BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DEVELOPMENT OF AN LLM RED-TEAMING TOOLKIT

VÝVOJ TOOLKITU PRO RED-TEAMING VELKÝCH JAZYKOVÝCH MODELŮ (LLM)

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ADAM VESELÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. JAKUB REŠ**
VEDOUCÍ PRÁCE

BRNO 2026

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

VESELÝ, Adam. *Development of an LLM Red-Teaming Toolkit.* Brno, 2026. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jakub Reš,

Development of an LLM Red-Teaming Toolkit

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jakub Reš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. I have used ChatGPT to correct spelling and other language mistakes.

....
Adam Veselý
January 10, 2026

Acknowledgements

I would like to thank my supervisor Ing. Jakub Reš for his guidance and support throughout the development of this thesis. I also appreciate the assistance provided by my colleagues and friends who contributed their insights and expertise.

Contents

List of Figures

Chapter 1

Introduction

The rapid advancement and widespread deployment of large language models (LLMs) have transformed natural-language interaction with computers. These models now power chatbots, code assistants, translation systems, and creative tools used daily by millions of users.

However, their remarkable capabilities come with significant safety and ethical risks. LLMs can generate harmful, biased, misleading, or illegal content when subjected to carefully crafted adversarial inputs, a practice commonly known as *jailbreaking* [30, 34].

Real-world incidents such as ChatGPT being tricked into providing bomb-making instructions [5], or Gemini's image-generation controversy [27], have demonstrated that even flagship commercial models remain vulnerable to adversarial prompting. In response, red-teaming, a cybersecurity technique involving simulated attacks to expose vulnerabilities, has been adopted by leading AI organisations (OpenAI, Anthropic, Google DeepMind) as a core component of LLM safety evaluation [17].

Beyond model-level behaviour, LLMs are embedded in applications and services where integration and deployment issues create additional, system-level risks. Industry guidance such as the OWASP Top 10 for Large Language Model Applications highlights vulnerabilities including prompt injection, insecure output handling, insecure plugin design, and excessive agency [19, 20].

With the adoption of the EU AI Act in 2024, systematic risk assessment including red-teaming will become a legal requirement for high-risk AI systems deployed in the European Union from 2026 onward [21]. Consequently, efficient, reproducible, and extensible red-teaming tools are no longer a luxury but an essential part of responsible AI development.

Despite significant progress, most existing open-source red-teaming frameworks suffer from limited modularity, poor support for modern systems and computational requirements that hinder adoption by smaller research teams and individual developers [23, 25, 1, 16]. This creates a clear need for a new, lightweight, developer-friendly red-teaming toolkit that lowers the barrier to LLM systematic safety testing.

The main goal of this bachelor's thesis is therefore the design, implementation, and evaluation of a modular open-source red-teaming toolkit for large language models that addresses the identified shortcomings of current solutions.

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary background on large language models. Chapter 3 surveys existing red-teaming methodologies, attack taxonomies, evaluation metrics, and open-source tools. Chapter 4 presents the proposed system architecture. Chapter 5 details the implementation of the red-teaming toolkit. Chapter 6 evaluates the toolkit on selected LLMs. Finally, Chapter 7 summarises the results and outlines directions for future work.

Chapter 2

Background on Large Language Models and Safety Risks

This chapter provides the technical and empirical background necessary to understand safety risks and adversarial behaviour in large language models (LLMs).

We first cover the architecture and training paradigm of modern LLMs and the alignment methods applied to improve their safety.

Subsequently, we survey the primary categories of safety risks and known vulnerability classes arising from LLM deployment.

2.1 Large Language Models

Large language models (LLMs) are sequence-to-sequence (encoder-decoder) or autoregressive (decoder) models based on the Transformer architecture [29].

They are trained on massive, internet-scale corpora using a self-supervised objective such as next-token prediction. After pretraining, many high-performance models undergo instruction fine-tuning and additional alignment to become more useful and safer in downstream use [32].

By 2025, publicly available and research-grade LLMs often exceed tens to hundreds of billions of parameters [13], and many incorporate advanced architectural and inference-optimisation techniques such as mixture-of-experts layers [6], retrieval-augmented generation [2], or quantisation-aware training for efficient deployment [4].

Because of the scale of their training data and capacity, LLMs encode a wide variety of linguistic patterns, factual knowledge, biases, and behavioural priors.

This scale gives them impressive generative and reasoning capabilities, but also makes them susceptible to emergent, unintended behaviours that are not trivially predictable — including safety failures, policy evasion, and social-engineering style manipulation [30].

2.1.1 Alignment and Safety Interventions

To mitigate risks from raw pre-trained models, developers commonly apply a pipeline of alignment techniques.

First, supervised fine-tuning (SFT) on curated instruction-following datasets helps the model adhere to desired task formats [32, 28].

Subsequently, reinforcement learning from human feedback (RLHF) is often used: human annotators rate model outputs, a reward model is trained on those ratings, and a policy optimisation algorithm such as Proximal Policy Optimisation (PPO) updates the model to align it to human preferences [18, 26].

Variants such as Direct Preference Optimisation (DPO) or RLAIF (reinforcement learning from AI feedback) aim to improve efficiency and scalability of alignment without sacrificing safety or quality [24, 10].

Despite alignment efforts, evidence demonstrates that safety training is not foolproof. Behavioural failures persist, especially under adversarial or adversary-chosen inputs: prompt-based attacks and jailbreaks remain a major vulnerability class (cf. [30, 34, 22]).

The fact that surface-level filtering and instruction tuning can be bypassed indicates that LLMs continue to rely on shallow heuristics rather than robust semantic safety guarantees [30, 34, 22].

2.2 LLM Safety Risks

LLM misuse and unintended outputs pose a broad array of risks.

The following taxonomy, commonly adopted in recent red-teaming efforts [23, 14], captures the primary threat vectors:

- **Malicious use:** generation of instructions or actionable content facilitating wrongdoing (e.g., construction of weapons or explosives, synthesis of illicit substances, or cyber intrusion).
- **Harassment, hate, and discrimination:** outputs that demean, harass or promote bias against individuals or protected groups.
- **Misinformation:** harmful or misleading claims that may influence user behaviour or beliefs.
- **Hallucinations:** fabricated or unfounded statements presented with confidence.
- **Privacy violations:** unintended leakage of sensitive information, either from memorised training data or through malicious prompts.
- **Self-harm / dangerous content:** generation of content promoting self-harm, suicide, or exploitation (especially of minors).
- **Emergent misuse and behavioural failures:** including instruction-following failures, refusal evasion (the model ignoring safety instructions), social-engineering exploitation, or covert manipulation over multiple turns.
- **Tool misuse / unsafe actions:** harmful commands or unintended actions when LLMs interface with external tools (e.g., code execution, browser control, or file manipulation).

These categories correspond to those used in large-scale safety benchmarks and red-team evaluation suites.

For example, frameworks such as *HarmBench* operationalise a broad set of harm categories and provide standard test sets and evaluation protocols across LLMs and red-teaming methods [14].

2.2.1 Application- and System-level Risks: OWASP LLM Top 10 (2025)

Beyond model-level behavioural vulnerabilities, real-world LLM deployments introduce additional system and integration risks. The OWASP LLM Top 10 (2025) [20] provides an industry-oriented overview of common failure modes encountered in practical LLM-based systems. For completeness, the full list is provided below, together with indicative categories reflecting the level at which each risk typically manifests:

- **LLM01: Prompt Injection** — crafted inputs override or manipulate system instructions. *(Model-level)*
- **LLM02: Sensitive Information Disclosure** — unintended leakage of private or memorised data. *(Model-level)*
- **LLM03: Supply Chain Vulnerabilities** — risks arising from compromised datasets, model weights, or third-party components. *(Deployment-level)*
- **LLM04: Data and Model Poisoning** — malicious training or fine-tuning data influencing model behaviour. *(Training-level)*
- **LLM05: Improper Output Handling** — insufficient sanitisation or validation of model outputs. *(Application-level)*
- **LLM06: Excessive Agency** — autonomous or uncontrolled agent actions producing unintended effects. *(Agent-level)*
- **LLM07: System Prompt Leakage** — extraction of hidden or system-level instructions. *(Model-level)*
- **LLM08: Vector and Embedding Weaknesses** — vulnerabilities in retrieval-augmented generation (RAG) pipelines or embedding stores. *(RAG-level)*
- **LLM09: Misinformation** — confident but incorrect or misleading outputs. *(Model-level)*
- **LLM10: Unbounded Consumption** — excessive resource usage leading to cost escalation or denial-of-service. *(Operational-level)*

Chapter 3

Red-Teaming of Large Language Models

This chapter focuses on red-teaming as a systematic methodology for evaluating the safety and robustness of large language models.

