ARL-TR-10489 ● Nov 2026



DEVCOM ARMY RESEARCH LABORATORY

ModelScope: Evaluation Platform for Autonomous AI Agents

by Albert Yoshimoto and Jaime C. Acosta

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-10489 ● Nov 2026





ModelScope: Evaluation Platform for Autonomous AI Agents

Albert Yoshimoto
University of Texas at El Paso

Jaime C. Acosta
DEVCOM Army Research Laboratory

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
November 2026	Technical Report	6/1/2026	7/31/2026

4. TITLE AND SUBTITLE
ModelScope: Evaluation Platform for Autonomous AI Agents

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Albert Yoshimoto and Jaime C. Acosta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLA-ND
Adelphi, MD 20783

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-10489

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Jaime C. Acosta, 0000-0003-2555-9989

14. ABSTRACT
Autonomous AI systems must be evaluated as multistep, tool-using workflows rather than as isolated text generators. This report describes ModelScope, a project-based platform for configuring, executing, validating, and inspecting local or remote AI agents. ModelScope uses discoverable bot-type plugins to isolate bot-specific defaults, interfaces, execution logic, metric catalogs, and dashboards. Its current bot-types support shell commands, llama.cpp command-line and managed-server inference, CyberAgentFlow integration, and multi-container Proxmox Linux containers (LXC) batch evaluation. Correctness is established through named validation sets with command or prompt steps and explicit accepted-output checks; operational telemetry is evaluated by configurable dashboard bands. The platform preserves sanitized configurations, transcripts, tool traces, telemetry, and run history for audit and comparison. The report emphasizes engineering design and development progress. Earlier measurements affected by harness defects are excluded, and quantitative benchmarking is reserved for controlled post-fix trials with complete configuration and validation records.

15. SUBJECT TERMS
cybersecurity; autonomous AI agents; benchmarking; Model Context Protocol; llama.cpp; CyberAgentFlow; validation; Network, Cyber, and Computational Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	21

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Jaime C. Acosta	(915) 747-8012

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures v

List of Tables v

1. Introduction 1

2. Background and Design Requirements 2

3. Current ModelScope Architecture 2

3.1 Interfaces and Project Lifecycle 2

3.2 Plugin Model 3

3.3 Current Bot Types 4

4. Evaluation Workflow 4

4.1 Configuration and Execution Phases 4

4.2 Validation Sets 5

4.3 Correctness vs. Operational Labels 6

5. Model and Tool Execution 6

5.1 llama.cpp Runtimes 6

5.2 Execution Environments 7

5.3 MCP and Tool-Calling Compatibility 7

6. Observability and Analytical Dashboard 7

6.1 Plugin-Owned Metrics 7

6.2 Metric Bands and Artifacts 8

7. Proxmox LXC Batch Evaluation 9

8. Development Progression and Contributions 9

9. Limitations, Future Work, and Conclusion 10

9.1 Limitations 10
9.2 Future Work 11
9.3 Conclusion 11

10. References 12

List of Symbols, Abbreviations, and Acronyms 13

Distribution List 14

List of Figures

Figure 1. Project creation establishes a persistent evaluation unit and selects the bot-type plugin that owns its configuration and execution behavior. ... 3

Figure 2. Current ModelScope architecture. A project selects a plugin, which binds an execution environment to a model or agent, applies validation, records telemetry, and renders dashboard evidence. 4

Figure 3. Configuration of a) least-privilege MCP tool permissions and b) project prompts. Tool access and task instructions are versioned with the project..... 5

Figure 4. Validation and lifecycle controls: a) an accepted-output check supplies objective task evidence and b) completion commands perform post-run collection or cleanup. 5

Figure 5. Runtime configuration for a) local, SSH, or Proxmox PCT execution and b) a managed llama-server model with explicit context and sampling settings. 6

Figure 6. Metric-threshold configuration. Plugins expose only supported observations, and projects define the hard-fail through hard-pass interpretation bands. 8

