CAI Fluency: A Framework for Cybersecurity AI Fluency

Víctor Mayoral-Vilches[1], **Jasmin Wachter**[2], **Cristóbal R. J. Veas Chavez**[2], **Cathrin Schachner**[2], **Luis Javier Navarrete-Lozano**[1] **and María Sanz-Gómez**[1]

[1] **Alias Robotics**, Vitoria-Gasteiz, Álava, Spain, ✉ victor@aliasrobotics.com
[2] **University of Klagenfurt**, Klagenfurt, Austria, ✉ Jasmin.Wachter@aau.at

October 8, 2025

Abstract

This work introduces `CAI Fluency`, an an educational platform of the Cybersecurity AI (CAI) framework dedicated to democratizing the knowledge and application of cybersecurity AI tools in the global security community. The main objective of the CAI framework is to accelerate the widespread adoption and effective use of artificial intelligence-based cybersecurity solutions, pathing the way to *vibe-hacking* – the cybersecurity analogon to *vibe-coding*.

CAI Fluency builds upon the Framework for AI Fluency, adapting its three modalities of human-AI interaction and four core competencies specifically for cybersecurity applications. This theoretical foundation ensures that practitioners develop not just technical skills, but also the critical thinking and ethical awareness necessary for responsible AI use in security contexts.

This technical report serves as a white-paper, as well as detailed educational and practical guide that helps users understand the principles behind the CAI framework, and educates them how to apply this knowledge in their projects and real-world security contexts.

Keywords Red-teaming · Cybersecurity AI · Vibe Hacking · Cybersecurity Education · AI Fluency · Human-AI Interaction

1 Introduction and Motivation [†] [*]

1.1 CAI – a cybersecurity AI framework

The cybersecurity landscape is undergoing a dramatic transformation as AI becomes increasingly integrated into security operations. **We predict that by 2028, AI-powered security testing tools will outnumber human pentesters.** This shift represents a fundamental change in how we approach cybersecurity challenges. *AI is not just another tool - it's becoming essential for addressing complex security vulnerabilities and staying ahead of sophisticated threats. As organizations face more advanced cyber attacks, AI-enhanced security testing will be crucial for maintaining robust defenses.*

This work builds upon prior efforts[1] [1, 2, 3] and similarly, we believe that democratizing access to advanced cybersecurity AI tools is vital for the entire security community. That's why we're releasing

---
[1] https://github.com/aliasrobotics/cai

Cybersecurity AI (`CAI`) as an open source framework. Additionally, we launch `CAI Fluency` – a framework educating about Cybersecurity AI.

Our goal is to empower security researchers, ethical hackers, and organizations to build and deploy powerful AI-driven security tools. By making these capabilities openly available and by sharing our knowledge with the community, we aim to level the playing field and ensure that cutting-edge security AI technology isn't limited to well-funded private companies or state actors.

1.2 Ethical principles behind CAI and CAI Fluency

You might be wondering whether releasing CAI *in-the-wild* and sharing our know-how – given CAI's capabilities and security implications – is ethical.

Our decision to open-source this framework is guided by two core ethical principles:

1. **Democratizing Cybersecurity AI:** We believe that advanced cybersecurity AI tools should be accessible to the entire security community, not just well-funded private companies or state actors. By releasing CAI as an open source framework, we aim to empower security researchers, ethical hackers, and organizations to build and deploy powerful AI-driven security tools, leveling the playing field in cybersecurity.

2. **Transparency in AI Security Capabilities:** Based on our research results, understanding of the technology, and dissection of top technical reports, we argue that current LLM vendors are undermining their cybersecurity capabilities. This is extremely dangerous and misleading. By developing CAI openly, we provide a transparent benchmark of what AI systems can actually do in cybersecurity contexts, enabling more informed decisions about security postures.

CAI is built on the following core principles:

- **Cybersecurity oriented AI framework:** CAI is specifically designed for cybersecurity use cases, aiming at semi- and fully-automating offensive and defensive security tasks.

- **Open source, free for research:** CAI is open source and free for research purposes. We aim at democratizing access to AI and Cybersecurity. For professional or commercial use, including on-premise deployments, dedicated technical support and custom extensions reach out to obtain a license.

- **Lightweight:** CAI is designed to be fast, and easy to use.

- Modular and agent-centric design: CAI operates on the basis of agents and agentic patterns, which allows flexibility and scalability. You can easily add the most suitable agents and pattern for your cybersecurity target case.

- **Tool-integration:** CAI integrates already built-in tools, and allows the user to integrate their own tools with their own logic easily.

- **Logging and tracing integrated:** using phoenix, the open source tracing and logging tool for LLMs. This provides the user with a detailed traceability of the agents and their execution.

- **Multi-Model Support:** more than 300 supported and empowered by LiteLLM. The most popular providers:

  - **Anthropic:** `Claude 3.7, Claude 3.5, Claude 3, Claude 3 Opus`
  - **OpenAI:** `O1, O1 Mini, O3 Mini, GPT-4o, GPT-4.5 Preview`
  - **DeepSeek:** `DeepSeek V3, DeepSeek R1`

- **Ollama:** `Qwen2.5 72B`, `Qwen2.5 14B`, etc

⚠ Access to this library and the use of information, materials (or portions thereof), i**s not intended, and is prohibited, where such access or use violates applicable laws or regulations**. By no means the authors encourage or promote the unauthorized tampering with running systems. This can cause serious human harm and material damages.

By no means the authors of CAI encourage or promote the unauthorized tampering with computing systems. Please don't use the source code in here for cybercrime. Pentest for good instead. By downloading, using, or modifying this source code, you agree to the terms of the LICENSE as well as the DISCLAIMER file.

1.3 Closed-Source is NOT an Alternative

Cybersecurity AI is a critical field, yet many groups are misguidedly pursuing it through closed-source methods for pure economic return, leveraging similar techniques and building upon existing closed-source (*often third-party owned*) models. This approach not only squanders valuable engineering resources but also represents an economic waste and results in redundant efforts, as they often end up reinventing the wheel. Below in Table 1 we list some of the closed-source initiatives we keep track of and attempting to leverage genAI and agentic frameworks in cybersecurity AI:

	Autonomous Cyber	CrackenAGI	ETHIACK	Horizon3
Lakera	Mindfort	Mindgard	NDAY Security	Runsybil
Selfhack	SQUR	Sxipher[2]	Staris	Terra Security
Xint	XBOW	ZeroPath	Zynap	7ai

Table 1: A non-exhaustive list of closed source alternatives to CAI.

1.4 Why Education is Key to Democratizing Cybersecurity

The process of democratizing cybersecurity AI tools entails the provision of tools for researchers and students, but it also involves making these advanced technologies accessible to a broader audience beyond just experts and large organizations. While providing tools for researchers and studentsis a crucial step towards fostering a more inclusive and diverse cybersecurity community, the mere provision of tools is not enough, for the following reasons:

- **Complexity of AI Tools:** Generative and Agentic AI tools often come with a steep learning curve due to their complex nature. *Cybersecurity* AI tools additionally demand a good understanding of both cybersecurity principles and AI concepts. Proper education and documentation help in demystifying these complexities, making the tools more approachable for a wider audience.

- **Skill Gap:** There is a significant skill gap in the cybersecurity field, with a shortage of professionals who are proficient (aka. *fluent*) in both cybersecurity and AI. Educational materials can help bridge this gap by providing learning resources that cover the basics of AI in cybersecurity, thereby empowering more individuals to use these tools effectively.

- **Community Engagement:** Democratization is not just about access but also about fostering a sense of community. By providing educational resources and encouraging the use of these tools,

the cybersecurity community can engage more actively. This can lead to collaborative learning, the sharing of best practices, and the development of new applications for these tools.

- **Innovation and Adaptability:** Education encourages innovation and adaptability. When more people have access to and understand these tools, there is a higher likelihood of novel use cases and innovative applications emerging. This can accelerate the evolution of cybersecurity practices and solutions.

- **Inclusivity and Diversity:** Making cybersecurity AI tools accessible through education and documentation promotes inclusivity and diversity within the cybersecurity community. It allows individuals from various backgrounds, including underrepresented groups and those with few ressources, to participate and contribute. This diversity can lead to a richer set of perspectives and solutions.

- **Reducing Dependency on Vendors:** *Over-reliance* on vendors for cybersecurity solutions can limit the ability of organizations and individuals to customize and adapt solutions to their specific needs. By educating users on how to use open-source tools, the community becomes less dependent on commercial solutions and more capable of developing customized solutions.

- **Continuous Learning and Improvement:** Cybersecurity is a rapidly evolving field, with new threats and technologies emerging continuously. Educational resources and documentation that are regularly updated can help the community stay current with the latest developments and best practices in using AI for cybersecurity.

In summary, while providing access to cybersecurity AI tools is a critical step towards democratization, it is the education and documentation that truly enables their widespread adoption and effective use across the entire security community. By lowering the barriers to entry and fostering a culture of learning and collaboration, the democratization of cybersecurity AI tools can lead to a more robust, diverse, and resilient cybersecurity ecosystem.
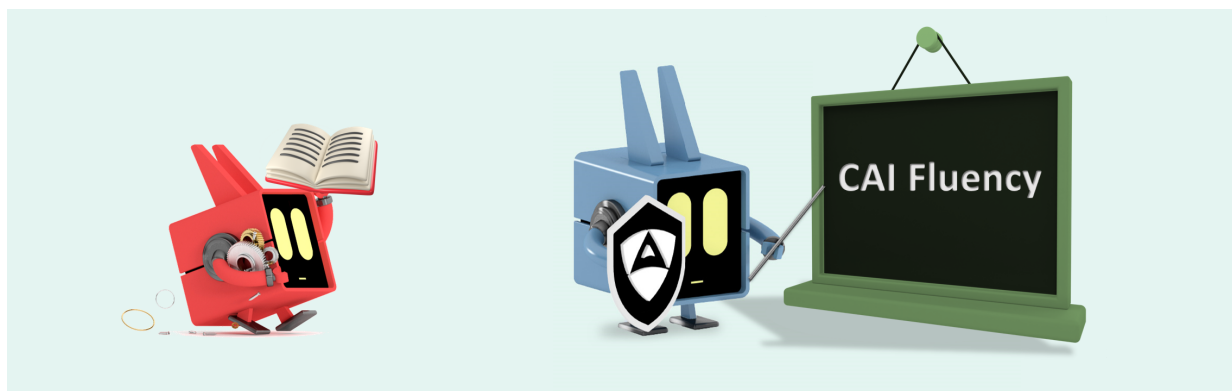

Figure 1: *CAI Fluency* – A new platform dedicated to education and documentation on cybersecurity AI.

*For this reason, we decided to create a new platform – **CAI Fluency** – dedicated to education and documentation, profound lectures and tutorials, ensuring that CAI is usable and beneficial for the entire security community.*

1.5 Introducing CAI Fluency: An Educational Framework for Cybersecurity AI Use and Deployment

CAI Fluency is part of the official CAI GitHub. It is an an *educational platform of the CAI framework dedicated to democratizing the knowledge and application of cybersecurity AI tools in the global security community*. The main objective is to enable the widespread adoption and effective use of artificial intelligence-based cybersecurity solutions.

This platform aims to minimize barriers and cultivate a culture of continuous learning and collaboration through educational resources, hands-on tutorials and practical guides that help users understand how to interact with CAI framework, and ultimately apply this knowledge in their projects and real-world security contexts.

CAI Fluency is grounded in a comprehensive theoretical framework that we detail in Section 1.4, which adapts established AI fluency principles specifically for cybersecurity applications, ensuring that practitioners develop both technical proficiency and ethical awareness.

Through this commitment to education and knowledge sharing, CAI Fluency helps foster a more inclusive, collaborative and resilient Cybersecurity AI community.

This document serves as an extensive technical report and user guide on the CAI arichitecture, tools and foundations, complementing the `GitHub repository` [4] as well as the technical report [1].

1.6 Theoretical Foundation: The Framework for AI Fluency in Cybersecurity

To establish a solid educational foundation for CAI Fluency, we build upon the *Framework for AI Fluency* developed by Dakan and Feller [5]. This framework provides a comprehensive structure for understanding and developing competencies in Human-AI interaction, which we adapt specifically for cybersecurity applications.

The integration of this framework is particularly relevant to CAI because it addresses the fundamental challenge of bridging the gap between AI capabilities and human expertise in cybersecurity contexts. As we will explore in subsequent sections, CAI implements ReAct agent models that require sophisticated human-AI collaboration - precisely the domain where AI fluency becomes critical.

1.6.1 AI Fluency Defined

Dakan and Feller define **AI Fluency** as the *ability to work*

- *effectively,*

- *efficiently,*

- *ethically,* and

- *safely*

within *emerging modalities of Human-AI interaction*. In the cybersecurity context, this means understanding how to leverage AI tools not merely as efficiency engines, but as authentic thinking partners for conducting meaningful security work while maintaining the highest standards of ethical practice.

This definition becomes particularly important when considering the ReAct framework (detailed in Section 2.2), where the iterative reasoning-action cycle requires practitioners to effectively delegate tasks, describe requirements, discern outputs, and maintain diligence throughout the process.

1.6.2 Three Modalities of Human-AI Interaction in Cybersecurity

Building on Dakan and Feller's framework, we adapt and extend their three modalities of interaction for cybersecurity contexts. These modalities are particularly relevant to cybersecurity practitioners and directly align with CAI's architectural design:

Modality 1: Automation (AI Performs Security-Defined Tasks) In this modality, AI systems perform cybersecurity tasks *independently based on direct human instructions*. This corresponds to single-agent ReAct implementations in CAI, where practitioners delegate specific security tasks to AI agents. This is particularly valuable for:

- **Reconnaissance automation:** Automated scanning, enumeration, and information gathering using CAI's built-in tools

- **Log analysis:** Processing large volumes of security logs and identifying patterns through specialized agents

- **Report generation:** Creating standardized vulnerability reports and documentation via CAI's reporting agents

- **Routine security tasks:** Password policy checks, basic compliance validation using CAI's validation tools

Modality 2: Augmentation (AI and Human Collaborate on Security Tasks) This modality involves AI and human *co-defining* and *co-executing* security tasks iteratively. It focuses on enhancing human cybersecurity expertise through AI thinking partnership and aligns with CAI's multi-agent patterns and Human-In-The-Loop (HITL) capabilities:

- **Threat hunting:** Collaborative analysis of complex attack patterns using CAI's swarm patterns for distributed analysis

- **Vulnerability assessment:** Joint evaluation of security weaknesses through CAI's specialized assessment agents

- **Incident response:** Real-time collaboration during security incidents using CAI's coordination mechanisms

- **Security research:** Exploring novel attack vectors through CAI's research and development patterns

Modality 3: Agency (Human Configures AI for Independent Security Operations) In this advanced modality, humans configure AI systems to *independently perform future security tasks*, potentially for other users or in autonomous security operations. This corresponds to CAI's most sophisticated agentic patterns:

- **Autonomous monitoring:** AI systems configured through CAI patterns that independently detect and respond to threats

- **Adaptive defense systems:** AI agents that evolve security postures based on emerging threats using CAI's learning mechanisms

- **Security training simulators:** AI tutors and adversarial training environments built with CAI's educational patterns

- **Intelligent security orchestration:** AI systems that coordinate complex security workflows using CAI's orchestration capabilities

1.6.3 The Four Core Competencies for Cybersecurity AI Fluency

Building on Dakan and Feller's original framework's "4 D's" (Delegation, Description, Discernment, and Diligence), we adapt and extend these competencies specifically for cybersecurity applications and integrate them with CAI's architectural principles:

Delegation - Strategic AI Tool Selection for Security Goals *Delegation* in cybersecurity involves identifying when and how to use AI tools effectively in security processes while understanding the unique capabilities and limitations of various AI technologies in security contexts. This competency is essential for effective use of CAI's agent ecosystem.

Subcategories:

- **Security Goal Awareness:** Understanding security objectives and threat landscapes to effectively integrate AI into security workflows using CAI's specialized agents

- **Security Platform Awareness:** Knowledge of CAI's agent capabilities, tool integrations, and appropriate use cases for different security scenarios

- **Security Task Delegation:** Optimal assignment of security tasks between human expertise and CAI's agent capabilities, including selection of appropriate agentic patterns

Description - Effective Communication with AI for Security Tasks *Description* encompasses skills needed to communicate security requirements, constraints, and objectives to AI systems, including crafting prompts that guide CAI agents toward producing useful security-related outputs.

Subcategories:

- **Security Product Description:** Articulating desired security outcomes and characteristics to CAI agents through effective prompting and configuration

- **Security Process Description:** Engaging in iterative dialogue with CAI agents for complex security analysis and investigation using the ReAct framework

- **Security Performance Description:** Defining how CAI agents should behave in security-critical scenarios and user interactions through proper configuration of agentic patterns

Discernment - Critical Evaluation of AI Security Outputs *Discernment* involves critically evaluating AI-generated security outputs, understanding their quality, relevance, potential biases, and security implications. This is crucial for maintaining security integrity when using CAI's capabilities.