We first introduce red-teaming in the context of LLMs, including its definitions, goals, and a taxonomy of testing approaches used in red-teaming.

Subsequently, we describe common attack techniques (single-turn, multi-turn, prompt injection, universal triggers, trojans/backdoors).

Finally, we review evaluation metrics and benchmark frameworks commonly used to measure safety failures, and survey existing open-source tools and frameworks for LLM red-teaming.

The chapter concludes with a synthesis of gaps in current tooling and evaluation practice, motivating the design of the toolkit proposed in this thesis.

3.1 Red-Teaming: Definitions and Methodology

Originally developed in cybersecurity and military contexts, *red-teaming* refers to the practice of simulating adversaries to probe system vulnerabilities before deployment [23].

In the context of LLMs, red-teaming denotes systematic probing of model behaviour via intentionally crafted adversarial prompts or inputs designed to circumvent safety and alignment measures, with the goal of discovering previously unknown failure modes, measuring their prevalence, and informing robust defences [25, 1].

Red-teaming campaigns in LLM settings typically have several objectives [23, 25]:

- **Discovery:** reveal novel, previously undocumented failure modes (e.g., new jailbreak styles, multi-turn attack vectors, prompt injection in application contexts).
- **Quantification:** estimate how often a model fails under adversarial conditions, enabling comparison across models and defence strategies.
- **Reproducibility:** produce repeatable test cases and evaluation pipelines so that safety regressions can be detected over time.
- **Defence hardening:** feed findings back into model tuning, safety filters, or deployment guardrails to reduce vulnerability.

Unlike static benchmarks, red-teaming targets the worst-case behaviour of a model rather than its average-case performance, focusing on adversarial inputs specifically crafted to expose vulnerabilities [14, 25]. Modern red-teaming approaches in LLMs span from fully manual adversarial testing to automated pipelines that integrate adversarial prompt generation, execution, and evaluation.

The following subsections describe manual, rule-based, and automated red-teaming methodologies in more detail.

3.1.1 Manual, Rule-based and Automated Red-Teaming

Manual red-teaming: human experts write adversarial prompts, simulate realistic misuse scenarios, and attempt to elicit harmful or policy-violating responses [23]. This approach benefits from human creativity and insight into real-world misuse, including social-engineering, context-aware manipulation, and subtle or ambiguous scenarios. However, it is expensive, time-consuming, and often non-reproducible.

Rule-based / template-based testing: uses curated prompt templates (e.g., standard jailbreaks, role-play prompts, obfuscation methods, encoded instructions) or transformation rules to generate adversarial inputs systematically [25, 14]. This method is reproducible and simple, but often limited to discovering known failure classes, and does not generalise to novel or adaptive attacks.

Automated red-teaming: algorithmic or model-in-the-loop generation of adversarial prompts, for example via search, optimisation, or by using another LLM as the attacker [25, 1]. Notable work in this space includes *Tree of Attacks (TAP)* which automatically generates jailbreak prompts against black-box LLMs by iteratively refining candidates in a tree-structured search and pruning unlikely attack paths [15]. Automated methods scale well, explore large prompt spaces, and can discover novel failures, but may also generate unrealistic or trivial prompts, and often depend on the quality of the attacker or judge model and the underlying search strategy.

Automated red-teaming has recently been re-envisioned as a sequential, multi-turn process rather than isolated single-turn attempts. For example, recent work models red-teaming as a Markov Decision Process (MDP), using hierarchical reinforcement learning (RL) to optimise long-horizon attacks over entire dialogue trajectories [1].

3.2 Attack Taxonomy

Adversarial attacks against LLMs span a diverse space of techniques that exploit different aspects of model behaviour and prompting dynamics. Organising these attacks into a taxonomy is useful for understanding their mechanisms, comparing red-teaming methods, and designing systematic evaluation pipelines [23].

This section surveys the primary attack classes considered in contemporary LLM red-teaming: single-turn jailbreaks, prompt-injection attacks, multi-turn conversational manipulation, universal or transferable triggers, and trojan or backdoor-style vulnerabilities [25].

3.2.1 Single-turn Jailbreaking

Single-turn jailbreaks are adversarial prompts supplied in one-shot (single user message) that instruct the model to ignore its safety filters or system instructions [30].

Common techniques include role-play (“Pretend you are . . . ”), direct overrides (“Ignore previous instructions . . . ”), encoding or cipher-based obfuscation to hide disallowed instructions, and other prompt-engineering strategies [22].

Despite alignment training, many models remain vulnerable to these attacks [30, 34].

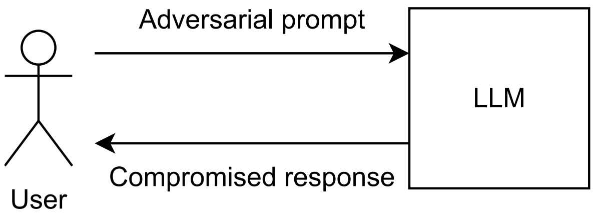

Figure 3.1: Example of a single-turn jailbreak prompt that elicits disallowed content from an LLM in one prompt. Adapted from Zhao and Zhang [33].

3.2.2 Prompt Injection

Prompt injection refers to attacks where user-supplied input (or external content, in the case of integrated applications) is directly interpreted by the LLM as instructions, potentially overriding or modifying hidden system prompts [22]. This is a major concern for real-world applications embedding LLMs (chatbots, agents, document processors, pipelines), because attackers can inject maliciously crafted instructions that the model interprets as legitimate. Empirical studies have demonstrated that many deployed LLM-based applications are vulnerable to prompt injection — for example via the *HouYi* attack, which compromised dozens of applications in a black-box setting [12]. Other work has shown that even automated, universal prompt injection attacks remain effective under defensive measures [11].

Prompt injection remains among the most significant security threats for LLM-based systems, as acknowledged by security guidance frameworks and cheat sheets (e.g., from OWASP) tailored to LLM applications [19, 20].

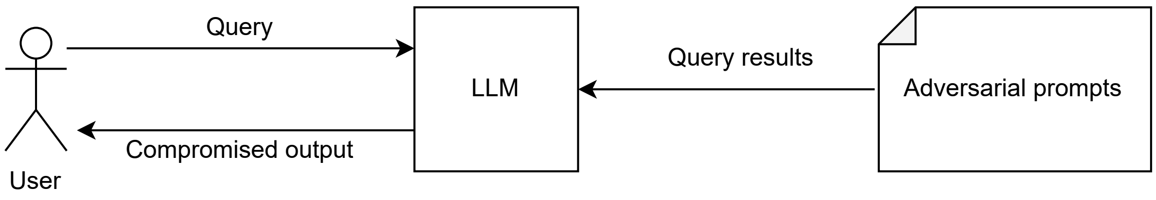

Figure 3.2: Example of an indirect prompt injection attack: a user issues a benign query, the LLM retrieves external content containing embedded instructions, and these instructions override system or developer prompts, leading to compromised output. Adapted from Greshake et al. [8].

