each answering a different question: is the output usable at all, are the values right, and is the record faithful to its source. Figure 6.1 expands the evaluation stage of the pipeline in Figure 5.1 into those three layers.

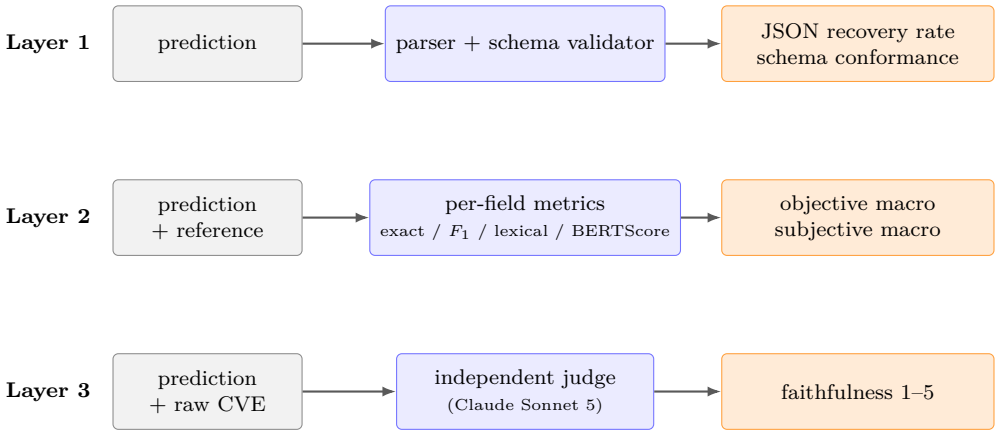


Figure 6.1. The three evaluation layers. Only Layer 2 consults the reference; the judge in Layer 3 sees the source record instead, which is what makes it reference-free.

6.2.1 Structural Validity

Structural validity measures whether the pipeline returns a machine-readable object.

The **JSON-valid rate** is the share of test records from which the inference pipeline recovers a JSON object. The parser first reads the complete response, then checks inside a fenced code block, and finally scans for the first balanced object. If a response ends before the object is closed, it also attempts to close an open string

and add the missing braces. An output counts as unparseable only when these steps fail. The metric reflects the usable output of the pipeline after parsing, not the syntax of the untouched model response.

The pipeline also restores the CVE identifier from the source record when a parsed object omits `cve_id`. The structural and content scores are computed after this step, so they describe the complete extraction pipeline. No other VIPER field is inserted during parsing.

The **schema conformance rate** asks the stricter question of whether the parsed object is actually a VIPER record. The schema validator checks the seven top-level fields and their nested requirements, verifies types and enumerated values, and rejects unexpected fields.

The two are not the same measurement, and the gap between them is significant. A model can produce flawless JSON that is not a VIPER record at all.

6.2.2 Content Accuracy

Content accuracy measures how closely each field matches its reference value. The metric depends on the kind of field because one comparison rule would not suit all of them.

Two helpers are shared by all of them. Text is normalized before comparison by lowercasing, trimming, and collapsing runs of whitespace, written $\nu(x)$; and a value is reduced to its set of alphanumeric tokens, written $\tau(x)$. Overlap between two token sets is measured by the Jaccard index,

$$J(A, B) \;=\; \frac{|A \cap B|}{|A \cup B|}\,. \tag{6.1}$$

Throughout, $N = 200$ is the number of test records, and $s_f(i) \in [0,1]$ is the score of field f on record i.

Exact match. The evaluation uses exact match for nine objective categorical and scalar fields. These values can be checked without interpretive matching. A CVE identifier, CVSS vector, or severity label is counted as correct only when it agrees with the reference. For a prediction p and reference r,

$$s_{\text{exact}}(p, r) \;=\; \begin{cases} \mathbb{1}[\,|p - r| \le \delta\,] & \text{if both are numeric,} \\ \mathbb{1}[\,\nu(p) = \nu(r)\,] & \text{otherwise,} \end{cases} \tag{6.2}$$

with a tolerance of $\delta = 0.1$ on the base score. Two nulls normalize to the same empty string and count as agreement. Two equivalences are applied and documented: a null base score is treated as equal to 0.0, since the target-generation instructions allow either when no CVSS exists, and `adjacent_network` is treated as equal to

`adjacent`, since they name the same CVSS value. Both are applied identically to every model.

Set F_1. The evaluation uses set F_1 for the five list fields. If the reference holds three URLs, a model that finds two should not score the same as one that finds none, which is what exact match would give.

Matching between items is soft. Two entries a and b count as the same when their token sets are equal or overlap enough,

$$m(a,b) \;=\; \mathbb{1}\Big[\, \tau(a) = \tau(b) \;\;\vee\;\; J\Big(\tau(a), \tau(b)\Big) \geq \theta \,\Big], \qquad \theta = 0.6, \tag{6.3}$$

so a difference regarding punctuation or word order does not destroy the score. Given a predicted list P and a reference list R, each predicted item is matched greedily against the first reference item not yet taken, which makes the pairing one-to-one; TP denotes the number of pairs formed. Then

$$\mathrm{Prec} = \frac{TP}{|P|}, \qquad \mathrm{Rec} = \frac{TP}{|R|}, \qquad F_1 = \frac{2\,\mathrm{Prec}\cdot\mathrm{Rec}}{\mathrm{Prec}+\mathrm{Rec}}\,. \tag{6.4}$$

The harmonic mean penalizes an imbalance between precision and recall. A cautious model that predicts one correct item out of three earns $\mathrm{Prec} = 1$ but $\mathrm{Rec} = 0.33$, giving $F_1 = 0.50$; a model that lists ten items to be sure of covering the three earns $\mathrm{Rec} = 1$ but $\mathrm{Prec} = 0.3$, giving $F_1 = 0.46$. Both receive a moderate score despite perfect performance on one side of the calculation.

Two empty lists score 1, which correctly rewards a model for saying nothing when the record says nothing, and a mismatch where exactly one side is empty scores 0.

Lexical similarity. The evaluation uses lexical similarity for the six free-text fields, where two answers can both be right while worded differently. The metric averages two classical measures, one sensitive to character order and one not:

$$\mathrm{sim}(p,r) \;=\; \tfrac{1}{2}\Big[\, \mathrm{SM}\Big(\nu(p), \nu(r)\Big) \;+\; J\Big(\tau(p), \tau(r)\Big) \,\Big]. \tag{6.5}$$

The first term is the ratio returned by Python's `SequenceMatcher`, an extension of Ratcliff–Obershelp pattern matching [46]. It recursively identifies matching subsequences and reports

$$\mathrm{SM}(a,b) \;=\; \frac{2M}{T}\,, \tag{6.6}$$

where M is the number of matched characters and T the combined length of the two strings. The second term is the token Jaccard index of Equation (6.1), which

ignores order entirely. Averaging them means a prediction is rewarded both for using the right words and for arranging them similarly.

The measure is deterministic and needs no model, so the objective macro stays independent of any learned component. Its weakness is that it is purely lexical: "SQL injection is possible" and "an attacker can execute arbitrary SQL" share one token and score low, although they say the same thing.