Subcategories:

- **Security Product Discernment:** Evaluating the quality and security relevance of CAI-generated security analysis and recommendations

- **Security Process Discernment:** Assessing the effectiveness of Human-AI collaboration in security contexts using CAI's tracing and monitoring capabilities

- **Security Performance Discernment:** Evaluating CAI systems' effectiveness in independent security operations and adjusting configurations accordingly

Diligence - Ethical and Responsible AI Use in Security *Diligence* refers to responsible use of AI in cybersecurity, including ethical considerations, transparency, and accountability for security decisions made with CAI assistance.

Subcategories:

- **Security Creation Diligence:** Responsible use of CAI tools while maintaining ethical security practices and awareness of potential misuse

- **Security Transparency Diligence:** Clear communication about CAI involvement in security assessments and decisions, leveraging CAI's built-in tracing capabilities

- **Security Deployment Diligence:** Taking responsibility for CAI-assisted security outputs, including thorough validation and risk assessment using CAI's validation tools

1.6.4 Framework Integration with CAI Architecture

This adapted framework provides the theoretical foundation for CAI Fluency's educational approach, ensuring that cybersecurity practitioners develop not just technical skills, but also the critical thinking and ethical awareness necessary for responsible AI use in security contexts.

The framework's integration with CAI's technical architecture becomes evident in several key areas:

- **ReAct Implementation:** The framework's emphasis on iterative reasoning-action cycles aligns perfectly with CAI's ReAct-based architecture (Section 2.2)

- **Multi-Agent Coordination:** The three modalities map directly to CAI's single-agent, multi-agent, and autonomous agentic patterns (Section 6)

- **Human-In-The-Loop Integration:** The framework's focus on human-AI collaboration supports CAI's HITL capabilities (Section 6.7)

- **Educational Scaffolding:** The competency framework provides a structured approach to learning CAI's capabilities progressively
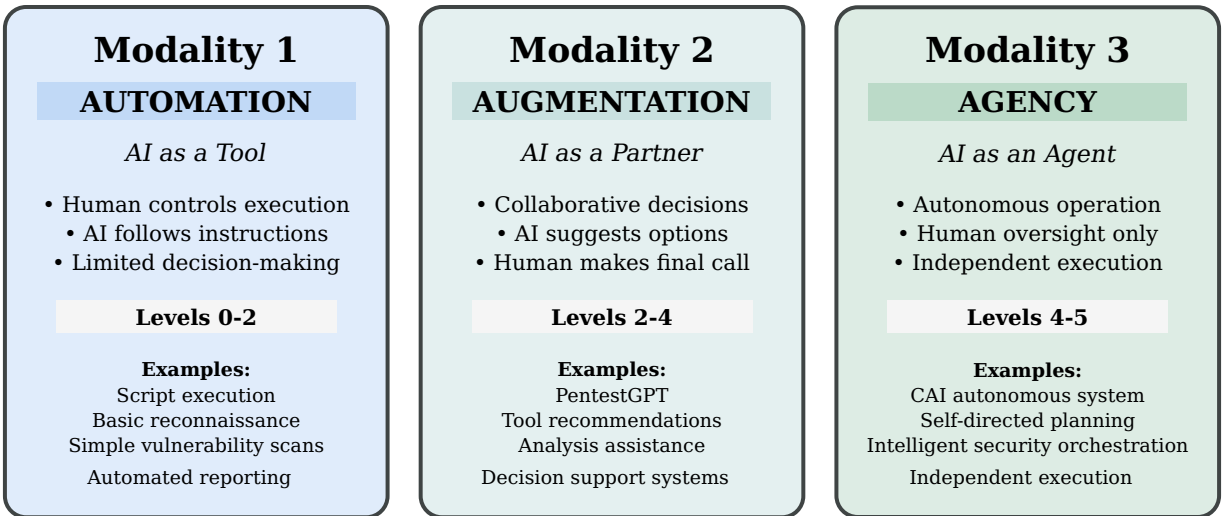
As we proceed through the technical sections of this document, we will repeatedly reference how these AI fluency competencies apply to specific CAI implementations, providing practitioners with both theoretical understanding and practical guidance for effective cybersecurity AI deployment.

1.7 Framework for Cybersecurity AI Fluency

Building upon the foundational AI Fluency framework, we now present a novel **Framework for Cybersecurity AI Fluency** that specifically addresses the unique challenges and opportunities of AI integration in cybersecurity contexts. This framework synthesizes insights from the Dakan-Feller AI Fluency model with the cybersecurity automation taxonomy developed by Mayoral-Vilches [2], creating a comprehensive educational and operational framework tailored for security practitioners.

1.7.1 Cybersecurity AI Automation Levels

Drawing from robotics principles and cybersecurity-specific requirements, we adopt and extend the 6-level taxonomy (Level 0-5) that distinguishes automation from autonomy in Cybersecurity AI [2]. This taxonomy provides a crucial foundation for understanding the current capabilities and limitations of AI systems in security contexts:

Modality 1	Modality 2	Modality 3
AUTOMATION	**AUGMENTATION**	**AGENCY**
AI as a Tool	*AI as a Partner*	*AI as an Agent*
• Human controls execution • AI follows instructions • Limited decision-making	• Collaborative decisions • AI suggests options • Human makes final call	• Autonomous operation • Human oversight only • Independent execution
Levels 0-2	**Levels 2-4**	**Levels 4-5**
Examples: Script execution Basic reconnaissance Simple vulnerability scans Automated reporting	**Examples:** PentestGPT Tool recommendations Analysis assistance Decision support systems	**Examples:** CAI autonomous system Self-directed planning Intelligent security orchestration Independent execution

← **Human Control**   **Increasing AI Autonomy** →

Figure 2: The three modalities of Human-AI interaction in cybersecurity contexts, showing the progression from automation through augmentation to agency, aligned with CAI automation levels.

Level	Autonomy Type	Plan	Scan	Exploit	Mitigate	
0	No tools	✗	✗	✗	✗	*Impossible in practice*
1	Manual	✗	✗	✗	✗	Metasploit [6], MulVAL [44]
2	LLM-Assisted	✓	✗	✗	✗	PentestGPT [7]
3	Semi-automated	✓	✓	✓	✗	AutoPT [8], Vulnbot [9]
4	Cybersecurity AIs	✓	✓	✓	✓	**CAI** [1]
5	Autonomous	✓	✓	✓	✓	*Aspirational*

Table 2: The autonomy levels in cybersecurity, adapted from [1] and SAE J3016 [10] driving automation levels. I classify cybersecurity autonomy from Level 0 (no tools) to Level 5 (full autonomy). Table outlines capabilities each level allows a system to perform autonomously: Planning (strategizing actions to test/secure systems), Scanning (detecting vulnerabilities), Exploiting (utilizing vulnerabilities), and Mitigating (applying countermeasures).

Level 0: **No Automation** Traditional manual cybersecurity operations with no AI assistance. Human operators perform all security tasks including monitoring, analysis, and response using conventional tools and methodologies.

Level 1: **Assistance** AI provides basic assistance to human operators, such as automated alert filtering, simple pattern recognition, or basic log aggregation. The human maintains full control and decision-making authority.

Level 2: **Partial Automation** AI systems can perform specific security tasks independently under human supervision, such as automated vulnerability scanning, basic threat detection, or routine compliance checking. Human oversight is required for validation and action approval.

Level 3: Conditional Automation AI systems can execute complex security workflows autonomously within defined parameters, such as automated incident triage, threat hunting, or vulnerability assessment. Human intervention is required for edge cases and strategic decisions.

Level 4: High Automation AI systems operate with significant independence in security operations, handling complex scenarios and making tactical decisions. Human oversight focuses on strategic guidance and exception handling.

Level 5: Full Autonomy Theoretical level where AI systems operate completely independently in cybersecurity contexts. This remains aspirational and raises significant concerns about accountability and control in security-critical environments.

1.7.2 Novel Cybersecurity AI Fluency Modalities

Inspired by the original framework but adapted for cybersecurity contexts and aligned with automation levels, we propose three specialized modalities that reflect the unique requirements of security operations:

Figure 3: The relationship between cybersecurity AI automation levels (0-5) and the three modalities of interaction, showing how different levels of automation align with different interaction patterns in CAI.

Modality I: Supervised Security Automation (Levels 0-2) This modality encompasses human-supervised AI operations where security practitioners maintain direct control over AI systems. It corresponds to CAI's single-agent patterns and basic automation capabilities:

- **Guided Reconnaissance:** AI-assisted information gathering with human validation at each step

- **Supervised Scanning:** Automated vulnerability assessments with human interpretation of results

- **Assisted Analysis:** AI-powered log analysis and pattern recognition with human oversight

- **Directed Response:** Human-initiated automated responses to common security events

Modality II: Collaborative Security Intelligence (Levels 2-4) This advanced modality involves dynamic Human-AI collaboration where both parties contribute expertise to complex security challenges. It aligns with CAI's multi-agent coordination and swarm patterns:

- **Adaptive Threat Hunting:** AI and human expertise combined to identify sophisticated threats through iterative hypothesis testing

- **Intelligent Incident Response:** Coordinated human-AI teams managing complex security incidents with distributed decision-making

- **Strategic Vulnerability Management:** AI-driven prioritization combined with human strategic assessment of organizational risk

- **Collaborative Red Teaming:** Human creativity augmented by AI's systematic exploration of attack vectors

Modality III: Autonomous Security Orchestration (Levels 4-5) This modality represents the most advanced form of cybersecurity AI deployment, where AI systems operate with significant independence while maintaining human oversight for strategic decisions:

- **Autonomous Defense Systems:** AI systems that independently detect, analyze, and respond to security threats within defined parameters

- **Adaptive Security Posture:** AI systems that automatically adjust security configurations based on evolving threat landscapes

- **Intelligent Security Training:** AI tutors and simulation environments that adapt to learner needs and emerging threat scenarios

- **Predictive Security Operations:** AI systems that anticipate and prepare for potential security events before they occur

1.7.3 Adapted Core Competencies: The "4 C's" of Cybersecurity AI Fluency

While maintaining the structural integrity of the original framework, we introduce cybersecurity-specific adaptations of the core competencies, renamed as the "4 C's" to reflect their security-focused nature:

Command - Strategic AI Deployment in Security Operations *Command* encompasses the ability to strategically deploy and direct AI systems within cybersecurity contexts, understanding when and how different automation levels are appropriate for specific security challenges.
Subcategories:

- **Threat-Aware Goal Setting:** Establishing security objectives that leverage appropriate AI automation levels based on threat assessment and organizational risk tolerance

- **Security Technology Mastery:** Deep understanding of cybersecurity AI tools, their capabilities, limitations, and appropriate deployment contexts across automation levels

- **Tactical Resource Allocation:** Optimal distribution of human and AI resources across security operations, considering automation capabilities and human oversight requirements
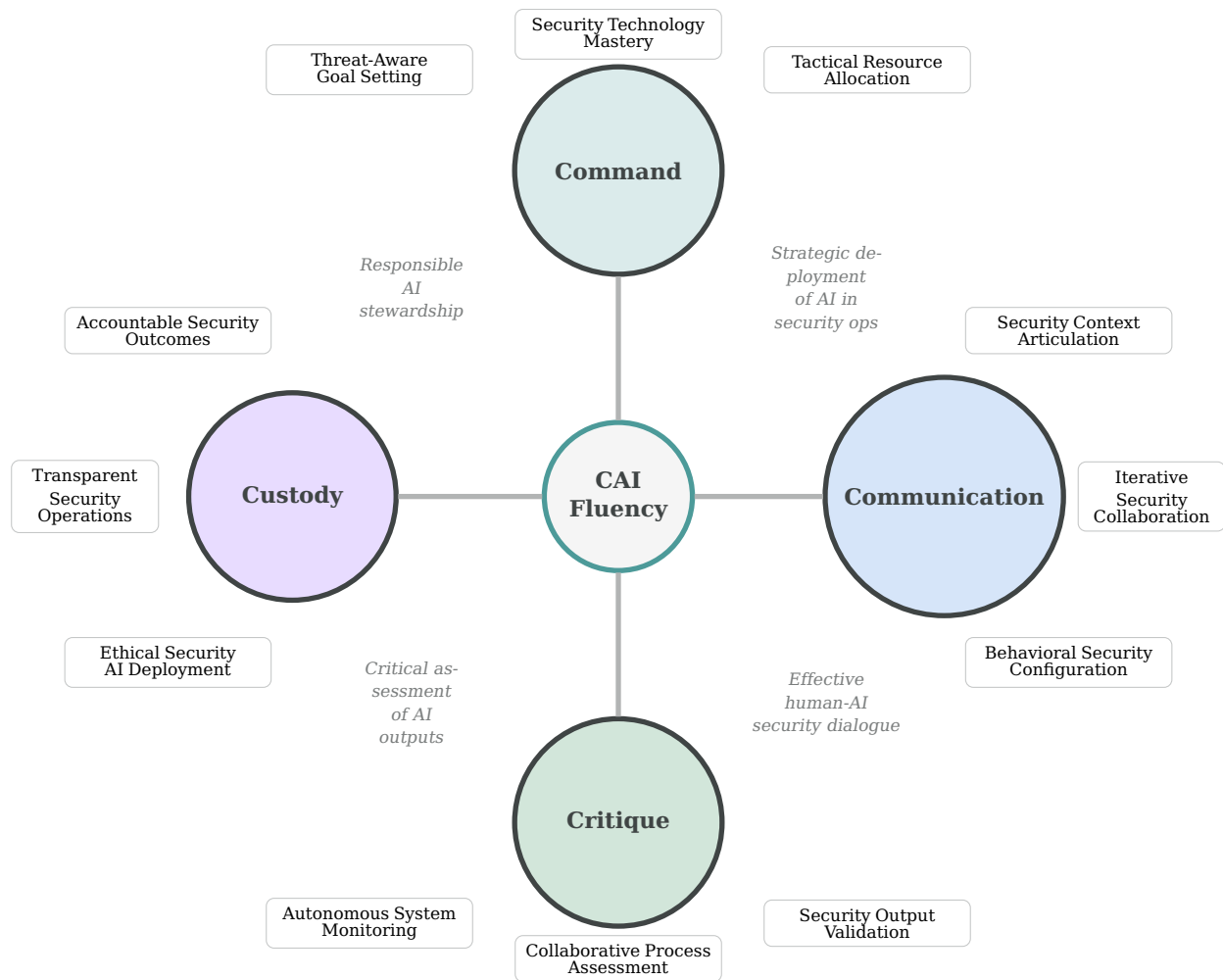
Figure 4: The 4 C's framework for Cybersecurity AI Fluency, showing the four core competencies (Command, Communication, Critique, and Custody) with their respective subcategories tailored for security practitioners.

Communication - Effective Human-AI Security Dialogue *Communication* involves the specialized skills needed to interact effectively with AI systems in security contexts, including the ability to translate complex security requirements into AI-understandable instructions.

Subcategories:

- **Security Context Articulation:** Clearly communicating threat landscapes, security requirements, and operational constraints to AI systems

- **Iterative Security Collaboration:** Engaging in dynamic dialogue with AI systems for complex security analysis, investigation, and response planning

- **Behavioral Security Configuration:** Defining appropriate AI behaviors for different security scenarios and automation levels

Critique - Critical Assessment of AI Security Outputs *Critique* involves the sophisticated evaluation of AI-generated security outputs, understanding their implications, limitations, and potential security risks or benefits.

Subcategories:

- **Security Output Validation:** Rigorous evaluation of AI-generated security analysis, recommendations, and automated actions for accuracy and completeness

- **Collaborative Process Assessment:** Evaluating the effectiveness of human-AI security workflows and identifying optimization opportunities

- **Autonomous System Monitoring:** Continuous assessment of AI system performance in security operations, including detection of drift or degradation

Custody - Responsible Stewardship of AI Security Systems *Custody* represents the highest level of responsibility in cybersecurity AI deployment, encompassing ethical use, accountability, and long-term stewardship of AI systems in security contexts.

Subcategories:

- **Ethical Security AI Deployment:** Ensuring AI systems are used responsibly in security contexts, avoiding potential for misuse or harm

- **Transparent Security Operations:** Maintaining clear documentation and communication about AI involvement in security decisions and actions

- **Accountable Security Outcomes:** Taking full responsibility for the consequences of AI-assisted security operations, including thorough validation and risk management

1.7.4 Integration with CAI Architecture and Pedagogy

This Framework for Cybersecurity AI Fluency provides the theoretical foundation for CAI Fluency's educational methodology, ensuring that security practitioners develop competencies aligned with the realities of modern cybersecurity AI deployment. The framework's integration with CAI's technical architecture manifests in several key areas:

- **Automation Level Awareness:** Educational content is structured around the 6-level taxonomy, helping practitioners understand current capabilities and limitations

- **Modality-Specific Training:** Different learning paths for supervised automation, collaborative intelligence, and autonomous orchestration scenarios

- **Competency-Based Assessment:** Evaluation frameworks based on the 4 C's, ensuring comprehensive skill development

- **Progressive Complexity:** Learning pathways that advance through automation levels as competencies develop

This framework serves as the pedagogical backbone for the practical implementations and case studies presented in subsequent sections, providing both theoretical grounding and practical guidance for effective cybersecurity AI deployment.

