



# Vectra Guard: Sandboxing & Execution Control Design

## Feature Overview

Vectra Guard enforces **container-based sandboxing** for all risky CLI executions. Every script or agent-driven command runs in an isolated container (Docker/Podman or similar), which fully separates it from the host system [1](#) [2](#). This follows best practices (e.g. Google's kubectl-AI notes that "isolating each agent's commands in its own container" protects the host and ensures consistent behavior [1](#)). Containers provide cross-platform, process-level isolation [2](#) and allow packaging all required tools/dependencies inside, improving portability and reproducibility [1](#) [2](#).

Within the sandbox, users can run **whole shell scripts or interactive agent sessions**. Vectra Guard limits filesystem and environment visibility to only what is explicitly mounted or passed in. (By default, only the project directory or specified mounts are visible; the rest of the host FS is hidden or read-only.) This goes beyond typical agent shells that sandbox only to the current directory [3](#): Vectra Guard can custom-isolate arbitrary directories and env vars per session.

All executions are fully **logged and tagged**. Every command, its stdout/stderr, and exit code are captured locally. For traceability, Vectra Guard attaches structured metadata (agent name or user, script path, invocation source, start/end time, working directory, etc.) to each session. (This mirrors designs like Gemini CLI's audit log – "records every AI-initiated command, stored locally per session" [4](#).) Logs and metadata are organized by session (and optionally by agent), so one can easily trace back who ran what, when, and where.

By default, sandboxing aims for minimal overhead and high confidence. Base container images can be cached or reused to speed startup. Users enable sandbox mode with simple CLI flags or config (e.g. a `--sandbox` switch, similar to Gemini's `--sandbox` option [5](#)). In sandbox mode, Vectra Guard transparently intercepts each command, runs it in the container, logs it, and then returns output to the user. This gives developers a reproducible, "clean" execution environment (with only declared mounts and deps) [6](#) while protecting the host system from harmful side effects.

## Supported Workflows (Examples)

- **Run a shell script in isolation:** A user can execute an existing script entirely inside a sandbox. For example:

```
# Run script.sh inside a Ubuntu 22.04 sandbox, mounting the project dir
# as /workspace
vectraguard sandbox run --image ubuntu:22.04 \
--mount /home/alice/project:/workspace \
```

```
--env NODE_ENV=production \
script.sh
```

Vectra Guard will launch a container from `ubuntu:22.04`, mount `/home/alice/project` into `/workspace` (read-write by default), set `NODE_ENV=production`, and run `script.sh`. All commands executed by the script, plus their outputs and errors, are logged locally with the session's metadata. The host outside `/home/alice/project` remains inaccessible inside the container.

- **Interactive agent or shell session:** A developer or AI agent can drop into an interactive sandboxed shell. For example:

```
# Start an interactive Debian shell named "guard-shell" with no network
access
vectraguard sandbox shell \
--image debian:bookworm \
--name guard-shell \
--no-network \
--mount /home/alice/project:/workspace
```

This command opens a new shell inside a container (`debian:bookworm`). Only `/workspace` is writeable (mapping from the host project) and no external network traffic is allowed (`--no-network`). The user or agent can run commands freely inside, but any filesystem changes (outside `/workspace`) are confined to the container. All input and output are logged.

- **Agent-driven workflows:** Vectra Guard integrates with CLI-based AI agents. For instance, an AI assistant can be started in sandbox mode:

```
# Launch AI agent session "dev-assistant" with its own isolated environment
vectraguard agent start --name dev-assistant \
--prompt "Refactor the code for performance" \
--mount /home/alice/project:/workspace
```

The tool will create a sandbox container (using a default or specified image) and run the agent loop inside it. Every command the agent issues is run in that container, with the agent's name "dev-assistant" logged in metadata. The agent can read/write only in `/workspace` (the mounted project), and cannot escape to other host paths. The entire session (commands, outputs) is recorded under a unique session ID for "dev-assistant".

Each workflow above can control **mount points** (read-only or read-write) and **environment variables**. Users may specify which host paths to expose and which to hide, giving precise control over the sandboxed view. Vectra Guard can also optionally clone a Git repo inside the container or load files via a setup script, ensuring that the workspace inside the container is reproducible and isolated <sup>6</sup>.