Figure 7. Post-run evidence: a) execution detail retains validation and trace context and b) the analytical dashboard summarizes validation separately from plugin-owned operational metrics..... 9

List of Tables

Table 1. Design considerations informing current direction..... 2

Table 2. Current bot types. 4

Table 3. Schedule and milestones. 10

1. Introduction

AI evaluation is increasingly concerned with systems that plan, invoke tools, interpret intermediate results, and revise their behavior over several turns. A conventional benchmark can score a static response, but it does not necessarily reveal whether an autonomous agent selected the right tool, remained within an operational boundary, completed cleanup, or produced evidence that a task succeeded. ModelScope[1] was developed to make those behaviors observable and reproducible in local, remote, and resource-constrained environments.

The original internship assignment was to research evaluation frameworks and implement a graphical platform that could configure and execute evaluations of large language models (LLMs), prompts, Model Context Protocol (MCP)-enabled agents, retrieval-augmented generation workflows, and related AI systems. The platform was also expected to capture execution logs and artifacts, measure performance and task completion, and generate information useful for comparative research.

The internship objectives were operationalized as the following engineering goals:

- Provide a graphical interface for configuring repeatable evaluation projects.
- Support more than one model or agent execution pattern without embedding every integration into the core application.
- Preserve prompts, commands, validation evidence, tool activity, errors, timing, and model telemetry.
- Distinguish objective task success from descriptive performance thresholds.
- Permit execution on a workstation, a remote host, or controlled Proxmox Linux containers.
- Produce run histories and exports that can support later comparative analysis.

ModelScope changed substantially during June and July 2026. The most important result was therefore not a single benchmark score, but the refinement of the platform itself—an early scenario- and metric-centric prototype was replaced by a persistent project model and a discoverable plugin architecture. This report documents that current implementation. Earlier performance figures affected by platform defects are not treated as benchmark findings.

2. Background and Design Requirements

Early work reviewed DeepEval, Promptfoo, Ragas, TruLens, LangSmith evaluation features, and Phoenix. These systems contributed useful concepts for assertion-based testing, retrieval evaluation, traces, experiment tracking, and observability.[2–6] ModelScope did not attempt to reproduce all these products. Its design emphasis became local agent execution, explicit command and prompt validation, llama.cpp telemetry, MCP tool restrictions, and deployment environments that can be isolated from cloud services.

Six design considerations informed the current direction: flexible evaluation execution, portable project configuration, broad support for model and agent integrations, observable validation of success claims, comprehensive telemetry, and an extensible integration model (Table 1). Together, these additions could allow evaluations to run locally or through remote adapters; help recreate projects without carrying over volatile session state; support direct models, command-line bots, managed servers, MCP tools, and CyberAgentFlow (CAF); ground success claims in observable validation rather than evaluator impressions alone; capture both model-processing and runtime behavior; and avoid hard-coding each bot into the Streamlit application.[7,8]

Table 1. Design considerations informing current direction.

Design considerations	Current ModelScope response
Local and remote execution	Local, SSH, and Proxmox PCT/LXC environment adapters
Reproducible configuration	Project files, autosave, duplication, and import/export
Multiple agent forms	Six bot-type plugins spanning shell, llama.cpp, CAF, and ProxBatch
Objective correctness	Named validation sets with explicit accepted-output checks
Operational observability	Run artifacts plus plugin-owned timing, token, throughput, and server-load metrics
Extensibility	Discovery from built-ins, plugin directories, Python entry points, and configured paths

Notes: SSH = secure shell; PCT = Proxmox container toolkit command and execution interface; LXC = Linux containers

3. Current ModelScope Architecture

3.1 Interfaces and Project Lifecycle

The current Streamlit application presents three primary interfaces: Configuration, Execute, and Analytical Dashboard. Configuration defines the project, selected bot type, execution environment, model or command runtime, prompts, permissions, validation sets, and dashboard thresholds. Execute runs the project and shows phase

and trace activity. Analytical Dashboard interprets preserved run records, metric bands, and plugin-specific views.

