NODEXLOOP 4.6.0 · nodexloop.orivael.dev

NodeXLoop

The User Guide: basic to advanced

Build, debug and govern AI agent loops: on the canvas, inside your codebase, from Claude Code or Codex through the NodeXLoop MCP server, and out to n8n, Zapier, Make and any MCP server.

Debug AI agents like you debug code.

How to read this guide

Part	What you will get out of it	Who it is for
1 · Basics	What NodeXLoop is, the tabs, the node palette, the node inspector panel field by field, your first run in Sim and Live, breakpoints.	Everyone. Start here.
2 · The guard	What the Axiom guard checks, why loops stop early, and how to read a block.	Anyone running live models.
3 · While coding	The VS Code extension: graphs as .loop.yaml files, code bindings, Run, Search and Module nodes, settings.	Developers.
4 · NXL MCP	Letting Claude Code or Codex build, check, simulate and debug loops for you.	Developers using AI coding agents.
5 · Integrations	Webhooks to n8n / Zapier / Make, the MCP Tool node, and Autopilot.	Automation and platform owners.
6 · Teams	Accounts, enrollment keys, the review pipeline, forks, the activity log and live rooms.	Team leads.
7 · Playbook	Seven end-to-end recipes, a cheat sheet, limits and troubleshooting.	Anyone getting to full use.
8 · Outside the editor	The same .loop.yaml in a notebook or Colab: flow and org views, inspecting the code behind a node with highlighting, line-to-node lookup, and a real debug loop over any public repo.	ML and research engineers, anyone running pipelines.
9 · Every panel	A reference to every screen: the header and back navigation, all ten tabs, the Fleet robot debugger, every dialog and drawer, the extension-only simulator and SimReady flows, and what changes on a tablet or phone.	Everyone, as a map.

PART 1

Basics

NodeXLoop is a visual environment for building, debugging and governing AI agent loops. A graph is an agent loop you can step through: **agent** → **tool** → **critic** → **retry** → **gate** → **output**. Unlike a general automation tool, the unit of work is the reasoning loop itself, so you get the things you expect from a code debugger: breakpoints, step-through, a full trace of every call, and a governor that stops loops that have stopped improving.

Everything runs against **your own models and keys**. Nothing is proxied through Orivael, and the guard runs on your machine.

Two ways to run it

	Web build	VS Code extension (recommended for real work)
Where	nodexloop.orivael.dev in any browser	Install **NodeXLoop** (publisher **orivael**) from the VS Code Marketplace or Open VSX, then run **NodeXLoop: Open Builder** from the command palette.
Storage	This browser's localStorage	VS Code state; API keys in the OS keychain
Model calls	From the browser (some providers need CORS)	From the extension host, so Anthropic and local Ollama work with no CORS issues
Code features	Export / import .loop.yaml by hand	Live two-way sync with .loop.yaml, code bindings, Run, MCP Tool and Module nodes, workspace files

Channel	How
VS Code Marketplace	Search **NodeXLoop** in the Extensions view (publisher orivael).
Open VSX	For VSCodium, Cursor and Gitpod: **orivael/loop-graph-engineer**.
Direct .vsix	From the GitHub releases page; each release lists its SHA-256. Extensions → ··· → Install from VSIX.
Browser	nodexloop.orivael.dev. Sim runs whole loops offline with no account and no key.
Browser + IDE (experimental)	ide.orivael.dev: editor, terminal, Python and git beside the canvas. Access by request.

The tabs

Tab	What it is for
Compose	Describe an outcome in a sentence; get a proposed loop from the template catalog, shape it in conversation, create it on the canvas. Calls no model.
Builder	The canvas. Drag nodes from the palette, connect them, edit them in the inspector, run.
Code	The loop as its .loop.yaml beside a pair that proposes diffs you accept or reject.
Templates	Ready-made loops, grouped: Build, Debug your company, Debug your life, Enterprise operations, Governance, Meta, Outcome-based coding (SMB), Bootstrapped startup, Established SMB & enterprise.
Runs	Run history, streaming node by node. The Channel shows a run as the nodes talking; Output, Timeline and Logs beside it; an exportable audit report per run.
Fleet	A robot debugger: deploy wheeled, legged, tracked or fixed robots (or SimReady assets, in the extension), set breakpoints, step, inject faults, export traces. Shows only what you deploy.
Orgs	Sign in, join an organisation, submit loops for review, see released ones.
Autopilot	Proposes simulated repairs for failed runs, inside budgets and approval rules you set.
Flow	Team projects: fork a teammate's loop without touching the original, and one attributed activity log per project (forks, edits, runs, submissions). See Part 6.
Help	Getting started, model registry, keyboard shortcuts.

Tabs that do not fit the window move into **More** at the end of the strip. The back button (‹), `Esc` and `Alt+Left` go back to the previous tab. Part 9 walks every tab and panel in detail.

The node palette

Group	Node	What it does
Agents	Research / Coder / Writer / Critic / Reader	A model call with a role-specific system prompt. Each node picks its model, temperature and iteration limit in the inspector. The Critic emits a score (**SCORE: 0.87**) that drives loops.
Flow control	Loop controller	Sends work back for another pass until the score threshold or iteration limit is hit.
	Gate / branch	Routes on a condition, or stops for a human.
	Merge / join	Waits for several branches and combines them.
Capabilities	Module capability	A governed Python package, installed into an isolated environment after a seven-check gate.
	Swarm	Several agents working one step together.
	Run command	Runs a command on your machine (extension only, off by default).
	Web search	SearXNG, Brave or Tavily; results are marked as untrusted data.
	MCP tool	Calls one tool on one MCP server from your workspace's .mcp.json (Part 5).
I/O	Topic input / Final output	Where a run starts and where its result lands.

Edges are either **normal** (forward) or **loop** (back to an earlier node, usually from a critic, labelled with why, e.g. *revise*). Double-click the canvas to add a node; scroll to zoom; `Ctrl+K` (`Cmd+K` on a Mac) opens the command menu.

The right panel: everything about the node you select

Click a node and the right-hand panel becomes its **inspector**. This is where a node's behaviour lives: the canvas shows the shape of the loop, the inspector holds what each step actually does. Changes save as you make them, go into Undo, and (in the extension) are written to the .loop.yaml.

Part of the panel	What it shows
Header	The node's name, and under it a one-line description of its type (for example what a Critic does).
Settings tab	Everything you can configure: model, prompt, limits, attachments, breakpoint, permissions, code binding, and the fields specific to the node type. Covered field by field below.
Memory tab	A preview of shared memory keys and scope. **Not connected yet**: changing it has no effect on a run.
Logs tab	This node's log lines from the **most recent run**, each with its time and level (info, success, warning, error). Empty until you have run the loop.
Nothing selected	A hint: double-click the canvas to quick-add a node, double-click a node to rename it.
An edge selected	The edge panel instead: its type (Normal or Loop / feedback), its label, and Delete edge.

On a narrow screen the panel folds away; the **Selected node panel** button in the top bar shows or hides it.

Settings fields every node has

Field	What it does	Tip
Node name	The label on the canvas and in the run log.	Name it after its job ("Security critic"), not its type.
Description	Free text, shown in the graph and kept in the .loop.yaml.	Write the node's contract: what goes in, what must come out.
Attachments	Tick files to give this node as context. The count shows next to the label.	In the browser, upload files or drop them on the canvas. In VS Code these are your workspace files.
Breakpoint — pause runs here	The run stops **before** this node so you can read what it is about to send. Same as F9.	Put one on the critic when a loop misbehaves.