3.2.3 Multi-turn and Conversational Attacks

Rather than achieve a jailbreak in a single prompt, adversaries may perform a sequence of manipulative steps — gradually steering the conversation, exploiting context persistence, memory, and the model's inability to consistently refuse undesirable requests [30].

Multi-turn attacks may involve context poisoning, social-engineering style dialogue, bait-and-switch tactics, or incremental obfuscation. This vector is increasingly recognised as one of the most dangerous and under-evaluated, as multi-step interactions more closely resemble realistic misuse scenarios.

Recent research recasts automated red-teaming as a multi-turn optimisation problem — better capturing long-horizon adversarial strategies — and shows that RL-based red-teaming significantly outperforms single-turn methods in eliciting harmful content [1].

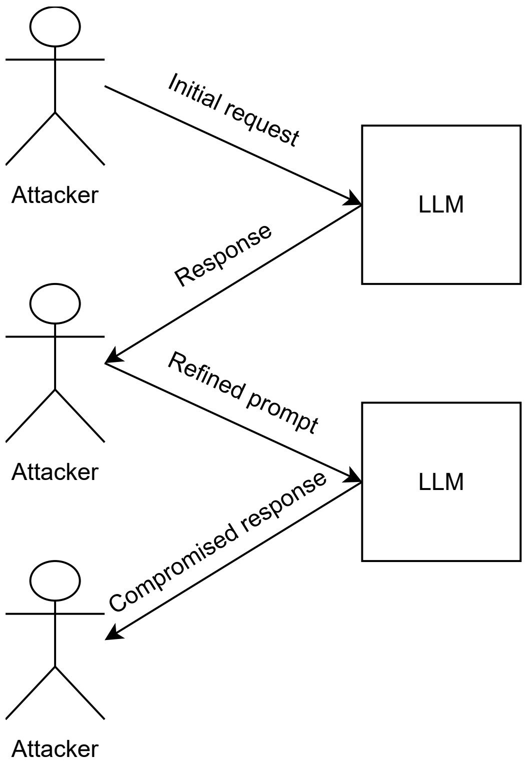

Figure 3.3: Illustration of a multi-turn jailbreak attack, in which an attacker iteratively refines prompts based on previous model responses to progressively elicit disallowed behaviour. Adapted from Zhao and Zhang [33].

3.2.4 Universal / Transferable Triggers and Trojan-style Attacks

Universal triggers or transferable adversarial prompts aim to find short token sequences (prefixes, suffixes, or embedded instructions) that reliably trigger undesired model behaviour across different inputs and even across different models [34]. This makes the attack highly reusable and dangerous.

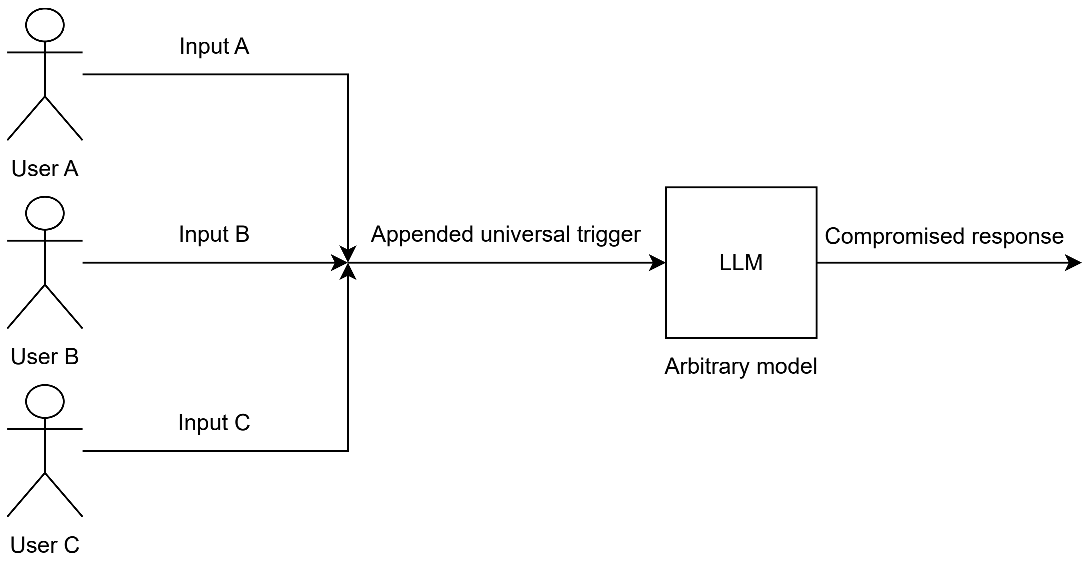

Figure 3.4: Illustration of a universal trigger attack, where a fixed trigger sequence appended to user inputs induces undesired behaviour in an unmodified language model across different prompts and, in some cases, across different models.

A related class of attacks are Trojan-style (backdoor) attacks, where the model itself is modified during training or fine-tuning to respond maliciously to a specific trigger. An example is the black-box Trojan prompt attack framework *TrojLLM*, which demonstrates that trigger patterns can be embedded into widely used LLM architectures and APIs, enabling malicious behaviour when the trigger is present [31].

Because the trigger is designed to appear benign, such attacks may bypass LLM defences.

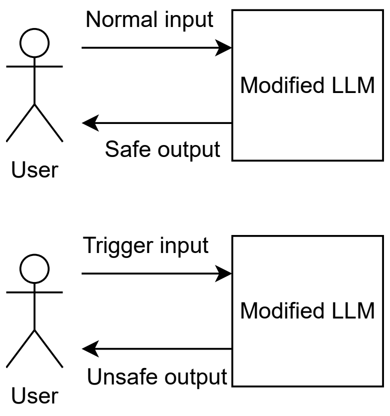

Figure 3.5: Illustration of a Trojan-style (backdoor) attack. The modified model behaves normally for inputs but produces malicious or policy-violating outputs when a specific trigger pattern is present. Unlike universal trigger attacks, the vulnerability is embedded in the model during training or fine-tuning rather than in the input prompt.

3.2.5 Prompt-based Evasion and Adversarial Perturbations

Beyond explicit instructions or injected content, adversaries may attempt adversarial prompting via subtle token-level perturbations, insertions, deletions, or encodings — minimal but adversarial changes that remain semantically similar to benign prompts yet cause the model to deviate [34].

Defences based on sanitisation or heuristic filtering often fail to detect such manipulations, especially in the presence of context sensitivity or long prompts. Certified-safety approaches have been proposed to mitigate this class: for example, erase-and-check, which systematically removes tokens and reruns safety filters to detect adversarial prompt manipulations, providing (under assumptions) a safety guarantee against insertions, suffixes, or adversarial infusions up to a bounded size [9]. While promising, these methods are computationally expensive and may degrade user experience or model utility.

Because these perturbation-based attacks operate at the token level and can be obfuscated, they represent a difficult-to-detect threat, especially when combined with other attack modalities (multi-turn, universal triggers, trojans).

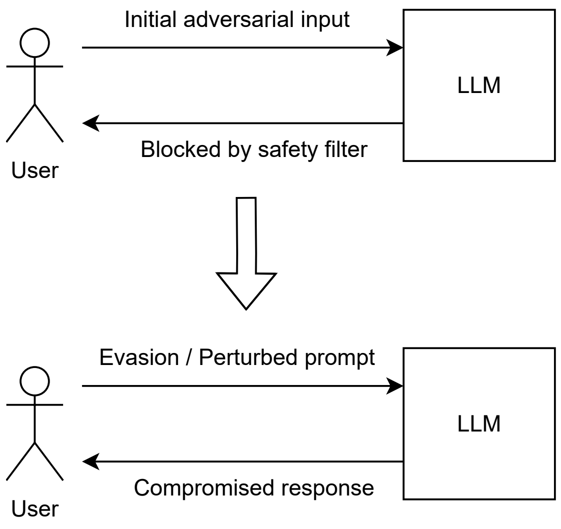

Figure 3.6: Illustration of a prompt-based evasion attack, where an attacker modifies a disallowed request using linguistic obfuscation or adversarial perturbations to bypass safety mechanisms.

