How to Use Headroom with VS Code on a Remote Linux Server

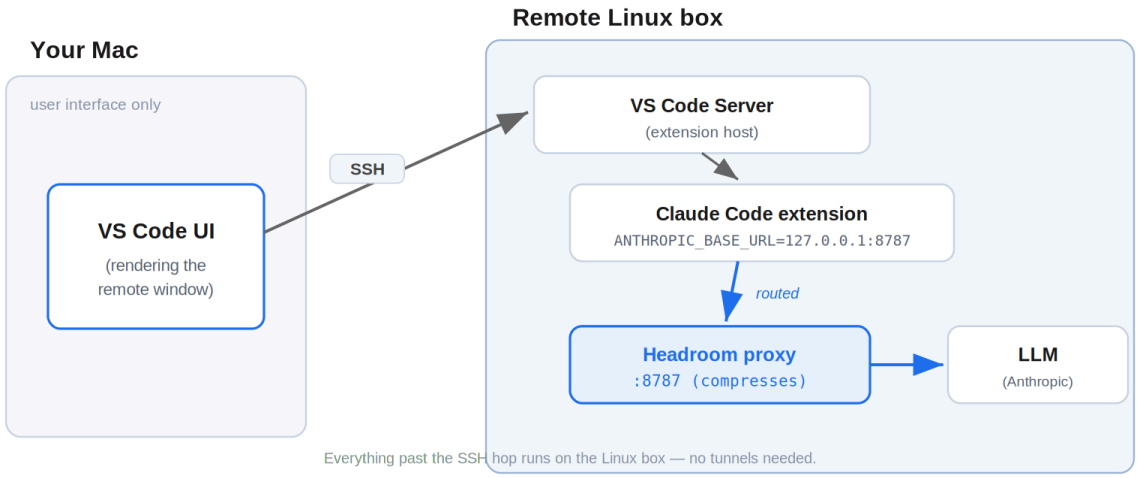
A practical guide to token-compression for Claude Code over Remote-SSH

Headroom is a local compression layer that sits between your AI coding agent and the LLM provider. It compresses tool outputs, logs, file contents, and conversation history *before* they reach the model — typically cutting 60–95% of tokens while preserving the answers. This guide walks through running Headroom alongside the Claude Code VS Code extension when your code lives on a remote Linux box and you drive VS Code from a Mac over Remote-SSH.

The Key Idea: Everything Runs on the Linux Box

With VS Code Remote-SSH, your Mac is only the user interface. The Claude Code extension's server component, the integrated terminal, and any process VS Code spawns all run on the remote Linux host. Your Mac just renders the window.

That means the entire Headroom loop — the proxy, the Claude Code process, and the environment variable that connects them — lives on the Linux box. Nothing has to cross the SSH boundary. When you point Claude Code at `http://127.0.0.1:8787`, "localhost" refers to the Linux server, not your Mac. No SSH tunnels or port forwarding are required.



Part 1 — Install Headroom on the Linux Server

The `headroom` command-line tool is the **Python** package. There is an npm package named `headroom-ai`, but that is only the TypeScript *library* for embedding compression in Node code — it does not provide the CLI. Don't rely on it for this.

Requirements

- **Python 3.10 or newer.** Many RHEL/Rocky/Alma systems ship Python 3.9 as the system interpreter, which is too old. Check first:

```
python3 --version
```

1. Install a modern Python (if needed)

On RHEL 9 / Rocky / Alma:

```
dnf install -y python3.12 python3.12-pip
python3.12 --version
```

If `python3.12` isn't offered, list what's available and substitute any ≥3.10 version:

```
dnf list available 'python3.1*'
```

2. Install the build prerequisites

Headroom has a Rust core and pulls in C/C++ extensions (such as `hnswlib`). Install the development headers and compilers up front to avoid mid-build failures:

```
dnf install -y python3.12-devel gcc gcc-c++ rust cargo
```

Without `python3.12-devel`, the C++ extension build fails with `fatal error: Python.h: No such file or directory`. Without `rust`/`cargo`, the Rust core build stalls or errors.