A project is the unit of reproducibility. Users can create, name, duplicate, import, export, and autosave projects. Project configuration is hydrated into session state only for the active project and bot type (Figure 1). Execution preserves the project configuration with run results while sanitizing sensitive values. History views allow completed sessions to be revisited without rerunning the agent. This lifecycle replaced the prototype's emphasis on choosing a built-in scenario from a registry.

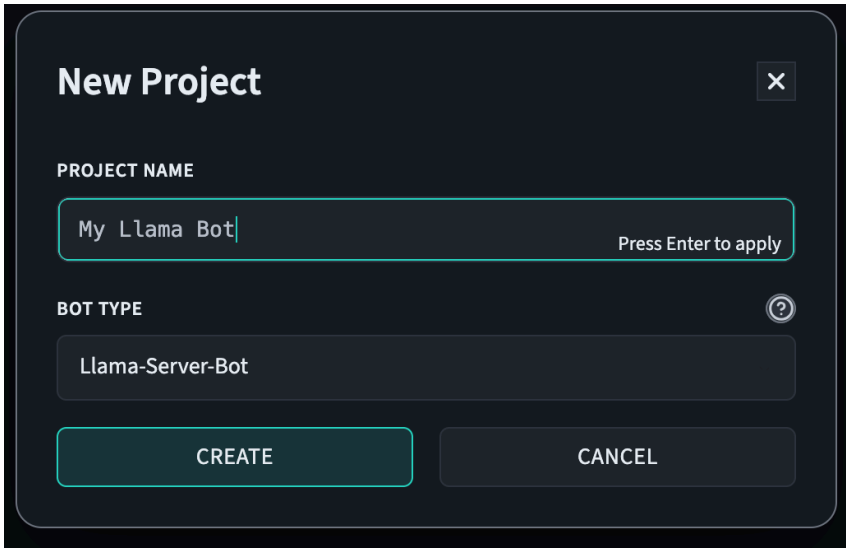


Figure 1. Project creation establishes a persistent evaluation unit and selects the bot-type plugin that owns its configuration and execution behavior.

3.2 Plugin Model

Bot integrations implement the BotTypePlugin contract. A plugin supplies identifying metadata, project defaults, state hydration, configuration and execution rendering, an evaluator, metric specifications, and optional result or dashboard hooks. This boundary keeps bot-specific behavior outside the central application. It also allows each integration to define only the telemetry that it can measure credibly.

The registry discovers built-ins in the core bot-types package, locally installed plugins, Python entry points in the modelscope.bot_types group, user plugin directories, and paths configured through the MODELSCOPE_BOT_PLUGIN_PATH environment variable. Defaults, widget state, evaluator state, and dashboards remain isolated by project and bot type. Figure 2 shows the current data path.

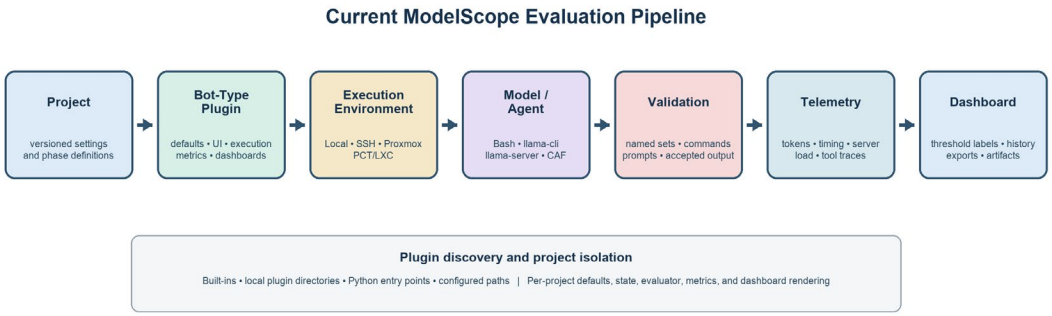


Figure 2. Current ModelScope architecture. A project selects a plugin, which binds an execution environment to a model or agent, applies validation, records telemetry, and renders dashboard evidence.