Fields for model-driven nodes

Agents (Research, Coder, Writer, Critic, Reader), Loop, Gate, Merge, Topic input and Final output all call a model, so they also have:

Field	What it does	Tip
Model	Picked from your model registry. The default model is marked **·default**, favourites with a **star**; a model that cannot run says why (for example a missing key) right in the list.	Use a cheap or local model for drafts and a strong one for the critic.
Temperature	0 to 2. Choosing a model fills in that model's default.	Low (0–0.3) for coders and critics, higher for writers.
System prompt	The instructions sent to this agent on every call.	For a critic, end with a **SCORE: 0.0–1.0** line; the loop logic reads it.

Fields for loops and critics

Field	What it does
Max iterations	1 to 20. The hard cap on passes around the loop.
Quality threshold	0 to 1. The score at which the loop counts as done.
Auto-terminate on success	Stop as soon as the threshold is met instead of using every iteration.
Pass full context	Send the whole history back each pass, not just the latest draft. Better revisions, more tokens.

The guard's loop governor (Part 2) can still stop a loop earlier than these settings if it stops improving.

Fields specific to one node type

Node	Fields and buttons
Topic input	**Prompt**: what the loop should accomplish. It is the run's starting prompt and the title of the output document.
Final output	**Output format**: Markdown document, Plain text, or JSON bundle.
Module capability	**Package** (shows *Pinned: version* once installed) · **Resolve & install…** runs the governed install · **Load context** fills **API context injected downstream**: the package's API surface, prepended to every later node at zero token cost to this node.
Swarm	**Swarm manifest** (JSON contract naming providers and worker models) · **Working directory** (inside the workspace) · **Run swarm** / **Stop** · **Latest result**.
Run command	**Command** (no shell: arguments split on spaces, quotes keep spaces together, no pipes or redirects) · **Working directory** (blank = workspace root) · **Run now…** · **Last output** with its exit code. Stdout goes downstream; a non-zero exit becomes a FAILED line a critic or gate can act on.
Web search	**Query** (blank means `{{input}}`: the upstream output cut to 300 characters) · **Max results** (1–10) · **Search now…** · **Last results**, marked as untrusted data for the next node.
MCP tool	**MCP server** (from .mcp.json, with its transport; Reload after editing the file) · **Tool**, each labelled *read-only*, *writes (asks first)*, *may delete (asks first)* or *undeclared (asks first)* · the tool's description and arguments (* = required) · **Arguments (JSON)** · **Call now…** · **Last result**. In the browser it is a placeholder that sim runs log.

Run command, Swarm, Web search and MCP tool do not call a model, so they have no Model, Temperature or System prompt fields.

Runtime permissions (agent nodes)

Research, Coder, Writer, Reader and Critic nodes end with a **Runtime permissions (guard policy)** block:

Permission	Options	Enforced?
File scope	A path pattern, e.g. /src/**	Recorded
Network	Allow / DENY	Enforced as a host allowlist in the extension
Git	None / Branch writes only / Unrestricted	Recorded
Shell	Free text, e.g. npm test only	Recorded
Secrets access	On / off	Recorded

Each node's permissions are written into its guard manifest and the .loop.yaml, so reviewers see what a step was allowed to do in the same diff as the change.

Implementation (jump to code)

The last block on every node binds it to code: a **file** path and a **Symbol** (function or class). **Open** jumps to it in VS Code. Under the fields a status line tells you where you stand: *binding not checked yet*, *resolves to line N*, or, in red, why it no longer resolves. See Part 3.

Debugging with the panel. Set a breakpoint on the suspect node, run, and when it pauses read its Settings (is it the model, the prompt or the threshold?) and its Logs tab (what actually happened last time). Change one field, run again, compare.

Your first run, in three steps

1. **Add a model.** Open the model registry (Help → Model registry, or the model field on any agent) and paste a key, or point it at Ollama or any OpenAI-compatible endpoint. OpenAI, Anthropic, Google and OpenRouter are built in. A node whose model has no key shows **amber** before you press Run, rather than failing mid-run.
2. **Open a template.** Start from one rather than an empty canvas. The prompts are the template, and a working loop is easier to change than to invent. *Code review cycle* is the shortest way to see the shape.
3. **Run it, then break it.** Press Run and watch the Runs tab stream node by node. Then select a node, press `F9` to set a breakpoint and run again: execution pauses *before* that node so you can read exactly what it was about to send. Step, continue or stop from the debug bar.

Sim mode and Live mode

	Sim	Live
Model calls	None. Synthetic agent text, real orchestration.	Your registered model APIs.
Cost	Nothing	Your provider's tokens
Guard	Not invoked (nothing to govern)	Every prompt and completion is checked
Use it to	Learn a template's shape, check wiring, test Autopilot	Get real output; see real blocks and early stops

Tip. A guard block you are trying to demonstrate will not happen in Sim. Switch to Live with a cheap or local model (Ollama) when you want to see the governor work.

The model registry, field by field

Field	What it does
Provider and endpoint	OpenAI, Anthropic, Google, OpenRouter, Ollama and WebLLM are built in; any OpenAI-compatible endpoint works.
WebLLM	A model that runs in this browser on WebGPU: no key, no server. The built-in is Qwen2.5 1.5B; the first call downloads it once (about 1 GB). Web app only (Chrome or Edge); inside VS Code use Ollama.
Duplicate as custom	Copies a built-in model to a new entry you can point at another endpoint (NIM, vLLM, a gateway) without losing its defaults.
Rate in / Rate out	Your $ per 1M tokens. Blank still counts tokens; only the dollar figure is withheld.
Keep alive (Ollama)	Re-sent after every call: `30m`, `24h`, or `-1` to hold the model in VRAM until Ollama restarts. Blank leaves Ollama's own timeout.
Reasoning mode (Ollama)	Off sends `reasoning_effort: none` so the whole token budget goes to the answer. On, a thinking model reasons first; the reasoning goes to the run log, not to the next node.

Thinking models. A thinking model with reasoning on and a small token budget can spend the whole budget thinking and hand the next node an empty answer. Turn reasoning off for workers, or raise the budget.

The model harness

Every model call runs inside a harness, set at the top of the model registry. A node with nothing for it to do makes exactly the call it made before. Every repair, failover and forced answer is a real call: guarded, counted in tokens and cost, and written to the run log, which ends with a harness summary.

Setting	What it does
Output contract (inspector)	What the node must produce: a JSON object with given keys, exactly one label, a final SCORE: line, a pattern, or anything non-empty. Checked in code, never by a model. Met means well-formed, not correct. Critics check SCORE: by default.
Repair turns	A reply that misses its contract is sent back with the exact problem. Default 1.
State the contract on the first call	Off by default: the first call is what the node asks, and the contract is spelled out only in a repair. Stated up front, it changed correct answers on small models.
Backup models (inspector)	Tried in order when the node's model fails, or still misses its contract. Put a cheap model first and a strong one as backup. Stop and guard blocks never fail over.
Thinking budget	For models with Reasoning mode on: when thinking runs past the budget, the model is asked for its answer with reasoning off. Measured on a 4B model: 34% fewer output tokens, same accuracy, more input tokens.
Cap label replies	64 tokens for a label. A model the cap cuts short is retried without it and not capped again.
Compress the goal prompt	Removes politeness wrappers and doubled words from the prompt you typed; never drops a content word, never touches model output or code.
Reuse identical calls	Same model, prompt and settings returns the earlier reply this session. Off by default.