1.8 How to Read this Document

For clarity and ease of use, this document is structured into 9 chapters that are categorized as either theoretical or practical. This distinction is intended to facilitate a more accessible reading experience for academia and practitioners, allowing each audience to navigate seamlessly between conceptual foundations and hands-on applications according to their specific interests and needs.

To facilitate easy navigation, the respective chapters are either marked with a † for theorerical Chapters and/or * for practical chapters.

1.8.1   Relevant Chapters for Academia and Technicians

The following chapters contain the foundational aspects and technical niceities of the CAI framework.

- **1 Introduction and Motivation** $^{\dagger *}$
- **2 Preliminaries: Foundational Concepts in CAI** †
- **3 Computational Models of Language from Theoretical Computer Science** †
    - This chapter can be skipped for a fast-lane theoretical introduction.
- **4 Neural Models: A Subset of Statistical Language Models** †
    - This chapter can be skipped for a fast-lane theoretical introduction.
- **5 The Evolution of Language Models: From Pure Generation to Agentic Interaction** †
- **6 CAI Architecture** $^{\dagger *}$

1.8.2   Relevant Chapters for Practitiones

The following chapters contain the practical and hands-on aspects of the CAI framework.

- **1 Introduction and Motivation** $^{\dagger *}$
- **6 CAI Architecture** $^{\dagger *}$
- **7 Getting Started** *
- **8 Quickstart, CAI Commands and Use** *
- **9 Development** *

A Note to the Readers

We encourage practitioners and academia to read CAI's the technical report at https://arxiv.org/pdf/2504.06017. Moreover, please note that CAI is in active development, so do not expect it work flawlessly. Instead, contribute by raising an issue or sending a pull request

Contents

List of Figures

List of Tables

2 Preliminaries: Foundational Concepts in CAI [†]

2.1 Formal vs. Probabilistic Language Models, LLMs and ReAct Agents

To understand the core functionalities of CAI, its limitations and use, you need to get familiar with its components and their foundations. The first important concept in this context is the ReAct Framework.

2.2 Motivation: CAI is a ReAct Framework

CAI generally implements the ReACT agent model [11]. ReAct stands for Reasoning and Acting, and it represent a powerful architecture where a Large Language Model (LLM) performs reasoning steps in natural language, generates code snippets or actions, and then executes them in an external environment (e.g. a Python interpreter or an API call). The results of the execution are then fed back into the LLM for further reasoning. The process is illustrated in the picture below.

Reasoning Cycle

Figure 5: Conceptual Drawing: Key Steps in ReAct agent models.

Key Steps in ReAct

1. **Reasoning step:** The LLM analyzes the problem using natural language.

2. **Action generation:** The LLM generates a code snippet (e.g., a function or API call).

3. **Execution:** The code is executed in a sandboxed environment.

4. **Observation:** The result of the execution is returned to the LLM.

5. **Next step:** The LLM refines its plan based on the new information.

We will explain the details of the ReAct framework at a later stage. For now we focus on the fundamental tension the ReAct framework bridges: LLMs are *probabilistic* models of natural language, while code must adhere to the rules of a *formal* language.

- Programming languages such as Python, JavaScript, or C++ are formal systems: they have strictly defined syntactic and semantic rules. A program is either valid or invalid, and small errors can result in total failure.

- In contrast, LLMs like GPT are probabilistic models trained on natural and programming language corpora. They learn to approximate what "correct" output should look like based on frequency and co-occurrence patterns, rather than internalizing grammar rules. They do not know the Python grammar – they have learned to simulate Python-like code based on what they have seen.

The ReAct loop operates right at the intersection of these two formalisms, feeding LLM output (i.e. Code) into external environments and reasoning based on the received output in a closed feedback loop.

2.3 Bridging Formal and Probabilistic Language Models in ReAct Frameworks

To understand why code generated by large language models (LLMs) is often syntactically correct but semantically flawed, we need to recognize a fundamental mismatch between two kinds of language modeling.

2.3.1 LLM-Generated Code Often Needs Revision: The Mismatch Between Language Models

Formal vs. Statistical Language Models. A programming language is a *formally defined language*. It has a precise syntax and semantics governed by strict rules (often defined via context-free grammars or stricter variants).

LLMs are probabilistic models of natural language, while code must adhere to the rules of a formal language.

A compiler or interpreter expects exact compliance with those rules and will reject any deviation – no matter how small. An LLM, by contrast, is a *statistical (probabilistic) language model*. It learns from examples, not from formal rules. It doesn't "know" syntax or semantics in the symbolic sense; rather, it has learned to approximate plausible-looking code based on patterns in the training data.

This leads to a structural mismatch:

- The programming language expects precision. It requires complete syntactic and semantic correctness.

- The LLM produces probability-weighted guesses, even for code – predictions that are often right, but not guaranteed to be.

Formal vs. Probabilistic Language Models As a result, LLM-generated code may:

- Be syntactically valid, but semantically incorrect (wrong logic, misuse of an API)

- Include placeholder variables or inconsistent naming

- Mix styles or make invalid assumptions about types, scopes, or libraries

Thus, LLMs generated code that "looks right", while not "being right".

Aspect	Formal Language Model (e.g., programming languages)	Probabilistic Language Model (e.g., LLMs)
Defined by	*Symbolic grammar (e.g., context-free)*	*Statistical patterns in data modelled by statistical model*
Output validity	*Must be exact (strict syntax and semantics)*	*Approximate; best-guess predictions*
Error handling	*Hard failure on rule violation*	*Soft failure or degradation of quality*
Interpretation	*Deterministic (parser, compiler)*	*Contextual, probabilistic.*
Learning method	*Hand-crafted rules and specifications*	*Data-driven learning from corpora.*

Table 3: Formal vs. Probabilistic Language Models.

3 Computational Models of Language from Theoretical Computer Science [†]

3.1 Formal Models of Language

A language is a collection of sentences of finite length all constructed from a finite alphabet (or, where our concern is limited to syntax, a finite vocabulary) of symbols, cf. [45]. Accordingly, a *grammar* is a "machine" that enumerates the sentences of a language.[3] In the study of language from a computational perspective, formal models, such as grammars, serve as essential tools for describing and analyzing the structure and behavior of languages – both artificial (e.g., programming languages) and natural. These models stem from theoretical computer science and formal language theory, and they provide mathematical frameworks for specifying which sequences of symbols are valid within a given language.

The motivation behind these models is twofold:

1. **Descriptive:** To formally characterize linguistic phenomena such as syntax, structure, and grammaticality.

2. **Computational:** To develop algorithms that can recognize, parse, or generate valid linguistic expressions efficiently. One of the foundational classes of formal models in this context is that of finite automata, which are used to define and analyze regular languages – the simplest class of formal languages.

3.1.1 Finite Automata and Regular Languages

Finite automata are abstract computational machines designed to recognize patterns in strings. They consist of a finite set of states, a transition function, an initial state, and one or more accepting states. When processing a string of symbols, the automaton transitions between states based on the current input symbol and its transition function. If the automaton ends in an accepting state after processing the entire input, the string is considered accepted by the automaton.

Regular Languages   A language is classified as a *regular language* if it can be recognized by some finite automaton or equivalently described by a regular expression. *Regular expressions* are symbolic notations that define search patterns and are widely used in programming and text processing.

---

[3]Informally, one can think of languages as a set of strings, and a grammar as a set of rules how to generate all those strings.

Regular languages are powerful enough to capture many simple and repetitive patterns, such as:

- Strings that begin or end with certain characters

- Strings that contain repeated subsequences

- Sets of strings with constrained lengths or character sequences

Formally, regular languages are *closed* under operations such as union, concatenation, and Kleene star (repetition). This makes them robust and mathematically tractable.

Limits of Regular Languages   Despite the aforementioned capabilities, finite automata have significant limitations. They lack memory beyond their current state, which means they cannot handle nested or recursively structured patterns. For instance, they cannot recognize balanced parentheses or match long-range dependencies – phenomena commonly found in natural and programming languages.
   Therefore:

- Finite automata can decide whether a string belongs to a regular language, but they cannot express hierarchical or recursive structures.

- This leads to the realization that not all languages of interest, particularly natural languages, are regular.

As a result, more expressive models, such as context-free grammars, are required to capture the complexity of such languages.

3.1.2   Context-Free Grammars (CFGs)

Context-free grammars consist of a vocabulary and a set of *production rules* (also called substitution rules). Their goal is to generate all valid strings (i.e., symbol sequences) of a language using these rules. A *production rule* takes the form $A \rightarrow w$ meaning that an occurrence of A in a string can be replaced by w. Here, both A and w are symbols drawn from the vocabulary of the grammar.

A context-free grammar G is formally defined by the set of it production rules. In this context:

- A *derivation* is a sequence of production rule applications from G.

- A *string* (composed only of terminal symbols) is derivable from G if such a derivation exists.

- The *language $L(G)$* generated by the grammar is the set of all strings derivable from the start symbol S using the rules in G.

- Languages generated by context-free grammars are known as *context-free languages*.

Context-free languages strictly include the class of regular languages –i.e., those defined by finite automata – thus forming a proper superset of them.

Iteration vs. Recursion   Finite automata make use of iteration: they loop arbitrarily many times, processing repeated patterns of symbols. In contrast, context-free grammars rely on recursion: a production rule may refer to the same symbol it defines. This recursive structure allows for the deep nesting of constructs and enables rules to relate elements that may be arbitrarily far apart in a string.

Properties of Context-Free Grammars

A grammar is termed context-free if all its rules are formulated independently of the context in which they are applied.This allows for a well-defined *parsing* process, in which:

- One can determine whether a string adheres to the grammar, and

- A syntax tree (also called a parse tree) representing the structure of the string can be constructed.

A program that performs this analysis is called a *parser*. Parsers are widely used in the processing of programming languages. In computational linguistics, there are ongoing efforts to describe natural language using context-free grammars.

Beyond Finite Automata: Structural Interpretation Unlike finite automata, context-free grammars do more than define the set of valid expressions in a language – they also implicitly assign structure to these expressions. They associate derivation trees (also known as parse trees or syntax trees) with sentences of the language. This process, known as parsing, represents the automatic syntactic analysis of sentences

3.1.3   Context-Sensitive Languages and Their Role in Natural Language Processing

Figure 6: The Chromsky Hierarchy of formal languages.

Figure 17 depicts Context-Sensitive Languages as a *Type 1-language* in the Chomsky hierarchy. In formal language theory, Noam Chomsky [45] proposed a hierarchy of grammars that classifies languages based on the computational complexity required to generate them:

1. **Type 3** – Regular languages, generated by finite automata

2. **Type 2** – Context-free languages (CFLs), generated by pushdown automata

3. **Type 1** – Context-sensitive languages (CSLs), generated by linear bounded automata

4. **Type 0** – Recursively enumerable languages, generated by Turing machines

Context-sensitive languages (Type 1) occupy a crucial position in this hierarchy: they are strictly more powerful than context-free languages, but still more constrained than unrestricted languages.

A context-sensitive grammar allows production rules of the form: $\alpha\ A\ \beta \rightarrow \alpha\gamma\beta$ where A is a non-terminal, α, β are strings (possibly empty), and γ is a non-empty string. The application of the rule depends on the context provided by α, β.

Context Sensitivity in Natural Language: An Example

Consider the phenomenon of subject-verb agreement in English in the two senteces *"The dog barks"* and *"The dogs bark"*. In this case, the verb form must agree in number with the subject. This dependency can be managed by a context-free grammar with some effort. However, natural language may exhibit more complex dependencies that require context-sensitive mechanisms.

Example: Cross-serial Dependencies in Swiss German   A classic illustration (see [46]) comes from Swiss German, where verb order and noun dependencies are interleaved in a way that cannot be captured

Some natrural language characteristics phenomena exceed the expressiveness of regular & context-free languages.

by context-free grammars. Consider the sentence: *"... dass mer d'chind em Hans es huus händ wele laa hälfe aastriiche"* (... that we the children have wanted to let Hans help paint the house). Here, each noun (object) must correspond with a specific verb, and the dependencies cross each other in a non-nested fashion. This structure cannot be represented by a context-free grammar and instead requires the additional expressive power of a context-sensitive grammar. This example shows that natural languages are not fully context-free and often require context-sensitive grammatical frameworks to be accurately modeled.

3.2   Computational Models for Context-Sensitive Languages

To process context-sensitive languages computationally, we need models more powerful than pushdown automata. The appropriate model is the linear bounded automaton (LBA) - a restricted form of Turing machine where the tape is limited to a length proportional to the input. LBAs can recognize context-sensitive languages, but they are computationally expensive and less practical for large-scale parsing tasks.

Thus, natural language processing (NLP) systems rarely use true context-sensitive grammars due to efficiency concerns. Instead, they often rely on approximations (e.g., mildly context-sensitive grammars like Tree Adjoining Grammars or Combinatory Categorial Grammars) or on statistical (aka. probabilistic) models.

Conclusion

Natural languages may display context-sensitive characteristics that go beyond the expressive power of regular and context-free grammars. These characteristics are reflected in long-distance dependencies, agreement phenomena, and structural ambiguities.

From the perspective of the Chomsky hierarchy:

- Finite automata and context-free grammars are insufficient to fully model natural language.

- Context-sensitive grammars offer the necessary expressive power, but at the cost of computational tractability.

Modern approaches, such as LLMs, offer an alternative by learning context-sensitive patterns *implicitly* through data-driven methods, providing practical but approximate solutions to the problem of natural language understanding.

3.3 Models for Natural Language

In this Chapter, we will ealobarte the details on and contrast the relevant computational language modeling paradigms. See [47] for further reading.

3.3.1 From Formal Grammars to Probabilistic Language Models

Large Language Models (LLMs), such as ChatGPT, represent a different paradigm for modeling language. They do not follow the strict generative frameworks of formal grammars, but instead:

- Learn probabilistic relationships between tokens based on vast amounts of real-world text.

- Capture context-sensitive behavior implicitly through attention mechanisms and deep neural networks.

- Do not explicitly enforce syntactic or semantic constraints but instead learn to approximate them statistically.

This makes LLMs highly effective for natural language, which is full of contextual subtleties. However, their probabilistic nature makes them less suited to handling formal languages (such as programming code), where precise syntax and unambiguous interpretation are crucial.

Thus, LLMs *simulate grammaticality rather than guaranteeing it.* Their strength lies in pattern generalization, not formal rule enforcement.

3.3.2 Language Models: A Probabilistic Perspective

From a formal standpoint, a *language* model is a probabilistic model [47] that estimates the likelihood of a sequence of words. Given a word sequence $W = w_1, w_2, \ldots, w_n$, the task of the language model is to model or estimate

$$P(W) = P(w_1, w_2, \ldots, w_n)$$

However, due to the sparse data problem, we cannot reliably estimate probabilities for long sequences directly from data. Therefore, earlier language models from natural language modelling (e.g., n-gram models) make simplifying assumptions to decompose the probability:

$$P(W) \approx P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_2) \cdot \ldots \cdot P(w_n \mid w_{n-1})$$

This *Markov approximation* assumes that each word depends only on the previous word (bigram) or a limited context (trigram, etc.). While computationally efficient, these models have limited ability to capture long-range dependencies or hierarchical structures – hallmarks of natural language.

3.3.3 Mathematical Definition as Stochastic Processes

Language models represent sequences – such as sentences – as ordered chains of elements (e.g., characters or words). In stochastic language models, these elements are treated as random variables

$X_1, X_2, \ldots$, forming a discrete-time stochastic process.

To enable the same model to handle sequences of varying length n, the beginning and end of the sequence are typically marked by two additional random variables, X_0 and X_{n+1}, which take on a special value – often denoted by a symbol such as $\bot$ or $<s>$ representing a start or end token. The probability of a specific sequence $w_1, \ldots, w_n$ can then be expressed as the *joint probability* of all elements in the sequence, including the start and end markers:

$$P\left(X_0 = \bot \land X_1 = w_1 \land \cdots \land X_n = w_n \land X_{n+1} = \bot\right).$$

A common shorthand notation for this is $P\left(\bot, w_1, \ldots, w_n, \bot\right)$. According to the law of total probability, this joint probability can be decomposed into a product of conditional probabilities:

$$
\begin{aligned}
P\left(\bot, w_1, \ldots, w_n, \bot\right) = \\
P\left(X_0 = \bot\right) \\
\cdot P\left(X_1 = w_1 \mid X_0 = \bot\right) \\
\cdot P\left(X_2 = w_2 \mid X_0 = \bot, X_1 = w_1\right) \\
\cdots \\
\cdot P\left(X_n = w_n \mid X_0 = \bot, X_1 = w_1, \ldots, X_{n-1} = w_{n-1}\right) \\
\cdot P\left(X_{n+1} = \bot \mid X_0 = \bot, X_1 = w_1, \ldots, X_n = w_n\right).
\end{aligned}
$$

Or more concisely:

$$P\left(\bot, w_1, \ldots, w_n, \bot\right) = P\left(\bot\right) \cdot P\left(w_1 \mid \bot\right) \cdot P\left(w_2 \mid \bot, w_1\right) \cdots P\left(w_n \mid \bot, w_1, \ldots, w_{n-1}\right) \cdot P\left(\bot \mid \bot, w_1, \ldots, w_n\right).$$

This formulation reflects the fundamental approach of probabilistic language models: they estimate the likelihood of an entire sequence by chaining together conditional probabilities that represent the model's learned expectations of which elements are likely to follow others, given the preceding context.