3. Install pipx and then Headroom

`pipx` installs the CLI into its own isolated environment while exposing the `headroom` command on your PATH:

```
python3.12 -m pip install --user pipx
python3.12 -m pipx ensurepath
source ~/.bashrc          # reload PATH

pipx install --python python3.12 "headroom-ai[all]"
```

The `[all]` extras include the ML compression model, memory store, MCP server, and proxy. For a lighter install (only the proxy and MCP, not the cross-agent memory features), use:

```
pipx install --python python3.12 "headroom-ai[proxy,mcp]"
```

4. Verify

```
headroom --version
headroom --help
```

Behind a corporate proxy or SSL inspection? If the build fails with `CERTIFICATE_VERIFY_FAILED`, the Rust toolchain or runtime assets are being intercepted. Installing `rust`/`cargo` from dnf first means the build won't fetch its own toolchain. Two runtime assets are also fetched over TLS — the ONNX runtime from cdn.pyke.io and the compression model from huggingface.co. If those are blocked, point your CA bundle via REQUESTS_CA_BUNDLE / SSL_CERT_FILE, or run with compression disabled (pure gateway mode), which needs neither asset.

Part 2 — Run the Proxy as a Persistent Service

The VS Code extension can spawn a Claude Code process at any time, so the proxy must already be running and must survive your SSH session dropping. Don't rely on `headroom wrap` for the extension path — that command launches its *own* terminal Claude Code. For the extension, start the proxy standalone.

Quick approach: tmux or nohup

```
# tmux (recommended — you can reattach to see logs)
tmux new -s headroom
headroom proxy --port 8787
# detach with Ctrl-b then d

# or nohup
nohup headroom proxy --port 8787 > ~/.headroom/proxy.out 2>&1 &
```

Durable approach: a systemd user service

Create `~/.config/systemd/user/headroom.service`:

```
[Unit]
Description=Headroom compression proxy
After=network.target

[Service]
ExecStart=%h/.local/bin/headroom proxy --port 8787
Restart=on-failure

[Install]
WantedBy=default.target
```

Then enable it:

```
systemctl --user daemon-reload
systemctl --user enable --now headroom
loginctl enable-linger "$USER"   # keeps it running after you log out
```

Confirm it's listening:

```
curl -s -o /dev/null -w "%{http_code}\n" http://127.0.0.1:8787/   # any response = up
```

Part 3 — Point the VS Code Extension at Headroom

The Claude Code VS Code extension reads a setting called `claudeCode.environmentVariables` — a JSON object injected into every Claude Code process the editor starts. This is the cleanest way to route the extension through the proxy without touching shell files.

The critical detail: use the Remote settings scope

Because the extension runs on the Linux box, the variable must land in a settings scope that the remote process can see. VS Code keeps separate settings sinks under Remote-SSH:

- **User (Local)** — lives on your Mac. Wrong scope; the remote extension won't see it.
- **Remote [SSH: yourhost]** — lives on the Linux box. ✔ Use this.
- **Workspace** — the remote project folder. Also works.

Steps:

1. Connect VS Code to the remote host over SSH.
2. Open the Command Palette → **Preferences: Open Remote Settings (JSON)** for your SSH host (or open Settings and select the Remote [SSH: yourhost] tab).
3. Add the setting:

```
"claudeCode.environmentVariables": {
    "ANTHROPIC_BASE_URL": "http://127.0.0.1:8787"
}
```

4. Reload the remote window: Command Palette → **Developer: Reload Window**. This restarts the extension host so it picks up the new variable.

Verify traffic is flowing

With the proxy running and the window reloaded, do something in Claude Code (ask it to read a file, run a command). Then on the Linux box:

```
headroom dashboard                # live savings dashboard
# or tail the proxy log
tail -f ~/.headroom/logs/proxy.log
```

If you see requests arriving, you're compressed. If the log stays silent while you use the extension, the variable isn't reaching the remote process — double-check you set it in the **Remote**, not **User (Local)**, scope.