The Runs tab and audit reports

Every run is recorded with its steps, per-node inputs and outputs, iterations, score, calls, tokens, estimated cost and every governance decision. The **audit report** exports provenance, cost, decisions and the full log in a print-ready form. Estimated figures (token savings, costs) are labelled as estimates.

- **Replay** any run node by node, including runs streamed live into the tab.
- **Compare**: arm one run, pick a second, and see the two side by side: score, calls, tokens, stop reason.
- **Repeat-shape suggestion**: run the same loop shape three times and NodeXLoop offers to save it as a template or submit it to your org. Any graph can also be saved from ··· → **Save as template**.
- **Attachments are read, not listed**: uploaded PDFs and decks have their text extracted, so a run works on the content rather than a file name.

> **Experimental templates.** Experimental means the shape is real and runnable, not that the output is trustworthy. The driver loop produces a draft and installs nothing. The legal drafter is not a legal service and not legal advice. Both say so on the canvas, in the input node, on every run.

PART 2

The Axiom guard

The guard runs **inside** the executor, not beside it. It is on by default and works with no network. It does three jobs:

- **Preflight.** Before a run starts, prompts are classified against the Axiom policy packs. A prompt-injection input costs **zero** model calls: it is blocked before the first one.
- **Per call.** Every prompt and completion is classified before it moves. A block halts *its own branch*; the rest of the graph carries on.
- **Loop governor.** Each critic iteration is judged on whether it is still making progress. A loop that has stopped improving is cut early; a loop still improving is left alone.

Why a loop stopped

Stop reason	What it means
LOOP_STALLED	The draft barely changed (novelty under 2%) for the patience window.
LOOP_FLAT	The score gain was below the minimum for the patience window, or the score went down (stops at once).
LOOP_DIVERGENT	The commitment trend (confidence, score, length) broke twice in a row.
CONSTITUTIONAL_DRIFT	Successive iterations drifted toward a block. Halts the whole branch, not just the loop.
BUDGET_EXHAUSTED	The run's step or wall-clock budget ran out.

Measured on the shipped app: a critic loop whose model returns the same draft every time costs **20 model calls with the guard off and 6 with it on**; a loop that is genuinely improving costs the same 12 either way.

Tuning it

Open the guard panel (the **guard** chip in the status bar) to choose policy packs and set the loop governor: **patience** (stalled iterations tolerated before stopping) and **minimum score gain** (below this, an iteration is not progress).

- **Escalation** (off by default) sends borderline verdicts to the hosted Axiom firewall for a second opinion. A remote *block* is authoritative; a remote *allow* never overturns a local block. If the network fails, the local verdict stands. The guard never fails open.
- Each decision produces a **manifest**. Local manifests are sealed with a per-install key (tamper-evident on your machine); **sealedBy: axiom** appears only when the firewall's own answer is the one reported.
- **Per-node runtime permissions** (files, network, shell, git, secrets) are recorded into each node's manifest and the .loop.yaml. Network is enforced as a host allowlist in the extension; the rest are declarations today.

> **Read the score honestly.** The loop score comes from the critic's own text. A critic that never emits **SCORE:** falls back to 0.75, which the flat-score rule reads as no progress. Make your critic prompts end with a score line.

PART 3

Using NodeXLoop while you code

In VS Code the graph is a file in your repo, next to the code it drives. It reviews in the same pull request and reverts with the same `git revert`.

The graph is a .loop.yaml

The builder keeps a `.nodexloop/<name>.loop.yaml` in sync both ways:

Direction	Behaviour
Graph → file	Written on every change, debounced 700 ms so dragging does not rewrite the file mid-gesture.
File → graph	Edit the YAML by hand or pull a branch and the canvas redraws. A malformed file never overwrites the graph: the error and line show on the red link chip.
During a run	External edits wait until the run finishes.
Diffs	Positions and edges are one line each; prompts use readable block scalars; no timestamps; re-saving an unchanged graph produces no diff.

```
apiVersion: nodexloop/v1
kind: LoopGraph
metadata:
  name: Release Agent
spec:
  nodes:
    - id: 2
      type: coder
      name: Coder
      position: { x: 320, y: 180 }
      breakpoint: true
      impl:
        file: src/agents/coder.ts
        symbol: runCoder
      config:
        model: gpt-4o
        temperature: 0.2
        maxIterations: 4
        systemPrompt: |-
          You are a careful engineer.
        permissions:
          files: /src/**
          network: deny
  edges:
    - { from: 1, to: 2, type: normal }
    - { from: 7, to: 2, type: loop, label: revise }
```

Bind nodes to code

- Set `impl: { file, symbol }` on a node in the inspector. **Open** jumps to the declaration using VS Code's own symbol provider, as accurate as Go-to-Definition.
- Bound nodes show a rail on the canvas, which turns **red** when the symbol is renamed or deleted, so a stale binding is reported rather than silently pointing at nothing.
- Opening a file highlights the nodes bound to it; **NodeXLoop: Reveal Binding in .loop.yaml** jumps the other way.
- NodeXLoop does not try to infer a graph from arbitrary code. You draw the loop; it keeps the link honest.

CodeLens and the Problems panel

You do not need the canvas open to see the links. In any source file with a bound function, two CodeLens lines sit above the declaration: one opens the graph, the other jumps to the binding in the .loop.yaml.

```
NodeXLoop node '6' — open graph  |  binding: debug_repo.loop.yaml:70
def check(ck: dict, tests: list[str], reply: str) -> dict:
```

Problems panel entry	Meaning
Stale binding (warning)	The symbol is gone from the file: renamed or deleted. Rebind or update impl.
Broken binding (error)	The bound file does not exist.
Axiom guard would block this at run time (warning)	A prompt in the .loop.yaml classifies as HARM or DECEIVE with the runtime's own classifier. Rewrite it or the node will not run.
NodeXLoop guard preflight (warning)	A prompt line matches a preflight pattern (for example an injection phrase).

Whole-line YAML comments are never linted: they never reach a model. Bindings resolve against the **first** workspace folder, and the scan covers up to 20 .loop.yaml files (node_modules, .git, out and dist are skipped); in a repo with more, open the folder that holds the loops you care about.

Nodes that touch your machine

Node	How to turn it on	Safety
Run command	Setting `loopGraph.allowRunNodes` = true, in a trusted workspace	Runs without a shell, confined to the workspace.
Web search	`loopGraph.search.provider` = searxng / brave / tavily. Keys via **Set Web Search API Key**.	Query is guard-checked before it leaves; results are guard-checked before a model reads them.
Module capability	**NodeXLoop: Add Module Capability...**	Registry allowlist, typosquat check, version pin, sha256, install-script warning, licence; installs into an isolated environment with a signed receipt.
MCP tool	`loopGraph.mcp.enabled` = true (Part 5)	Guard on arguments and results; approval modal for non-read tools.
Workspace files	Attach files to a node in the inspector	Run output can be written back into the workspace.