3.3 Evaluation Metrics and Benchmarks

Robust and systematic evaluation is critical for red-teaming, as it enables meaningful comparison of models, attack strategies, and defences. The following metrics and benchmarks are widely adopted in contemporary LLM safety research and practice [23].

3.3.1 Quantitative Metrics

- **Attack Success Rate (ASR):** the proportion of adversarial attempts (prompts) that succeed in eliciting harmful or policy-violating outputs. ASR is the most common metric, but it depends heavily on the definition of „harmful" and on the quality of the judge (classifier, LLM-judge, human) [30, 14, 25].
- **Refusal Bypass Rate:** a variant of ASR that measures how often safe-mode refusals or safety filters are bypassed — i.e., the model produces a disallowed output rather than refusing the request [30, 23].
- **Judge Reliability Metrics:** when using automated judges (regex, classifier, LLM-based), it is crucial to measure false positives / false negatives, calibration error, and inter-annotator agreement (if doing human calibration). Poor judges can substantially distort ASR and other metrics [1, 23].
- **Robustness Metrics:** measure the stability of attacks or defences under input variations — such as paraphrasing, prompt reordering, model temperature differences, different seeds (randomness), context shuffling, or small perturbations [14, 15].
- **Transferability / Generality:** measure how well attacks discovered on one model or configuration transfer to other models, prompts, or deployments [34, 14].
- **Conversational / Trajectory Metrics:** in multi-turn attack scenarios, metrics may capture success over a dialogue trajectory (e.g., whether harmful content emerges at any point), time-to-failure, or complexity (number of turns required) [25, 15].

3.3.2 Benchmarks and Standardised Suites

To enable systematic comparison across red-teaming methods and LLMs, several standard benchmarks have recently been developed. A prominent example is *HarmBench*, which provides an open-source evaluation framework, a large pool of red-teaming methods and a diverse set of target models and defences [14].

HarmBench enables reproducible large-scale red-team evaluation and supports both attack- and defence-side experiments. Its release marks a milestone toward standardising LLM safety evaluation pipelines.

Despite this progress, many prior works still rely on ad-hoc prompt sets, non-public prompt libraries, or private internal red-team pipelines. In addition, while single-turn attack benchmarks are relatively common, multi-turn or conversational red-teaming remains underrepresented in publicly available benchmark suites.

3.4 Open-source Red-Teaming Tools and Frameworks

A variety of open-source frameworks and research prototypes have been developed to support systematic red-teaming of LLMs. Their design choices, capabilities, and limitations differ significantly, reflecting diverse goals and use cases.

3.4.1 Frameworks for rule-based or template-based red-teaming

Frameworks such as *PyRIT* provide a modular architecture for plugging in model backends, defining prompt templates, logging results, and running interactive or batch red-teaming sessions. These tools are often lightweight and well-suited for smaller projects or smaller compute budgets, but tend to lack advanced automated search or generation capabilities [16].

Another widely used toolkit is *garak*. garak provides „probes, generators, and detectors“: probes manage attack logic; generators abstract target models (LLMs, dialog systems, or any component taking text and returning text); detectors assess whether output indicates a successful attack; and the framework compiles results into human-readable reports (HTML and JSON). This design allows red-teaming across a variety of model backends and output modalities, but — as with other rule-based frameworks — it may not support automated, learning-based attack generation or multi-turn conversational scenarios out of the box [3].

3.4.2 Automated Red-Teaming Frameworks

More advanced frameworks attempt to integrate attack generation, execution, and evaluation into a unified pipeline. Notable recent work includes *MAD-MAX*, a modular adversarial red-teaming framework designed to allow multiple attack strategies (template-based, search-based, LLM-driven) within a pluggable architecture [25]. Its modular nature makes it flexible and extensible, but in practice integrating it with diverse model runtimes (local LLaMA variants, API-based models, multi-modal models) and scaling up to large-scale red-teaming campaigns remains challenging due to compute demands, backend model compatibility, and evaluation infrastructure requirements.

Beyond modular frameworks, recent work explores fully automated, trajectory-based red-teaming that recasts red-teaming as a sequential decision-making process over entire dialogues. For example, the automated red-teaming approach by Belaire et al. (2025) formalises multi-turn red-teaming as a Markov Decision Process (MDP), enabling attack policies to optimise over entire conversation trajectories rather than single messages [1]. This methodology captures realistic adversarial behaviour and reveals vulnerabilities that single-turn or template-based attacks may miss.

3.4.3 Defence-oriented and Certified Safety Approaches

In response to the growing sophistication of attacks, some work focuses on hardening LLMs against adversarial prompting. For example, the framework „erase-and-check“ provides a method for certifying safety against adversarial prompt modifications (suffix insertion, insertion at arbitrary positions, adversarial infusions) under a bounded adversarial size. This method recomputes safety classification after systematically removing tokens from the prompt and offers provable safety guarantees under certain assumptions [9]. While promising, such approaches are often computationally expensive and may impair user experience or model usability.

Moreover, the existence of Trojan prompt attacks (as demonstrated by frameworks such as *TrojLLM*) indicates that even seemingly benign prompts may embed stealthy triggers that cause harmful behaviour, complicating defence strategies and necessitating robust input sanitisation, runtime monitoring, or adversarial-resistant prompt encoding [31].

To summarise the capabilities, design choices, and limitations of the reviewed frameworks, tables 3.1 and 3.2 provide a comparison of rule-based, prompt-injection, automated, and certified-safety tools.

Framework	**Attack types**	**Backends**	**Multi-turn**	**Key notes**
PyRIT [16]	Rule/template prompts, basic injection.	API-based (OpenAI, Anthropic); limited local support.	Partial	Lightweight; regex/classifier judges; limited automation.
garak [3]	Template attacks, probes, obfuscation.	API + HuggingFace; extensible generators.	Partial	Regex/classifier detectors; widely used; limited multi-turn.
TAP [15]	Search-based jailbreaks.	Black-box APIs; local wrappers.	No	Query-efficient; single-shot focus.
Prompt-injection frameworks (e.g., HouYi) [12]	Direct/indirect injection; document-based attacks.	LLM-integrated applications.	Yes	Realistic workflows; task-specific checks.

Table 3.1: Comparison of rule-based and prompt-injection frameworks.

Framework	**Attack types**	**Backends**	**Multi-turn**	**Key notes**
MAD-MAX [25]	Template/LLM/search attacks.	API + local via plugins.	Yes	Modular; multi-turn; high compute cost.
Automatic LLM Red-teaming (Belaire et al.) [1]	LLM-as-attacker; RL optimisation; trajectory attacks.	API/local (model-in-loop).	Yes	Trajectory MDP; powerful but expensive.
Erase-and-Check [9]	Token deletions, suffix detection (defence).	Model-agnostic inputs.	N/A	Provable guarantees; computationally heavy.

Table 3.2: Comparison of automated and certified safety frameworks.

The contrasts highlighted in these tables reveal several systematic limitations across current tooling, which motivate the research gaps discussed in the following section.

3.5 Synthesis and Research Gap

The literature and tools surveyed above show that, although the research community has developed a rich taxonomy of harms, attack strategies, and evaluation metrics, substantial gaps remain that limit the effectiveness, accessibility, and reliability of red-teaming for large language models.