Part 4 — Fallback: the Integrated Terminal

The extension can be finicky about a custom `ANTHROPIC_BASE_URL` across releases — in some versions, pointing it at a non-Anthropic base URL bounces you to a login screen (it treats the custom URL as a third-party gateway needing its own token). This matters more if you authenticate via a Claude Max/Pro login rather than an API key.

If that happens, fall back to the terminal flow, which is rock-solid and already runs on the Linux box:

1. In VS Code, open the **integrated terminal** (View → Terminal). Because of Remote-SSH, this terminal is a shell on the Linux server.
2. Run:

```
headroom wrap claude
```

`headroom wrap claude` starts (or reuses) the proxy, installs the rtk shell-output compressor, registers the MCP retrieve tool, and launches Claude Code with `ANTHROPIC_BASE_URL` already pointed at the proxy. You do your Claude Code work in that terminal session rather than the extension UI — same model, same project, fully compressed.

This fallback needs nothing from Part 3. It’s self-contained, so it’s also a good way to confirm Headroom works at all before fighting with extension settings.

Optional Extras

- **Serena MCP** (code-graph / semantic tooling) is skipped if uvx isn't installed. To enable it, install uv on the Linux box: `curl -LsSf https://astral.sh/uv/install.sh | sh`, then re-run your wrap or restart the extension.
- **Output token reduction** trims what the model writes back (preambles, restated code, over-thinking on routine steps). It's off by default — set `export HEADROOM_OUTPUT_SHAPER=1` before launching the proxy.
- **Check your savings** anytime with `headroom perf`, or `headroom dashboard` while the proxy is running.

Troubleshooting Quick Reference

Symptom	Likely cause	Fix
`headroom: command not found`	Installed only the npm library, or PATH not reloaded	Install the Python package via pipx; source ~/.bashrc
`No matching distribution found`	System Python < 3.10	Install python3.12 and use it explicitly
`Python.h: No such file`	Missing dev headers	dnf install -y python3.12-devel gcc gcc-c++
`Install spinner hangs forever`	Rust toolchain fetch blocked	dnf install -y rust cargo, then retry
`CERTIFICATE_VERIFY_FAILED`	SSL inspection intercepting fetches	Install Rust from dnf; set REQUESTS_CA_BUNDLE
`Proxy log silent while using extension`	Env var in wrong settings scope	Move it to Remote [SSH] scope, reload window
`Extension shows login screen`	Custom base URL treated as gateway	Use the integrated-terminal headroom wrap claude fallback
`Proxy dies when SSH drops`	Not running as a service	Use tmux, nohup, or the systemd user service

Summary

1. Install the **Python** headroom CLI on the Linux box (Python 3.10+, dev headers, Rust).
2. `headroom proxy --port 8787` as a persistent service on the Linux box.
3. Set `claudeCode.environmentVariables` → `ANTHROPIC_BASE_URL: http://127.0.0.1:8787` in the Remote settings scope, then reload the window.
4. If the extension misbehaves, open the remote integrated terminal and run `headroom wrap claude` instead.

Because Remote-SSH puts everything on the Linux host, there are no tunnels to manage — the proxy, the agent, and the routing all live in one place.

About EVOKNOW.AI

Founded in 2002 by serial entrepreneur Mohammed Kabir, EVOKNOW, Inc. designs and delivers custom technology solutions for mid-size enterprises with complex requirements, particularly those operating across multiple international markets. For more than two decades the company has helped businesses bridge the gaps between knowledge, implementation, and practical adoption of new technology.

In recent years EVOKNOW has expanded its expertise to harness artificial intelligence across enterprise computing — from complex data analysis and 24/7 fraud and security management to content creation and agentic automation. Its engineering teams work fluently across all major Large Language Model APIs, building sophisticated agentic solutions spanning content moderation, code-quality analysis, and intelligent automation. The team also brings deep Linux and cloud expertise across AWS and Akamai environments, along with edge-AI work on NVIDIA hardware.

EVOKNOW's founder, Kabir, discovered Linux in college and became a respected expert and author of dozens of books — including the first Security and Optimization title for Red Hat Press and the *Apache Server Bible*. That open-source heritage continues to shape the company's engineering culture today.

Learn more at **evoknow.com**.