Commands

Command	Use it to
NodeXLoop: Open Builder	Open the canvas in a full editor panel.
NodeXLoop: Reveal Binding in .loop.yaml	Jump from code to the node that implements it.
NodeXLoop: Add Module Capability...	Gate and add a Python package as a node capability.
NodeXLoop: Set Web Search API Key...	Store a Brave or Tavily key in the OS keychain.
NodeXLoop: Set MCP Secret...	Store a value for a `${VAR}` used in .mcp.json.
NodeXLoop: Add Simulator from Hugging Face Space...	Import a pinned, hash-verified robot simulator into the Fleet.
NodeXLoop: Fleet Simulators... / Set Simulator Escalation Level...	Build, open and govern imported simulators.

Settings

Setting	Default	Meaning
loopGraph.allowRunNodes	false	Let Run nodes execute commands.
loopGraph.mcp.enabled	false	Allow MCP Tool nodes (trusted workspaces only).
loopGraph.mcp.configPath	.mcp.json	Which file lists the MCP servers.
loopGraph.search.provider	none	searxng, brave or tavily.
loopGraph.search.searxngUrl	http://localhost:8080	Your SearXNG instance.
loopGraph.pythonPath / nodePath	auto-detect	Interpreters used for module and simulator environments.
loopGraph.checkForUpdates	true	Only matters for a hand-installed .vsix; store installs update themselves.

PART 4

NXL MCP: let your coding agent drive

The NodeXLoop MCP server gives Claude Code, Codex or any MCP client the same guard, graph checker, simulator and trace viewer the canvas uses. Your agent can build a loop, check it, simulate it, read the trace and fix it, without leaving the terminal. **The guard is in the wire. Sim is callable; live is not.** There is deliberately no live-run tool: real model spend stays a human action in the app.

Set it up

You need Node 18+ and a NodeXLoop checkout. Then register the server:

```
# Claude Code (the double dash matters)
claude mcp add nodexloop -s user -- node /path/to/nodexloop/mcp/nodexloop_mcp.mjs
claude mcp list            # look for "Connected"; `add` does not validate

# Team rollout: -s project writes a committable .mcp.json
claude mcp add nodexloop -s project -- node nodexloop/mcp/nodexloop_mcp.mjs

# Codex
codex mcp add nodexloop --command node --args /path/to/nodexloop/mcp/nodexloop_mcp.mjs
```

For the module resolver tool, compile the extension once: `cd loop-graph-vscode && npm install && npm run compile`.

The tools

Tool	What it does	Safety
nxl_guard_preflight	Classify a prompt or plan against the policy packs. Verdict, class, signals, sealed manifest. **A block is authoritative.**	read-only, offline
nxl_build_loop	Turn a description, template id or draft .loop.yaml into a validated graph plus a check report. Starts from a shipped template when one matches.	read-only
nxl_compile_check	Static check: schema, ids, edges, reachability, loop bounds, impl bindings resolve to real files and symbols, guard lint on every prompt, module pins.	read-only
nxl_sim_loop	Sim run: canned agents, real orchestration, real guard. Returns run id, steps, stop reason, verdicts, est. tokens saved. Iterations capped at 5.	no model calls, no network
nxl_get_trace	Replay a run or one step: inputs, outputs, verdicts, logs.	read-only
nxl_module_resolve	Gate a Python package; optionally install it into an isolated env with a receipt.	gated write
nxl_build_and_harden	build → check → sim → mechanical repair, up to 5 rounds. Lands in **sandbox review**, never release.	budget-capped, human-gated
nxl_suggest_templates	Templates (and your org's released ones) that match a task.	read-only

Make your agent use it every time

Put this rule in your repo's `CLAUDE.md` or `AGENTS.md`:

```
When creating or modifying any .loop.yaml: always run nxl_compile_check
before finishing, and nxl_sim_loop before claiming it works. A loop is
"done" when sim passes with no guard blocks - not when the YAML parses.
```

Two working patterns

Pattern A: your agent drives, NodeXLoop supplies ground truth

```
nxl_build_loop("triage tickets, human approves divisions")
nxl_compile_check     -> {errors: [{node: 3, issue: "impl symbol 'route' not found in track.py"}]}
fix, re-check         -> ok
nxl_sim_loop          -> {stop_reason: LOOP_FLAT, score: 0.71, run_id: nxl-...}
nxl_get_trace(run, 7) -> critic stalled on empty retrieval
fix the critic, re-sim -> score 0.93 -> done
```

Pattern B: hand the cycle to nxl_build_and_harden

It builds, checks, simulates and applies mechanical repairs (drops dangling edges, bounds unbounded loops, sets missing thresholds). It consults **no model**. A guard block or a broken binding stops it for a human; a low score comes back for your agent to fix. The result waits in sandbox review; your org's pipeline is the only path to release.

Prompts that work well

- "Build a NodeXLoop loop that drafts release notes from merged PRs, with a critic and a human gate. Check it and sim it."
- "Preflight this system prompt with nxl_guard_preflight before we put it in the coder node."
- "The sim stopped LOOP_FLAT. Pull the trace for the critic step and tell me why."
- "Bind the coder node to src/agents/coder.ts::runCoder and re-run compile check."
- "Is there a shipped template for vendor evaluation? Start from it."

Where things are kept

NXL_MCP_STATE	State directory (default **.nodexloop/** in your repo): runs, receipts, sandbox, isolated envs.
AXIOM_MASTER_KEY	If set, receipts are sealed with a key derived from it (sealed_by: axiom); otherwise a per-process key (sealed_by: local).
NXL_PYTHON	Interpreter used for isolated environments.
NXL_API	Org API for released templates and sandbox submission.
NXL_MCP_SUBMIT=0	Never submit hardened loops to the org.

> **Remember.** A sim pass proves orchestration and policy, not model quality. Run it Live in the app before you trust the output.

PART 5

Integrations

All three integrations are off until you turn them on, and all three run under the same guard.

Webhooks: n8n, Zapier, Make

Integrations → Webhooks. Paste the receiver's URL, optionally set a signing secret, choose events. **Send test** posts a test.ping and shows you the exact envelope. The last 50 deliveries are listed with their status.

Event	Fires when
run.started	A run begins (node, edge and agent counts, mode).
run.finished	A run completes or is stopped (status, duration, iterations, score, calls, tokens, cost, guard summary, up to 4,000 chars of output).
guard.blocked	The guard blocks a node, before or during the run. Carries the verdict and a **SHA-256 of the input**, never the blocked text.
loop.stalled	The guard stopped a loop early (iterations, estimated tokens saved).
submission.released	An org review releases a submission.

With a secret set, each request carries `X-NodeXLoop-Signature: sha256=HMAC(secret, timestamp + "." + raw_body)`. Verify over the **raw** body and reject timestamps older than about five minutes. Delivery retries once on a network error or 5xx and never interrupts a run.

```
const crypto = require('crypto');
function verify(secret, timestamp, rawBody, signature) {
  const want = 'sha256=' + crypto.createHmac('sha256', secret)
    .update(timestamp + '.' + rawBody).digest('hex');
  return signature.length === want.length
    && crypto.timingSafeEqual(Buffer.from(signature), Buffer.from(want));
}
```

> **n8n.** Turn on the Webhook node's **Raw Body** option. A self-hosted Code node needs NODE_FUNCTION_ALLOW_BUILTIN=crypto. From the web build the receiver must allow cross-origin requests; from the extension it needs nothing, and Zapier catch hooks only work from the extension.