3.3 Current Bot Types

The July 2026 source tree exposes six active bot types (Table 2).

Table 2. Current bot types.

Bot type	Execution purpose	Metric ownership
Bash-Bot	Runs configured command workflows through an execution environment	No dedicated metric catalog
Llama-CLI-Bot	Invokes llama-cli for prompt and command workflows	Common run metrics
Llama-Server-Bot	Starts or connects to an OpenAI-compatible llama-server	Common plus token and server metrics
CAF Standard	Executes the standard CyberAgentFlow command-line integration	Common run metrics
CAF + llama.cpp	Combines CAF workflow execution with managed llama.cpp inference	Common plus server metrics
Llama-Server-ProxBatch	Runs one managed server workflow across selected Proxmox containers	Per-container and roll-up metrics

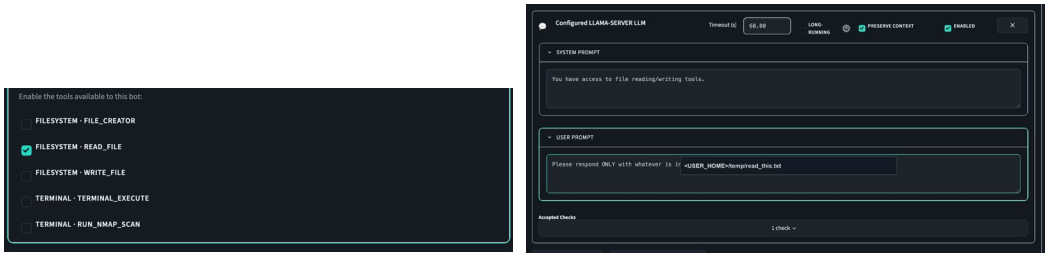
Several prototype elements are intentionally absent. The built-in scenario registry and scenario presets, the generic batch runner, the Comparison tab, the Target tab, and standalone CAF configuration and dashboard tabs were removed. CAF is now represented through bot-type plugins, and batch evaluation is implemented by Llama-Server-ProxBatch rather than by a generic command.

4. Evaluation Workflow

4.1 Configuration and Execution Phases

A user selects a bot type and execution target, configures the model or command runtime, restricts MCP tool permissions, and defines prompts and system instructions (Figure 3). Every project can organize work into Startup, Validation,

and Completion phases. Startup prepares the environment or launches dependencies. Validation performs the commands and prompts that establish task success. Completion performs final collection or cleanup. The evaluator preserves phase timing, stdout and stderr, prompts, responses, tool calls, validation decisions, exceptions, cancellation state, and available telemetry.



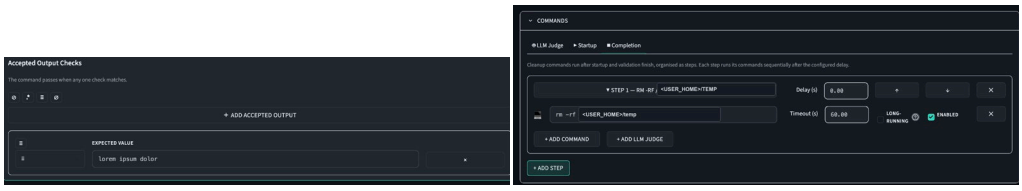
a) b)

Figure 3. Configuration of a) least-privilege MCP tool permissions and b) project prompts. Tool access and task instructions are versioned with the project.

4.2 Validation Sets

Validation sets are the principal correctness mechanism (Figure 4). A project can contain multiple named sets, each with enabled or disabled command and prompt steps. Steps may include a timeout, a delay, and cleanup behavior. Output checks can require an exact string, a regular expression match, no output, or ignored output. An ignored output is useful when the step is operational but not itself evidence of success. Multiple actionable checks represent acceptable alternatives, enabling a validation step to recognize more than one correct form without weakening the underlying criterion.