3.3.4 Enter Large Language Models (LLMs)

LLMs, such as GPT, extend the probabilistic modeling paradigm by leveraging deep learning architectures, particularly transformers, which use self-attention mechanisms to model dependencies across entire sequences –regardless of their length.

Unlike traditional probabilistic language models, such as n-grams, LLMs:

  - Estimate $P\left(W\right)$ using *neural representations* of language learned from massive corpora.

  - Do not rely on hand-crafted rules or fixed context windows.

  - Use distributed representations (embeddings) and training objectives such as next-token prediction to generalize from observed patterns.

LLMs thus implicitly learn context-sensitive behavior, not by encoding grammars, but by observing countless examples where such behavior is evident. They generalize from statistical regularities across the data.

Conclusion   Large language models represent a fundamental shift in how we model linguistic knowledge:

  • They do not correspond to any one level of the Chomsky hierarchy, since they are not symbolic processors.

- They do not store or apply explicit grammar rules, yet they can produce grammatically plausible output, often consistent with context-sensitive grammars.

- Their success comes from learning from *data*, not from *formal derivations*.

In summary, language are not formal language processors in the classical sense, but the example of LLMs proves their effectivity at approximating even highly structured linguistic phenomena – simply because they have seen enough examples to learn their statistical signatures.

4 Neural Models: A Subset of Statistical Language Models [†]

Neural networks, also known as *artificial neural networks (ANN)* are powerful tools to learn and extract patterns from data. The design of neural networks is inspired by the functioning of the human brain, or, more exactly, the interaction of neurons, see [48]. Figure 7 is a schematic illustration of a natural neuron.

Figure 7: Schematic Drawing of a Natural Neuron.

- A *Neuron* (Figure 7) is a nervous cell, that is connected to other neurons, passing electric signals to connected neurons. The brain consists of Billions of Neurons, making up our consciousness.

- The *Axon* of a neuron sends a signal to the Dendrit of another Neuron.

- An Axon has several *terminals*.

- Axon terminals and Dendrits are connected via *synapses* (Figure 8).

- Synapses can amplify or damp signals. If an incomming signal is sensed, the receiving neuron itself might fires a signal (electrical activity).

When do neurons fire? The *activation* of a neuron depends on several parameters:

1. the number of signals received

2. the type of synapses (damping or amplfying)

3. randomness – activations need not be dertministic.

4.1 Basic Structure of an Artifical Neuron – the Perceptron

The simplest neural model – the *perceptron* – imitates the functioning of a single neural cell, cf. [48]. The structure is depicted in Figure 11. Similar to the neuron that aggregates and processes signal

Figure 8: Synapses connect two or more neurons.

(a) Natural Neuron   (b) Perceptron

Figure 9: Artificial Neurons – Perceptrons (b) – mimick the structure and behaviour of natural neural cells (a).

inputs from the dendrites in the nucleolus, before passing them on to other neurons in the axon terminals, the perceptron processes the input features $x_1, ..., x_n$ from left to right, producing an output y_j.

- **Inputs:** x_1 to x_n represent the input features of the perceptron. These can be various types of observations: Images, Audio signals, Text (e.g., words or characters).

- **Output:** y_j is the output predicted by the perceptron. Often, the output is related to a classification problem (e.g., in a binary classification task). In this case, the output is a class label of a set of predefined classes.

- **Parameters:** Weights: the weights w_{ij} and bias b_j (see Figure 12) are the *learnable parameters* of the model. They correspond to the synapses, cf. figure 8, damping and aplifying the input signals. These are the only unknowns in a basic neural model. They are learned through *supervised learning*.

- **Node Computation:** Each unit x computes the weighted sum of its inputs.

- **Activation Function:** The activation function applies a transformation to x, often mapping it to a limited range such as $[0, 1]$ (e.g., via a sigmoid function or via thresholding).

Figure 10: Perceprtons consist of input features $x_1, ..., x_n$, weights $w_{1j}, ..., w_{nj}$, an aggregation node that aggregates the input signals to x, an activation function $f(x)$ and the output y_j.

Figure 11: Image of a sigmoid neuron activation function f which maps the aggregates input activation x to a value in a limited range, here $[0, 1]$. In this example, due to the mapping to $[0, 1]$ outputs y_j can be interpreted as probabilities of class labels.

4.2 Supervised Learning

In *supervised learning*, labeled instances x_1 to x_n and their corresponding outputs y_j are used as training samples. In an interative learning process, the weights of the perceptron are to adjusted to fit the data. The aim is to modify the parameter in a way that the error – the *loss* – between true outputs and predicted output is as small as possible, cf. Figure 12.

In supervised learning, the model receives feedback based on whether its predictions are correct.

- **Correct Prediction:** No update needed; the weights remain unchanged.

- **Incorrect Prediction:** At least one weight must be updated to reduce error.

The updates are computed using *gradient descent*. For a complete mathematical description, we refer the interested reader to [49]

Figure 12: Depiction of the weights and bias update in supervised perceptron learning.

4.3 Neural Networks with Hidden Layers

Note that *artificial neural networks* (ANNs) may contain multiple neurons in the output layer, particularly when handling multi-class classification. Networks may also include *hidden* layers, allowing them to model more complex patterns, see Figure 13.

Figure 13: Multi-layer Perceptron Neural Network with hidden layers and two outputs.

Example: Log Analysis

Let us assume we want to train a neural network for anomaly detection, in order to detect cyberattacks (e.g. DDoS) using the log traces. For simplicity assume the log trances are of fixed length.

- **Input Layer:** The input training instances $x_1, ..., x_n$ represent the logs traces in form of words/tokens. For simplicity, let us assume we have one neuron for each word/token in the vocabulary.

- **Output Layer:** The output y_j represents the set of cyberattacks, or, in case of binary classification, whether or not a cyberattack occured. The respective output neuron indicates whether a particular attack is detected (1 = yes, 0 = no). The value is a real number between 0 and 1, representing the probability that a certain attack has occured.

4.4   Supervised Training of ANNs – the Process

In supervised learning of artificial neural networks, the training process consists of two steps: *forward propagation* and *backpropagation* [50].

Forward Propagation:

1. A single or multiple training instances are fed into the network.

2. Each neuron computes its output based on current weights.

3. The signal passes forward through the layers until it reaches the output.

4. The model's predicted output is compared to the true label.

Figure 14: Multi-layer Perceptron – the signal (green – activated, white – not activated) passes though the network.

At this point we, compute the difference between the true vs. predicted output labels and distinguish two cases:

Case 1: *Correct Prediction*

→ No changes are needed. We move to the next training instance.

Case 2: *Incorrect Prediction*

→ The parameters need to be changed. Note however, that, when a single neuron modifies one of its parameters, subsequent neurons may also need to adjust their connection weights according to the learning equation. "Hence, we need to solve a set of the mutually coupled learning equations." [51] This coupled learning problem is solved using *Backpropagation*, a stochastic descent method [52]:

- The error is propagated backwards through the network, from the output to the input layer.
- The weights are updated using gradient descent, based on the computed gradients of the loss function.

Once the input layer is reached, the weights have been adjusted. The next instance can then be processed using the updated parameters.

4.5 The Loss Function

The *loss function* quantifies the error on training instances. It serves as the objective function to minimize during training and it quantifies the difference between tue and predicted outputs of the ANN.

Common loss functions include [47]:

- *Cross-entropy loss* (for classification)
- *Mean squared error (MSE)* (for regression)

4.6 Gradient Descent

Gradient descent(see, e.g. [52]) is an iterative optimization algorithm used to minimize the loss. The idea is to take repeated steps in the opposite direction of the gradient of the function at the current point, because this is the direction of steepest descent. Following the steepest descent, see Figure 15.

4.7 Learning Rate, Batch Size, Epochs and Iteration

- In training, the *learning rate* determines how large the updates to the weights are during training. The learning rate corresponds to the changes applied in each gradient descent step, see table 4. The optimal rate depends on the form of the functionand needs to be determined in the training.

- The *batch size* is the number of instances propagated forward before the network updates its weights. Common sizes are powers of 2: 32, 64, 128, 256.

- One complete pass through the entire training dataset is called a *training epoch*. The number of training epochs thus determines how often the whole dataset it propagated throught the model in order to uodate the parameters.

Figure 15: Effect of the learning rate on convergence in gradient descent methods. A small learning rate (a) requires many updates before reaching the minimum point. A large learning rate (b) causes drastic changes in the updates. The optimal rate (c) lies between the two extremes and results in optimal convergence behaviour.

Learning Rate	Effect
High	*Faster convergence, but may cause overshooting of global minima (unstable).*
Low	*More stable, but slow learning and risk of getting stuck*
Moderate	*good balance is typically most effective*

Table 4: Effect of Learning Rate on Gradient Descent Convergence.

- One forward-backward pass over a single batch is called an *iteration*.

After the initial terminology has been introduced, the next chapter provides a detailed description about the evolution of the *ReAct-Framework*.

5 The Evolution of Language Models: From Pure Generation to Agentic Interaction [†]

In this chapter, we will give an overview on the evolution of large language models, related prompting techniques (*prompt engineering*) and agentic frameworks. For further reading, we refer the interested reader to *the Prompt Engineering Guide*, see [12].

Figure 16: Illustration of the Evolution of ReAct models from language models via reasoning models and tool use.

5.1 Towards the ReAct Framework

Modern language models (LMs) have undergone a significant evolution, progressing from purely statistical text generators to sophisticated agents capable of complex reasoning and tool use (i.e. LLMs). Below, we outline this trajectory in four stages:

- **Basic LLMs** → Section 5.2
- **Reasoning LLMs** → Section 5.3
- **Agentic LLMs**, as well as → Section 5.4
- **ReAct Frameworks** → Section 5.5

5.2 Basic LLMs: Statistical Sequence Modeling

Generative language models, such as GPT-2 [13] and early GPT-3 [14] variants, were trained to *predict the next token in a sequence* based solely on large-scale text corpora. *Tokens* are the basic units that an LLM uses to process texts. A token can be a complete word, a punctuation mark or just a part of a word.

Core Capability The core capabilities of these models is the generation of fluent, coherent text by learning statistical *co-occurrence patterns* of tokens, so called *n-grams*, which were generalized via the models' self-attention mechanism.

Limitations Due to their purely generative nature, basic LLMs posses several inherit limitations.

1. *Lack of explicit multi-step reasoning.* Basic LLMs produce an output *stream*, and thus lack the ability to rethink and iteratively refine their output.

2. *No interaction with external data or execution environments.* Basic large language models generate new tokens based on their internal parameters only and have no access to external sources and tools.

3. *They are susceptible to superficial "hallucinations" and inconsistencies in complex tasks.* Lacking external tools, the output cannot be verfied or validated.

5.3   Reasoning LLMs: Chain-of-Thought and Internal Deliberation

To improve performance on tasks requiring multi-step inference (e.g., arithmetic, logic puzzles, multi-fact question answering), a method called Chain-of-Thought (CoT) Prompting was developed [15]. To create a chain-of-thought prompt, users typically attach instructions such as "Describe your reasoning step by step" or "Explain your answer in steps" to their request. This encourages the LLM to generate intermediate steps before the final answer, making the process more transparent and accurate.

Core Capability CoT encourages the model to produce intermediate reasoning steps in natural language before arriving at a final answer. It enables the LM to "think out loud", decomposing complex problems into smaller sub-problems.

Compared to basic LLMs, reasoning LLMs show significant *gains on benchmarks* requiring sequential logic. Moreover, they provide *enhanced transparency*, since instructors and users can inspect the model's reasoning chain.

Limitations The Chain-of-Thought (CoT) still occurs entirely within the LM's parameters; there is no external verification or data retrieval involved in the reasoning process. Moreover, errors in early reasoning steps propagate through to the final answer. Therefore, despite enhanced reasoning capabilities compared to vanilla LLMs, reasoning models still are still restricted in terms of knowledge base and external verification [15].

5.4   Agentic LLMs: Integrating Tools and External Environments

Agents based on LLMs, hereafter also referred to as *LLM agents* for short, integrate LLM applications that can perform complex tasks by using an architecture that combines LLMs with key modules such as scheduling and memory. When building LLM agents, an LLM serves as the main controller or "brain" that controls a sequence of operations required to complete a task or user request.

Core Capability Agents allow language models to act - to query databases, call APIs, execute code, or interact with other software - thus overcoming information constraints and validating reasoning steps. The LLM agent may require key modules such as scheduling, memory and tool utilization.

5.4.1 Building Blocks

Generally speaking, an LLM agent framework can consist of the following core components:

- **Agent/brain** the LLM serves as the core coordinator.

- **Memory:** manages the agent's past behaviors. It stores intermediate results and context from the tools and enhances the model with long-term memory and data

- **Tools:** Tools correspond to a set of tool(s) that enables the LLM agent to interact with external environments, such as Wikipedia Search API, code interpreter and math engine.

  - *Tool API Interface:* The LM is extended with a predefined set of callable functions (search, calculator, code execution).

- **Planning Module:** The LM devises a sequence of tool uses to satisfy the user's query, often framed as a plan in natural language.

  - *Execution and Observation:* Each tool invocation returns structured observations that feed back into the LM's next planning step.

Wang et al., (see [16]) formalise various planning modules, differentiating between planning modules *with feedback* and *without feedback*. The latter are termed [12] *Act-Only* language models and often fail to solve complex tasks. Examples of *Act-Only LMs* (acc. to [16]) include the following:

- `WebGPT`: Browses the web to gather citations [17].

- `HuggingGPT`: HuggingGPT [18] employs ChatGPT to orchestrate Hugging Face AI models to for multi-modal problem solving.

- `SayCan`: Plans robot actions by combining LLM reasoning with affordance models. [19]

- `Toolformer and frameworks like LangChain`: Automatically learn when and how to call external tools during generation. [20, 21]

To overcome this challenge, one can use a mechanism that allows the model to iteratively reflect and refine the execution plan based on past actions and observations: `ReAct`.

5.5 The ReAct Framework: Synergizing Reasoning and Acting

ReAct – short for Reasoning + Acting – formalizes an integrated loop in which an LM interleaves internal deliberation with external tool use:

The `ReAct-loop` (in general and in CAI) comprizes of five steps:

1. **Reasoning ("Think"):** The model generates one or more natural-language thoughts, articulating what to do next or why.

2. **Acting ("Act"):** Also termed 'action generation'. In this step the action is generated. The LLM selects and an external tool based on its reasoning.

3. **Execution ("Execute"):** The action or code is executed in a sandboxed environment.

4. **Observation ("Observe"):** The environment or tool returns results (data retrieval, computation output, API response), which become new context.

5. **Iterative Loop:** The LM uses the updated context to refine its next "Thought" and "Action", enabling error correction, dynamic planning, and more reliable outcomes.

Figure 17: The ReAct-loop comprizes of five steps: Thinks, Act, Execute, Observe, Iterate

5.6 Why Iteration Matters

The iterative nature of ReAct-Frameworks posesses several advantages over Act-Only and pure Reasoning LMs:

- *Overcomming the Formal-Probabilistic Bridge.* Because LLMs are not formal processors, the code they generate might not be correct. ReAct provides a runtime check – a way to bring formal verification into a probabilistic process. But the need for iteration reflects the mismatch: LLMs can't guarantee correctness, so they must observe and revise.

- *Error Mitigation.* If a code snippet fails or a data lookup yields no result, the model can analyze the failure and adjust its approach.

- *Dynamic Reasoning.* ReAct enables adaptive strategies – branching plans, conditional tool use, and exploration of alternative solutions.

- *Grounding.* ReAct anchors the LM's internal reasoning in concrete, verifiable outputs from the external environment.

In effect, the ReAct Framweork makes LLMs behave more like programmers: it enables the agent to write code, test it, learn from the result, and iterate.

5.6.1 Agent Profiles or Personas

Although not mandatory, an agent can be assigned a *profile* or *persona* to define their role [16]. This profiling information is typically written in the prompt, which can contain specific details such as role details, personality, social information and other demographic information.