MCP Tool node: any MCP server as a governed step

A node that calls one tool on one MCP server and passes the result on. Servers come from the workspace's .mcp.json, the same file Claude Code reads, so GitHub, Notion, Postgres, Slack, Drive or Axiom's own server all work.

1. Set `loopGraph.mcp.enabled` to true (trusted workspace required).
2. List servers in .mcp.json:

```
{ "mcpServers": {
    "github": { "type": "http", "url": "https://api.githubcopilot.com/mcp/",
                "headers": { "Authorization": "Bearer ${GITHUB_TOKEN}" } },
    "axiom":  { "command": "python", "args": ["axiom_mcp_server.py"] } } }
```

1. Store credentials with **NodeXLoop: Set MCP Secret**. ${VAR} and ${VAR:-default} resolve from the OS keychain, then the environment; they never reach the graph or the webview.
2. Drop an **MCP tool** node, pick a server and tool. Required fields are prefilled, the first string with `{{input}}`.

Step	What happens
Arguments	Guard-checked before they leave. A block stops the branch; the server is never contacted.
Required fields	Missing required arguments are refused before the call.

Step	What happens
Effect	Tools annotated read-only run directly. Anything else asks in a VS Code modal before **every** call, showing the arguments.
Result	Marked untrusted and guard-checked before any model reads it, so an injection in a tool's output stops the branch.
Sim runs	Never start a server; they log which tool would have been called.

Autopilot: supervised self-repair

A tab with one master switch. Register debugging templates, choose when Autopilot may step in, and set its limits. When a finished run fires a trigger, it runs a **simulated** diagnose → repair with a matching template. Every decision, including doing nothing, goes to the Activity feed.

- **Templates** (register only): paste JSON with a nodes[] array, or YAML with name:, nodes: and edges:. Optionally declare which triggers a template handles.
- **Sensitivity** sets the score threshold from 0.50 (conservative) to 0.95 (aggressive).
- **Triggers**: run failed, guard stopped a stalled or flat loop, score below threshold, guard blocked a node. Healthy runs are never touched.
- **Approval**: every repair / only guard-block and high-token (≥18k) repairs / report only. Pending repairs show in **This hour** with before → after sim scores and **Approve & apply** / **Reject**.
- **Guardrails**: hourly repair and token budgets (each repair estimated at ~8–26k tokens before it starts), a same-failure cooldown, and a circuit breaker: 3 below-threshold repairs in a row pause Autopilot until a human clicks **Reviewed — resume**.
- Approved JSON templates replace the canvas (Undo brings the old graph back). YAML templates record intent only.

> **Honest scope.** Autopilot's repairs are simulated and its scores are triage estimates, not measurements. It cannot trigger itself, and settings stay in this browser or VS Code; nothing is sent anywhere.

PART 6

Teams: Orgs, review, forks and rooms

The unit of sharing is a whole reviewed graph, and the point is that somebody read it. Model API keys never leave your machine on any of these paths: the shared graph is key-free, and the server refuses a payload carrying one.

Accounts and enrollment (Orgs tab)

- **Real accounts**: a username and password. The display name is what approvals and forks are attributed to; it is deliberately not the credential. Disabling an account ends the session it already holds.
- **Employee enrollment keys**: an admin mints one per person: labelled, use-capped, expiring, revocable, and attributed to whoever redeems it. A new organisation joins by key only. Keys and admin codes are stored hashed and shown exactly once.
- **Device-only mode** stays available for solo or offline work, with no account and no server; the app says which mode you are in wherever it matters.
- Org admins also get a terminal console, `nxl-admin`, for the same calls in bulk (mint, list and revoke keys, manage accounts, rotate the admin code).

The review pipeline

Stage	Who moves it
submit	Anyone in the org, from the canvas or through nxl_build_and_harden.
sandbox	Where submissions and hardened loops wait. Nothing here is loadable by the org.
approved	A **different** account from the submitter: separation of duties, keyed on the unique username.
released	Again not the submitter. Released loops show in the org and in nxl_suggest_templates, and fire submission.released.

Every transition is attributed. Keep the .loop.yaml in git as well: the PR is your code review, the org review is your release gate.

Flow: fork a loop, watch the team

- **Fork without touching the original.** Copy-on-write branches off a teammate's graph; the parent is never written to.
- **One activity log per project**, rendered as chat: a colour per person, an icon per action (start, fork, edit, run, submit); click a row for detail. Runs log themselves with the governor's savings.
- **Visibility is enforced by the server**, not the view: you see a project only if you are attached to it.

Live rooms

Share a graph as a room link and edit it together: graph sync, presence and peer cursors. The link is the whole capability, Drive-share style, so share it only with people who should edit. Rooms run in the browser app; the extension points you there, because its webview blocks the outbound socket.

PART 7

Playbook: getting full value

Recipe 1 · Stop paying for loops that have stopped improving

1. Open a critic-loop template, set a real model, keep the guard on.
2. Make sure the critic prompt ends with a SCORE: line.
3. Run Live. When it stops LOOP_STALLED or LOOP_FLAT, open the run's audit report to see iterations, calls and estimated tokens saved.
4. Tune patience and minimum score gain in the guard panel to match how your loop really converges.

Recipe 2 · Let Claude Code build and debug a loop in your repo

1. Register the MCP server (Part 4) and add the CLAUDE.md rule.
2. Ask: “Build a loop for X, bind the coder to path::symbol, then check and sim it.”
3. When sim stops early, ask it to read the trace and fix the step. Repeat until sim passes with no guard blocks.
4. Open the .loop.yaml in the builder and run it Live yourself. Commit graph and code in one PR.

Recipe 3 · Get a Slack alert from n8n when the guard blocks something

1. In n8n: Webhook node (POST, Raw Body on) → Code node verifying the signature → Slack node.
2. In NodeXLoop: Integrations → Webhooks, paste the n8n URL, set a secret, tick guard.blocked and loop.stalled.
3. Press Send test and confirm n8n received and verified it. The alert gets the verdict and a hash, not the text.

Recipe 4 · Pull real data into a loop with the MCP Tool node

1. Add a GitHub (or CRM, or database) server to .mcp.json and its token via Set MCP Secret.
2. Topic input → MCP tool (e.g. search issues with {{input}}) → Writer → Critic → Gate → Output.
3. Prefer read-only tools; write tools ask before every call, which is the point.

Recipe 5 · Supervised self-repair with Autopilot

1. Register two or three debugging templates, each declaring the triggers it handles.
2. Start in **Report only** for a week and read the Activity feed.
3. Move to **Every repair**, then **Guarded** once you trust it. Keep budgets tight; let the breaker do its job.

Recipe 6 · Map a pipeline you already have, then inspect it

1. Draw the pipeline's stages as nodes and bind each to the function that does the work (Part 3).
2. nxl_compile_check until every binding resolves, then nxl_sim_loop.
3. In VS Code: CodeLens over each function, Problems panel for drift. In a notebook: flow, org and show (Part 8).

Recipe 7 · Debug a failing test in a public repo with a local model

1. Open the Colab notebook (Part 8), run sections 0, 1 and 1b, then section 7.
2. Give it owner/name and the failing pytest id. It reproduces the failure before the model sees anything.
3. Read the trace: each diff under the node that wrote it, the real test result under the node that ran it.
4. A passing patch is left applied and saved; review it (the gate) before using it anywhere.