## Container and Isolation Model

- **Container runtime:** Vectra Guard relies on Docker/Podman or equivalent. For each session it launches a fresh container (or reuses a warm container) from a specified image. The container runs as an unprivileged user (not root) whenever possible, and Linux capabilities are minimized. This enforces OS-level isolation: network namespaces, PID namespaces, and mount namespaces ensure the process sees only the container's view. As Gemini CLI notes, container-based sandboxes provide "complete process isolation" <sup>2</sup>.
- **Filesystem mounts:** By default, no host filesystem is visible inside the container except explicitly bound volumes. Typically the project directory is mounted read-write; everything else (e.g. `/`, `/etc`, host home) is either not mounted or mounted read-only. Users can add additional `--mount hostPath:containerPath[:ro]` flags to expose more paths. For example, common read-only mounts might include shared libraries or tool directories. Any path not mounted is effectively "hidden" from the sandbox process. This aligns with the principle of "deny-all, allow only specified paths" for maximum safety.
- **Environment isolation:** The container's environment (PATH, env vars, shell, etc.) is separate from the host's. Vectra Guard sets a clean default env inside the container; it then injects only whitelisted variables (via `--env KEY=VAL` flags or an env-file) into the sandbox. Secrets or host credentials are not passed by default. This prevents an agent from seeing sensitive host env vars.
- **Network policies:** By default, sandbox containers have **no outbound network**. (Alternatively, Vectra Guard can create a docker network with restrictive firewall rules.) If needed, specific domains or subnets can be whitelisted using internal iptables inside the container (similar to the Claude Code devcontainer example, which "drops all traffic by default, then whitelists specific domains" <sup>7</sup>). This ensures agents or scripts cannot ping arbitrary external endpoints unless explicitly allowed.
- **Portability and tooling:** Each sandbox image should bundle all necessary CLI tools (git, bash, curl, etc.). As one GitHub issue observes, packaging tools like `kubectl`, `jq`, etc. inside the container "improve portability" and consistency <sup>1</sup>. Vectra Guard can come with a set of common sandbox images, or allow users to bring custom images. The container's OS (Debian, Alpine, etc.) can be chosen to match the developer's needs.
- **Ephemeral containers:** After a session ends, Vectra Guard stops and removes the container (unless a user opts to pause/save state). This guarantees no leftover processes. All persistent state is written to the mounted volumes or to the logged outputs on the host. The containers themselves are disposable.

## Metadata Tagging Format

Vectra Guard collects rich metadata for each session to aid auditing. For each run or session, it creates a **metadata record** (e.g. a JSON or YAML file) with fields such as:

```
{
  "session_id": "20251219T101053Z-9af4c3",
  "agent": "dev-assistant",
  "script": "script.sh",
  "origin": "vectraguard-cli",
  "start_time": "2025-12-19T10:10:53Z",
  "end_time": "2025-12-19T10:12:10Z",
  "cwd": "/workspace",
  "user": "alice",
  "image": "ubuntu:22.04",
  "mounts": ["/home/alice/project:/workspace"],
  "description": "Nightly maintenance script"
}
```

Key fields include: a unique `session_id` or timestamp; the `agent` name or user initiating it; the main `script` or command file; the invocation `origin` (CLI or agent); start/end timestamps; the container `image` used; and any relevant context (current working directory, user, mount info, etc.). These metadata entries are written at the start (and updated at the end) of each sandbox session. All log files produced in that session reference this session ID. This structured tagging ensures one can quickly answer questions like “which agent ran this command and when” or “which session produced these outputs.”

## File/Log Structure

Logs are stored on the host in a directory tree under a standard Vectra Guard folder (for example `~/.vectraguard/logs/` or a project-specific `.vectraguard/` directory). Each session gets its own subfolder (named by timestamp or ID). For example:

```
~/.vectraguard/logs/
├── session-20251219T101053Z-9af4c3/
│   ├── metadata.json      # (metadata as above)
│   ├── commands.log       # list of commands run in order
│   ├── stdout.log         # combined stdout of all commands
│   └── stderr.log         # combined stderr of all commands
└── session-20251220T083015Z-7b2a8f/
    ├── metadata.json
    ├── commands.log
    ├── stdout.log
    └── stderr.log
...
...
```

Within each session folder, `commands.log` contains a timestamped record of every shell command executed, in the order run. The `stdout.log` and `stderr.log` files capture the standard output and error streams, respectively, from the entire session. (Alternatively, commands and outputs could be

interleaved in one log file, but separate logs make parsing easier.) This design follows the audit practice of Gemini CLI, which “records every AI-initiated command... stored locally per session” <sup>4</sup>.