5.7 Limitations of Single Agents

Intitially, ReAct is a single agent model, meaning it inherits the draw-backs of single agent models. Single agent models are subject to the following limitations:

- **Context Window Limits:** Language models can only consider a limited amount of text (tokens) at once, restricting their ability to handle large documents or long conversations.

- **Hallucination / Lack of Accuracy:** Even with external tools, single agents sometimes generate confident but incorrect or unverifiable responses, due to a lack of real-time validation or external checking.

- **Single Task Execution:** Most LLMs handle one prompt at a time, limiting their efficiency for complex or multi-part problems.

- **Lack of Collaboration:** A single agent lacks the capacity for division of labor or specialized reasoning, reducing its performance on tasks that benefit from teamwork or expert roles.

5.8   Solutions: Parallelization and Multi-Agent Systems

To overcome the limitations of single agents, *multi-agent systems* have gained popolarity in generative AI systems Common use-cases include the following:

- **Distributed Processing and Cross-Validation:** Multiple agents can evaluate the same task independently, increasing reliability through majority voting or peer review.

- **Shared Tasks – Divide and Conquer, Meta-Agents:** Tasks are split among agents by a controller (meta-agent), each solving a portion before merging their outputs. Example: `AutoGen`[22], `LangGraph` [20]

- **Multi-tasking and Specialized Agents:** Assigning different roles to agents (e.g., planner, coder, evaluator) increases efficiency and facilitates expertise-based problem solving, cf. [23, 24].

5.9   The Challenge: Coordination of Agents

In *multi-agent systems*, agents may use structured messages or APIs to exchange results, feedback, or queries, mimicking human collaboration [25]. The following terminology is used in this context: After

the basic terminology of agentic LLM frameworks has been discussed, we proceed to elaborating on the how the *ReAct* framework is realized within the CAI architecture.

Term	Definition
Agentic Patterns	Designing systems that balance autonomy and coordination without excessive complexity is task-dependent. Various ways to do so include Hierarchical Patterns, Swarm Patterns etc. The overarching term describing the setup is called *Multi-Agent Architecture* or *Agentic Pattern* [20] [26].
Human-In-The-Loop (HITL)	Acc. to [27] the term *Human-In-The-Loop* (HITL) refers to semi-autonomous systems where model developers continuously integrate human feedback into different steps of the model deployment workflow, see Section 6.7.
Turns	As in group discussions, *turns* are used to prevent overlap or confusion in AI-agent communication. A turn typically represents the workflow from the incoming to an outgoing message of the agent. During a turn, multiple interactions with other agents may take place (e.g. tool use). An interaction is a bilateral exchange between an agent and another agent or the environment.
Handoffs	*Handoffs* are sub-agents that the agent can delegate to. You provide a list of handoffs, and the agent can choose to delegate to them if relevant. They allow passing on context and results smoothly between agents. This is particularly useful in scenarios where different agents specialize in distinct areas.
Tracing	*Tracing* defines the act of collecting a comprehensive record of events during an agent run: LLM generations, tool calls, etc.

Table 5: Multi-Agent LLM Systems - Basic Terminology.

6 CAI Architecture [†] [*]

In this Chapter, we describe how the ReAct Framework is realized in the CAI software stack. CAI focuses on making cybersecurity agent *coordination* and *execution* lightweight, highly controllable, and useful for humans. To do so it builds upon 7 pillars: `Agents`, `Tools`, `Handoffs`, `Patterns`, `Turns`, `Tracing` and `HITL`. We recommend the interested reader and practitioners to take a look at the following files in the source folder of the CAI GitHub repository as a starting point for using CAI:

- __init__.py
- cli.py - entrypoint for command line interface
- util.py - utility functions
- agents - Agent implementations
- internal - CAI internatl functions (endpoints, metrics, logging, etc.)
- prompts - Agent Prompt Database
- repl - CLI aestethics and commands
- sdk - CAI command sdk
- tools - agent tools

Figure 18: Conceptual Drawing: ReAct Model Architecture in CAI

6.1 Agents

At its core, CAI abstracts its cybersecurity behavior via Agents and agentic Patterns (see Section 5). An Agent in an *intelligent system that interacts with some environment.* More technically, within CAI we embrace a robotics-centric definition wherein an agent is anything that can be viewed as a system perceiving its environment through sensors, reasoning about its goals and and acting accordingly upon that environment through actuators (definition adapted from [28]).

In cybersecurity, an `Agent` interacts with systems and networks, using peripherals and network interfaces as sensors, reasons accordingly and then executes network actions as if actuators.

And, as mentioned before, in CAI, Agents implement the ReACT (Reasoning and Action) agent model [11].

```
1  import os
2  from openai import AsyncOpenAI
3  from cai.sdk.agents import Agent, Runner, OpenAIChatCompletionsModel
4
5  #Function to run the agent
6  async def run_agent(agent, message):
7      response = await Runner.run(agent, message)
8      return response
```

```
1  #Initialize the agent
```

```
 2   ctf_agent = Agent(
 3       name="CTF Agent",
 4       instructions="""You are a Cybersecurity expert Leader""",
 5       model=OpenAIChatCompletionsModel(
 6               model=os.getenv('CAI_MODEL', "openai/gpt-4o"),
 7               openai_client=AsyncOpenAI(),))
 8
 9   #Define the task message
10   messages = [{"role": "user",
11                "content": "CTF challenge: TryMyNetwork. Target IP: 192.168.1.1"}]
12
13   #Run the agent
14   response = await run_agent(agent = ctf_agent, message = messages)
```

6.2 Tools

Tools let cybersecurity agents take actions by providing interfaces to execute system commands, run security scans, analyze vulnerabilities, and interact with target systems and APIs - they are the core capabilities that enable CAI agents to perform security tasks effectively.

In CAI, tools include built-in cybersecurity utilities (like LinuxCmd for command execution, WebSearch for OSINT gathering, Code for dynamic script execution, and SSHTunnel for secure remote access), function calling mechanisms that allow integration of any Python function as a security tool, and agent-as-tool functionality that enables specialized security agents (such as reconnaissance or exploit agents) to be used by other agents, creating powerful collaborative security workflows without requiring formal handoffs between agents. Please refer to Section 8.3 for a detailed description of available tools and their functioning.

```
 1   import os
 2   from openai import AsyncOpenAI
 3   from cai.sdk.agents import Agent, Runner, OpenAIChatCompletionsModel
 4   from cai.tools.reconnaissance.generic_linux_command import generic_linux_command
 5   from cai.tools.reconnaissance.exec_code import execute_code
 6
 7   #Function to run the agent
 8   async def run_agent(agent, message):
 9       response = await Runner.run(agent, message)
10       return response
```

```
1   #Initialize the agent and set tools
2   ctf_agent = Agent(
3       name="CTF Agent",
4       instructions="""You are a Cybersecurity expert Leader""",
5       model=OpenAIChatCompletionsModel(
6               model=os.getenv('CAI_MODEL', "openai/gpt-4o"),
7               openai_client=AsyncOpenAI(),),
8       tools=[generic_linux_command, execute_code])
9
10  #Define the task message
11  messages = [{"role": "user",
12              "content": "CTF challenge: TryMyNetwork. Target IP: 192.168.1.1"}]
13
14  #Run the agent
15  response = await run_agent(agent = ctf_agent, message = messages)
```

6.3 Handoffs

Handoffs allow an Agent to delegate tasks to another agent, which is crucial in cybersecurity operations where specialized expertise is needed for different phases of an engagement.

In our framework, Handoffs are implemented as tools for the LLM, where a **handoff/transfer function** like transfer_to_flag_discriminator enables the ctf_agent to pass control to the flag_discriminator_agent once it believes it has found the flag. This creates a security validation chain where the first agent handles exploitation and flag discovery, while the second agent specializes in flag verification, ensuring proper segregation of duties and leveraging specialized capabilities of different models for distinct security tasks.

```
1   import os
2   from cai.sdk.agents import Agent, Runner, OpenAIChatCompletionsModel, function_tool
3   from openai import AsyncOpenAI
4
5   #Function to run the agent
6   async def run_agent(agent, message):
7       response = await Runner.run(agent, message)
8       return response
9
10  #Custom function for flag discriminator
11  @function_tool
12  def transfer_to_flag_discriminator():
13      """
14      Transfer the flag to the
15      flag_discriminator_agent to check if
16      it is the correct flag
17      """
18      return flag_discriminator_agent
```

```
1    #Initialize the CTF agent
2    ctf_agent = Agent(
3        name="CTF Agent",
4        instructions="""You are a Cybersecurity expert Leader""",
5        model=OpenAIChatCompletionsModel(
6                model=os.getenv('CAI_MODEL', "openai/gpt-4o"),
7                openai_client=AsyncOpenAI(),),
8        tools = [transfer_to_flag_discriminator])
9
10   #Initialize the Flag Discriminator Agent
11   flag_discriminator_agent = Agent(
12       name="Flag Discriminator Agent",
13       instructions="""You are a Cybersecurity expert facing a CTF challenge.
14               You are in charge of checking if the flag is correct.""",
15       model=OpenAIChatCompletionsModel(
16               model=os.getenv('CAI_MODEL', "openai/gpt-4o"),
17               openai_client=AsyncOpenAI(),))
18
19   #Define the task message
20   messages = [{"role": "user",
21               "content": "CTF challenge: TryMyNetwork. Target IP: 192.168.1.1"}]
22
23   #Run the agent
24   response = await run_agent(agent = ctf_agent, message = messages)
```

6.4 Patterns

Recall that agentic `Pattern` is a structured design paradigm in artificial intelligence systems where autonomous or semi-autonomous agents operate within a defined *interaction framework* (the pattern) to achieve a goal. These `Patterns` specify the organization, coordination, and communication methods among agents, guiding decision-making, task execution, and delegation.

An agentic pattern (AP) can be formally defined as a tuple:

$$AP = (A, H, D, C, E)$$

wherein:

- A (Agents): A set of autonomous entities, $A = \{a_1, a_2, ..., a_n\}$, each with defined roles, capabilities, and internal states.

- H (Handoffs): A function $H : A \times T \to A$ that governs how tasks T are transferred between agents based on predefined logic (e.g., rules, negotiation, bidding).

- D (Decision Mechanism): A decision function $D : S \to A$ where S represents system states, and D determines which agent takes action at any given time.

- C (Communication Protocol): A messaging function $C : A \times A \to M$, where M is a message space, defining how agents share information.

- E (Execution Model): A function $E : A \times I \to O$ where I is the input space and O is the output space, defining how agents perform tasks.

When building `Patterns` in CAI, we generally classify them among one of the categories in Table 6, though others exist.

Agentic Pattern categories	Description
Multi Agent Patterns	
Swarm (Decentralized)	Agents share tasks and self-assign responsibilities without a central orchestrator. Handoffs occur dynamically. *An example of a peer-to-peer agentic pattern is the `CTF Agentic Pattern`, which involves a team of agents working together to solve a CTF challenge with dynamic handoffs.*
Hierarchical	A top-level agent (e.g., "PlannerAgent") assigns tasks via structured handoffs to specialized sub-agents. Alternatively, the structure of the agents is harcoded into the agentic pattern with pre-defined handoffs.
Chain-of-Though (Sequential Workflow)	A structured pipeline where Agent A produces an output, hands it to Agent B for reuse or refinement, and so on. Handoffs follow a linear sequence. *An example of a chain-of-thought agentic pattern is the `ReasonerAgent`, which involves a Reasoning-type LLM that provides context to the main agent to solve a CTF challenge with a linear sequence.*
Auction-Based (Competitive Allocation)	Agents "bid" on tasks based on priority, capability, or cost. A decision agent evaluates bids and hands off tasks to the best-fit agent.
Single Agent Patterns	
Recursive	A single agent continuously refines its own output, treating itself as both executor and evaluator, with handoffs (internal or external) to itself. *An example of a recursive agentic pattern is the `CodeAgent` (when used as a recursive agent), which continuously refines its own output by executing code and updating its own instructions.*

Table 6: Patterns in CAI - Framwork-specific Terminology and Description.

6.4.1 Building Custom Patterns in CAI

Building a `Patterns` is rather straightforward and only requires to link together `Agents`, `Tools` and `Handoffs`. For example, the following builds an offensive Pattern that adopts the Swarm category:

```
1   import os
2   from cai.sdk.agents import Agent, Runner, OpenAIChatCompletionsModel, function_tool
3   from openai import AsyncOpenAI
4   from cai.agents.red_teamer import redteam_agent
5   from cai.agents.thought import thought_agent
6   from cai.agents.mail import dns_smtp_agent
```

```
 1   #Custom function for dns agent
 2   @function_tool
 3   def transfer_to_dns_agent():
 4       """
 5       Use THIS always for DNS scans and domain reconnaissance about dmarc and dkim registers
 6       """
 7       return dns_smtp_agent
 8
 9   #Custom function for red team agent
10   @function_tool
11   def redteam_agent_handoff(ctf=None):
12       """
13       Red Team Agent, call this function empty to transfer to redteam_agent
14       """
15       return redteam_agent
16
17   #Custom function for thought agent
18   @function_tool
19   def thought_agent_handoff(ctf=None):
20       """
21       Thought Agent, call this function empty to transfer to thought_agent
22       """
23       return thought_agent
24
25   # Register handoff functions to enable inter-agent communication pathways
26   redteam_agent.tools.append(transfer_to_dns_agent)
27   dns_smtp_agent.tools.append(redteam_agent_handoff)
28   thought_agent.tools.append(redteam_agent_handoff)
29
30   # Initialize the swarm pattern with the thought agent as the entry point
31   redteam_swarm_pattern = thought_agent
32   redteam_swarm_pattern.pattern = "swarm"
```

6.5 Turns and Interactions

During the agentic flow (conversation), we distinguish between **interactions** and **turns**.

- **Interactions** are sequential exchanges between one or multiple agents. Each agent executing its logic corresponds with one *interaction*. Since an `Agent` in CAI generally implements the ReACT agent model, each *interaction* consists of 1) a reasoning step via an LLM inference and 2) act by calling zero-to-n `Tools`. This is defined in `process_interaction()` in core.py.

- **Turns:** A turn represents a cycle of one or more interactions which finishes when the `Agent` (or `Pattern`) executing returns `None`, judging there're no further actions to undertake. This is defined in `run()`, see core.py.

Note: CAI Agents are not related to Assistants in the Assistants API. They are named similarly for convenience, but are otherwise completely unrelated. CAI is entirely powered by the Chat Completions API and is hence stateless between calls.

6.6 Tracing

CAI implements AI observability by adopting the OpenTelemetry standard and to do so, it leverages Phoenix which provides comprehensive tracing capabilities through OpenTelemetry-based instrumen-

tation, allowing you to monitor and analyze your security operations in real-time. This integration enables detailed visibility into agent interactions, tool usage, and attack vectors throughout penetration testing workflows, making it easier to debug complex exploitation chains, track vulnerability discovery processes, and optimize agent performance for more effective security assessments.

6.7 Human-In-The-Loop (HITL)

6.7.1 Levels of Autonomy in Cybersecurity

In robotics, **automation** refers to systems that execute predefined tasks without human intervention, while **autonomy** requires something fundamentally different:

The distinction between automated and autonomous cybersecurity AI is not academic pedantry – it's a critical safety issue.

adaptive behavior. Autonomous systems must interact with their environment, reason about uncertainty, and tailor their actions to environments and situations never explicitly programmed. The key difference between automated and autonomous systems lies in the capability of *intelligent adaptation* to changing environments.

The cybersecurity domain, comprizing of dynamic environments, adversarial actors, incomplete information, and the need for creative problem-solving, resembles the robotics setting. Similarly, "AI security tools" intelligent adaption capabilities range over spectrum from no automation to true autonomy.

Adapting the well-established SAE J3016 levels of driving automation [10], [2] proposes six levels of cybersecurity autonomy, from Level 0 (no tools) to Level 5 (fully autonomous):

- **Level 0 – No Tools:** all tasks are performed manually, without any computational tools whatsoever. Since using a terminal or text editor constitutes tool usage, true Level 0 operation is limited to offline spying tasks.

- **Level 1 – Manual Tools:** tool use (e.g. Metasploit [6] or Nmap [29]) provides assistance and execute commands. Tool input and strategic decisions what to do next, however, remain human-driven.

- **Level 2 – LLM-Assisted:** LLMs, such as **PentestGPT** [3], assist while humans execute. The AI is an intelligent assistant, not an independent actor.

- **Level 3 – Semi-Automated:** The AI system can execute complete attack sequences in specific, well-defined scenarios. Tools like AutoPT [8] and Vulnbot [9] alongside many others operate here – they can autonomously scan, exploit, and report findings, but **require human intervention** for edge cases, validation, and mitigation strategies.

- **Level 4 – Cybersecurity AIs:** Systems aim to handle the complete security assessment lifecycle in security scenarios with minimum human intervention. Nontheless, these systems require human oversight.

- **Level 5 – Fully Autonomous:** The aspirational goal where AI handles all cybersecurity tasks in all conditions without human intervention.