Cancellation is tracked separately from failure so that an aborted run is not misreported as a completed negative result. Similarly, skipped validation, missing output, and cleanup errors are retained as distinct conditions. This distinction is necessary for later data-quality screening.



a) b)

Figure 4. Validation and lifecycle controls: a) an accepted-output check supplies objective task evidence and b) completion commands perform post-run collection or cleanup.

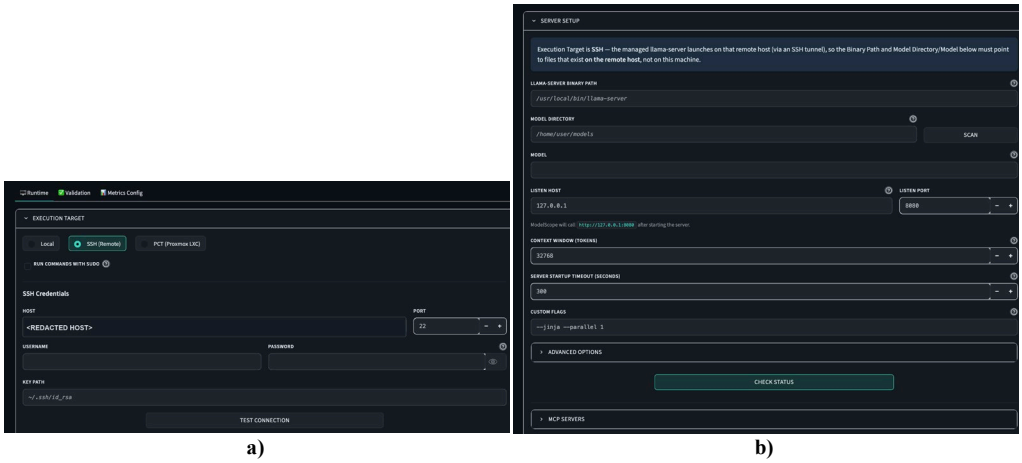
4.3 Correctness vs. Operational Labels

Validation and dashboard thresholds answer different questions. Validation asks whether the intended task was completed under the configured checks. Dashboard bands ask how a completed run behaved; for example, whether latency or throughput fell into an expected operating range. A favorable latency band cannot turn a failed validation into a successful task, and a slow but correctly completed run remains a validation success with an unfavorable performance label. Keeping these decisions separate prevents operational preferences from becoming implicit ground truth.

5. Model and Tool Execution

5.1 llama.cpp Runtimes

ModelScope supports direct llama-cli execution and managed llama-server execution.[8] Projects can select a generative pretrained transformer (GPT)-generated unified format (GGUF) model, context size, sampling parameters, and server options (Figure 5). Model discovery assists with selecting available artifacts, but the selected model path and relevant controls are preserved in project state. Managed-server integrations launch the server, poll its health endpoint, execute OpenAI-compatible requests, capture Prometheus telemetry when available, and tear the service down at the end of the workflow. Cancellation and recovery paths attempt to stop subprocesses and avoid leaving stale services behind.



a) b)

Figure 5. Runtime configuration for a) local, SSH, or Proxmox PCT execution and b) a managed llama-server model with explicit context and sampling settings.

5.2 Execution Environments

The environment abstraction provides local execution, SSH execution, and Proxmox PCT execution. Local projects run commands and model processes on the ModelScope host. SSH projects use a remote shell while preserving the same evaluator interface. PCT projects invoke commands in a selected Linux container through the Proxmox host. Environment selection is therefore part of the project configuration rather than a separate Target tab.

5.3 MCP and Tool-Calling Compatibility

For MCP-enabled workflows, ModelScope discovers a server's tool catalog and allows the project to grant a restricted subset of tools. This least-privilege selection becomes evaluation metadata, making it possible to determine whether two runs had the same capabilities.[9] The client supports native OpenAI-compatible *tool_calls* messages and a fallback <*tool_call*> representation emitted by some local models.