This structure allows quick access by session ID. (Optionally, logs could also be organized under agent directories: e.g., `~/.vectraguard/logs/dev-assistant/<session>/...`.) No external log server is needed – everything is local. The logs and metadata are human-readable (JSON/text) and timestamped for debugging and audits.

## CLI UX (Sample Commands & Flags)

Below are illustrative CLI commands and flags for Vectra Guard’s sandbox mode:

```
# 1. Run a script in a sandbox with custom image and mounts:  
vectraguard sandbox run [options] <script-path>  
# Example:  
vectraguard sandbox run \  
  --image ubuntu:22.04 \  
  --mount /home/alice/project:/workspace \  
  --mount /tmp/logs:/tmp/logs:ro \  
  --env FILESERVER_URL=https://example.com \  
  --network=none \  
script.sh  
  
# 2. Start an interactive sandbox shell (with specified mounts and no network):  
vectraguard sandbox shell [--image IMAGE] [--mount HOST:CONT[:ro]] [--name  
SESS_NAME] [--read-only /path] [--no-network]  
# Example:  
vectraguard sandbox shell \  
  --image debian:bookworm \  
  --mount /home/alice/project:/workspace \  
  --read-only /etc \  
  --name guard-shell \  
  --no-network  
  
# 3. Launch an AI agent session in sandbox mode:  
vectraguard agent start --name <agent-name> --prompt "<instructions>" [--  
sandbox] [--image IMAGE] [--mount ...]  
# Example:  
vectraguard agent start --name dev-assistant \  
  --prompt "Audit the code for security issues" \  
  --sandbox \  
  --image python:3.10-slim \  
  --mount /home/alice/project:/workspace  
  
# 4. Review logs of past sessions (future feature):
```

```
vectraguard logs list  
vectraguard logs show session-20251219T101053Z-9af4c3
```

- `--image` : specifies the container image to use (default could be a built-in base like Alpine or Ubuntu).
- `--mount` : bind-mounts a host path into the container (append `:ro` for read-only).
- `--env` : passes an environment variable into the sandbox.
- `--no-network` (or `--network=none`) : disables network inside the sandbox by default.
- `--name` : names the session for easier reference.
- `--read-only` : marks a mount point as read-only if supported.

These commands are designed to be intuitive to developers (similar to `docker run` flags). They ensure that sandbox mode is opt-in (e.g. via a `sandbox` subcommand or `--sandbox` flag) and configurable per run.

## Security Considerations and Guardrails

- **Strong isolation:** By default, each sandbox container runs as a non-root user with no special privileges. Container capabilities like `SYS_ADMIN`, `NET_ADMIN`, etc., are dropped. Users should not run containers in privileged mode. This prevents containers from mounting or tampering with host resources. As noted in the context of agent shells, simple workarounds should not break the isolation <sup>3</sup>.
- **Filesystem restrictions:** The default policy is “deny everything, allow only needed paths.” No sensitive host directories (e.g. `/etc`, `/var/lib`, `/home/otheruser`) are mounted. Write access is allowed only on specific mounts (project/workspace). This defends against malicious or buggy scripts that might try destructive actions (the infamous “`rm -rf /`” scenario). As the Hugging Face analysis warns, unsupervised agents can hallucinate dangerous commands <sup>8</sup>, so we mitigate by limiting what they can actually reach.
- **Network lockdown:** Network is turned off by default (`--no-network`). If network is needed, it should be explicitly enabled and possibly filtered. For example, a default-deny firewall inside the container (using iptables) can be used to whitelist only trusted domains <sup>7</sup>. This prevents any AI session from exfiltrating data or reaching untrusted services without approval.
- **Command approval (optional):** Vectra Guard can implement a user-confirmation step for risky actions. Drawing from Gemini CLI 0.9’s design, every planned command could be shown to the user for approval <sup>9</sup>. By default, non-destructive read-only actions could auto-run, while writes or network calls require confirmation. This human-in-the-loop guardrail adds safety against unexpected AI behavior.
- **Limited privileges:** The container should not run in Docker “privileged” mode, and should use a minimal user namespace mapping so that the container’s root is not the host’s root. Any credentials or tokens are passed only as needed. Vectra Guard itself should drop its own elevated privileges (e.g. if installing binaries, it can use user namespaces or capabilities).

