// WPR-CLD-003 · PUBLISHED 3 OCTOBER 2026 · UPDATED 23 NOVEMBER 2026

Self-Hosted CI Runners: The Job That Waited for a Sleeping PC

Where CI spend goes, which runner hosts cut it, when they're cheaper and when they aren't, and how to set up, run and secure each one, from a Windows PC to a fleet of spot instances

Jason Newell · MAX Research Collective · Report WPR-CLD-003 · Published 3 October 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/self-hosted-ci-runners.html
Cite as: Newell, J. (2026). Self-Hosted CI Runners: The Job That Waited for a Sleeping PC. MAX Research Collective, WPR-CLD-003. https://www.maxresearchcollective.com/articles/self-hosted-ci-runners.html

The PC that went to sleep

Point a workflow at a self-hosted runner, let the Windows PC behind that runner go to sleep, and GitHub does exactly what it was told. It holds the job and waits, for up to a day, for a machine that won't answer until somebody touches the mouse. Nothing fails and nothing alerts. The release simply doesn't ship.

That stall is the whole subject in miniature. Continuous integration, usually shortened to CI, is the service that builds and tests every change pushed to a repository. Every hosted CI service rents you its machines by the minute. A self-hosted runner moves that work onto a machine you control and takes the per-minute charge to zero. In exchange it hands you a new set of jobs: keeping the machine awake, patched, isolated and honest about whether it's online. The sleeping PC is what happens when only the first half of that trade gets done. Both halves matter.

This piece was prepared for the AlienInterface project, the repository behind the M.U/T.H/E.R (Model Unification / Tool Harness / Execution Runtime) desktop app. That repository now runs its CI on the owner's Windows PC when the PC is on, and on

GitHub's own machines when it is off. It covers who runs self-hosted runners and why, and where they can live: a Windows PC, the Windows Subsystem for Linux, Docker containers, a network-attached storage box, Kubernetes, virtual machines and rented cloud instances. It shows how to set each one up and use it. It also says when the move is cheaper, when it isn't, and what each option costs in money, time and risk. Prices are list prices as of 30 September 2026, and several hosted rates are quoted at GitHub's January 2026 list. Check the linked sources before committing money, because these numbers move.

A glossary sits near the end for the terms that recur, and every term is also explained where it first appears.

Is it cheaper? The short answer

Per minute, it almost always is. GitHub Actions, the CI service built into GitHub, charges $0.006 a minute for a 2-core Linux runner on a private repository, $0.010 for Windows and $0.062 for macOS. A self-hosted runner is the same agent program running on a machine you control, and GitHub bills nothing per minute for it. You pay for the hardware, the electricity and the upkeep instead.

Whether the move saves money overall depends on three things. The first is what you already own. A PC, a NAS or an old laptop that is already paid for makes the hardware cost zero, so the runner pays from its first day. The second is what you run. macOS minutes cost ten times Linux minutes, so ten thousand of them cost $620 against $60 for Linux, and moving them pays back fastest. The third is how much of your own time the runners eat — the cost that sinks badly run fleets.

It is not cheaper in four situations. Public repositories already get standard hosted runners free, and self-hosting them adds risk with no saving. A team that stays inside its monthly free allowance saves nothing on the invoice, though it may still gain build speed. A runner platform that takes a day a week of an engineer's time can cost more than the minutes it replaced. And bursty Mac work on rented cloud Macs carries a 24-hour minimum, which makes per-job use expensive.

The savings come from five places in the stack:

- Compute minutes are the obvious one. Windows and macOS minutes are the most expensive, so those move first.
- Special hardware is the second. GPUs, many-core machines and Macs are billed even on public repositories when a vendor hosts them, and a spare desktop or a Mac mini replaces them.
- Parallelism and plan fees are the third. Some platforms, Azure DevOps among them, sell concurrency by the job, and self-hosted concurrency is cheaper or free.
- Build time is the fourth, and often the largest. A persistent runner keeps dependency caches, Docker layers and compiler caches warm between runs, so builds get faster and people wait less.
- Storage and network are the fifth. Pull-through mirrors and artifact stores on a NAS cut registry pulls, rate-limit failures, cloud egress charges and storage billed over quota.