BERTScore [47] addresses exactly that weakness. Its name derives from Bidirectional Encoder Representations from Transformers (BERT), although related encoder models can supply its embeddings. It compares meaning rather than surface form. Both texts are passed through a transformer, which yields one contextual embedding per token: $x = (x_1, \ldots, x_k)$ for the reference and $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_l)$ for the prediction. Each token is then matched to its most similar counterpart by cosine similarity, with recall matching every reference token and precision every predicted token:

$$R_{\mathrm{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} x_i^\top \hat{x}_j \,, \qquad P_{\mathrm{BERT}} = \frac{1}{|\hat{x}|} \sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} x_i^\top \hat{x}_j \,, \tag{6.7}$$

$$F_{\mathrm{BERT}} = \frac{2\, P_{\mathrm{BERT}}\, R_{\mathrm{BERT}}}{P_{\mathrm{BERT}} + R_{\mathrm{BERT}}} \,. \tag{6.8}$$

Here, a match represents semantic proximity between contextual embeddings. Because raw cosine similarities occupy a narrow band well above zero, scores are rescaled against a baseline computed on random sentence pairs, so that an unrelated pair sits near 0 rather than near 0.8. The embeddings come from `roberta-large`.

BERTScore is reported for the same six free-text fields. Reading it alongside lexical similarity helps distinguish different wording from different content.

Aggregation. A field's score is the mean of its per-record scores over the whole test set,

$$S_f \;=\; \frac{1}{N} \sum_{i=1}^{N} s_f(i) \,, \tag{6.9}$$

and the two macro averages are the unweighted means of those field scores over the eleven objective and nine subjective fields respectively,

$$\mathrm{OBJ} = \frac{1}{|\mathcal{F}_{\mathrm{obj}}|} \sum_{f \in \mathcal{F}_{\mathrm{obj}}} S_f \,, \qquad \mathrm{SUBJ} = \frac{1}{|\mathcal{F}_{\mathrm{subj}}|} \sum_{f \in \mathcal{F}_{\mathrm{subj}}} S_f \,. \tag{6.10}$$

Averaging over fields rather than over records gives every field equal weight, so a field that is easy and frequent cannot dominate a field that is hard.

6.2.3 LLM-as-a-Judge

The final layer evaluates the extraction as a whole. A prediction can score well field by field and still misrepresent the vulnerability, and it can word things differently from the reference while being entirely correct.

The independent judge receives the raw CVE record and the prediction, then rates on a scale from 1 to 5 how faithfully and completely the extraction captures the record. It never sees the reference, which makes this a reference-free measurement and means it does not inherit whatever the reference gets wrong. Each prediction is graded on its own rather than compared against another, following the single-answer grading protocol described by Zheng et al. [39]; because there is no pair of answers to order, the position bias they document for pairwise judging cannot arise.

Claude Sonnet 5 performs the independent evaluation. DeepSeek generated the training targets, while Llama produced the predictions, so assigning the judge role to Claude keeps the three roles separate. Using the teacher as judge would be circular because it would grade students trained to imitate it, and evaluators have been shown to score their own generations more highly than human annotators do [48]. The same judge and prompt are used for all four models.

In the MT-Bench experiments of Zheng et al., GPT-4's judgments showed agreement with human experts comparable to the agreement measured between human evaluators [39]. This result motivates the additional measurement, but it remains an approximation of human assessment rather than a substitute for it. The same work documents verbosity and self-enhancement biases that may also affect the judge used here.

6.3 Comparison Protocol and Failure Handling

The evaluation compares Llama 3.1 8B and Llama 3.2 1B in base and fine-tuned form. LM Studio uses the settings described in Section 5.7 for each model, and every model processes the same 200 records. The base models receive the full schema in their system prompt; the fine-tuned models do not.

Excluding unparseable outputs would measure content accuracy only on successful generations and would give each model a different denominator. To evaluate end-to-end performance across the complete test set, the content metrics assign a score of zero to every unparseable output. Writing $\mathcal{P}$ for the set of records a model parsed, the content score becomes

$$S_f \;=\; \frac{1}{N} \sum_{i \in \mathcal{P}} s_f(i). \tag{6.11}$$

An unparseable output scores zero on every content field. Structural rates also use all 200 records by construction.

The judge calculation follows the same policy for unparseable outputs, assigning them the floor score of 1. Judge API failures are different: because no judgment was returned, those records are omitted from the judge average. If $\mathcal{J} \subseteq \mathcal{P}$ contains the parsed records with a returned judge score, then

$$\bar{g} = \frac{\sum_{i \in \mathcal{J}} g_i + 1 \cdot n_{\text{fail}}}{|\mathcal{J}| + n_{\text{fail}}}. \tag{6.12}$$

6.4 Results

The main comparison asks whether fine-tuning improves the 8B checkpoint over supplying the complete schema in its prompt. Table 6.1 reports the results for all four model configurations, while Figure 6.2 summarizes their four main measures. The 8B results are discussed first; Section 6.5 then considers how reducing the model to 1B changes the outcome.

Table 6.1. Results for all four models. Content metrics use all 200 test records; judge averages use the available scores and penalize unparseable outputs.

	Llama 3.1 8B		Llama 3.2 1B	
Metric	base	fine-tuned	base	fine-tuned
JSON recovered	100%	99%	85.5%	90.5%
Schema conforming	79.5%	**98%**	3.5%	68%
Objective macro	0.828	**0.938**	0.489	0.759
Subjective macro	0.577	**0.750**	0.400	0.570
BERTScore	0.718	**0.823**	0.464	0.654
Judge (1–5)	3.37	**4.51**	1.96	3.40

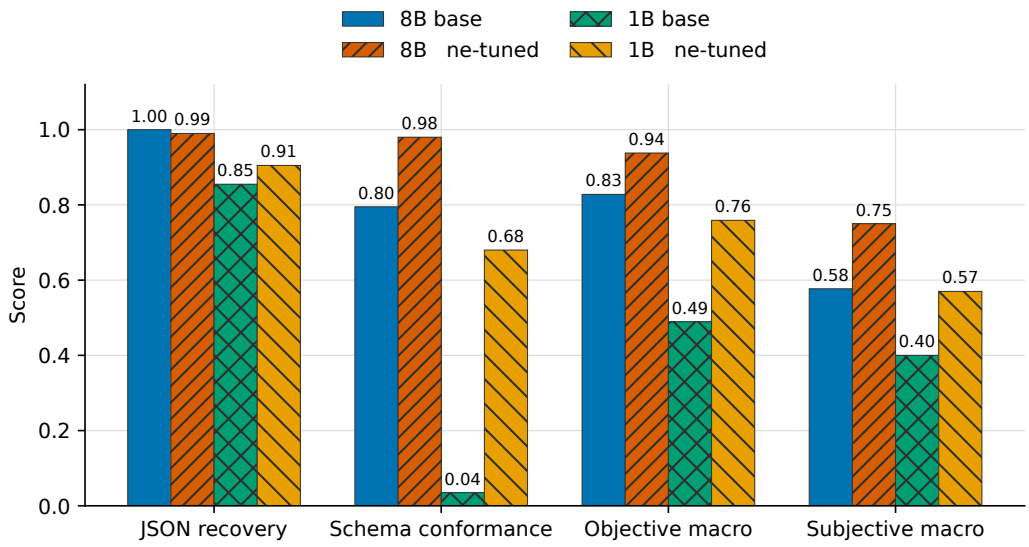


Figure 6.2. Headline metrics for the four models.

The fine-tuned 8B model leads on schema conformance, both content macros, BERTScore, and judged faithfulness. The base model retains a one-point advantage in JSON recovery. These measurements come from separate parts of the evaluation: the parser and schema validator assess structure, field-level metrics use the human-reviewed reference, and the judge sees the source record instead.