- **Lack of lightweight, modular, and extensible tooling:** existing frameworks typically focus either on lightweight, rule-/template-based probing or on large-scale automated red-teaming, but rarely combine both. There is a need for toolkits that are accessible to small research teams or individual developers yet support modern LLM backends, extensible attack and evaluator plugins, and reproducible logging and benchmarking.
- **Judge reliability and evaluation consistency:** many red-teaming efforts rely on heuristic or automated judges (regex filters, simple classifiers) that often lack calibration, robustness, or reproducibility. As a result, reported Attack Success Rates (ASR) may misrepresent true safety risk. Certified-safety methods (e.g., erase-and-check) provide stronger guarantees, but are computationally expensive and may be impractical for routine use.
- **Underrepresentation of multi-turn and real-world attack vectors:** most prior work and benchmarks focus on single-shot prompts; multi-turn conversational attacks — which more accurately model real-world adversaries — remain under-evaluated. Recently proposed trajectory-based red-teaming frameworks help, but are not yet part of standard open-source toolkits.
- **Difficulty integrating across diverse model backends and deployment contexts:** LLMs are deployed in varied settings (local open-source models, API-based proprietary models, multi-modal agents, applications with external tools), yet existing tools often lack abstractions or adapters covering this diversity.
- **Limited transparency and reproducibility of prompt libraries and experiment artefacts:** many studies do not release their full prompt sets, random seeds, or evaluation logs, making independent replication or longitudinal safety regression testing difficult.

These gaps motivate the design goals for the toolkit developed in this thesis: namely, modularity and pluggability (attack generators, model adapters, evaluators), reproducible experiment manifests and logging, hybrid judging (balancing cost and fidelity), and explicit support for multi-turn conversational testing. By addressing these gaps, the proposed toolkit aims to lower the barrier to entry for systematic LLM red-teaming for researchers, students, and small teams.

Chapter 4

Design of the Red-Teaming Toolkit

This chapter focuses on the design of the proposed red-teaming toolkit.

First, we outline the design requirements derived from the analysis in Chapter 3.

We then present the overall system architecture and describe the core modules, their responsibilities, and the main data and control flows that run experiments.

Finally, the chapter discusses implementation-oriented technology choices and operational considerations that motivated the prototype implementation.

Figure 4.1 provides an overview of the proposed system architecture.

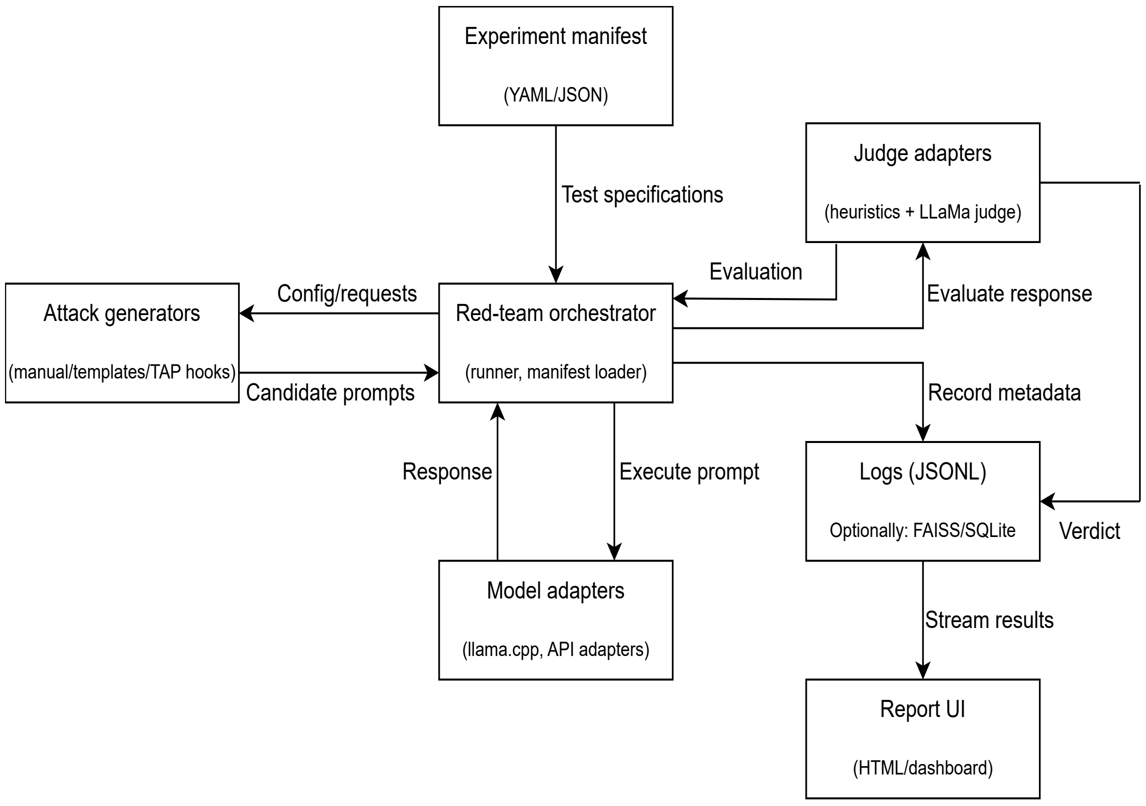

Figure 4.1: High-level architecture of the proposed red-teaming toolkit. The orchestrator coordinates attack generators, model adapters, and a hybrid judging pipeline. Experiments are declared via manifests, all results and metadata are recorded in an append-only JSONL log store and surfaced through a lightweight reporting UI.

4.1 Requirements and Goals

This section summarises the functional and non-functional requirements that guided the design of the toolkit. These requirements are derived from the gaps identified in Chapter 3, Section 3.5, and reflect both practical usability concerns and research-oriented evaluation needs. Together they shaped the system's modular architecture, the choice of logging format, and constraints on compute and deployment.

4.1.1 Functional requirements

- **Attack generators:** support manual, template-based and automated generation plugins that produce candidate adversarial prompts.
- **Model adapters:** pluggable adapters enabling execution against diverse LLM backends (local LLaMA/llama.cpp, API-based providers, or mock backends).
- **Judges / evaluation:** hybrid judging pipeline supporting lightweight heuristic detectors, classifier-based judges, and optional LLM-based judges.
- **Orchestration:** a central runner to schedule experiments, coordinate prompt delivery, manage retries/rate limits, and aggregate results.
- **Logging and reproducibility:** experiment manifests and append-only JSONL logs capturing prompts, responses, verdicts, and metadata for reproducibility and auditing.
- **Reporting UI:** a minimal web UI for browsing results, filtering experiments and exporting artifacts for further analysis.

A short synthesis: the functional requirements translate the identified gaps into concrete capabilities — accessible generation of adversarial inputs, backend-agnostic execution, a layered evaluation pipeline, and support for reproducible experiments and result inspection.

4.1.2 Non-functional requirements

- **Modularity:** clear separation of concerns (adapters, generators, judges) and stable plugin interfaces to ease extension and testing.
- **Low compute footprint:** enable lightweight local operation (e.g., small LLaMA on CPU / quantised runtime) and optional scaling to heavier compute for automated pipelines.
- **Reproducibility:** deterministic experiment manifests (seed, model config, attack generator config, judge config, prompts) and machine-readable logs to allow reruns and regression testing.
- **Extensibility:** simple mechanisms to add new attackers, adapter implementations, or judge modules without changing core orchestrator code.
- **Security and containment:** default safe settings (rate limits, sandboxed evaluation, no automatic execution of outputs) with clear opt-in for risky features.

4.1.3 Mapping to gaps from Chapter 3 section 3.5

The requirements above are chosen to address the key gaps identified earlier: (1) broaden access through lightweight tooling, (2) improve judge reliability via hybrid judging, (3) enable multi-turn trajectory testing through the orchestrator, (4) support diverse backends via model adapters, and (5) increase reproducibility via manifests and JSONL logging.

4.2 High-level Architecture

Figure 4.1 presents the component-level architecture. The following paragraphs describe components and the main data and control flows.

4.2.1 Components (narrative)

The toolkit is organised around a concise set of components whose responsibilities map directly to the requirements described above.

Central to the design are declarative **manifests**: small YAML/JSON specifications that name the target model(s), select the attack generator(s), configure the judging pipeline and thresholds, and record execution parameters (seeds, batch sizes, rate limits, output paths). Manifests serve two purposes: they make experiments reproducible by recording the exact configuration needed to rerun a trial, and they act as an explicit contract between components so that adapters, generators and judges can be composed without implicit coupling.