- **Image integrity:** All sandbox images should come from trusted sources and be regularly patched. Optionally, Vectra Guard can verify image digests or use an internal registry. Avoid pulling random internet images at runtime.
- **Logging & audit:** Because all activity is logged, any incident can be traced. The audit logs themselves are protected (only root or the invoking user can read them). This transparency deters malicious commands, since “everything you do is recorded.” (This is akin to Gemini’s audit log which “records every AI-initiated command” <sup>4</sup>.)
- **Fail-safe defaults:** If the sandbox environment fails to initialize, Vectra Guard should refuse to run the command on the host. It should never silently fall back to executing on the host. Clear error messages must inform the user if sandbox launch fails (e.g. Docker not available).

By combining these measures, Vectra Guard gives developers confidence that even if an AI-generated script is flawed or malicious, it cannot harm the host system. The isolation is explicit and verifiable.

## Future Extensibility

- **Remote & parallel sandboxes:** In future, Vectra Guard could orchestrate sandboxes on remote hosts or cloud containers for parallel workloads. LangChain’s docs point out that remote sandboxes enable parallel execution and reproducibility <sup>10</sup>. For instance, a Kubernetes or AWS Fargate integration could spin up multiple container sandboxes concurrently for heavy tasks.
- **Network policy enhancements:** More advanced networking controls can be added (e.g. service meshes, proxy whitelists, or hardware-enforced enclaves). Inspired by the Claude Code approach, dynamic iptables or eBPF rules could selectively allow traffic for specific APIs.
- **Filesystem security modules:** Beyond containers, integrating Linux LSMs like Landlock (or using Anthropic’s sandbox-runtime) could offer fine-grained, container-less isolation. This would let untrusted binaries confine themselves without full virtualization.
- **Replay & analysis tools:** While the initial focus is on logging, future versions might let users query or replay sandbox sessions (for example, re-running a logged command in the container to reproduce an issue).
- **Secret management:** Integration with secrets managers (e.g. Vault, AWS Secrets) could allow injecting time-limited credentials into sandboxes without exposing them on the host.
- **Extensible hooks and policies:** Plugins or hooks could enable custom pre- and post-execution policies (e.g. automated code linting before running, or automatic diffing of file changes). This follows the trend of CLI agents supporting extensions. <sup>11</sup>
- **User interface:** A GUI or dashboard could visualize sessions, allow downloading logs, or configuring sandbox settings via a config file or interactive prompts.

Overall, Vectra Guard's sandbox mode is designed to be a solid foundation for safe command execution. By building on containerization and thorough logging, it addresses current security needs while leaving room for evolving workflows and integrations.

**Sources:** The design draws on best practices from AI-enabled shells and dev sandboxes. For example, Google's Gemini CLI and Anthropic's Claude Code both advocate container sandboxes for safety [9](#) [2](#), and analyses of agentic CLIs emphasize that environment-level isolation is needed to stop hallucinated destructive actions [8](#) [5](#). Our approach also follows documented guidelines for sandboxing AI agents using Docker and firewalls [1](#) [7](#), adapting them to Vectra Guard's CLI-centric model.

---

[1](#) Support container-based sandbox for agent execution to improve safety and portability · Issue #334 ·

GoogleCloudPlatform/kubectl-ai · GitHub

<https://github.com/GoogleCloudPlatform/kubectl-ai/issues/334>

[2](#) Sandboxing in the Gemini CLI | Gemini CLI

<https://geminicli.com/docs/cli/sandbox/>

[3](#) [8](#) Agentic CLIs Can Do So Much More Than Code-Gen

<https://huggingface.co/blog/danielrosehill/agentic-cli-beyond-the-repo>

[4](#) [5](#) [9](#) [11](#) Google Gemini CLI 0.9: interactive coding, terminal agents, and context-aware development

<https://www.datastudios.org/post/google-gemini-cli-0-9-interactive-coding-terminal-agents-and-context-aware-development>

[6](#) [10](#) Deep Agents CLI - Docs by LangChain

<https://docs.langchain.com/oss/python/deepagents/cli>

[7](#) Sandboxing AI Coding Agents: Network Firewall + Restricted Shell Environment

<https://mfyz.com/ai-coding-agent-sandbox-container/>