Structure. The largest single effect of fine-tuning is on schema conformance, which rises from 79.5% to 98%. JSON validity barely moves, and in fact the base model is marginally ahead at 100% against 99%, a difference of two records. This is the clearest illustration of why the two structural metrics are separate. The parser recovers an object from every base-model response, but those objects do not always form valid VIPER records. Most schema violations follow two recurring patterns: 22 predictions use a severity label outside the allowed set, and 16 give the product field a type the schema does not permit.

Output failures. The 8B base model produces no unparseable outputs, compared with 2 from the fine-tuned 8B model. Their judge averages use denominators of 200 and 199, respectively, because one API call for the fine-tuned model returned no score.

Content. The objective macro rises from 0.828 to 0.938 and the subjective macro from 0.577 to 0.750. The subjective figure is lower for both models, which is

expected: those fields ask for interpretation, and even a good answer may be worded differently from the reference.

Faithfulness. The judge average rises from 3.37 to 4.51. Figure 6.3 shows the distribution behind those averages, which is more revealing than the mean alone.

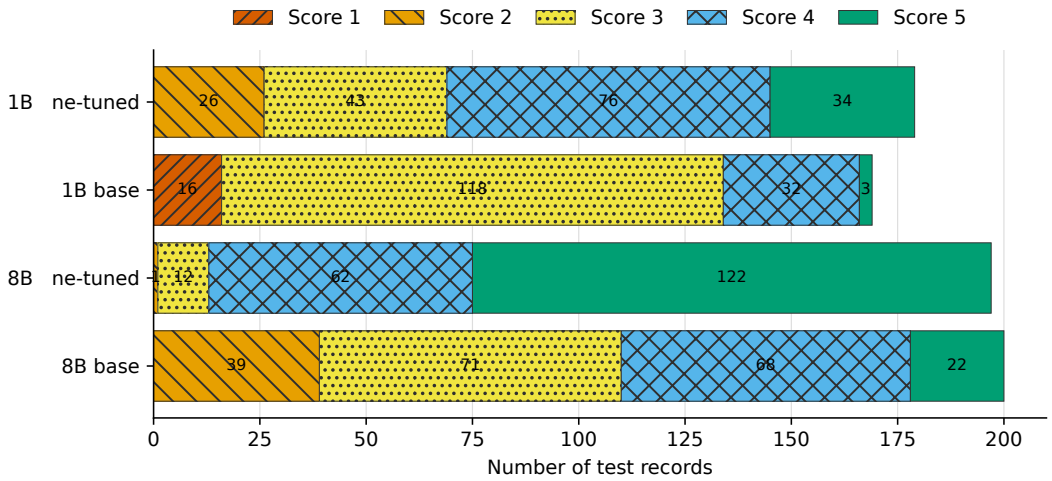


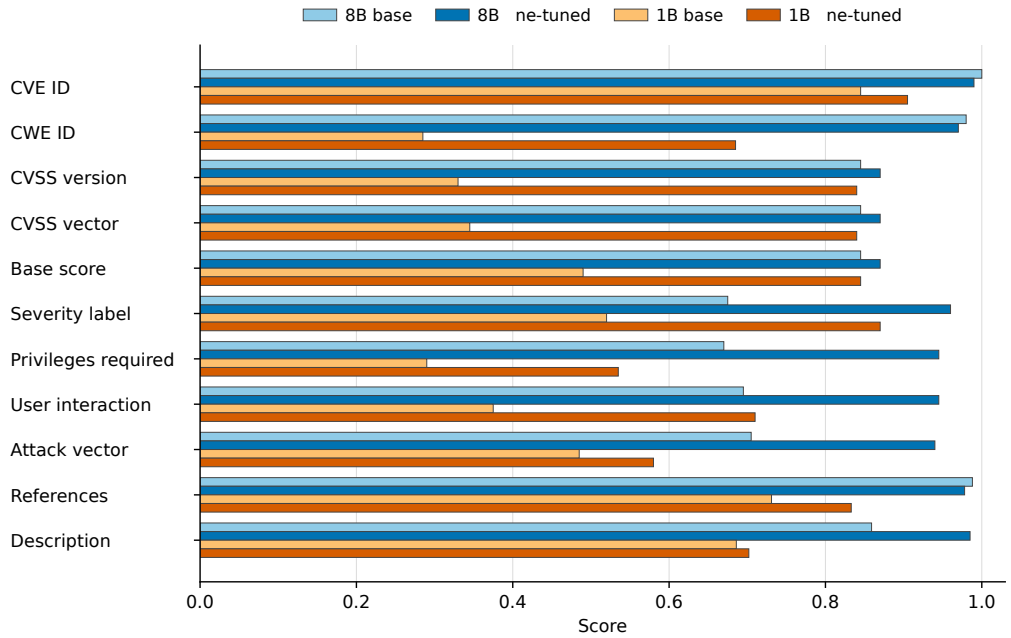
Figure 6.3. Distribution of judge scores. Counts cover the graded predictions; unparseable outputs are scored 1 in the reported average.

For the base 8B model the most common score is 3, with 39 records at 2 and only 22 at 5. For the fine-tuned model 122 of the graded records score 5 and only one scores below 3. Fine-tuning sharply reduces the lower end of the distribution.

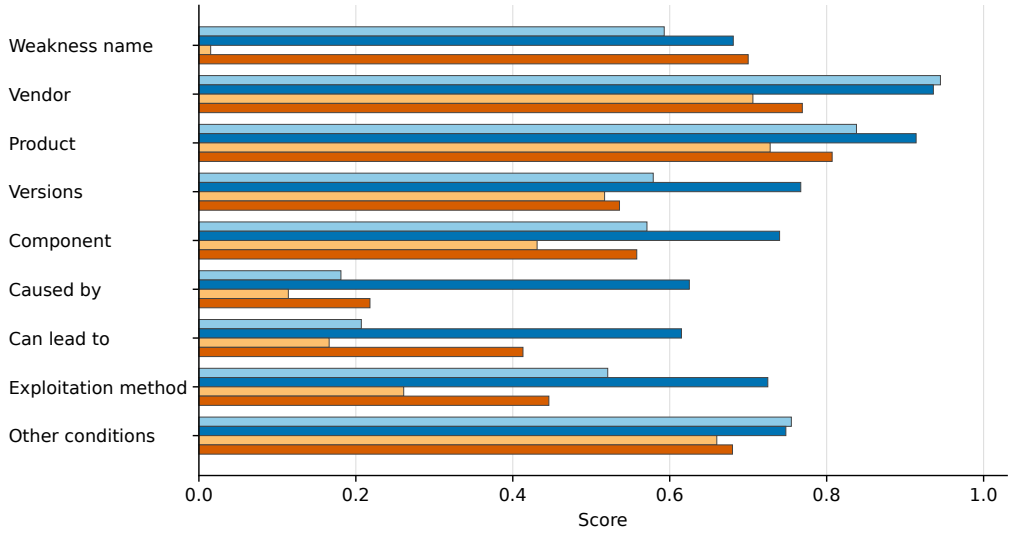
Per-field behavior. Fine-tuning does not improve every part of the record equally. Figure 6.4 presents the twenty individual scores.