The cost you add is operations: patching, disk cleanup, runner updates, security isolation and your time. Most of this guide is about keeping that small. The tools are golden images, infrastructure as code, ephemeral runners, monitoring, and a switch that falls back to hosted runners whenever your own machines are offline. That switch is the fix for the sleeping PC.

The order matters. Start with the PC you already own, running a Windows service runner for Windows jobs and WSL2 or Docker for Linux jobs. Leave public repositories on hosted runners, and add a hosted fallback switch so nothing waits on a sleeping machine. Add a NAS or mini PC for always-on work and caches, and a Mac mini for iOS. Bring in Kubernetes or spot instances only when you have enough parallel jobs to justify an autoscaler.

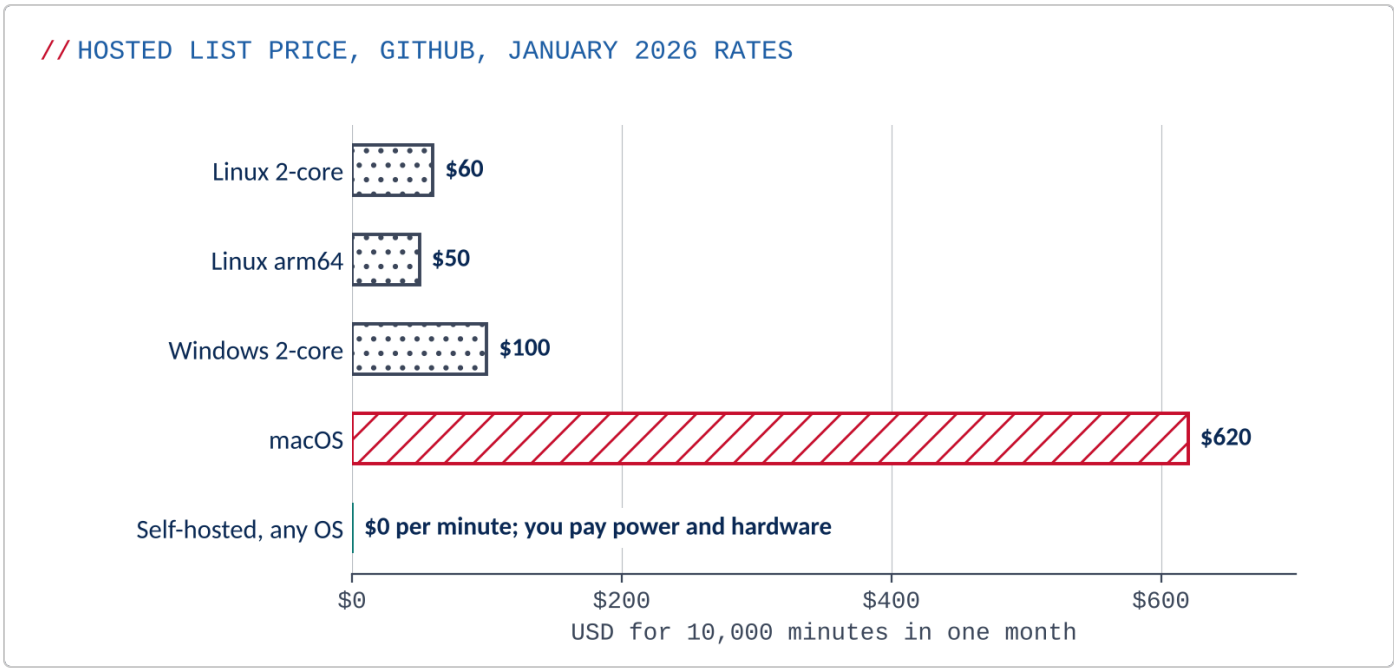


Figure 1. Ten thousand hosted macOS minutes cost $620, ten times the Linux price, while the same minutes on a self-hosted runner cost nothing per minute.