6.7.2 CAI is a semi-autonomous framwork

Our framework aims for *Level 4* capabilities by combining multiple specialized agents (for pentesting, bug hunting, blue teaming, etc.) with seamless tool integration and a human supervisor overseeing the AI's choices. While CAI explores autonomous capabilities, we recognize that *effective security*

operations still require human teleoperation providing expertise, judgment, and oversight in the security process. In this taxonomy, CAI delivers a framework for building Cybersecurity AIs with a strong emphasis on *semi-autonomous* operation, as the reality is that **fully-autonomous** cybersecurity systems remain premature and face significant challenges when tackling complex tasks.

Accordingly, the *Human-In-The-Loop* (HITL) module is a core design principle of CAI, acknowledging that human intervention and teleoperation are essential components of responsible security testing.

Figure 19: Despite rapid progress in AI, **human expertise remains a critical component of effective cybersecurity AI systems**. The "*Human-In-The-Loop (HITL)*" is the end-user or security analyst who provides the prompts, guidance, and final judgment on the AI's findings.

Through the `cli.py` interface, users can seamlessly interact with agents at any point during execution by simply pressing `Ctrl+C`. This is implemented across core.py and also in the REPL abstractions REPL.

7 Getting Started *

In this chapter we provide a detailed description how to setup CAI in your OS or container environment.

7.1 Install

To install CAI framework on different platforms the system needs to ensure that Python 3.12 is installed on your machine. After that a Python virtual environment has to be set up to isolate dependencies. The environment is activated when the cai-framework package is installed. A .env file with API keys and default settings is generated. Finally the CAI command-line tool is launched.

Platform-specific steps are applied as needed, such as `Homebrew` on macOS, Personal Package Archives on Ubuntu or manual compilations on Android.

In the following sections, you will find the OS or environment specific installation commands to install CAI.

7.1.1 OS X

```
1   brew update && \
2       brew install git python@3.12
3
4   # Create virtual environment
5   python3.12 -m venv cai_env
6
7   # Install the package from the local directory
8   source cai_env/bin/activate && pip install cai-framework
9
10  # Generate a .env file and set up with defaults
11  echo -e 'OPENAI_API_KEY="sk-1234"\nANTHROPIC_API_KEY=""\nOLLAMA=""\nPROMPT_TOOLKIT_NO_CPR=1
12  \nCAI_STREAM=false' > .env
13
14  # Launch CAI
15  cai  # first launch it can take up to 30 seconds
```

7.1.2 Ubuntu 24.04

Installing CAI in Ubuntu 24.04 is done using the following commants in the `CLI`:

```
1   sudo apt-get update && \
2       sudo apt-get install -y git python3-pip python3.12-venv
3
4   # Create the virtual environment
5   python3.12 -m venv cai_env
```

```
1   # Install the package from the local directory
2   source cai_env/bin/activate && pip install cai-framework
3
4   # Generate a .env file and set up with defaults
5   echo -e 'OPENAI_API_KEY="sk-1234"\nANTHROPIC_API_KEY=""\nOLLAMA=""
6   \nPROMPT_TOOLKIT_NO_CPR=1
7   \nCAI_STREAM=false' > .env
8
9   # Launch CAI
10  cai  # first launch it can take up to 30 seconds
```

7.1.3 Ubuntu 20.04

To launch CAI in Ubuntu 20.04, please proceed with the following commands:

```
1   sudo apt-get update && \
2       sudo apt-get install -y software-properties-common
3
4   # Fetch Python 3.12
5   sudo add-apt-repository ppa:deadsnakes/ppa && sudo apt update
6   sudo apt install python3.12 python3.12-venv python3.12-dev -y
7
8   # Create the virtual environment
9   python3.12 -m venv cai_env
10
11  # Install the package from the local directory
12  source cai_env/bin/activate && pip install cai-framework
13
14  # Generate a .env file and set up with defaults
15  echo -e 'OPENAI_API_KEY="sk-1234"\nANTHROPIC_API_KEY=""\nOLLAMA=""\nPROMPT_TOOLKIT_NO_CPR=1
16  \nCAI_STREAM=false' > .env
17
18  # Launch CAI
19  cai  # first launch it can take up to 30 seconds
```

7.1.4 Windows WSL

To install CAI in the Windows Subsystem for Linux (WSL), please open a terminal and execute the following commands:

```
1
2   sudo apt-get update && \
3       sudo apt-get install -y git python3-pip python3-venv
4
5   # Create the virtual environment
6   python3 -m venv cai_env
```

```
1   # Install the package from the local directory
2   source cai_env/bin/activate && pip install cai-framework
3
4   # Generate a .env file and set up with defaults
5   echo -e 'OPENAI_API_KEY="sk-1234"\nANTHROPIC_API_KEY=""\nOLLAMA=""\nPROMPT_TOOLKIT_NO_CPR=1
6   \nCAI_STREAM=false' > .env
7
8   # Launch CAI
9   cai  # first launch it can take up to 30 seconds
```

7.1.5 Android

To set up CAI in android, please process the following commands:

```
1   # Get new apt keys
2   wget http://http.kali.org/kali/pool/main/k/kali-archive-keyring/
3   kali-archive-keyring_2024.1_all.deb
4
5   # Install new apt keys
6   sudo dpkg -i kali-archive-keyring_2024.1_all.deb && rm kali-archive-keyring_2024.1_all.deb
7
8   # Update APT repository
9   sudo apt-get update
10
11  # CAI requieres python 3.12, lets install it (CAI for kali in Android)
12  sudo apt-get update && sudo apt-get install -y git python3-pip build-essential
13  zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev libssl-dev libreadline-dev
14  libffi-dev libsqlite3-dev wget libbz2-dev pkg-config
15  wget https://www.python.org/ftp/python/3.12.4/Python-3.12.4.tar.xz
16  tar xf Python-3.12.4.tar.xz
17  cd ./configure --enable-optimizations
18  sudo make altinstall # This command takes long to execute
19
20  # Clone CAI's source code
21  git clone https://github.com/aliasrobotics/cai && cd cai
22
23  # Create virtual environment
24  python3.12 -m venv cai_env
25
26  # Install the package from the local directory
27  source cai_env/bin/activate && pip3 install -e .
28
29  # Generate a .env file and set up
30  cp .env.example .env  # edit here your keys/models
31
32  # Launch CAI
33  cai
```

7.1.6 Docker

There is also the possibilty to run the framework directly in a Docker container without additional environmental configuration. For that the Github repo has to be cloned to the local machine and opened in an IDE e.g. in Visual Studio Code. After opening the project folder a pop-up appers in the lower right corner saying "Folder contains a Dev Container configuration file. Reopen folder to develop in container." By clicking on "Reopen in Container" the remote connection initializes and starts the container (this may take some time). Once the container is running a new terminal can be opened directly in VS Code, leading to a shell session with user root in workspace.

Alternatively, a seperate terminal can be started accessing the docker container via

```
1   docker exec -it <containerID> bash
```

After entering the session with root in `workspace`, the tool can be started with the command `cai`.

```
1   (root@1ae1d8c63301)-[/workspace]
2   # cai
```

7.2   Initial Configuration

CAI uses a `.env` file to manage key settings like API keys, model selection, and feature toggles via environment variables. This section explains how to set up the file using the provided .env example and outlines important requirements. It also shows how to configure a custom endpoint for advanced setups or self-hosted models.

7.2.1   Setup the .env File

CAI leverages the .env file to load configurations at launch. To facilitate the setup, the repo provides an exemplary .env file as a template for configuring CAI's setup and your LLM API keys to work with the desired LLM models.

> ⚠️ CAI does not provide API keys for any model by default. In order to use CAI, users need to integrate their own LLM API keys or host their own models. To use a specific LLM in CAI, the respective API keys or URLs have to be set in the respective environment variables.
> Moreover, the OPENAI_API_KEY must not be left blank. It should contain either "sk-123" (as a placeholder) or your actual API key.

If you are using `alias0` model, make sure you have installed a CAI version >0.4.0 and that you have an .env.example file to be able to use it.

```
1   OPENAI_API_KEY="sk-1234"
2   OLLAMA=""
3   ALIAS_API_KEY="<sk-your-key>"  # note, add yours
4   CAI_STEAM=False
```

7.2.2   Custom OpenAI Base URL Support

CAI supports configuring a custom OpenAI API base URL via the `OPENAI_BASE_URL` environment variable. This allows users to redirect API calls to a custom endpoint, such as a proxy or self-hosted OpenAI-compatible service. Below you can find an Example of a `.env` entry configuration:

```
1   OLLAMA_API_BASE="https://custom-openai-proxy.com/v1"
```

Or directly from the command line:

```
1   OLLAMA_API_BASE="https://custom-openai-proxy.com/v1" cai
```

7.3 Environment Variables

To use private models, users are provided with a .env.example file. After renaming it as .env., users can fill in in their corresponding API keys to use CAI. A list of possible environment variables is shown in Table 7.

Variable	Description
Swarm (CTF_NAME)	Name of the CTF challenge to run (e.g. "picoctf_static_flag")
CTF_CHALLENGE	Specific challenge name within the CTF to test
CTF_SUBNET	Network subnet for the CTF container
CTF_IP	IP address for the CTF container
CTF_INSIDE	Whether to conquer the CTF from within container
CAI_MODEL	Model to use for agents
CAI_DEBUG	Set debug output level (0: Only tool outputs, 1: Verbose debug output, 2: CLI debug output)
CAI_BRIEF	Enable/disable brief output mode
CAI_MAX_TURNS	Maximum number of turns for agent interactions
CAI_TRACING	Enable/disable OpenTelemetry tracing
CAI_AGENT_TYPE	Specify the agents to use (boot2root, one_tool...)
CAI_STATE	Enable/disable stateful mode
CAI_MEMORY	Enable/disable memory mode (episodic, semantic, all)
CAI_MEMORY_ONLINE	Enable/disable online memory mode
CAI_MEMORY_OFFLINE	Enable/disable offline memory
CAI_ENV_CONTEXT	Add dirs and current env to LLM context
CAI_MEMORY_ONLINE_INTERVAL	Number of turns between online memory updates
CAI_PRICE_LIMIT	Price limit for the conversation in dollars
CAI_REPORT	Enable/disable reporter mode (ctf, nis2, pentesting)
CAI_SUPPORT_MODEL	Model to use for the support agent
CAI_SUPPORT_INTERVAL	Number of turns between support agent executions
CAI_WORKSPACE	Defines the name of the workspace
CAI_WORKSPACE_DIR	Specifies the directory path where the workspace is located

Table 7: List of Environment Variables

7.4 OpenRouter Integration

The Cybersecurity AI (CAI) platform offers seamless integration with OpenRouter, a unified interface for Large Language Models (LLMs). This integration is crucial for users who wish to leverage advanced AI capabilities in their cybersecurity tasks. OpenRouter acts as a bridge, allowing CAI to communicate with various LLMs, thereby enhancing the flexibility and power of the AI agents used within CAI.

To enable `OpenRouter` support in CAI, you need to configure your environment by adding specific entries to your .env file. This setup ensures that CAI can interact with the OpenRouter API, facilitating the use of sophisticated models, such as Meta-LLaMA. The following code snippet depicts how users can configure `OpenRouter` support in CAI.

```
1  CAI_AGENT_TYPE=redteam_agent
2  CAI_MODEL=openrouter/meta-llama/llama-4-maverick
3  OPENROUTER_API_KEY=<sk-your-key>  # note, add yours
4  OPENROUTER_API_BASE=https://openrouter.ai/api/v1
```

7.5 Model Context Protocol (MCP) Integration

CAI supports the Model Context Protocol (MCP) for integrating external tools and services with AI agents. MCP is supported via two transport mechanisms:

SSE (Server-Sent Events) - For web-based servers that push updates over HTTP connections:

```
1  CAI>/mcp load http://localhost:9876/sse burp
```

STDIO (Standard Input/Output) - For local inter-process communication:

```
1  CAI>/mcp stdio myserver python mcp_server.py
```

Once connected, you can add the MCP tools to any agent:

```
 1   CAI>/mcp add burp redteam_agent
 2   Adding tools from MCP server 'burp' to agent 'Red Team Agent'...
 3                              Adding tools to Red Team Agent
 4    _____
 5   | Tool                          | Status | Details                               |
 6   |                               |        |                                       |
 7   | send_http_request             | Added  | Available as: send_http_request       |
 8   | create_repeater_tab           | Added  | Available as: create_repeater_tab     |
 9   | send_to_intruder              | Added  | Available as: send_to_intruder        |
10   | url_encode                    | Added  | Available as: url_encode              |
11   | url_decode                    | Added  | Available as: url_decode              |
12   | base64encode                  | Added  | Available as: base64encode            |
13   | base64decode                  | Added  | Available as: base64decode            |
14   | generate_random_string        | Added  | Available as: generate_random_string  |
15   | output_project_options        | Added  | Available as: output_project_options  |
16   | output_user_options           | Added  | Available as: output_user_options     |
17   | set_project_options           | Added  | Available as: set_project_options     |
18   | set_user_options              | Added  | Available as: set_user_options        |
19   | get_proxy_http_history        | Added  | Available as: get_proxy_http_history  |
20   | get_proxy_http_history_regex  | Added  | Available as: get_proxy_http_history_regex |
21   | get_proxy_websocket_history   | Added  | Available as: get_proxy_websocket_history |
22   | get_proxy_websocket_history_regex | Added | Available as: get_proxy_websocket_history_regex |
23   | set_task_execution_engine_state | Added | Available as: set_task_execution_engine_state |
24   | set_proxy_intercept_state     | Added  | Available as: set_proxy_intercept_state |
25   | get_active_editor_contents    | Added  | Available as: get_active_editor_contents |
26   | set_active_editor_contents    | Added  | Available as: set_active_editor_contents |
27   |_____|
28
29   Added 20 tools from server 'burp' to agent 'Red Team Agent'.
30   CAI>/agent 13
31   CAI>Create a repeater tab
```

You can list all active MCP connections and their transport types:

```
 1   CAI>/mcp list
```

8   Quickstart, CAI Commands and Use *

8.1   Quickstart

To start CAI after installing it, the user has to type `cai` in the command line interface:

```
1   # cai
2
3           CCCCCCCCCCCC      ++++++++   ++++++++       IIIIIIIIIII
4         CCC::::::::::::C  +++++++++        +++++++++  I::::::::I
5       CC:::::::::::::::C +++++++++          +++++++++ I::::::::I
6      C:::::CCCCCCCC::::C ++++++++    ++     ++++++++ II::::::II
7     C:::::C       CCCCCC ++++++     ++++      ++++++   I::::I
8    C:::::C               ++++     ++++++      ++++     I::::I
9    C:::::C               ++++                 ++++     I::::I
10   C:::::C                ++                    ++      I::::I
11   C:::::C                 +    ++++++++++++++   +      I::::I
12   C:::::C                     ++++++++++++++++++       I::::I
13   C:::::C                     +++++++++++++++++        I::::I
14    C:::::C       CCCCCC       ++++++++++++++          I::::I
15     C:::::CCCCCCCC::::C         ++++++++++++          II::::::II
16      CC:::::::::::::::C          ++++++++++           I::::::::I
17        CCC::::::::::::C           +++++               I::::::::I
18          CCCCCCCCCCCC              ++                 IIIIIIIIIII
19
20                    Cybersecurity AI (CAI), vX.Y.Z
21                        Bug bounty-ready AI
22
23   CAI>
```

The `cai` command initializes CAI and provides a prompt to execute any security task you want to perform. The navigation bar at the bottom displays important system information, which facilitates the user in understanding the environment while working with CAI.

You can find a quick demo video to help you get started with CAI. In the following Sections, the basic steps – from launching the tool to running your first AI-powered task in the terminal – are described in a beginner friendly manner.

8.2   CAI Command Reference

After starting CAI, users can directly enter natural language instructions in the command line interface, e.g. 'Do a port scan on machine ...' or execute CAI commands. CAI commands start with a slash / and possibly require arguments.

8.2.1   Quick Shortcuts

The following shortcuts help navigate quickly in CAI:

- ENTER - execute input

- ESC + ENTER - Multi-line input

- TAB - Command completion

- ↑/↓ - Command history

- `Ctrl+C` - Interrupt/Exit

- Use `help` for detailed command help

- Use `help quick` for a quick guide

- Use `help commands` for all commands

- `shell [COMMAND]` - Execute shell commands

- Use the `$` prefix for quick shell: `$ ls`

8.2.2 Environment Commands

To set the workspace, change the environment variables or run a ducker container, the following commands come handy:

- `workspace set [NAME]` - Set workspace directory

- `config` - Manage environment variables

- `virt run [IMAGE]` - Run a Docker container

Example: to set the CAI workspace directory, use the `workspace set [NAME]` command:

```
1  CAI>/workspace set ctf_name
```

8.2.3 Manage LLMs

To manage models, either via `Model Context Protocol` or `CAI API`, users can use the following commands

- `mcp load [TYPE] [CONFIG]` - Load MCP servers

- `model [NAME]` - Change AI model

Find an example of MCP use with CAI commands below:

```
1  CAI>/mcp load sse http://localhost:3000
2  CAI>/mcp add server_name agent_name
```

Example: To change the LLM, use the cai command `model`.

```
1  CAI>/model claude-3-haiku
```

8.2.4 Agent Management in CAI

For a detailed description on available built in agents (cf. `agent list` their use and tools, see Section 8.4. For an overview of build in multi-agent patterns, see Section 8.5.