The four CVSS-derived fields move furthest at 8B: privileges required from 0.670 to 0.945, user interaction from 0.695 to 0.945, attack vector from 0.705 to 0.940, and severity label from 0.675 to 0.960. These are precisely the fields whose correct value has to be derived by decoding a CVSS vector rather than copied from the text. The base model often assigns incorrect values to these fields, while the fine-tuned model applies the mapping more consistently. Among the subjective fields the causal ones gain most, with `caused_by` rising from 0.181 to 0.625 and `can_lead_to` from 0.207 to 0.615, though both remain among the hardest fields for every model.

Five fields change little: the identifier, the CWE, the reference list, the vendor, and the extra conditions. Their base-model scores are already at or above 0.75, leaving less room for improvement than in the weaker fields.



(a) Objective fields.



(b) Subjective fields.

Figure 6.4. Per-field scores for the objective and subjective field groups. Light and dark shades distinguish the base and fine-tuned checkpoints at each model size. Both panels use the same horizontal scale.

6.5 Comparison Across Model Sizes

The 1B results describe schema conformance and extraction quality at the smaller model size.

Structural validity at 1B. The parser recovers a JSON object from 85.5% of the base 1B responses, but only 3.5% form valid VIPER records. Among the 200 records, 29 produce no recoverable object, 161 parse but collapse a required object to a scalar by writing `"vulnerability_type": null`, 3 parse with some other violation, and 7 are fully valid.

This pattern accounts for the great majority of failures. The schema permits both members of the object to be null, so `{"name": null, "cwe_id": null}` would pass validation. The 1B model instead replaces the required object with a bare `null`. Repairing only that pattern would raise conformance from 3.5% to 33%, which is a diagnostic rather than a result. This estimate identifies the schema's nesting structure as the dominant source of failure, although it does not establish the quality of the extracted content. The fine-tuned 1B shows the same class of error in milder form, with 39 records promoting fields to the top level that belong inside `threat_information`.

The 1B base and fine-tuned models produce 29 and 19 unparseable outputs, respectively. Their judge averages both use 198 records because two API calls in each configuration returned no score. Across all four configurations, LM Studio recorded 23 responses that reached the 2048-token generation limit. The parser recovered 12 of them. Eleven unparseable outputs were consistent with truncation, as summarized in Table 6.2. This classification is based on the output content because the completion logs do not associate each token-limit event with a CVE identifier.

Table 6.2. Responses that reached the 2048-token limit and unparseable outputs consistent with truncation.

Model	Reached limit	Consistent with truncation
8B base	2	0
8B fine-tuned	3	2
1B base	4	0
1B fine-tuned	14	9

Fine-tuning gains at each model size. Table 6.3 compares the improvement at each size.

Table 6.3. Improvement from fine-tuning at each model size.

Gain from fine-tuning	8B	1B
Schema conforming	+18.5 pt	+**64.5** pt
Objective macro	+0.110	+**0.270**
Subjective macro	+0.173	+0.170
BERTScore	+0.105	+**0.190**
Judge (1–5)	+1.14	+**1.44**

The smaller model gains more on every measure except the subjective macro, which improves by almost the same amount at both sizes: +0.170 against +0.173. Fine-tuning produces a much larger structural gain at 1B, while the improvement on interpretive fields remains similar. The largest per-field change in the study is the weakness name at 1B, rising from 0.015 to 0.700. The very low base score reflects its frequent replacement of the containing object with `null`.

Comparison between the fine-tuned 1B and prompted 8B models. The fine-tuned 1B scores 0.759 against 0.828 on the objective macro and 0.654 against 0.718 on BERTScore. Their judged faithfulness is similar, at 3.40 for the fine-tuned 1B and 3.37 for the prompted 8B. Schema conformance remains lower for the fine-tuned 1B, at 68% against 79.5%, while the fine-tuned 8B reaches 98%.

6.6 Discussion

The results answer the three questions from Section 4.3.

RQ1: The schema can be learned implicitly. The fine-tuned 8B model conforms to the formal schema on 98% of records while never being shown the schema at inference time. Its system prompt names the format and asks for JSON, nothing more. The model learned enough of the schema from the training examples to reproduce its structure without receiving the formal definition at inference time.

RQ2: Fine-tuning improves on prompting. For the tested configuration, fine-tuning produces a substantial improvement over the full-schema baseline prompt. The base model conforms on 79.5% of records and is judged 3.37 out of 5. Reading a handful of its outputs can give the impression that it handles the task, but the complete test set reveals that roughly one record in five is not a valid VIPER record. A downstream system would have to handle those failures.

RQ3: Comparison across model sizes. At 1B the schema is followed on only 3.5% of records with the baseline prompt, while fine-tuning raises conformance to 68%. The gain from fine-tuning is larger for the smaller checkpoint on almost every measure. A fine-tuned 1B approaches a prompted 8B and matches it on judged faithfulness, but does not reach the fine-tuned 8B on any measure. For the two checkpoints tested, reducing model size is associated with lower schema conformance and content accuracy.

The field-level scores add two observations. Fine-tuning improved more than formatting alone: some of the largest 8B gains occur in fields derived from the CVSS vector, suggesting that the model learned mappings present in the teacher-generated examples. The hardest fields for every model are the causal ones, `caused_by` and `can_lead_to`, which are also the fields where the reference itself is least certain. The method is least reliable where the training targets require the most interpretation.

The interpretation of these results depends on several aspects of the experimental design. The subjective reference was reviewed by one person, the language-model judge was not validated against human ratings on this test set, and each checkpoint was fine-tuned once. The 8B and 1B models also belong to different Llama releases, while the smaller checkpoint reused the 8B training configuration. These conditions limit how broadly the observed differences can be attributed to fine-tuning or model size. Chapter 7 examines these limitations in full and outlines the experiments needed to address them.

One metric-specific qualification remains relevant here. The same token-overlap rule is used for every list field, including references. Two different URLs may still match when their token sets overlap by at least 0.6, making the reference-list score more permissive than exact URL comparison.

Taken together, the results support fine-tuning as an effective way to teach the VIPER format to the tested models, with the 8B checkpoint providing the strongest balance of structural reliability and content quality. Chapter 7 brings the findings together and considers the work needed before the extractor could be used in an operational setting.

Chapter 7

Conclusion and Future Work

Software vulnerability records combine standardized identifiers and fields with descriptions written in free-form prose. Those descriptions may contain the affected component, the cause of a vulnerability, the conditions required for exploitation, and its possible consequences. People can interpret this information, but automated tools need it in a consistent structure. The proposed approach addresses that gap by defining the VIPER format and training open-weight language models to produce it locally from raw CVE records.

The experiments compared prompted and fine-tuned versions of two model sizes on the same test set. They examined whether the schema could be learned from examples, whether fine-tuning improved the structure and content of the generated records, and how performance differed between the 8B and 1B checkpoints. The findings answer those questions while revealing limits in the training targets, evaluation reference, model comparison, and test corpus.

The contributions and research findings are summarized first. Their limitations define how far the conclusions can be extended and lead directly to the proposed future work. The chapter closes by returning to what the experiments establish.

7.1 Summary of Contributions

The work produced five connected contributions.

1. **The VIPER schema.** VIPER defines a consistent JSON representation for information extracted from CVE records. Its seven required top-level fields cover the vulnerability type, affected system, threat information, severity, references, and original description. The formal schema also defines the required nested structure, accepted data types, and restricted values used during validation.

2. **A distillation dataset.** The published corpus contains 989 raw CVE records paired with VIPER targets generated by a stronger teacher model. Fifteen conversations exceeded the sequence limit, leaving 974 examples for the training runs, as described in Section 5.6. These examples allowed the models to learn both the required structure and the interpretation needed for fields that cannot be copied directly from the source.