Cheat sheet

Want to...	Do this
Learn a loop for free	Templates → open one → Sim → Run
Pause before a node	Select it, F9, run
See why a loop stopped	Runs → the run → stop reason and audit report
Version a graph	Use the extension; commit .nodexloop/*.loop.yaml
Check a prompt is safe	nxl_guard_preflight, or Live run with the guard on
Have an agent build loops	Register NXL MCP + CLAUDE.md rule
Notify another tool	Integrations → Webhooks
Use an external tool mid-loop	MCP Tool node + .mcp.json
Auto-triage failures	Autopilot tab, Report only first
Compare two runs	Runs → arm one → pick another
Hold an Ollama model in VRAM	Model registry → Keep alive -1
Edit a loop with a teammate live	Browser app → share a room link
See which loop a function belongs to	CodeLens above it, or li.which("file.py:123") in a notebook
Read a node's code, key lines marked	li.show(loop, node, root, highlight=r"regex")

Known limits

- Sim mode does not invoke the guard; blocks only appear in Live.
- Loop scores are reported by the critic model, not measured.
- Per-node file, shell, git and secret permissions are recorded, not enforced. Network is enforced as a host allowlist in the extension.
- There is no live-run MCP tool, by design.
- MCP read-only annotations are the server's claim; the guard checks arguments and results regardless. Servers that do not annotate their tools ask for approval on every call.
- OAuth-only remote MCP servers are not supported; use one that accepts a bearer token.
- Inbound triggers (n8n calling NodeXLoop) are not available yet.
- Live rooms are browser-only; the extension's webview cannot open the socket.
- The notebook inspector (Part 8) resolves Python bindings only, and its graphs need graphviz (preinstalled on Colab).

Troubleshooting

Symptom	Fix
Node shows amber	Its model has no key. Add one in the model registry.
Anthropic or Ollama fails in the browser	CORS. Use the VS Code extension, which calls from the host.
The guard never blocks my test prompt	You are in Sim. Switch to Live.
A loop stops LOOP_FLAT immediately	The critic is not emitting SCORE:, or the score went down. Check the trace.
Link chip is red	The .loop.yaml failed to parse. The chip shows the reason and line; the canvas is untouched.
MCP Tool node refuses to run	Enable loopGraph.mcp.enabled, trust the workspace, open a folder, check .mcp.json.
claude mcp list shows nodexloop not connected	Check the path and Node 18+; `add` does not validate.
Webhook signature fails in n8n	Turn on Raw Body; verify over the raw bytes, not re-serialised JSON.
No CodeLens over a bound function	Open the repo root as the first workspace folder; more than 20 .loop.yaml files are not all scanned.
A thinking model's node returns nothing	Turn Reasoning mode off for that model, or raise its token budget.
Autopilot has gone quiet	Look for the red breaker banner, the budget lines or cooldown entries in the Activity feed.

PART 8

Loops outside the editor: notebooks and Colab

A .loop.yaml is just a file, so the graph-to-code link does not need VS Code. Axiom ships a small Python module, `research/code_harness/loop_inspect.py`, that reads the same file in a Jupyter or Colab notebook and gives you there what the extension gives you in the editor. The worked example is the qwen-coder training notebook, `research/finetune/colab_qwen_coder_mt.ipynb`, which uses two loops:

Loop file	What it maps
loops/qwen_coder_mt.loop.yaml	The training run: repo list → mine verified fixes → base model attempt → replay attempt → training set → SFT → held-out eval (loops back to SFT) → ship decision → GGUF. 9 nodes, every working node bound.
loops/debug_repo.loop.yaml	One failing test in a public repo: prepare → reproduce + locate → prompt → model diff → apply + re-test (loops back to the model) → human review → patch. 8 nodes.

Both pass nxl_compile_check (all 13 bindings resolve) and nxl_sim_loop with no guard blocks. In VS Code they open on the canvas like any other loop; the notebook does not replace that, it brings the same map to where the pipeline actually runs.

The five views

Call	What you get
`li.flow(loop, status)`	The graph drawn with graphviz. Nodes are filled by type and outlined by status: **green** done, **grey** not yet, **red** failed, **purple** your decision, **blue** running. Loop edges are dashed red with their label.
`li.org(loop, root, notes)`	The org table: one row per node with its type, its job, and the function that implements it as **file:start-end symbol()**, plus any columns you add: where it runs, what it reads and writes, which notebook section runs it.
`li.show(loop, node, root, highlight=...)`	The node's card (type, description, model, limits, prompt, binding) above its function's source, with real line numbers and every line matching the regex highlighted, plus a count of matches.
`li.which(loop, "file.py:123", root)`	Code → graph: the node whose bound function contains that line.
`li.show_trace(result, loop)`	A debug run, turn by turn: each diff syntax-highlighted under the node that wrote it, the verdict under the node that tested it, and the feedback that went round the loop edge in a fold-out.

```
import loop_inspect as li
LOOP = "research/code_harness/loops/qwen_coder_mt.loop.yaml"

li.flow(LOOP, {1: "done", 2: "done", 3: "todo", 8: "you"})   # the map
li.org(LOOP, ROOT, ORG)                                        # who does what, where
li.show(LOOP, "Replay attempt", ROOT,
        highlight=r"git apply|run_pytest|feedback")            # the code, key lines marked
li.which(LOOP, "research/code_harness/repair_turns.py:150", ROOT)
```

Why the bindings matter here. The views read the same impl bindings nxl_compile_check verifies. Rename a function and the org table says **MISSING symbol** instead of showing the wrong code, the same way the extension's rail turns red.

Status that follows the run

Give each node in your org notes the file it writes, and status falls out of the filesystem: a node is done when its output exists. The notebook redraws the flow at the end of every section, so a long run reads as a graph filling in, and after a disconnect you can see at a glance where to resume.

```
ORG = {2: {"runs": "VM CPU", "writes": "/content/gitdbg/pairs.json", "section": "3"}, ...}
status = {i: "done" if os.path.exists(o["writes"]) else "todo" for i, o in ORG.items()}
display(li.flow(LOOP, status))
```

The debug loop, end to end

`research/code_harness/debug_repo.py` is the debug loop as plain functions, one per node, so the graph and the code stay one-to-one:

Node	Function	What really happens
1 Repo + failing test	prepare	Clone the public repo; its own Python 3.12 venv; detect a src/ layout.
2 Reproduce	reproduce	Run the test. If it already passes there is nothing to debug and the loop stops.
3 Locate sources	locate	Source files named in the long traceback, tests excluded (or the ones you list).
4 Prompt	build_prompt	The exact format the model was trained on: buggy files, tests, pytest tail.
5 Model diff	repair_loop	The model (Ollama, temperature 0) sends a unified diff.
6 Apply + re-test	check	git apply, then the test. A diff that edits the test is refused. A failed attempt is reversed with git apply -R, so only the model's own change is undone.
6 → 5 loop		The real git or pytest error goes back to the model, up to ROUNDS more times.
7 Human review	you	A passing patch is left applied and saved as a .patch. Read it before using it.

Every observation is real. Nothing in the loop is simulated: the failure the model reads is the one pytest printed. Stale bytecode is cleared between attempts, because an earlier attempt of the same size written in the same second would otherwise leave a .pyc that Python still trusts.