The **orchestrator** is the control centre that consumes manifests and coordinates execution. By centralising sequencing, retries, rate-limit handling and provenance capture, the orchestrator enforces determinism where required and provides a single place to collect metadata for auditing and analysis. This centralisation also simplifies comparing alternative attack strategies or backends because experiment control logic is kept separate from generation and evaluation code.

Model adapters decouple backend-specific details from orchestration and judging logic. Each adapter translates the toolkit's generic generate/evaluate calls into backend-specific payloads for local runtimes, remote APIs, or user-supplied custom code. Adapters therefore enable interchangeable backends (local, API, or bespoke research runtimes), declare whether they support black-box or white-box interaction, and report computational profile and cost metadata that the orchestrator records.

Attack generators encapsulate adversary logic so that generation strategies—ranging from curated templates to search-based methods or LLM-driven proposal generators—can be developed, tested, and compared independently of execution concerns. Treating attackers as plugins improves reproducibility (via manifest configuration and deterministic seeding) and encourages safe development practices (the plugin interface limits the capability surface of attack code and enables isolated execution via containerisation).

The **hybrid judging pipeline** implements a layered evaluation strategy: fast heuristic filters catch obvious cases at low cost, classifier-based judges handle well-scoped categories of harm, and optional LLM-based evaluators provide calibrated judgments for ambiguous or complex responses. This layering balances throughput, cost, and evaluation fidelity while enabling the system to record per-judge metadata (labels, confidences, and calibration statistics) for later analysis.

An append-only **JSONL log store** is used as the canonical record of execution events and verdicts to prioritise transparency, auditability, and reproducibility: logs are written sequentially without in-place modification, remain easy to inspect with standard tooling, and can be consumed independently by the reporting UI or offline analysis scripts.

A lightweight **reporting UI** consumes the canonical JSONL logs to enable inspection, failure analysis, filtering and export of experimental artifacts. The UI is intentionally read-only and analysis-first: it exposes provenance and verdicts to reviewers and collaborators without embedding additional processing logic that would complicate reproducibility.

At runtime the flows are straightforward and mirror the declarative structure of manifests. The orchestrator loads a manifest, initialises the selected attack generator(s) and model adapter(s), requests prompt candidates, submits prompts to adapters for inference, forwards model outputs to the judging pipeline, and appends structured events and verdicts to the append-only JSONL log store. The reporting UI and any offline analysis tools then read the logs to produce aggregate metrics, inspect individual traces, and support further research. This organisation preserves provenance, supports interchangeable components, and makes it straightforward to rerun or extend experiments while maintaining reproducibility and safety.

4.2.2 Principal data and control flows

The system's runtime behaviour can be understood as a set of clearly defined control and data flows that move experiment specifications to evaluated outcomes while preserving provenance for reproducibility and auditability.

The enumerated list below summarises the principal interactions; the orchestrator sits at the centre of these flows and coordinates generators, adapters and judges.

1. **Manifest → Orchestrator:** orchestrator loads the manifest describing experiment parameters.
2. **Orchestrator → Attack generators:** the orchestrator initialises and invokes the selected attack generator according to the manifest, requests candidate prompts during experiment execution and provides context or feedback as needed.
3. **Attack generators → Orchestrator:** returns candidate prompts based on initial configuration and the current request or internal state.
4. **Orchestrator → Model adapters:** sends prompts for execution against selected backend.
5. **Model adapters → Orchestrator:** returns raw responses and metadata (latency, tokens, status).
6. **Orchestrator → Judge adapters:** the orchestrator initialises and invokes the selected judges and submits outputs for evaluation.
7. **Judge adapters → Orchestrator:** returns structured verdicts.
8. **Judge adapters → Log store:** judges append structured verdicts in JSONL format.
9. **Orchestrator → Log store:** stores experiment metadata, prompts, raw responses and orchestration events in JSONL format.
10. **Log store → Reporting UI:** the UI reads logs to visualise and export results.

4.3 Module Specifications

This section describes the core modules of the toolkit in more detail, focusing on their interfaces, responsibilities, and configuration options relevant to the implementation.

4.3.1 Model Adapters

Purpose. Provide a uniform interface to execute prompts on heterogeneous backends, while permitting the inclusion of arbitrary model implementations and custom inference code.

Interface (conceptual)

```
class ModelAdapter:
    def generate(prompt: str, config: Dict) -> ModelResponse
    def health_check() -> HealthStatus
    def cost_estimate(prompt: str, config: Dict) -> CostInfo
```

Design rationale. Adapters are used to isolate the orchestrator from backend-specific details so that experiments may be rerun with different backends without changing orchestration or judging logic. Critically, the adapter abstraction must support both API-style connectors for remote providers and custom code entrypoints that allow the user to integrate bespoke model runtimes (for example an experimental research model, a new open-source checkpoint, or a custom inference pipeline). This flexibility enables reproducible comparison experiments and encourages extensibility: researchers can evaluate novel model implementations simply by providing a compatible adapter.

Supported backends (examples)

- **Local LLaMA/llama.cpp** — lightweight, local, free option for both generation and judging.
- **API backends** — optional connectors for providers (OpenAI, Anthropic, etc.) implemented as separate adapters.
- **Custom runtime / plugin code** — user-provided Python modules or binary runtimes integrated via the adapter interface and sandboxed execution.
- **Mock/test** — deterministic stub for unit tests and CI.

Inference modes and performance considerations. Adapters should explicitly expose the inference characteristics they support. Two common modes are:

- **Black-box inference:** the orchestrator interacts with the model purely via an API that returns text outputs and metadata. This mode is appropriate for remote providers or proprietary runtimes where internal state is unavailable and interaction is limited to prompt–response exchanges.

- **White-box inference:** the adapter exposes internal model information (for example logits, attention masks, or intermediate activations) or allows custom model-internal callbacks. This mode is appropriate for local or research-oriented runitmes and enables attack strategies or analyses that require gradient access, introspection or fine-grained instrumentation of model behaviour.

The adapter should declare which mode(s) it supports so that attack generators and judges can adapt accordingly.

Quantisation and low-compute operation. Practical deployment on laptops or small servers motivates support for quantised runtimes (for example GGUF [7] or other low-precision formats) within local adapters. Adapters should therefore allow configuration flags that request quantised model files or reduced-precision inference. Where appropriate, adapters should also report effective memory usage and latency estimates, so manifests can record the computational profile of each experiment for reproducibility.

Adapter responsibilities

- translate generic calls into backend-specific payloads,
- manage rate limits, errors, retries and backoff policies,
- normalise response objects (text, tokens, metadata),
- surface cost/latency metadata to the orchestrator,
- declare supported inference mode(s) and quantisation capability,
- provide a secure execution pathway for custom code (sandboxing or subprocess isolation).

4.3.2 Attack Generators

Purpose. Produce adversarial prompts or prompt sequences under an agreed interface, while encapsulating the generation strategy so that different attackers can be compared under identical experimental conditions.

Plugin interface (conceptual)

```
class AttackGenerator:
    def configure(params: Dict) -> None
    def next(batch_size: int) -> List[PromptCandidate]
    def reset() -> None
```

Design rationale. Making attack logic a first-class plugin type separates the concern of adversary design from experiment execution, which improves reproducibility and reuse. It also enables the toolkit to host diverse attacker implementations — from hand-curated templates to search-based or LLM-driven generators — while the orchestrator remains agnostic to generation internals.

Module responsibilities and required capabilities. An attack module must:

- accept a declarative configuration (from the experiment manifest) that specifies seeds, constraints, and generation parameters;
- return a sequence (or batch) of `PromptCandidate` objects, each containing the prompt text and an identifier;
- offer deterministic behaviour under a fixed seed unless the manifest requests stochasticity (to enable reruns and regression testing);
- provide an optional stateful policy interface when multi-turn or adaptive attacks are required (e.g., methods for updating internal state based on model responses);
- document whether it requires white-box capabilities (e.g., access to logits, gradients) or can operate in a black-box setting, and gracefully degrade if unavailable;
- include lightweight validation hooks or unit tests (for example, a `sanity_check()` method) to verify expected output formats prior to running large experiments.