3. **Two locally deployable extractors.** Llama 3.1 8B and Llama 3.2 1B were fine-tuned with LoRA, merged with their adapters, quantized, and served locally. Neither fine-tuned model receives the complete schema during inference, and neither depends on the teacher or an external inference service after deployment.

4. **A controlled comparison of prompting and fine-tuning.** The evaluation compares each fine-tuned model with its corresponding prompted base model on the same 200 test records and through the same local runtime. The base models received the complete schema, while the fine-tuned models received the minimal prompt used in training. This design measures whether learning from examples offers an advantage over supplying the format through prompting.

5. **An evaluation framework for structure and content.** The evaluation in Chapter 6 separates JSON recovery from formal schema conformance and measures individual fields according to the type of comparison they require. It also uses an independent language-model judge to assess faithfulness to the source. The 200 draft references were checked against their raw CVE records, and 124 were corrected before the final evaluation.

Together, these contributions satisfy the three constraints defined in Chapter 1. VIPER provides the fixed format, the merged and quantized models support local inference, and the minimal inference prompt tests whether the fine-tuned models learned the schema from examples.

The three research questions defined in Chapter 4 can now be answered directly. Chapter 6 provides the detailed interpretation of the results, while Table 6.1 brings the headline measurements together.

RQ1: Schema learning. Both fine-tuned models learned enough of the VIPER schema from their training examples to reproduce it without receiving the complete definition at inference time. The 8B model produced schema-conforming records for 98% of the test set, while the 1B model reached 68%. The difference between them is large, but both results indicate that the required structure can be learned from examples. The 8B checkpoint reproduced it more reliably, indicating that performance is sensitive to the checkpoint used.

RQ2: Effect of fine-tuning. Fine-tuning improved structural conformance and content quality at both model sizes. For the 8B model, schema conformance rose from 79.5% to 98%, the objective macro from 0.828 to 0.938, and the subjective macro from 0.577 to 0.750. Its average judge score also increased from 3.37 to 4.51. The 1B model showed the same overall direction, with schema conformance rising from 3.5% to 68% and its judge score from 1.96 to 3.40. JSON recovery was the only headline measure that did not improve in both comparisons: the 8B result decreased slightly from 100% to 99%. Overall, the measurements indicate that fine-tuning offered a clear advantage over the full-schema prompts used for the corresponding base models.

RQ3: Comparison across model sizes. The 1B checkpoint benefited considerably from fine-tuning but remained less reliable than the fine-tuned 8B model. Its 68% schema conformance was below the 8B model's 98%, while its objective macro reached 0.759 against 0.938 and its judge score reached 3.40 against 4.51. The fine-tuned 1B model obtained a judge score similar to the prompted 8B model, but it remained lower in schema conformance and field-level accuracy. For the two checkpoints examined here, the smaller model was markedly less reliable despite its reduced size. Because the checkpoints belong to different Llama releases, the observed difference cannot be attributed to parameter count alone.

The field-level results in Figure 6.4 indicate that fine-tuning affected more than the output structure. Some of the largest improvements at 8B occurred in privileges required, user interaction, attack vector, and severity label. These values are derived from the CVSS vector instead of being copied directly from the record. Their improvement suggests that fine-tuning taught the model recurring mappings present in the training examples, in addition to the structure of the output. This interpretation remains limited to the tested examples and does not establish that the model learned a general CVSS scoring procedure.

7.2 Limitations

The findings apply to the models, records, prompts, and evaluation procedure used in these experiments. The reported improvements should be interpreted with the following limitations in mind.

Reference quality. The subjective fields in the test reference began as outputs from one teacher model. All 200 records were subsequently compared with their raw CVE records, and 124 references were corrected. This review removed many clear errors, but it was performed by one reviewer. A second independent annotation is unavailable, so inter-annotator agreement cannot be reported. The subjective

metrics consequently measure agreement with a carefully checked reference, not with an established ground truth shared by multiple experts.

Judge validity. The language-model judge was kept separate from the teacher and the evaluated models, but it was not validated against human ratings on this test set. Its scores provide an additional view of faithfulness, particularly when an answer differs in wording from the reference, but they should not be treated as a replacement for expert assessment. Using one judge also leaves open the possibility that another model would assign different scores or rank the systems differently.

Synthetic training targets. The student models learned from targets generated by a single teacher. Errors and preferences in those targets may carry over to the fine-tuned models. Two of the 989 targets were later found to violate the formal schema; Section 5.4 describes these cases. The causal fields also remained among the weakest fields in the evaluation. These observations justify combining automatic validation with human inspection when synthetic supervision is used for interpretive security data.

Experimental comparison. The experiments include one fine-tuning run for each checkpoint, so they do not measure variation across random seeds or repeated runs. The 1B run reused the 8B training configuration without a separate hyperparameter search. Its reported performance describes that configuration and should not be interpreted as the best result achievable at 1B. In addition, Llama 3.1 8B and Llama 3.2 1B belong to different model releases. Their comparison reflects the complete checkpoints used in the experiment and cannot isolate parameter count as the only cause of the difference. The prompted baselines provide a consistent full-schema comparison, but no systematic prompt search was performed.

Corpus coverage. The training and test records contain English descriptions published from 2021 to 2026. Older CVE formats, other languages, and reporting conventions outside this period were not tested. The 200-record test set supports a consistent comparison among the four model configurations, but it does not represent every product category, assigning authority, or style found in the complete CVE List.

Reproducibility. The dataset snapshot, prompts, configurations, and generated artifacts preserve much of the experimental procedure. Exact repetition is nevertheless not possible from the surviving material. The teacher targets were sampled at a non-zero temperature through a changing API model name, and not every software version or random choice used during training was recorded. Repeating

the pipeline should produce a comparable experiment, but not necessarily identical targets, checkpoints, or predictions.

Operational validation. Local inference was demonstrated, but the thesis did not benchmark latency, throughput, energy consumption, or memory use across different user machines. It also evaluated VIPER records as extraction outputs rather than inputs to an operational security system. The thesis thus establishes extraction quality only under the tested conditions; it does not establish VIPER's effectiveness in vulnerability prioritization, incident response, or automated mitigation.

7.3 Future Work

Output constraints and training-target quality. Many of the observed failures involve invalid nesting, missing required objects, or values outside an allowed set. Grammar-constrained generation could prevent many of these structural errors by limiting each generated token to choices permitted by the schema. For outputs that complete successfully, a grammar derived from the formal schema could enforce the required types, nesting, and allowed values. The `llama.cpp` runtime supports grammar-based constraints [49], so this extension would fit the existing local deployment pipeline. It would not prevent truncation or ensure that the extracted content is faithful to the source. Content evaluation would still be necessary. A future training pipeline should also validate every synthetic target against the formal schema and reject or correct invalid examples before they reach the student model.

Reference and judge validation. A stronger evaluation would begin with independent annotation of at least part of the test set. Comparing two reviewers would provide an agreement measure for the interpretive fields and reveal which VIPER definitions require further clarification. Human ratings of a representative sample of predictions could then be compared with the language-model judge, showing whether its scores and model ranking agree with expert assessment. Repeating the evaluation with additional judge models would indicate how much the result depends on the particular judge selected for this thesis.