The Colab notebook, section by section

Section	What it does
0 Settings	Drive folder, sizes, and EXTRA_REPOS: your own public repos to mine.
1 Code	From an axiom.tar on Drive, or a clone with a GH_TOKEN Colab secret.
1b Flow & org	The training loop's flow and org table, a node inspector with highlighting, and line-to-node lookup.
2 Repo list	The vetted permissive list plus your extras; each extra's licence is read from GitHub and only permissive ones (MIT, BSD, Apache, ISC…) enter training. Others can still be debugged in section 7.
3 Mine	Commits whose test fails at the parent and passes with the fix; re-runs skip repos already mined.
4 Repair turns	The base model tries each bug; each miss becomes user → wrong attempt → real error → gold fix.
5 Train	Baseline eval, a smoke run, then bf16 LoRA with loss on the final turn only, and a Q4_K_M GGUF.
6 Eval	The new model on the same held-out bugs, side by side with the base, with Wilson intervals.
7 Debug any public repo	The debug loop above, with the flow, the trace and an inspector for its code.

Bring your own pipeline

1. Write a .loop.yaml whose nodes are your stages, each with impl: { file, symbol } on the function that does it.
2. Run nxl_compile_check until every binding resolves, then nxl_sim_loop. Sim proves the wiring, not the results.
3. In your notebook: li.flow for the map, li.org with your own notes, li.show on any stage you need to read.
4. Keep the loop in the same commit as the code. A renamed function then fails compile check, the extension's Problems panel and the notebook's org table together.

PART 9

Panel reference: every screen, one by one

This part walks every panel NodeXLoop shows, in the order you meet them: the header, the ten tabs from left to right, the dialogs and drawers, and the extension-only flows. Each one lists what is on it, what each control does, and what happens to it on a narrow panel or a tablet. Parts 1 to 8 explain *why*; this part is the map.

The header

One row across the top, the same on every tab.

Control	What it does
Logo	Back to the Builder canvas.
Palette button (left)	Show or hide the node palette. Below 1180px wide the palette is a drawer that slides over the canvas.
‹ Back	Goes back to the previous tab. Disabled when there is nowhere to go; its tooltip names where it goes. Same as `Esc` (when nothing is open), `Alt+Left` and the mouse back button.
Tabs	Compose, Builder, Code, Templates, Runs, Fleet, Orgs, Autopilot, Flow, Help. Tabs that do not fit move into **More** at the end of the strip (it says how many, e.g. "4 more"); the tab you are on is always shown.
Project name	Click to rename the loop. Next to it, **Saved** or **Unsaved**.
Save	Saves the project (`Ctrl+S`).
More actions (three dots)	Open project…, New graph, Save as template, Import JSON, Export JSON, Export .loop.yaml (git-native), Copy share link, Welcome & about, Keyboard shortcuts.
Undo / Redo	`Ctrl+Z` / `Ctrl+Shift+Z`. Hidden below 620px; the shortcuts still work.
Ctrl+K button	The command palette: search and run any action. `Ctrl+K` / `Cmd+K`.
Files (paper clip)	The Files dialog (below).
Integrations	Google, GitHub, Slack, Webhooks, Control Room (below). A dot means one is connected.
Models (gear)	The model registry (Part 1).
Drawer button (right)	Below 1500px the secondary controls move into a panel that slides down from the header. They are moved, not copied: same buttons, same shortcuts.
Architect	Opens the Loop Architect chat beside the canvas (below). On a phone it lives in the drawer.
Run loop	Runs the graph in Sim or Live. Always pinned to the right edge, at every width.

Getting around

Key or gesture	What it does
`Esc`	Closes the innermost thing first: a menu, dialog, the command palette, the node selection, a drawer, an open run thread. When nothing is open it goes back a tab. In a text field it only leaves the field; press it again to go back. Works with focus inside the Fleet too.
`Alt+Left`**, mouse back**	Back a tab.
More	Opens the list of tabs that did not fit. Picking one switches to it and shows its tab.
Vertical wheel on the tabs	Scrolls the strip sideways if a tab is still clipped (rare: More takes them first).

Compose

Describe an outcome in a sentence and shape the loop in conversation. **No model is called** while you compose; every reply says what the code actually did.

Area	What is on it
Loops (left)	+ starts a new conversation; your saved loops (click to open on the canvas); starting points from the template catalog; Sim or Live mode at the bottom. Below 1280px it is a drawer: **Loops** button at the top of the conversation opens it, and a tap outside or `Esc` closes it.
Conversation (centre)	An example sentence and six starting points when empty. Each reply proposes a template-based loop with the words it matched on, a step list and its return paths. Follow-ups understand iterations, pass score, review, a name, or a different template; anything else is said not to have changed anything. **Open canvas** goes to the Builder.
Review choice	"Pause for my review" puts a breakpoint on the output node; "Run straight through" removes it.
Loop contract (right)	Read off the graph: intent, the guard's local verdict, participants, guardrails (iterations and pass score steppers, review, gates, guard, mode), and checks (structure, prompt lint). **Create loop** puts it on the canvas; **Create and run** also starts a run. Both are disabled while a check fails.
Narrow	Below 940px the contract moves under the conversation and the page scrolls.

> **Going back.** After Create the Builder opens on the new loop. `Esc` or ‹ returns to Compose with the draft and the conversation intact.

Builder

Area	What is on it
Node palette (left)	Every node type, grouped (Part 1). Drag one onto the canvas, or double-click the canvas to quick-add. Below 1180px it is a drawer opened from the palette button.
Canvas	Nodes and edges. Drag to move, drag from a port to connect, scroll to zoom, double-click a node to rename. A red ● on a node is a breakpoint. Zoom controls and fit-to-view sit bottom-left, with a minimap of the whole graph (hidden on small screens).
Inspector (right)	The selected node's Settings, Memory and Logs tabs, or the selected edge (Part 1). An agent's Settings include Output contract and Backup models (Part 1, the model harness). Below 1180px it is a drawer that opens when you select something; a tap on the shaded canvas closes it.
Status bar (bottom)	Agents active and run state, node and edge counts, the guard's state, hints, the Ctrl+K hint, and the version.

Code

The loop as its `.loop.yaml`, beside a pair that proposes diffs. No model is called here either.

Area	What is on it
Explorer (left)	The file, every node (click to jump to it in the YAML), any `.axiom` specs, and whether the loop is linked to a repo file. Below 1280px it is a drawer: the button at the start of the path bar opens it.
Editor (centre)	The YAML, in sync with the canvas. Type to edit; your edit goes through the same review as a proposal. The path bar has **Export** (or **Link** in the extension).
Pair (right)	Ask for a change ("make the critic stricter", "at most 2 iterations", "pause before publishing", renames) and get a diff to **Accept** or **Reject**. Accept goes through Undo, and rewrites a linked file. `@node` names a node.
Narrow	Below 940px the pair moves under the editor and the page scrolls.

Templates

Area	What is on it
Start from a template	Starred templates are pinned here.
Your templates	Graphs you saved with More actions → Save as template.
Built-in templates	51 loops in nine groups (Part 1). Each card opens the loop on the canvas; the star pins it. Every template is checked to build a runnable graph.