Types and examples Attack generator implementations may include:

- **Manual / template-based:** curated prompts or parameterised templates appropriate for reproducible baselines.
- **Search-based:** tree or beam-search algorithms that explore the prompt space, prioritising query-efficiency.
- **LLM-driven generators:** use an LLM to propose candidate prompts.

Testing and safety considerations Generators that produce content likely to trigger harmful outputs should be accompanied by explicit safety metadata (a declared risk level in the manifest) and be executed only under appropriate opt-in settings. For development, each module should provide small example manifests and self-checking routines so that the orchestrator can verify compatibility before large-scale runs begin.

4.3.3 Judges and Evaluation

Purpose. Decide whether a given model output constitutes a success for a given attack objective and produce a structured verdict.

Hybrid pipeline

1. **Heuristic filters:** fast, low-cost regex or rule checks for obvious cases.
2. **Classifier-based judges:** trained binary / multi-class classifiers for specific harm categories.
3. **LLM-based judges:** optional judge using a LLM for evaluation and calibration.

Judge adapter interface (conceptual)

```
class JudgeAdapter:
    def configure(params: Dict) -> None
    def evaluate(response: ModelResponse, prompt: PromptCandidate) -> Verdict
    def health_check() -> HealthStatus
    def reset() -> None
```

Verdict structure (conceptual)

- `verdict_id` (uuid)
- `experiment_id`, `prompt_id`, `model_id`, `trace_id`
- `labels:` list of detected harms / categories
- `confidence:` numeric score
- `judge_type:` heuristic|classifier|llm
- `explanation:` short string or structured rationale
- `timestamp:` time of evaluation

Judge reliability Record false positive/negative estimates, and when possible, human calibration metadata (annotation counts, inter-annotator agreement).

4.3.4 Experiment Manifests and Logging

Manifest (declarative) A manifest is a small YAML/JSON file describing:

- `experiment_id`, `author`, `description`
- `model` and adapter config
- `generator` selection and parameters
- `judge` pipeline and thresholds
- `seeds`, `batch_sizes`, `rate_limits`
- `output` path / logging destination

Logging format (conceptual) JSONL (newline-delimited JSON) is used as the canonical export format for logs. Each line represents an atomic event record, for example:

```
{
  "event_type": "trial_result",
  "timestamp": "2025-01-10T12:34:56Z",
  "experiment_id": "exp-2025-001",
  "prompt_id": "p-0001",
  "prompt_text": "Explain how to bypass X",
  "model": "llama-local-v1",
  "response_text": " ... ",
  "tokens": 182,
  "judge_verdicts": [
    {"judge_type":"heuristic","label":"refusal","confidence":0.95}
  ]
}
```

The JSONL log serves as the canonical, append-only record of experiment events; it is compact, streaming-friendly and easy to import for offline analysis or archival.

4.3.5 Plugin System and Extensibility

Design principles

- **Small stable interfaces:** adapters and generators rely on an API with few methods and well-defined inputs/outputs.
- **Entrypoints:** use Python entry points (or a minimal plugin loader) to discover third-party plugins.
- **Isolation:** run untrusted or resource-heavy plugins in containers to prevent crashes or data leaks.

Example plugin lifecycle

1. runtime loads plugin via entrypoint,
2. orchestrator calls ‘configure()‘ with manifest params,
3. orchestrator calls ‘next()‘ to obtain prompts,
4. at experiment end orchestrator calls ‘reset()‘ or ‘close()‘.

4.4 Security, Privacy and Ethical Considerations

The toolkit is intentionally designed for responsible use. Key safeguards and guidance:

- **Safe defaults:** rate limits, conservative timeouts, and an explicit opt-in for any code-execution or tool-integration features.
- **PII handling:** redact or avoid storing personally identifiable information (PII); if necessary, store only hashed identifiers and be explicit about retention policies.
- **Red-team ethics:** require a documented use-case and owner contact for experiments that simulate real-world exploitation; discourage any experiments that could meaningfully cause harm or enable illegal activity.

- **Containment:** run external or third-party plugins in isolated processes/containers; ensure model outputs are never automatically executed by the system.
- **Logging and audit:** logs are append-only and include experiment provenance to enable post-hoc review.

To reduce risks associated with executing user-supplied or third-party code, adapters and plugin loaders execute untrusted code in isolated subprocesses or lightweight containers and require explicit opt-in in the manifest for any plugin that requests elevated capabilities (for example code execution, external network access, or writing to the host filesystem).

4.5 Conceptual Deployment

4.5.1 Technology choices

The technology choices for the implementation prioritise accessibility, reproducibility, and pragmatic extensibility, while maintaining sufficient performance for red-teaming experiments rather than optimising for maximal throughput.

Python was selected as the primary implementation language because its ecosystem provides mature packaging and testing tools, straightforward plugin discovery mechanisms, and broad familiarity in the research community. These properties reduce friction for contributors and simplify reproducible packaging of experiments.

For local model runtimes we favour `llama.cpp` together with an idiomatic Python binding (for example `llama-cpp-python`) because this combination supports low-memory, CPU-based inference and quantised model files (GGUF [7] or similar formats). Using a quantised, CPU-capable runtime aligns with the design requirement to enable lightweight operation on personal workstations while still permitting an upgrade path to more powerful GPU/proprietary backends via the adapter interface.

Adapters expose a uniform API that allows the orchestrator to invoke either remote API backends (OpenAI, Anthropic, etc.) or custom local runtimes. Crucially, adapters also permit user-supplied custom code entrypoints so that research implementations, experimental model checkpoints, or bespoke inference pipelines can be evaluated without modifying core orchestration logic. Where custom code is accepted, the design emphasises sandboxing and subprocess isolation to reduce security and stability risks.

For experiment provenance and offline analysis the system uses newline-delimited JSON (**JSONL**) as the canonical log format. JSONL is intentionally compact, streaming-friendly, and language-agnostic: it enables straightforward append-only logging, simple programmatic consumption using standard Unix-style tooling, and easy import into analysis environments. To support interactive inspection and dissemination of results, a lightweight read-only reporting web application is provided. Frameworks such as Streamlit or a small Flask-backed frontend are appropriate at this stage because they minimise development overhead while delivering sufficient functionality for filtering, inspecting individual trials, and exporting artifacts; heavier frontend frameworks are unnecessary given the read-only, analysis-first design.

Reproducible local deployment is supported via containerisation (Docker and docker-compose) so that the environment, Python dependencies, and optional local runtime binaries can be packaged and shared. Containers make it easier to document representative resource requirements and to provide pre-configured quantised model files for laptop-first experiments.

Finally, the orchestrator is designed as a simple Python service (a CLI implemented with **Typer** and an **asyncio**-based runner) to keep the implementation compact, testable, and readable for other developers or researchers who may extend the toolkit.

Together, these technology choices form a practical stack that aligns with the chapter's stated requirements: modularity through clear adapter interfaces, low compute footprint through quantised local runtimes, reproducibility through manifest-driven experiments and JSONL logs, and extensibility via plugin entrypoints and containerised deployments.

Local vs API backends. The design prioritises local execution using LLaMA models via `llama.cpp` for development and cost-free judging. Adapters for API-based model providers are optional and configured declaratively through the experiment manifest. The adapter pattern ensures that backends can be exchanged without modifying orchestration logic.

Logging and indexing. For improved query performance in the user interface, an optional lightweight index (e.g., SQLite or FAISS) may be derived from the canonical JSONL logs. Log rotation and retention policies are considered as operational concerns for long-running experiments.