Model and training experiments. Future experiments should select a configuration for the smaller checkpoint and vary the learning rate, adapter rank, number of epochs, and sequence length. Repeating each configuration with multiple random seeds would show whether the reported improvements are stable. A cleaner study of model size would compare checkpoints from the same model family and release, reducing the architectural differences present in the current 8B–1B comparison.

Several teachers could also generate candidate targets for reconciliation before fine-tuning, reducing dependence on the interpretations and recurring errors of one model.

Broader coverage and operational evaluation. Extending the corpus to older CVE records, additional languages, and a wider range of assigning authorities would test whether the extractor generalizes beyond recent English descriptions. Runtime tests should measure latency, throughput, memory use, and energy consumption across several hardware configurations. A downstream security process should then consume the resulting VIPER records. One candidate is the automated mitigation loop proposed by Bringhenti et al. [31]. An experiment of this kind would reveal whether the current fields support operational decisions or whether the schema needs information that the extraction-focused evaluation did not consider.

7.4 Closing Remarks

Under the tested conditions, the experiments indicate that an open-weight language model can learn a nested extraction format from examples and use it during local inference. The fine-tuned 8B model provided the strongest result, reaching 98% schema conformance while also improving the objective, subjective, and faithfulness measures over its prompted base version. Fine-tuning contributed more than valid formatting: it improved the model's ability to recover and interpret information from the source records.

Fine-tuning made the 1B checkpoint considerably more reliable, but it did not close the gap with the fine-tuned 8B model. The smaller model moved from rarely following the complete schema to producing conforming records for most of the test set. Its remaining gap suggests a trade-off between a smaller deployment model and extraction reliability. Confirming that trade-off would require the runtime measurements proposed in the Future Work section. The appropriate checkpoint will depend on how much structural failure and content error a downstream application can tolerate.

VIPER, the distillation dataset, the locally deployed models, and the evaluation framework provide concrete artifacts for the next stage of the work. They do not yet constitute a validated operational vulnerability-management system, but they make the extraction problem measurable and expose the points that need further improvement. Stronger target validation, repeated model experiments, broader data, human evaluation, and downstream testing would determine whether the approach can move from a controlled experiment to practical security use.

Appendix A

Reproducibility Materials

This appendix contains the machine-readable material needed to reproduce the experiments without interrupting the methodology with long schemas, prompts, and configuration files.

A.1 Complete VIPER JSON Schema

The validator uses the following JSON Schema Draft-07 document. It is divided into four consecutive fragments so that page breaks fall between complete groups of fields. Reading the fragments in order gives the original document without omissions.

Part 1: Document header and top-level requirements.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "VIPER",
  "description": "Vulnerability Intelligence Parsed Extraction Record",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "cve_id",
    "vulnerability_type",
    "affected_system",
    "threat_information",
    "severity",
    "references",
    "description"
  ],
  "properties": {
    "cve_id": { "type": "string" },
```

Part 2: Vulnerability and affected-system fields.

```
    "vulnerability_type": {
      "type": "object",
      "additionalProperties": false,
      "required": ["name", "cwe_id"],
      "properties": {
        "name": { "type": ["string", "null"] },
        "cwe_id": { "type": ["string", "null"] }
      }
    },
    "affected_system": {
      "type": "object",
      "additionalProperties": false,
      "required": ["vendor", "product", "versions", "component"],
      "properties": {
        "vendor": { "type": ["string", "null"] },
        "product": { "type": ["string", "null"] },
        "versions": { "type": "array", "items": { "type": "string" } },
        "component": { "type": ["string", "null"] }
      }
    },
```

Part 3: Threat-information fields.

```
    "threat_information": {
      "type": "object",
      "additionalProperties": false,
      "required": ["caused_by", "can_lead_to", "exploited_by",
      ↪  "preconditions"],
      "properties": {
        "caused_by": { "type": "array", "items": { "type": "string" } },
        "can_lead_to": { "type": "array", "items": { "type": "string" } },
        "exploited_by": { "type": ["string", "null"] },
        "preconditions": {
          "type": "object",
          "additionalProperties": false,
          "required": [
            "privileges_required",
            "user_interaction",
            "attack_vector",
            "other_conditions"
          ],
          "properties": {
            "privileges_required": {
              "type": ["string", "null"],
              "enum": ["none", "low", "high", null]
            },
            "user_interaction": {
              "type": ["string", "null"],
              "enum": ["none", "required", null]
            },
            "attack_vector": {
              "type": ["string", "null"],
              "enum": ["network", "adjacent", "local", "physical", null]
            },
            "other_conditions": {
              "type": "array",
              "items": { "type": "string" }
            }
          }
        }
      }
    },
```

Part 4: Severity and remaining top-level fields.

```
    "severity": {
      "type": "object",
      "additionalProperties": false,
      "required": ["cvss_version", "cvss_vector", "base_score",
      ↪  "severity_label"],
      "properties": {
        "cvss_version": { "type": ["string", "null"] },
        "cvss_vector": { "type": ["string", "null"] },
        "base_score": { "type": ["number", "null"] },
        "severity_label": {
          "type": ["string", "null"],
          "enum": ["LOW", "MEDIUM", "HIGH", "CRITICAL", null]
        }
      }
    },
    "references": { "type": "array", "items": { "type": "string" } },
    "description": { "type": "string" }
  }
}
```

Listing A.1. Complete VIPER JSON Schema, divided into four consecutive fragments at object boundaries.

A.2 Generation and Evaluation Prompts

A.2.1 Serialized Llama Conversation

The JSONL dataset stores roles and content separately. Axolotl applies the Llama 3 chat template during preprocessing, producing the following sequence. The CVE record and target are abbreviated here, but the control tokens and text surrounding them are reproduced exactly.

```
1  <|begin_of_text|><|start_header_id|>system<|end_header_id|>
2
3  You are a cybersecurity analyst who converts raw CVE records into the VIPER (Vulnerability
   ↪  Intelligence Parsed Extraction Record) JSON format. Respond with a single valid JSON
   ↪  object and nothing else.<|eot_id|><|start_header_id|>user<|end_header_id|>
4
5  Convert this raw CVE record into the VIPER schema:
6  {<compacted CVE record>}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
7
8  {<VIPER target>}<|eot_id|>
```

Listing A.2. Serialized Llama 3 training conversation. The CVE record and VIPER target are abbreviated; the surrounding instructions and control tokens are reproduced exactly.

A.2.2 DeepSeek Teacher Prompt

The target-generation script supplied the following system prompt. Its `VIPER_SCHEMA` constant is a compact description intended for prompting; it is not the formal validator in Section A.1.

```
You are an expert vulnerability intelligence analyst.
Your job is to read a raw CVE record (MITRE cvelistV5 JSON) and extract the
information into the VIPER (Vulnerability Intelligence Parsed Extraction Record)
schema below.
Output MUST be a single JSON object that exactly matches this schema:

{
  "cve_id": "string",
  "vulnerability_type": { "name": "string|null", "cwe_id": "string|null" },
  "affected_system": {
    "vendor": "string|null", "product": "string|null",
    "versions": ["..."], "component": "string|null"
  },
  "threat_information": {
    "caused_by": ["..."], "can_lead_to": ["..."],
    "exploited_by": "string|null",
    "preconditions": {
      "privileges_required": "none|low|high|null",
      "user_interaction": "none|required|null",
      "attack_vector": "network|adjacent|local|physical|null",
      "other_conditions": []
    }
  },
  "severity": {
    "cvss_version": "string|null", "cvss_vector": "string|null",
    "base_score": 0.0,
    "severity_label": "LOW|MEDIUM|HIGH|CRITICAL|null"
  },
  "references": ["..."], "description": "string"
}
```

Prompt continued: extraction rules.