Runs

Area	What is on it
Run history (left)	Every run, newest first, with status. **Select** lets you tick runs; **Clear all** or clear the ticked ones.
Channel	The run as a conversation between its nodes. Filter chips (All, Needs attention, one per participant). A flagged step opens a thread with the guard's verdict and that node's log lines. The box at the bottom adds a note to the run log and the audit report, or runs `/stop`, `/replay`, `/compare`, `/audit`, `@node`. Notes are recorded; they do not steer agents mid-run.
In this run (right)	Participants and iterations; click one to filter to it. Below 1100px this column is hidden and the **Overview** chip opens it in its place; × closes it.
Output / Timeline / Logs	The final output, the step timeline, and the raw log, for the same run.
Narrow	Below 700px the history sits above the run and the page scrolls.

Fleet: the robot debugger

The Fleet shows **only robots you deploy**. An empty floor says so and offers templates. Robots move on a small, stated motion model: commanded speed scaled by how each locomotion type copes with the surface, traction against friction (slip), a gait for legged bodies, and tilt from gait, roughness and slope. It is a harness for debugging loops that command robots, not a physics engine.

Control	What it does
Robots / Debug buttons	On a narrow panel, open the robot list and the debugger as drawers.
+ Deploy	Locomotion templates, SimReady assets already in this project, and (in VS Code) **Get from SimReady Central...**.
Pause / Run, Step	Pause the whole sim; **Step** advances one tick while paused. Keys: `Space`, `N`.
Speed	0.25× to 4×. Keys `[` and `]`.
Clock	Sim time, T+mm:ss.t.
+ Simulator (VS Code)	Import, build and open a simulator from a Hugging Face Space (below).

Template	How it moves
Wheeled AMR	Differential drive, 1.2 m/s. Slips on wet floor, stops at stairs.
Quadruped	Trot gait, 1.4 m/s. Handles gravel and stairs; careful on wet.
Humanoid biped	Walking gait with a balance margin; falls past 22° of tilt.
Tracked crawler	0.6 m/s, sure-footed everywhere here, heavy on battery.
Fixed arm	No locomotion: pick-and-place cycles and joint limits.

The floor has concrete, carpet, wet floor, gravel, a 12° ramp and stairs (each with its friction μ, roughness and slope), trust zones (Open Floor, Operations, Restricted), a charging dock, and a Live Sim Arena when a simulator is connected.

The debugger (select a robot)

Section	What is on it
Banner	NOMINAL, FALLEN, or the escalation level; "Paused on breakpoint: …" when one fired.
State	State, position and heading, surface, zone, odometer, governance distance.
Drive	Speed limit slider (10–100% of top speed). Mission: **Set goal on map** (click a point; it goes there and holds), **Draw patrol route** (click points, then Done; it loops through them), **Hold here**, **Resume patrol**, **New random route**. A goal in a zone the robot is not authorized for is allowed but flagged: entering it escalates.
Telemetry	Sparklines for speed, slip, tilt and battery, with warning lines.
Gait / Joints	Which feet are in stance; each joint against its limits (amber when within 5%).
Breakpoints	Pause the whole sim when: slip above 30%, tilt within 5° of a fall, it falls, it is stuck 2 s, it enters an unauthorized zone, a joint nears its limit, battery below 20%.
Inject a fault	Weak actuator (drifts off heading), pose sensor noise, push, spill underfoot.
SimReady asset	For an asset robot: package id, source and pinned commit, license, files on disk, and conformance checks passed per engine (PhysX, Isaac, MuJoCo, Newton…) from the package's own validation report.
Recover & export	Stand up / recover, Clear escalation, E-stop or Reboot, Send to dock, **Trace .jsonl** (every telemetry sample), **Debug graph** (the episode as a NodeXLoop graph), Undeploy.

The **Debug & governance log** at the bottom lists every event; click a line to select its robot. Deployed robots are remembered in this browser (or VS Code) until you undeploy them.

Orgs

Sign in, create or join an organisation with an enrollment key, submit loops for review, and see released ones. Details in Part 6.

Autopilot

Area	What is on it
On/off switch	Autopilot is off until you turn it on.
Counters	Repairs this hour, tokens this hour (estimated), waiting for approval, and the circuit breaker.
This hour	What it did, newest first.
Engagement policy	Score threshold, when to step in (run failed, guard stopped a stalled loop, score below threshold, guard blocked a node), human approval rules, max repairs and token budget per hour, same-failure cooldown.
Templates	The debugging templates it may use. Details in Part 5.

Flow

Team projects: fork a teammate's loop without touching the original, and read one attributed activity log per project. Details in Part 6.

Help

Getting started in three steps (add a model, open a template, run it then break it), the model registry, and keyboard shortcuts. More actions → Keyboard shortcuts and More actions → Welcome & about open the same material as dialogs.

Dialogs and side panels

Panel	Opened from	What is on it
Command palette	`Ctrl+K`	Search every action and run it with Enter. `Esc` closes it.
Model registry	Models (gear)	The Harness settings, then every model, whether it can run (or why not), its key fingerprint (never the key), and Add model. Part 1.
Files	Files (paper clip)	Drop or browse files. Text under 300 KB becomes run context; larger files register as metadata only. Dropping a file on the canvas creates an input node.
Integrations	Integrations	Google (Drive & Docs context), GitHub (repo context, verified against api.github.com), Slack (run summaries), Webhooks (signed events to n8n, Zapier, Make), Control Room (live telemetry stream).
Loop Architect	Architect	A chat beside the canvas: describe a loop and a checked draft lands on the canvas, runs in Sim and comes back as the nodes talking. Only nine node types.
Open project	More actions	Your saved projects.
Keyboard shortcuts	More actions, or `?`	Every shortcut.
Welcome & about	More actions	Install links for VS Code and Open VSX, and what NodeXLoop is.
Guard dialog	A blocked step	Why the guard stopped the run, and the matched pattern (Part 2).

Extension-only flows (VS Code)

Flow	How you start it	What happens
Simulator from a Space	Fleet → + Simulator, or the command **NodeXLoop: Add Simulator from Hugging Face Space…**	Pinned to a commit, every file hash-verified, nothing run. Building it (npm ci + vite) is a separate, confirmed step; it opens in its own governed panel and appears on the Fleet as a LIVE robot.
SimReady asset	Fleet → + Deploy → Get from SimReady Central…, or **NodeXLoop: Get SimReady Asset for the Fleet…**	Four steps: a source repo the NVIDIA SimReady Central Space lists, a package, how much (**specs**: metadata and physics layers, a few hundred KB; **preview**: plus thumbnails; **full**: the whole USD package, capped by `loopGraph.simReadyMaxMB`), and the motion template. Only public, ungated repos; pinned to a commit; every file checked against the tree hash and the package's BOM; nothing run; a receipt is sealed.
Escalation	**NodeXLoop: Set Simulator Escalation Level…**, **Set Agent Fleet Level (kill switch)…**	L1 warn to L4 terminate, applied by the extension and recorded.
Modules, search, MCP	**Add Module Capability…**, **Set Web Search API Key…**, **Set MCP Secret…**	See Part 3 and Part 5.

Small screens, tablets and phones

- The app fills the **visible** part of the browser window, so nothing sits under a tablet's or phone's toolbar. Panels that are taller than the screen scroll inside themselves.
- Below 1500px the header's secondary controls move into its drawer; below 1180px the palette and inspector become drawers; below 1280px Compose's loop list and the Code explorer do too; below 1100px the Run channel overview moves behind its chip; below 1080px the Fleet's robot list and debugger become drawers.
- Every drawer has a button that opens it and closes on `Esc` or a tap outside. Nothing that is hidden at a width is unreachable at that width.