Compatibility work at the end of the reporting period addresses models that place tool-call structures in a reasoning channel and models that require exact assistant/tool-message sequencing. The engineering goal is to recover a valid call without presenting internal reasoning as the final answer, append the assistant message containing tool_calls before the role=tool result, and provide diagnostics for empty streamed responses. These are platform improvements under active verification; they are not reported here as benchmark findings.

6. Observability and Analytical Dashboard

6.1 Plugin-Owned Metrics

ModelScope uses a plugin-based approach to evaluation metrics: each bot-type plugin declares the observations its evaluator can produce. Common run metrics include total latency and the number of prompts and commands executed. Managed llama-server plugins can additionally report prompt, completion, and total tokens; prompt and generation throughput; prompt and completion processing time; active and deferred requests; context high-water mark; decode calls; and busy slots per decode. When available from the underlying endpoint, these server values are recorded as run-scoped observations.

6.2 Metric Bands and Artifacts

For each applicable metric, the user can configure hard-fail, soft-fail, soft-pass, and hard-pass bands (Figure 6). The dashboard compares a completed observation with those inclusive boundaries and displays the resulting label. Bands are interpretive metadata; they do not modify validation, rerun the evaluator, or determine the process exit status.

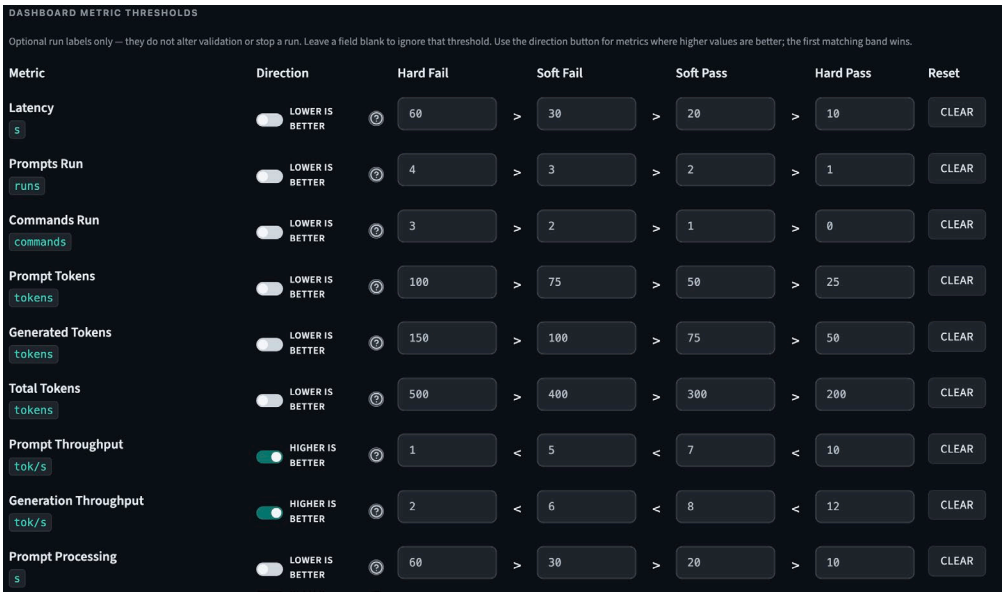
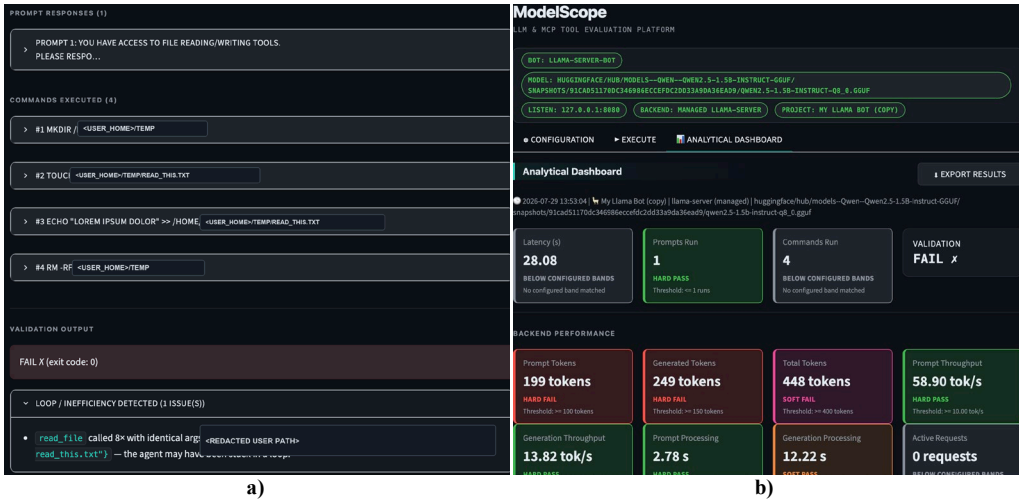