```
Extraction rules:
- "cve_id": copy from the record metadata (e.g. "CVE-2024-1234").
- "vulnerability_type.name": short human label of the weakness (e.g.
  "SQL Injection", "Out-of-bounds Write"). Use null if it cannot be determined.
- "vulnerability_type.cwe_id": the CWE identifier such as "CWE-79" if present in
  problemTypes; otherwise null.
- "affected_system": vendor, product and the affected versions. "versions" is a
  list of version strings (may be empty). "component" is the specific module /
  subsystem if mentioned, else null. Map the literal string "n/a" to null.
- "threat_information.caused_by": list of root causes (e.g. "improper input
  validation", "missing authentication"). Empty list if unknown.
- "threat_information.can_lead_to": list of impacts / consequences (e.g.
  "remote code execution", "information disclosure"). Empty list if unknown.
- "threat_information.exploited_by": how it is exploited (e.g. "crafted HTTP
  request", "malicious file"), or null.
- "threat_information.preconditions": derive from the CVSS vector string when one
  is present, otherwise infer from the description:
```

```
    * privileges_required  <- PR:N=none, PR:L=low, PR:H=high
    * user_interaction     <- UI:N=none, UI:R=required
    * attack_vector        <- AV:N=network, AV:A=adjacent, AV:L=local, AV:P=physical
  "other_conditions" is a list of any extra requirements (e.g. "victim must open
  a malicious page"); empty list if none.
- "severity": take the best available CVSS metric in the record. Fill
  cvss_version (e.g. "3.1"), cvss_vector (the vector string), base_score (number)
  and severity_label (LOW/MEDIUM/HIGH/CRITICAL). Use null / 0.0 when no CVSS data
  exists.
- "references": list of the reference URLs in the record (deduplicated).
- "description": the English description text from the record.

Only use information that is actually present in or directly implied by the
record. Never invent CVSS scores, vendors or CWEs. Use null (or empty lists)
whenever a value is genuinely unavailable. Respond with JSON only.
```

Listing A.3. System prompt used to generate the VIPER training targets.

Every teacher user turn then contains the fixed instruction "Convert this raw CVE record into the VIPER schema:", a newline, and the compacted CVE record.

A.2.3 Prompted Baseline and Evaluation Judge

The baseline uses the teacher prompt above with this exact suffix:

```
Do NOT call any tool or function and do NOT wrap the answer in a function call. Output ONLY
↪  the raw VIPER JSON object and nothing else.
```

The independent judge receives this system prompt:

```
You are a strict evaluator. Given a raw CVE record and a VIPER extraction, rate how faithfully
↪  and completely the extraction captures the record on a 1-5 integer scale (5 = perfect,
↪  fully grounded; 1 = wrong/hallucinated). Respond with ONLY a JSON object: {"score": <int>,
↪  "reason": "<short>"}.
```

Its user message consists of `RAW CVE:`, the compacted record, `VIPER EXTRACTION:`, and the prediction being graded.

A.3 Training and Inference Configuration

The verified 8B Axolotl configuration is reproduced below. The archive does not record the Axolotl version or an immutable model revision, and the configuration does not declare a random seed. Parameters absent from the file cannot be linked to a documented set of version-specific defaults.

```yaml
adapter: lora
base_model: unsloth/Meta-Llama-3.1-8B-Instruct
bf16: true
fp16: false
load_in_4bit: false
load_in_8bit: false
datasets:
  - path: mx003/viper
    type: chat_template
    field_messages: messages
    chat_template: llama3
val_set_size: 0.05
lora_r: 32
lora_alpha: 64
lora_dropout: 0.05
lora_target_modules: [q_proj, v_proj, k_proj, o_proj,
                      gate_proj, down_proj, up_proj]
gradient_accumulation_steps: 8
gradient_checkpointing: true
micro_batch_size: 2
num_epochs: 2
learning_rate: 0.0002
optimizer: adamw_torch_fused
lr_scheduler: cosine
warmup_ratio: 0.03
output_dir: ./outputs/mymodel
sequence_len: 4096
save_steps: 10
logging_steps: 5
flash_attention: false
sample_packing: true
group_by_length: false
train_on_inputs: false
special_tokens:
  pad_token: <|end_of_text|>
```

Listing A.4. Verified Axolotl configuration used for the 8B fine-tuning run.

The 1B file is identical except for these two entries:

```
base_model: unsloth/Llama-3.2-1B-Instruct
output_dir: ./outputs/mymodel-1b
```

The project runbook records the following 8B merge and upload commands. The 1B run uses its corresponding YAML file and repository name.