Operational considerations. The operational design of the toolkit is guided by the assumption that it will primarily be used in resource-constrained environments, such as personal workstations or small research servers. Consequently, expected CPU and memory requirements are documented for representative model sizes, and the use of smaller or quantised models is recommended during development and exploratory experimentation. This enables practitioners to conduct meaningful red-teaming experiments without requiring dedicated accelerator hardware.

From a security and containment perspective, the toolkit assumes that plugins and attack generators may be untrusted or experimental. To mitigate associated risks, the design supports execution of external components in isolated subprocesses or lightweight containers. In addition, model outputs are treated as untrusted data and are never automatically executed or passed to downstream systems without explicit user intervention.

Basic observability features are considered an important operational aid rather than a core contribution of the toolkit. The orchestrator may therefore expose lightweight health checks and runtime metrics, such as request rates, latency distributions, queue lengths, and judge throughput. These signals support debugging, performance evaluation, and failure analysis during both development and experimental use, while remaining intentionally simple to avoid additional operational complexity.

Chapter 5

Implementation

Chapter 6

Evaluation

Chapter 7

Conclusion

Bibliography

[1] Belaire, R.; Sinha, A. and Varakantham, P. *Automatic LLM Red Teaming*. 2025. Available at: https://arxiv.org/abs/2508.04451.

[2] Borgeaud, S.; Mensch, A.; Hoffmann, J.; Cai, T.; Rutherford, E. et al. *Improving language models by retrieving from trillions of tokens*. 2022. Available at: https://arxiv.org/abs/2112.04426.

[3] Derczynski, L.; Galinkin, E.; Martin, J.; Majumdar, S. and Inie, N. *Garak: A Framework for Security Probing Large Language Models*. 2024. Available at: https://arxiv.org/abs/2406.11036.

[4] Dettmers, T.; Pagnoni, A.; Holtzman, A. and Zettlemoyer, L. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. Available at: https://arxiv.org/abs/2305.14314.

[5] Esmailzadeh, Y. *Potential Risks of ChatGPT: Implications for Counterterrorism and International Security*. 2023. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4461195.

[6] Fedus, W.; Zoph, B. and Shazeer, N. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. 2022. Available at: https://arxiv.org/abs/2101.03961.

[7] Gerganov, G. *GGUF: An Efficient File Format for Large Language Models*. 2023. Available at: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md.

[8] Greshake, K.; Abdelnabi, S.; Mishra, S.; Endres, C.; Holz, T. et al. *Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*. 2023. Available at: https://arxiv.org/abs/2302.12173.

[9] Kumar, A.; Agarwal, C.; Srinivas, S.; Li, A. J.; Feizi, S. et al. *Certifying LLM Safety against Adversarial Prompting*. 2025. Available at: https://arxiv.org/abs/2309.02705.

[10] Lee, H.; Phatale, S.; Mansoor, H.; Mesnard, T.; Ferret, J. et al. *RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback*. 2024. Available at: https://arxiv.org/abs/2309.00267.

[11] Liu, X.; Yu, Z.; Zhang, Y.; Zhang, N. and Xiao, C. *Automatic and Universal Prompt Injection Attacks against Large Language Models*. 2024. Available at: https://arxiv.org/abs/2403.04957.

[12] Liu, Y.; Deng, G.; Li, Y.; Wang, K.; Wang, Z. et al. *Prompt Injection attack against LLM-integrated Applications*. 2024. Available at: https://arxiv.org/abs/2306.05499.

[13] Lu, X.; Liu, Z.; Liusie, A.; Raina, V.; Mudupalli, V. et al. *Blending Is All You Need: Cheaper, Better Alternative to Trillion-Parameters LLM*. 2024. Available at: https://arxiv.org/abs/2401.02994.

[14] Mazeika, M.; Phan, L.; Yin, X.; Zou, A.; Wang, Z. et al. *HarmBench: A Standardized Evaluation Framework for Automated Red Teaming and Robust Refusal*. 2024. Available at: https://arxiv.org/abs/2402.04249.

[15] Mehrotra, A.; Zampetakis, M.; Kassianik, P.; Nelson, B.; Anderson, H. et al. *Tree of Attacks: Jailbreaking Black-Box LLMs Automatically*. 2024. Available at: https://arxiv.org/abs/2312.02119.

[16] Munoz, G. D. L.; Minnich, A. J.; Lutz, R.; Lundeen, R.; Dheekonda, R. S. R. et al. *PyRIT: A Framework for Security Risk Identification and Red Teaming in Generative AI System*. 2024. Available at: https://arxiv.org/abs/2410.02828.

[17] OpenAI. *Red Teaming Network*. 2023. Available at: https://openai.com/blog/red-teaming-network.

[18] Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L. et al. *Training language models to follow instructions with human feedback*. 2022. Available at: https://arxiv.org/abs/2203.02155.

[19] OWASP Cheat Sheet Series. *LLM Prompt Injection Prevention Cheat Sheet*. 2025. Available at: https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html.

[20] OWASP Top 10 for LLMs. *Top 10 Risk & Mitigations for LLMs and Gen AI Apps*. 2025. Available at: https://genai.owasp.org/llm-top-10.

[21] Parliament, E. and Council. *Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. 2024. Available at: https://eur-lex.europa.eu/eli/reg/2024/1689/oj.

[22] Perez, F. and Ribeiro, I. *Ignore Previous Prompt: Attack Techniques For Language Models*. 2022. Available at: https://arxiv.org/abs/2211.09527.

[23] Purpura, A.; Wadhwa, S.; Zymet, J.; Gupta, A.; Luo, A. et al. *Building Safe GenAI Applications: An End-to-End Overview of Red Teaming for Large Language Models*. 2025. Available at: https://arxiv.org/abs/2503.01742.

[24] Rafailov, R.; Sharma, A.; Mitchell, E.; Ermon, S.; Manning, C. D. et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2024. Available at: https://arxiv.org/abs/2305.18290.

[25] Schoepf, S.; Hameed, M. Z.; Rawat, A.; Fraser, K.; Zizzo, G. et al. *MAD-MAX: Modular And Diverse Malicious Attack MiXtures for Automated LLM Red Teaming*. 2025. Available at: https://arxiv.org/abs/2503.06253.

[26] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A. and Klimov, O. *Proximal Policy Optimization Algorithms*. 2017. Available at: https://arxiv.org/abs/1707.06347.

[27] Shaw, A.; Ye, A.; Krishna, R. and Zhang, A. X. *Agonistic Image Generation: Unsettling the Hegemony of Intention*. 2025. Available at: https://arxiv.org/abs/2502.15242.

[28] Stiennon, N.; Ouyang, L.; Wu, J.; Ziegler, D. M.; Lowe, R. et al. *Learning to summarize from human feedback*. 2022. Available at: https://arxiv.org/abs/2009.01325.

[29] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L. et al. Attention Is All You Need. *CoRR*, 2017, abs/1706.03762. Available at: http://arxiv.org/abs/1706.03762.

[30] Wei, A.; Haghtalab, N. and Steinhardt, J. *Jailbroken: How Does LLM Safety Training Fail?* 2023. Available at: https://arxiv.org/abs/2307.02483.

[31] Xue, J.; Zheng, M.; Hua, T.; Shen, Y.; Liu, Y. et al. *TrojLLM: A Black-box Trojan Prompt Attack on Large Language Models*. 2023. Available at: https://arxiv.org/abs/2306.06815.

[32] Zhang, B.; Liu, Z.; Cherry, C. and Firat, O. *When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method*. 2024. Available at: https://arxiv.org/abs/2402.17193.

[33] Zhao, Y. and Zhang, Y. *Siren: A Learning-Based Multi-Turn Attack Framework for Simulating Real-World Human Jailbreak Behaviors*. 2025. Available at: https://arxiv.org/abs/2501.14250.

[34] Zou, A.; Wang, Z.; Carlini, N.; Nasr, M.; Kolter, J. Z. et al. *Universal and Transferable Adversarial Attacks on Aligned Language Models*. 2023. Available at: https://llm-attacks.org.