Figure 6. Metric-threshold configuration. Plugins expose only supported observations, and projects define the hard-fail through hard-pass interpretation bands.

A session preserves a sanitized configuration snapshot, phase results, timestamps, transcripts, tool traces, validation evidence, errors, and available metrics. Run history and dashboard exports provide the audit trail needed to compare controlled trials. Sensitive values are removed from exported configuration and screenshots included in this report (Figure 7) were selected from a sanitized quick-start guide rather than from operational hosts.



a) b)

Figure 7. Post-run evidence: a) execution detail retains validation and trace context and b) the analytical dashboard summarizes validation separately from plugin-owned operational metrics.

7. Proxmox LXC Batch Evaluation

Llama-Server-ProxBatch provides ModelScope its batch architecture. It is implemented as a bot-type plugin rather than as a generic command-line batch runner. The plugin scans the local Proxmox PCT inventory, excludes templates, and lets the user select multiple Linux containers plus a master container used for setup and configuration checks. It starts a managed llama-server inside each selected container, waits for service health, and applies the same Startup–Validation–Completion workflow to every container.[1,10]

Batch concurrency is configurable. The Execute interface reports per-container status and prefixed progress, supports cancellation, and permits failed containers to be retried. Each container retains its own validation result, prompts, tool activity, errors, timing, tokens, and available server metrics. The roll-up aggregates the container records while preserving the individual evidence needed to identify configuration mismatches or isolated failures.

8. Development Progression and Contributions

The platform progressed from early research and feasibility work into an integrated, extensible evaluation environment (Table 3). Initial efforts established the evaluation concepts, execution patterns, interface direction, and local model workflows. Subsequent development introduced project persistence, execution logging, remote connectivity, and agent integration support, followed by refinements to usability, session isolation, lifecycle management, and recovery

behavior. The current architecture is organized around a collaborative plugin model, with multiple bot types, plugin-owned metrics, and remote batch execution support.

Table 3. Schedule and milestones.

Period (2026)	Platform state and contribution
Early June	Requirements research, evaluation concepts, GUI reviews, llama.cpp setup, and MCP workflow exploration
June–early July	Initial GUI, evaluator, execution logging, SSH support, CAF integration, and project vision
Mid July	Project persistence, autosave behavior, imported/exported configuration, and interface corrections
Late July	CAF + llama.cpp bot development, light-mode and usability work, lifecycle/session isolation, and llama.cpp recovery fixes.
Current state	Collaborative plugin refactor, six bot types, plugin-owned metrics, and Proxmox LXC ProxBatch execution

This progression also refined the evaluation model. Earlier concepts emphasized built-in scenarios and a broad shared metric registry. The current design instead centers on projects, plugin-defined behavior, named validation sets, and preserved run evidence. This reduces coupling between the interface and individual integrations, lets new execution targets follow a consistent lifecycle, and ensures that metrics are presented only when they are meaningfully supported.

9. Limitations, Future Work, and Conclusion

9.1 Limitations

The principal limitation is limited end-to-end experimental coverage after the most recent fixes. Local models differ in their tool-call formatting, reasoning-channel behavior, streaming semantics, and tolerance for assistant/tool-message ordering. Server telemetry can also be absent or differ by llama.cpp build. Proxmox containers make parallel comparison practical, but container configuration, model placement, CPU contention, and retry history must be controlled before performance differences can be attributed to a model.