- `agent list` - List all available agents

- `agent select [NAME]` - Switch to specific agent

- `agent info [NAME]` - Show agent details

- `parallel add [NAME]` - Configure parallel agents

8.2.5   Session History and Memory Management

The following commands can be used to manage the conversation and interaction history of CAI.

- `memory` list - List saved memories

- `history` - View conversation history

- `compact` - AI-powered conversation summary

- `flush` - Clear conversation history

8.2.6   Quick Start Workflows

After the initial configuration is set up, it is time to test CAI and and perform the first cybersecurity tasks.

CTF Challenge

To have CAI solve a capture the flag (CTF) challenge, the following sample workflow can be followed.

```
1    /agent select redteam_agent
2    /workspace set ctf_name
3    Describe the challenge...
```

Bug Bounty

To setup a simple boug bounty hunter agent – *Bug Bounter Agent*) – to test a website, can proceed with the following commands.

```
1    /agent select bug_bounter_agent
2    /model claude-3-5-haiku
3    Test https://example.com
```

Parallel Reconnaissance

To spawn two agents (here: a `Red Team Agent` and a `Network Traffic Specialist`) in parallel, use the following commands:

```
1    /parallel add red_teamer
2    /parallel add network_traffic_analyzer
3    Scan 192.168.1.0/24
```

For advanced security testing and more sophisitcated analysis taylored to the users' needs, agents and agentic patterns can be customized.

In the folloging sections we provide a comprehensive overview of available agents (Section 8.4) and function tools (Section 8.3) that can be assigned to agents, as well as built in mutli-agent patterns (Section).

8.3 Tools Available in CAI (v0.5.2)

This section, Section 8.3 gives a comprehensive overview of the function tools and related modules available in CAI. Since not all users might be familiar with the tools usually used for network security analysis, this section also provides a short introduction to the most common tools as well as use of these functions.

We refer to `src/cai/tools/` in the repository for the source code of the tools.

8.3.1 Taxonomy of Function Tools

CAI provides different *tools* agents can use for security analysis. They are grouped in six major categories inspired by the *Cyber Kill Chain* (a cybersecurity attack model typically accredited to Lockheet Martin [30], see [31]), as well as `misc, web`, `others` and `web`.

8.3.2 The Cyber Kill Chain

The *Cyber Kill Chain* is a framework developed by Lockheed Martin to identify and prevent cyber intrusion activity. The framework is structured in disctinct stages, providing a structured way for security teams to identify what the adversaries must complete in order to achieve their objectives. The seven stages of the *Cyber Kill Chain* are:

1. **Reconnaissance:** The attacker gathers information about the target such as email addresses, systems, employees and potential entry points.

2. **Weaponization:** The attacker creates a malicious payload and pairs it with an exploit.

3. **Delivery:** The attacker transmits the exploit to the target.

4. **Exploitation:** The payload is triggered, exploiting a vulnerability in the target system.

5. **Installation:** The attacker installs malware to maintain a foothold.

6. **Command and Control:** The compromised system connects back to the attacker, enabling a command channel for remote manipulation of the victim.

7. **Actions on Objectives:** The attacker achieves their final goal such as stealing sensitive files, deploying ransomware,causing operational disruption, etc.

Similarly, we categorize the CAI tools into six major categories and four additional smaller categories:

Reconnaissance and Weaponization | Exploitation | Privilege Escalation | Lateral Movement | Data Exfiltration | Command and Control | Network | Web Tools | Miscellaneous Tools | Other Tools

8.3.3 Tool Overview

As of version V0.5.2, CAI supports the following tools (by group) and related utility modules are available. Note that function tools that can be used by agents (directly or indirectly) are displayed in turquoise color, whereas classes and utility modules will be displayed in gray.

1. **Reconnaissance and Weaponization - *reconnaissance***

   - curl (Function Tool)
   - generic_linux_command (Function Tool)

- netcat (Function Tool)
- netstat (Function Tool)
- nmap (Function Tool)
- shodan.py (Utility Module)
- shodan_search (Function Tool)
- shodan_host_info (Function Tool)

2. **Exploitation - *exploitation***

- strings_command (Function Tool)
- decode_hex_bytes (Function Tool)
- decode64 (Function Tool)
- execute_code (Function Tool)

3. **Privilege escalation - *escalation***

- currently empty

4. **Lateral movement - *lateral***

- currently empty

5. **Data exfiltration - *exfiltration***

- list_dir (Function Tool)
- cat_file (Function Tool)
- pwd_command (Function Tool)
- find_file (Function Tool)
- wget (Function Tool)

6. **Command and control - *control***

- run_ssh_command_with_credentials (Function Tool)
- ReverseShellClient (Class)

7. **Network - *network***

- capture_remote_traffic (Function Tool)
- remote_capture_session (Function Tool)

8. **Web Tools - *web***

- google_search.py (Utility Module)
- headers.py (Utility Module)
- query_perplexity (function tool)
- make_web_search_with_explanation (Function Tool)
- make_google_search (function tool)
- webshell_suite.py (Utility Module)

9. **Miscellaneous Tools - *misc***

- rag.py (Utility Module)

- `reasoning.py` (Utility Module)
- `think` (Function Tool)
- `execute_cli_command` (Function Tool)
- `execute_python_code` (Function Tool)

10. **Other Tools - *others***

- `scripting_tool` (Function Tool)

8.3.4 Reconnaissance Tools

curl A simple `curl` function tool to make HTTP requests to a specified target (argument). the function returns the output of running the `curl` command.

> `curl` is a command-line tool to transfer data from or to a server using various protocols such as HTTP, HTTPS, FTP, and more. It can be used to interact with web servers, APIs, etc. This way, curl can be leveraged to uncover server misconfigurations, erroreous access control (e.g. if it is possible to access unauthorzied endpoints). Moreover, curl can be used to ss controls (e.g., accessing unauthorized endpoints) check for information disclosure in HTTP headers. See [32] for futher reading.

generic_linux_command The tool `generic_linux_command` executes commands with session management. This tool can be used to run any command. The system automatically detects and handles regular commands (`ls`, `cat`, `grep`, etc.), interactive commands that need persistent sessions (`ssh`, `nc`, `python`, etc.). It also handles session management and output capture. The function call returns either the command output, the session ID for interactive commands and/or the status message.

Examples `generic_linux_command("ls -la")`.
The system should automatically detect and use the appropriate execution environment.

- **CTF:** Commands run in the CTF challenge environment when available
- **Container:** Commands run in Docker containers when the environment variable `CAI_ACTIVE_CONTAINER` is set
- **SSH:** Commands run via SSH when the environment variables `SSH_USER` and `SSH_HOST` are configured
- **Local:** Commands run on the local system as fallback

> Linux commands are foundational elements for system management, monitoring, and scripting. General Linux commands include `ls`, `ps`, and `grep` [33]. It's primarily used for monitoring system behavior, viewing running processes, open ports, and users, and automating tasks via shell scripts. This tool can help uncover misconfigured services, unwanted running processes, and file permission issues [33].

netcat This fuction tool provides a simple `netcat` tool to connect to a specified host and port. Additional arguments to pass to the `netcat` command are the host, port and (optionally) the payload data to send to the host.

The function returns the output of running the `netcat` command or an error message if connection fails.

> `netcat` (nc) is a versatile networking tool used for reading and writing data across network connections using TCP or UDP [34]. It finds application in port scanning, banner grabbing, setting up reverse shells, and debugging and testing network services. Using netcat can help uncover open and exposed ports, identify services with verbose banners revealing software versions, and potentially achieve remote command execution if used with reverse shell techniques [34].

netstat This function provides the `netstat` tool to list all listening ports and their associated programs. The function returns the output of running the `netstat` command or an error message if connection fails.

> The `netstat` command displays active network connections, listening ports, routing tables, and network interface statistics [35]. The tool commonly used for checking open and listening ports, monitoring network activity, and identifying unexpected or unauthorized connections. This tool assists in uncovering hidden or unauthorized services, malware communicating externally, and port misuse or conflict [35].

nmap This function provides a simple `nmap` tool to scan a specified `target` (argument). The function returns output of running the `nmap` command.

> `nmap` is a powerful network scanning tool used for discovering hosts and services on a network [29]. It is employed for host discovery, port scanning, OS and service fingerprinting, and vulnerability detection. `nmap` can help uncover open and vulnerable ports, identify running services and their versions, and detect weak configurations or outdated software [29].

shodan.py A Shodan search utility module for reconnaissance. This module provides helper functions to search *Shodan* for information about hosts, services, and vulnerabilities using the Shodan API.

shodan_search This function tool performs a *Shodan* search for information based on the provided Shodan search query (str) and limit (int), specifying the maximum number of results to return (default:10).

shodan_host_info This function tool provides detailed information about a specific `host` from Shodan.

> *Shodan* [36] is a search engine that "lets you find and explore devices and systems connected to the Internet" [37]. It provides detailed information about a specific IP address and assists in viewing open ports and services on a host, checking for known vulnerabilities and CVEs, and profiling devices from external IPs. Shodan services can help uncover vulnerable or outdated services, unintended exposure of internal systems, and potential entry points for attackers [36].

8.3.5 Privilege Esclalation Tools

Currently, there are no tools in CAI that fall into this specific category. This does not imply that there are no tools available to do so; rather they corresponding tools correspond to multiple categories and are located in other categories.

8.3.6 Lateral Movement Tools

As with *Privilege Escalation*, the corresponding tools can be assigned to multiple categories and can be found in other categories.

8.3.7 Exploitation Tools

strings_command   Given the file path as an argument, this function tool extracts the printable strings from a binary file.

decode_hex_bytes   The function decodes a string of hex bytes (Input format: "0xFF 0x00 0x63..." into ASCII text.

decode64   This function tool decodes an input string from Base64 format to ASCII text.

> As their Linux analogons `xxd` or `hexdump`, `base64` and `strings`, these functions can be used for threat hunting, and vulnerability assessment: The `strings` command can be used to analyze malware or suspicious files, as it can help identify potential malicious code, extract configuration data, credentials, or other sensitive information. Decoding hexdumps is useful when analyzing network captures, malware, or (suspicious) files that contain hexadecimal-encoded data, such as authentication credentials. Base64, onn the other hand, is often used in HTTP traffic in authorization and content-type headers, as well as for JSON Web Tokens and API keys. When analyzing HTTP traffic, being able to decode base64-encoded data helps identify authentication credentials or API keys, and helps uncover security issues, such as hardcoded credentials.

execute_code   The function tool `execute_code` create a file code, stores it and executes it. It allows for executing code provided in the programming languages depicted in Table 8: Bevore execution, the

Python (py)	PHP (php)	Bash (sh)	Shell (sh)	Ruby (rb)
Perl (pl)	Go (go)	Golang (go)	JavaScript (js)	JS (js)
TypeScript (ts)	TS (ts)	Rust (rs)	C# (cs)	CS (cs)
Java (java)	Kotlin (kt)	C (c)	C++ (cpp)	

Table 8: List of programming languages that can be processed by the `execute_code` function tool.

tool creates a permanent file with the provided code. Subsequently, it executes the code using the appropriate interpreter. The code can be executed repreatedly using the `generic_linux_command` tool.

The arguments of the function are the code snippet to execute, the programming language to use (default: python), the filename (default: exploit) as well as a timeout paramerer (default: 100 seconds). In case no code is provided the function returns an error message.

8.3.8 Data Exfiltration Tools

list_dir   Given the path and some optional additional arguments, function `list_dir` lists the contents of a directory. The output is the the output of the generic `ls` command.

cat_file As the generic Linux command, `cat` displays the contents of a file. As additional argument, the path to the fiile is passed to the `cat` command.

pwd_command A function to retrieve the path of the current working directory (*pwd*). Returns the absolute path of the current working directory.

find_file Given the filepath, `find_file` Finds a file in the filesystem.

wget Given the `url` (and optional additional argumentS), the function tool `wget` downloads files from the web.

8.3.9 Command and Control Tools

run_ssh_command_with_credentials The `run_ssh_command_with_credentials` tool executes remote commands via SSH using password authentication. The function takes the remote `host` address, then SSH `username`, `password` and SSH `port` (default: 22) as function arguments.

ReverseShellClient The command and control module implemented in `command_and_control.py` provides the class `ReverseShellClient` – a reverse shell client implementation that allows an LLM control and interact with remote shells. The module provides a flexible and interactive way for the LLM to interact with remote machines. The module

- Establishes a connection to a listener on the attacker's machine

- Allows the attacker to send commands to the remote machine

- Executes the commands on the remote machine and sends the output back to the attacker

- Provides a way for the attacker to interact with the remote machine's shell

- Manages shell sessions.

> A *reverse shell client* is a type of software tool that establishes a connection from a remote machine back to the attacker's machine, allowing the attacker to interact with the remote machine's shell.

8.3.10 Network Tools

capture_remote_traffic The `capture_remote_traffic` function tool captures network traffic from a remote virtual machine. It returns a pipe (a process with `stdout`) that can be read by `tshark`.

The function inputs are the target `ip, username, password, interface, port` (default: 22) and `timeout` (default: 10 seconds), as well as optional filters.

> Tshark [38] (the non-graphical counterpart of Wireshark [39, 40]) is a powerful, command-line based network protocol analyzer. Tshark can output data in a variety of (human-readable) formats and filter the capture by protocol, IP address, port, etc. It can be used to diagnose network issues on remote servers, gather and inspect captured traffic for intrusion detection or forensics. For details on .pcap and network analysis, see, e.g., [41].

remote_capture_session The `remote_capture_session` is a context manager/function tool ton capture remote traffic capture. The tool also automatically cleans up resources. The function inputs are the target `ip, username, password, interface` and `port` (default: 22), as well as optional filters.

8.3.11 Web Tools

google_search.py   This utility mulude provides methods to perform to perform Google searches in two modes:

1. The **regular search** returns URLs from standard Google search results

2. **Google dorking** returns URLs from searches using advanced Google search operators

headers.py   This utility module analyzes HTTP requests and responses using the `web_request_framework` function. The function provides utilities for making HTTP requests and analyzing the responses from a security testing perspective, including header analysis, parameter inspection, and security vulnerability detection.

webshell_suite.py   This helper module contains utilities for web exploitation, specifically for PHP webshell and curl generation and upload.

query_perplexity   The function tools queries the *Perplexity AI* API.[42] with a user prompt and returns the query output.

> 💡 *Perplexity AI* is an AI-powered search engine and answer engine. It combines the capabilities of LLMs with real-time web search. While traditional search engines return a list of links, Perplexity summarizes the information and presents natural language answers with direct citations.

make_web_search_with_explanation   Executes an intelligent web search via the AI service for relevant cybersecurity and CTF-related information. This function sends the provided query to the internet search engine and returns the response. It also uses the full context of the current CTF challenge.

make_google_search   Performs a google search and returns a list of search results. Each result contains an URL, the title and a text snippet.

8.3.12 Miscalleneous Tools

execute_python_code   The tool executes `Python` code (the input argument) and returns the output. Optional additional inputs include context in form of a dicitonary. The funtion returns the output of the Python program.

execute_cli_command   This *Command Line Interface* aka. `CLI` command function executes shell commands and processing their output. The function argument is the command to execute, which should be concise and focused. Avoid overly verbose commands with unnecessary flags or options. It returns the formatted command output and possibly truncates it.

8.3.13 Other Tools

rag.py   A utilities module for *Retrieval Augmented Generation (RAG)* to query and add data to a vector databases. This module is used by the memory agent.

The module constists of a function tool `query_memory` to retrieve relevant context from Previous CTFs executions. The function arguments are the search `query` to find relevant documents and `top_k` (default: 3), a parameter specifying the number of top results to return. The function either returns the most relevant matches from the vector database (formatted as a string) or a warning "No documents

found in memory."

The two other function tools in the module, `add_to_memory_episodic` and `add_to_memory_semantic` add relevant data to the persistent memory.