```
axolotl merge-lora config.yaml
grep -o '"chat_template"' outputs/mymodel/merged/tokenizer_config.json
huggingface-cli upload mx003/viper-llama31-8b-merged ./outputs/mymodel/merged
```

The merged repository was converted through the Hugging Face Space `ggml-org/gguf-my-repo`, selecting `Q4_K_M` for inference and `Q8_0` as a higher-precision reference. The archive contains no local conversion command or immutable Space revision. LM Studio served one GGUF at a time. Inference was invoked with `run_inference.py`; the script defines temperature 0, a 2048-token output limit, four stop strings, and one worker as its defaults.

A.3.1 Local Inference Environment

Table A.1. Hardware and runtime details recovered from the LM Studio configuration.

Item	Recorded value
Operating system	Windows, x86-64
Processor	Intel Core Ultra 7 256V
Graphics	Intel Arc 140V, integrated, 8 GB
Acceleration	Vulkan; full GPU offload requested
Backend	`llama.cpp-win-x86_64-vulkan-avx2` 2.27.1
Runtime slots	4
8B context length	22,784 tokens
1B context length	131,072 tokens

LM Studio 0.4.19 is the last application version recorded on the machine. The available metadata does not establish whether that version was already installed during the inference runs on 15–16 July 2026.

A.4 Dataset and Prediction Artifacts

Table A.2 lists the archived artifacts. When a response cannot be parsed into a JSON prediction, its raw text is retained. Each prediction directory consequently accounts for all 200 test records.

The source corpus was sampled from the official `cvelistV5` repository as it appeared at Git commit `24a246a8361d7ad4145bd38bf059d525aca3d538`, cloned on 4 April 2026. The commit uniquely identifies the repository snapshot used to construct the training and test sets.

Table A.2. Dataset and prediction artifacts found in the project archive.

Artifact	Parsed/lines	Raw failures
`dataset.jsonl`	989	–
`predictions/base`	200	0
`predictions/finetuned`	198	2
`predictions_llama3-2/base-1b`	171	29
`predictions_llama3-2/finetuned-1b`	181	19

The following public Hugging Face repositories were verified at the time of writing:

- training dataset: `https://huggingface.co/datasets/mx003/viper`;
- fine-tuned 8B model: `https://huggingface.co/mx003/viper-llama31-8b`;
- quantized 8B model: `https://huggingface.co/mx003/viper-llama31-8b-Q4_K_M-GGUF`;
- merged 1B model: `https://huggingface.co/mx003/viper-llama32-1b-merged`.

No immutable Hugging Face revisions were recorded during the experiment. A public repository for the final 1B GGUF file was not identified, and the four prediction sets are retained in the supplied project archive rather than in a public artifact repository.

Bibliography

[1] National Institute of Standards and Technology. *NVD Dashboard: Vulnerability Statistics*. 2024. URL: https://nvd.nist.gov/general/nvd-dashboard.

[2] CVE Program. *CVE List V5: Catalog of CVE Records in the CVE Record Format*. 2026. URL: https://github.com/CVEProject/cvelistV5.

[3] The MITRE Corporation. *CVE Program*. 2024. URL: https://www.cve.org/.

[4] CVE Program. *CVE Record Format (JSON Schema)*. 2024. URL: https://github.com/CVEProject/cve-schema.

[5] Forum of Incident Response and Security Teams. *Common Vulnerability Scoring System v3.1: Specification Document*. 2019. URL: https://www.first.org/cvss/v3.1/specification-document.

[6] Forum of Incident Response and Security Teams. *Common Vulnerability Scoring System version 4.0: Specification Document*. 2023. URL: https://www.first.org/cvss/v4.0/specification-document.

[7] Harold Booth, Doug Rike, and Gregory A. Witte. *The National Vulnerability Database (NVD): Overview*. ITL Bulletin. National Institute of Standards and Technology (NIST), Dec. 2013. URL: https://csrc.nist.gov/publications/detail/itl-bulletin/2013/12/the-national-vulnerability-database-nvd-overview/final.

[8] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. «Detecting and Augmenting Missing Key Aspects in Vulnerability Descriptions». In: *ACM Transactions on Software Engineering and Methodology* 31.3 (2022), pp. 1–27. DOI: 10.1145/3498537.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, et al. «Language Models are Few-Shot Learners». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[11] DeepSeek-AI. «DeepSeek-V3 Technical Report». In: *arXiv preprint arXiv:2412.19437* (2024).

[12] Aaron Grattafiori et al. «The Llama 3 Herd of Models». In: *arXiv preprint arXiv:2407.21783* (2024).

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. «LoRA: Low-Rank Adaptation of Large Language Models». In: *International Conference on Learning Representations (ICLR)*. 2022.

[14] Georgi Gerganov and llama.cpp contributors. *llama.cpp: LLM Inference in C/C++*. 2026. URL: `https://github.com/ggml-org/llama.cpp`.

[15] Meta AI. *Llama 3.2 Model Card*. 2024. URL: `https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md`.

[16] CVE Program. *Frequently Asked Questions*. 2026. URL: `https://www.cve.org/ResourcesSupport/FAQs`.

[17] CVE Program. *Process: Reserve a CVE ID and Publish a CVE Record*. 2026. URL: `https://www.cve.org/about/Process`.

[18] National Institute of Standards and Technology. *CVEs and the NVD Process*. 2026. URL: `https://nvd.nist.gov/general/cve-process`.

[19] CVE Program. *CVE Numbering Authority (CNA) Rules*. 2024. URL: `https://www.cve.org/ResourcesSupport/AllResources/CNARules`.

[20] The MITRE Corporation. *CWE: Common Weakness Enumeration*. 2024. URL: `https://cwe.mitre.org/`.

[21] The MITRE Corporation. *CWE Mapping and Navigation Guidance*. 2026. URL: `https://cwe.mitre.org/documents/cwe_usage/mapping_navigation.html`.

[22] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «A Novel Approach for Security Function Graph Configuration and Deployment». In: *IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021, pp. 457–463.

[23] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «A Novel Abstraction for Security Configuration in Virtual Networks». In: *Computer Networks* 228 (2023), p. 109745.

[24] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Serena Spinoso, Fulvio Valenza, and Jalolliddin Yusupov. «Improving the Formal Verification of Reachability Policies in Virtualized Networks». In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 713–728.

[25] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «Atomizing Firewall Policies for Anomaly Analysis and Resolution». In: *IEEE Transactions on Dependable and Secure Computing* 22.3 (2025), pp. 2308–2325.

[26] Daniele Bringhenti, Lucia Seno, and Fulvio Valenza. «An Optimized Approach for Assisted Firewall Anomaly Resolution». In: *IEEE Access* 11 (2023), pp. 119693–119710.

[27] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «A Two-Fold Traffic Flow Model for Network Security Management». In: *IEEE Transactions on Network and Service Management* 21 (2024), pp. 3740–3758.

[28] Daniele Bringhenti and Fulvio Valenza. «Optimizing Distributed Firewall Reconfiguration Transients». In: *Computer Networks* 215 (2022), p. 109183.

[29] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «Automating VPN Configuration in Computer Networks». In: *IEEE Transactions on Dependable and Secure Computing* 22.1 (2025), pp. 561–578.

[30] Daniele Bringhenti and Fulvio Valenza. «GreenShield: Optimizing Firewall Configuration for Sustainable Networks». In: *IEEE Transactions on Network and Service Management* 21.6 (2024), pp. 6909–6923.

[31] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. «A Looping Process for Cyberattack Mitigation». In: *IEEE International Conference on Cyber Security and Resilience (CSR)*. 2024, pp. 276–281.

[32] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. «Finetuned Language Models Are Zero-Shot Learners». In: *International Conference on Learning Representations (ICLR)*. 2022.

[33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, et al. «Training Language Models to Follow Instructions with Human Feedback». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2022.

[34] Meta AI. *Llama 3: Official Repository and Chat Format*. 2024. URL: `https://github.com/meta-llama/llama3`.

[35] Hugging Face. *Writing a Chat Template*. 2026. URL: `https://huggingface.co/docs/transformers/en/chat_templating_writing`.

[36] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. «QLoRA: Efficient Finetuning of Quantized LLMs». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[37] Georgi Gerganov and llama.cpp contributors. *llama.cpp Quantization Documentation*. 2026. URL: `https://github.com/ggml-org/llama.cpp/blob/master/tools/quantize/README.md`.

[38] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. arXiv preprint arXiv:1503.02531. 2015.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, et al. «Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena». In: *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. 2023. URL: `https://arxiv.org/abs/2306.05685`.

[40] JSON Schema Organization. *JSON Schema: A Media Type for Describing JSON Documents*. 2020. URL: `https://json-schema.org/specification`.

[41] DeepSeek. *JSON Output*. 2026. URL: https://api-docs.deepseek.com/guides/json_mode/.

[42] Axolotl AI. *Conversation Dataset Format and Chat Templates*. 2026. URL: https://docs.axolotl.ai/docs/dataset-formats/conversation.html.

[43] Axolotl AI. *Axolotl: Post-Training and Fine-Tuning for Large Language Models*. 2026. URL: https://github.com/axolotl-ai-cloud/axolotl.

[44] PyTorch Contributors. *Scaled Dot Product Attention*. 2026. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.

[45] LM Studio. *LM Studio REST API*. 2026. URL: https://lmstudio.ai/docs/developer/rest.

[46] Python Software Foundation. *difflib — Helpers for Computing Deltas*. 2026. URL: https://docs.python.org/3/library/difflib.html.

[47] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. «BERTScore: Evaluating Text Generation with BERT». In: *International Conference on Learning Representations (ICLR)*. 2020. URL: https://arxiv.org/abs/1904.09675.

[48] Arjun Panickssery, Samuel R. Bowman, and Shi Feng. «LLM Evaluators Recognize and Favor Their Own Generations». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024. URL: https://arxiv.org/abs/2404.13076.

[49] Georgi Gerganov and llama.cpp contributors. *GBNF Guide: Grammar-Constrained Generation in llama.cpp*. 2026. URL: https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md.