Validation sets provide objective checks only to the extent that the expected outputs represent adequate ground truth. Exact strings and regular expressions are useful for deterministic tasks but may be insufficient for open-ended analysis. The platform records hardware- and runtime-oriented telemetry, but broader operating-system utilization measurements and calibrated semantic judgments remain future work.

9.2 Future Work

The next research phase should freeze a source revision and execute repeated controlled trials across a wider set of GGUF models and quantizations. Trials should record processor and memory allocation, model-server flags, container images, tool catalogs, and validation ground truth. Analysis should include variance, failure-mode classification, cross-container reproducibility, and sensitivity to concurrency. Hardware utilization data should be integrated with the current latency, token, throughput, and server-load observations.

CyberAgentFlow should remain a supported integration target whose sessions can be evaluated through the same project and validation abstractions. ScenarioForge may later provide reproducible Common Open Research Emulator (CORE) network topologies and seeded cyber-range conditions for ModelScope evaluations. That integration is future work; ScenarioForge is not part of the current ModelScope execution architecture described in this report.

9.3 Conclusion

ModelScope now addresses the central internship objectives through a configurable graphical interface, a persistent project lifecycle, six extensible bot types, local and remote execution adapters, explicit MCP permissions, observable validation, preserved session artifacts, plugin-owned telemetry, dashboard bands, and Proxmox LXC batch execution. Its strongest result is an auditable engineering foundation for evaluating autonomous AI workflows while keeping correctness separate from operational performance.

The platform is not yet a completed comparative benchmark study. By excluding unreliable early measurements and defining the evidence required for post-fix trials, this report establishes a more defensible boundary between platform development and empirical claims. The current architecture supports the next step: repeated, controlled evaluation of local tool-using agents with complete validation and telemetry records.

10. References

1. ModelScope source code and project documentation. Local development repository. GitHub, Inc.; 2026.
2. Ip J, Vongthongsri K. confident-ai/deepeval. Ver. 4.2.2 [computer software]. GitHub, Inc.; 2026 [accessed 2026 July 29]. https://github.com/confident-ai/deepeval
3. Promptfoo intro. Promptfoo, Inc.; last updated 2026 Sep 8 [accessed 2026 July 29]. https://www.promptfoo.dev/docs/
4. Exploding Gradients. Ragas [documentation]. 2026 [accessed 2026 July 29]. https://docs.ragas.io/
5. Evaluate and trace AI agents. TruLens; c2026 [accessed 2026 July 29]. https://www.trulens.org/
6. Yoshimoto A. Friday status report, internal internship status report. University of Texas at El Paso (US); 2026 June 5.
7. Streamlit documentation. Snowflake Inc.; c2026 [accessed 2026 July 29]. https://docs.streamlit.io/
8. Gerganov G. llama.cpp: LLM inference in C/C++ [internet]. GitHub, Inc.; 2023 [accessed 2026 July 29]. https://github.com/ggml-org/llama.cpp
9. Model Context Protocol. Specification [internet]. Model Context Protocol a Series of LF Projects, LLC; 2024 [updated 2026 July 28; accessed 2026 July 29]. https://modelcontextprotocol.io/specification/
10. pct(1) Proxmox Server Solutions GmbH [software]. Proxmox Server Solutions GmbH; last updated 2026 Sep 18 [accessed 2026 July 29]. https://pve.proxmox.com/pve-docs/pct.1.html

List of Symbols, Abbreviations, and Acronyms

AI	Artificial Intelligence
CAF	CyberAgentFlow
CORE	Common Open Research Emulator
CPU	Central Processing Unit
GGUF	GPT-Generated Unified Format
GPT	Generative Pretrained Transformer
GUI	Graphical User Interface
LLM	Large Language Model
LXC	Linux Containers
MCP	Model Context Protocol
PCT	Proxmox Container Toolkit Command and Execution Interface
SSH	Secure Shell

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CB TECH LIB