> *Retrieval Augmented Generation (RAG)* [43] introduces an intermediary step to classical LLM inference. Rather than passing the input directly to the LLM, RAG instead uses the input to retrieve a set of relevant documents or passages from a database or corpus. The retrieved inputs are then concatenated with the original input and inputed to the LLM, which subsequenty generates the actual output. RAG thus has two sources of knowledge: the *parametric memory* (knowledge stored in the model parameters) and the *nonparametric memory* - the database from which RAG retrieves passages.

reasoning.py The `reasoning.py` utilities module provides reasoning tools for tracking thoughts, findings and analysis. Specifically, it provides utilities for recording and retrieving key information discovered during CTF progression, via the function tools

- `thought`, a tool used to express detailed thoughts and analysis during boot2root CTF;

- `write_key_findings` as well as

- `read_key_findings`; the tools to read and write key findings to a `state.txt` file to track critical informations with respect to the CTF, such as discovered credentials and vulnerabilities, privilege escalation vectors, system access details and other key findings needed for progression

think `think` is a function tool from the `reasoning.py` module. An agent, e.g. the thought agent, can use the tool to think about something. While the method cannot obtain new information or change the database, it can be used when complex reasoning or some cache memory is needed.

scripting_tool The scripting tool is a method to execute Python code directly in the memory. We advice the users to use this tool with caution since the function executes Python code directly.

Moreover, since code is directly executed, the user needs to import all the modules and libraries before use. If the command is empty or invalid or whenever potentially dangerous operations are detected an error is returned.

The function can handle the following arguments as inputs:

- Raw `Python` code

- Markdown formatted code

- Code with leading or trailing whitespace

Additional optional arguments include the usual command line arguments as well CTF context objects (required for tool interface). After the call, the function returns the output from the Python code.

8.4 Built-in Agents Available in CAI (v0.5.2)

As of Version 0.5.2, the following agents are available (see also: https://github.com/aliasrobotics/cai/tree/main/src/cai/agents). For a short introduction how to build your own agents and integrate function tools, please refer to Sections 6.1 and 8.3 for custom tool use documentation.

Agent (Nr.)	Key (Module)	Description
Blue Team Agent (1)	blueteam_agent (cai.agents.blue_teamer)	An agent that specializes in system defense and security monitoring. Expert in cybersecurity protection and incident response.
Bug Bounter (2)	bug_bounter_agent (cai.agents.bug_bounter)	An agent that specializes in bug bounty hunting and vulnerability discovery. Expert in web security, API testing, and responsible disclosure.
DFIR Agent (3)	dfir_agent (cai.agents.dfir)	DFIR Base Agent Digital Forensics and Incident Response (DFIR) Agent module for conducting security investigations and analyzing digital evidence. This agent specializes in system and network forensics, malware analysis, memory and disc forensics, evidence preservation, incident response, threat hunting as well as iimeline reconstruction.
Flag discriminator (4)	flag_discriminator (cai.agents.flag_discriminator)	An Agent focused on extracting the flag from the output. The agent calles the `CTF_Agent` if no flag is found.
CTF agent (5)	one_tool_agent (cai.agents.flag_discriminator)	A CTF Agent and profound command line tool expert, focused on conquering security challenges using generic linux commands. Expert in cybersecurity and exploitation.
DNS_SMTP_Agent (6)	dns_smtp_agent (cai.agents.mail)	The DNS SMTP Agent is a module for checking email configuration security.
Memory Agent	query_agent, semantic_builder, episodic_builder (cai.agents.memory)	Memory agent implementation realzing retrieval augmented generation (RAG) [43] for CAI. It stores long term memory in episodic and semantic formats. The episodic_builder stores *episodic memories* – chronological records of past interactions – in episodic format. The semantic_builder memorizes cross-exercise knowledge through similarity-based on historical experiences. The query_agent queries the memory system to retrieve relevant historical information from previous security assessments and CTF exercises.
Memory Analysis Specialist (7)	memory_analysis_agent (cai.agents.memory_analysis_agent)	An agent that specializes in network security analysis. Expert in monitoring, capturing, and analyzing network communications for security threats. Can call the DFIR Agent for help.

8.5 Predefined Patterns Available in CAI (v0.5.2)

As of Version 0.5.2, the following build in patterns are available in CAI. For a short introduction how to build your own patterns using agents, tools and handoffs, please refer to Section 6.4 for documentation.

Agent	Key (Module)	Description
Network Security Analyzer (8)	network_security_analyzer (cai.agents.network_traffic_-analyzer)	An agent for runtime memory analysis and manipulation. The Agent specializes in process memory examination, monitoring, and modification for security assessment, vulnerability discovery, and runtime behavior analysis.
Red Team Agent (9)	redteam_agent (cai.agents.red_teamer)	A red team base agent that specializes in bug bounty hunting and vulnerability discovery. Expert in web security, API testing, and responsible disclosure.
Replay Attack Agent (10)	replay_attack_agent (cai.agents.replay_attack_agent)	Replay attack and counteroffensive agent, specialized in network replay attacks, packet manipulation, and counteroffensive techniques for security testing and incident response. The agents objectives are to identify and exploit replay vulnerabilities, test protocol implementation security, simulate advanced persistent threats and evaluate defensive controls against replay attacks.
Reporting Agent (11)	reporting_agent (cai.agents.reporter)	The reporting agent creates professional security assessment reports in HTML.
Retester Agent (12)	retester_agent (cai.agents.retester)	An agent that specializes in vulnerability verification and triage. Expert in determining exploitability and eliminating false positives
Reverse Engineering Specialist (13)	reverse_engineering_agent (cai.agents.reverse_engineering_agent)	A reverse engineering and binary analysis. The agent specializes in firmware analysis, binary disassembly, decompilation, and vulnerability discovery using tools like Ghidra, Binwalk, and various binary analysis utilities.
Sub-GHz SDR Specialist (14)	subghz_sdr_agent (cai.agents.subghz_sdr_agent)	An agent for sub-GHz radio frequency analysis using HackRF One. The agent specializes in signal capture, replay, and protocol analysis for IoT, automotive, industrial, and wireless security applications.
Thought Agent (15)	thought_agent (cai.agents.thought)	A reasoning agent focused on analyzing and planning the next steps in a security assessment or CTF challenge.
Use Case Agent (16)	use_case_agent (cai.agents.usecase)	Agent that creates high-quality cybersecurity case studies, demonstrating how CAI tackles various security scenarios, CTF challenges, and cybersecurity exercises.
Wi-Fi Security Tester (17)	wifi_security_agent (cai.agents.wifi_security_tester)	A Wi-Fi security testing agent for wireless lan network security testing and penetration. The agent is expert in wireless attacks, password recovery, and communication disruption.

Table 9: List of Agents available in CAI

Pattern (Nr.)	Key (Module)	Description
Bug Bounty Triage Agent (18)	bb_triage_swarm_pattern (cai.agents.patterns.bb_triage)	**Swarm Pattern.** A cyclic swarm pattern for *bug bounty triage* operations. This module establishes a coordinated multi-agent system where specialized agents collaborate on vulnerability discovery and verification tasks. The pattern implements a directed graph of agent relationships, where each agent can transfer context (message history) to another agent through handoff functions, creating a complete communication network for comprehensive bug bounty and triage analysis.
Red Team Manager (19)	redteam_swarm_pattern (cai.agents.patterns.red_team)	**Swarm Pattern.** A cyclic swarm pattern for *red team operations*. This pattern establishes a coordinated multi-agent system where specialized agents collaborate on security assessment tasks. The pattern implements a directed graph of agent relationships, where each agent can transfer context (message history) to another agent through handoff functions, creating a complete communication network for comprehensive security analysis.
Offsec Pattern (20)	offsec_pattern (cai.agents.patterns.offsec)	**Parallel Pattern.** A parallel bug bounty and red team with different contexts for offensive security operations.
blue_team_red_team_share (21)	blue_team_red_team_share (cai.agents.patterns.red_blue-_team)	**Parallel Pattern.** A parallel security assessment pattern - a team of red and blue agents with *shared* context. This pattern demonstrates the use of the unified `Pattern` class for parallel agent execution, where both red and blue team agents *share* the same context.
blue_team_red_team_split (22)	blue_team_red_team_split (cai.agents.patterns.red_blue-_team_split)	**Parallel Pattern.** A parallel security assessment pattern combining red and blue team agents with *split* context. This pattern demonstrates the use of the unified `Pattern` class for parallel agent execution, where red and blue team agents operate with *separate* contexts for independent analysis.

Table 10: List of Pre-defined Patterns available in CAI

9 Development *

Development is facilitated via VS Code dev. environments. To try out our development environment, clone the repository, open VS Code and enter de dev. container mode:

9.1 Contribution

If you want to contribute to this project, use Pre-commit before submitting your merge request.

```
1   pip install pre-commit
2   pre-commit # files staged
3   pre-commit run --all-files # all files
```

9.2   Optional Requirements: caiextensions

Currently, the extensions are not available as they have been (largely) integrated or are in the process of being integrated into the core architecture. We aim to have everything converge in version 0.6.x. Coming soon!

9.3   Usage Data Collection

CAI is provided free of charge for researchers. To improve CAI's detection accuracy and publish open security research, instead of payment for research use cases, we ask you to contribute to the CAI community by allowing usage data collection. This data helps us identify areas for improvement, understand how the framework is being used, and prioritize new features. Legal basis of data collection is under Art. 6 (1)(f) GDPR – CAI's legitimate interest in maintaining and improving security tooling, with Art. 89 safeguards for research.

The collected data includes:

- Basic system information (OS type, Python version)

- Username and IP information

- Tool usage patterns and performance metrics

- Model interactions and token usage statistics

We take your privacy seriously and only collect what's needed to make CAI better. For further info, reach out to research[at]aliasrobotics.com. Users can disable some of the data collection features via the `CAI_TELEMETRY` environment variable. Nontheless, we encourage users to enable the feature and contribute back to research: `CAI_TELEMETRY=False cai`.

9.4   Reproduce CI-Setup locally

To simulate the CI/CD pipeline, you can run the following in the Gitlab runner machines:

```
1   docker run --rm -it \
2     --privileged \
3     --network=exploitflow_net \
4     --add-host="host.docker.internal:host-gateway" \
5     -v /cache:/cache \
6     -v /var/run/docker.sock:/var/run/docker.sock:rw \
7     registry.gitlab.com/aliasrobotics/alias_research/cai:latest bash
```

Acknowledgments

Licencing Agreement

This project is a combination of open-source components under the MIT License and proprietary additions licensed **for research purposes only**.

MIT-Licensed Components

Portions of this project are derived from `openai/openai-agents-python`, available under the MIT License. The original MIT-licensed code can be found in the `src/cai/agents` directory.

The full MIT License is included in the file `LICENSE-MIT` on the official GitHub repository.

Proprietary Additions

All additions, modifications, and new components authored by Alias Robotics S.L. – found in the `src/cai` – are licensed as follows:

Research-Use License

Copyright (c) [2025] Alias Robotics S.L.

Permission is granted to use, copy, and modify these components **solely for non-commercial research and academic purposes**, provided that this copyright notice and license are retained in all copies.

> ⛔ **Commercial, professional, or production use of these components is strictly prohibited without a commercial license.** To obtain a commercial license, please contact: https://aliasrobotics.com

References

[1] Víctor Mayoral-Vilches, Luis Javier Navarrete-Lozano, María Sanz-Gómez, Lidia Salas Espejo, Martiño Crespo-Álvarez, Francisco Oca-Gonzalez, Francesco Balassone, Alfonso Glera-Picón, Unai Ayucar-Carbajo, Jon Ander Ruiz-Alcalde, Stefan Rass, Martin Pinzger, and Endika Gil-Uriarte. Cai: An open, bug bounty-ready cybersecurity ai, 2025. URL https://arxiv.org/abs/2504.06017.

[2] Víctor Mayoral-Vilches. Cybersecurity ai: The dangerous gap between automation and autonomy, 2025. URL https://arxiv.org/abs/2506.23592.

[3] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. Pentestgpt: An llm-empowered automatic penetration testing tool. *arXiv preprint arXiv:2308.06782*, August 2023. URL https://arxiv.org/abs/2308.06782.

[4] Alias Robotics. Cai: Cybersecurity ai - an open bug bounty-ready artificial intelligence, 2025. URL https://github.com/aliasrobotics/cai. Accessed: 2025-06-27.

[5] Rick Dakan and Joseph Feller. Framework for ai fluency (practical summary document), 2025. URL https://ringling.libguides.com/ai/framework. Ringling.edu/ai/, Retrieved on January 2025.

[6] Rapid7. Metasploit framework, 2024. URL https://www.metasploit.com/. Accessed: 2024-04-01.

[7] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864, 2024.

[8] Benlong Wu, Guoqiang Chen, Kejiang Chen, Xiuwei Shang, Jiapeng Han, Yanru He, Weiming Zhang, and Nenghai Yu. Autopt: How far are we from the end2end automated web penetration testing? *arXiv preprint arXiv:2411.01236*, 2024.

[9] He Kong, Die Hu, Jingguo Ge, Liangxiong Li, Tong Li, and Bingzhen Wu. Vulnbot: Autonomous penetration testing for a multi-agent collaborative framework. *arXiv preprint arXiv:2501.13411*, 2025.

[10] SAE International. J3016_202104: Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. Technical report, SAE International, April 2021. URL https://www.sae.org/standards/content/j3016_202104/.

[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[12] Elvis Saravia. Prompt Engineering Guide. *https://github.com/dair-ai/Prompt-Engineering-Guide*, 12 2022.

[13] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[16] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on

large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

[17] OpenAI. Webgpt: Improving the factual accuracy of language models through web browsing, 2021. URL https://openai.com/index/webgpt/.

[18] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugging-gpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

[19] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[20] LangChain. Langgraph: A graph-based architecture for building ai applications, 2023. URL https://python.langchain.com/docs/concepts/architecture/#langgraph. Accessed through https://python.langchain.com/docs/concepts/architecture/#langgraph.

[21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[22] Microsoft. Autogen: Automating the next generation of software development, 2023. URL https://microsoft.github.io/autogen/stable/. Accessed through https://microsoft.github.io/autogen/stable/.

[23] Zhang, C., Wu and Y., Zeng, X. and et al. Autoagents: Enabling multi-agent collaboration in llms, 2023. URL https://arxiv.org/abs/2309.07864.

[24] Liang, Y. and Wu, Y.and et al. Autogen: Enabling next-gen multi-agent applications with llms, 2023. URL https://microsoft.github.io/autogen/stable/. Accessed: 2025-07-11.

[25] OpenAI. Agents, 2025. URL https://openai.github.io/openai-agents-python/agents/.

[26] J. Chomicki and N. Saeedloei. Coordinating multi-agent systems. in knowledge representation for ai planning. *Springer*, 2022.

[27] Zijie J. Wang, Dongjin Choi, Shenyu Xu, and Diyi Yang. Putting humans in the natural language processing loop: A survey, 2021. URL https://arxiv.org/abs/2103.04044.

[28] Stuart Russell and Peter Norvig. Ai a modern approach. *Learning*, 2(3):4, 2005.

[29] nmap. nmap, 2025. URL https://nmap.org/. Accessed 2025-07-11.

[30] Wireshark Developers. Wireshark, 2024. URL https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html. Accessed: 2025-07-11.

[31] Eric M Hutchins, Michael J Cloppert, Rohan M Amin, et al. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2011.

[32] curl.se. curl documentation, 2025. URL https://curl.se/docs/. Accessed 2025-07-11.

[33] Linux Command. Linux command, 2025. URL https://linuxcommand.org/. Accessed 2025-07-11.

[34] netcat man page. netcat man page, 2025. URL https://linux.die.net/man/1/nc. Accessed 2025-07-11.

[35] netstat man page. netstat man page, 2025. URL https://man7.org/linux/man-pages/man8/netstat.8.html. Accessed 2025-07-11.

[36] Shodan API. Shodan api, 2025. URL https://developer.shodan.io/api. Accessed 2025-07-11.

[37] Shodan Search. Shodan search, 2025. URL https://www.shodan.io/search. Accessed 2025-07-11.

[38] Wireshark Developers. Tshark, 2024. URL https://www.wireshark.org/docs/man-pages/tshark.html. Accessed: 2025-07-11.

[39] Wireshark Developers. Wireshark, 2024. URL https://www.wireshark.org/. Accessed: 2025-07-11.

[40] Guy Harris. Wireshark: A practical introduction to packet analysis. *IEEE Network*, 20(6):38–43, 2006. doi: 10.1109/MNET.2006.1688083.

[41] Chris Sanders. *Practical packet analysis: Using Wireshark to solve real-world network problems*. no starch press, 2017.

[42] Perplexity AI. Perplexity api documentation, 2022. URL https://docs.perplexity.ai/home. Accessed: 2025-07-11.

[43] Meta AI. Retrieval-augmented generation: Streamlining the creation of intelligent natural language processing models, 2023. URL https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-natural-language-proces Accessed through https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creatio

[44] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *14th USENIX Security Symposium*, pages 113–128, 2005.

[45] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[46] Stuart M Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, 1985.

[47] Yoav Goldberg. *Neural network methods for natural language processing*. Morgan & Claypool Publishers, 2017.

[48] Frank Rosenblatt. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Spartan Books, 1962.

[49] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[50] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[51] Shun-ichi Amari. Mathematical foundations of neurocomputing. *Proceedings of the IEEE*, 78(9):1443–1463, 1990.

[52] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5 (4-5):185–196, 1993.