Who runs self-hosted runners, and who shouldn't

Almost everyone who pays a CI bill is a candidate. The reason changes with size. A solo developer or a freelancer usually wants to stay inside the free minutes and get faster builds from the PC already under the desk. A small team of two to fifteen developers is usually paying for Windows or macOS minutes and can replace most of them with one mini PC, a NAS and a Mac mini. A mid-size company of fifteen to two hundred developers runs enough parallel test suites that autoscaling on cheap cloud capacity pays for a platform team's attention. An enterprise runs its own runners for governance and network access as much as for cost — its builds need to reach internal systems that hosted machines can't see.

Some industries self-host because the cloud can't do the job at all. Embedded and automotive teams need tests that touch physical boards; semiconductor design firms run tools whose licences are tied to on-premises hosts. Finance, healthcare, government and defense teams keep code and secrets inside networks that hosted runners never enter. The industry section later in this piece maps each sector to its setup.

Three groups should think twice. Open-source maintainers with public repositories already get free hosted minutes. Worse, a self-hosted runner attached to a public repository lets any stranger's pull request run code on the maintainer's machine. Teams with no one to own patching and disk space will find the operations cost arriving as 2 a.m. outages. And teams whose monthly usage sits comfortably inside their plan's included minutes will see no change on the invoice, so their case rests entirely on build speed.

How a CI runner works

The control plane and the agent

Every modern CI system splits into two halves. The control plane is the service that stores pipelines, queues jobs, holds secrets and collects logs and artifacts. GitHub Actions, GitLab, Azure DevOps, the Jenkins controller, Buildkite, CircleCI, Bitbucket Pipelines and Forgejo or Gitea all play this role. The agent — vendors variously call it a runner, an agent or a node — runs on a machine, asks the control plane for work, runs the job's steps and streams the results back.

A hosted runner is an agent on the vendor's machine, billed by the minute; a self-hosted runner is the same agent on your machine. From the pipeline's point of view the only difference is a label — self-hosted where ubuntu-latest used to be.



Figure 2. Every runner host on the right opens its own outbound connection to the control plane, so none of them needs an inbound port or a public address.

Three properties make self-hosting practical almost anywhere.

1. The connection is outbound only. The agent long-polls the control plane over HTTPS on port 443: it holds a request open and waits for the service to answer with a job. You open no inbound ports and need no public IP address. It runs behind NAT (the address translation a home router does), on a home connection, or inside a locked-down VLAN, which is a separately fenced segment of a network. Jenkins is the exception: its agents can connect inbound over WebSocket or JNLP, its older agent protocol, or the controller can reach agents over SSH.
2. The same program runs everywhere. The agent is a single binary or a small bundle for Windows, Linux and macOS, on both x64 and arm64 processors. Only the packaging differs: a Windows service, a systemd unit on Linux, a launchd job on macOS, a container, a Kubernetes pod or a VM image.

3. Labels route the work. Jobs ask for labels or tags (self-hosted, windows, gpu), and the control plane gives each job to any idle agent whose labels include every one requested. Labels are how work gets steered to the right hardware.

Persistent, ephemeral and just-in-time runners

A runner can live in one of three modes, and the mode decides most of the security and speed trade-offs that follow.

MODE	WHAT HAPPENS AFTER A JOB	STRENGTHS	WEAKNESSES	USE IT FOR
Persistent	The agent stays registered and takes the next job on the same disk	Warm caches, fastest builds, simplest setup	Leftover files and processes between jobs, state drift, a compromised job can affect the next	Trusted private repos on your own machines
Ephemeral (--ephemeral)	The agent takes exactly one job, then unregisters and exits; a supervisor starts a fresh one	Clean state per job, safe to scale	Needs a supervisor (container restart policy, autoscaler), cold caches	Containers, Kubernetes, VM pools
Just-in-time (JIT)	An orchestrator asks the API for a single-use config and boots a worker with it; no long-lived registration token exists	Most secure, no token lying around, precise scaling	Requires an orchestrator and API credentials	Autoscaled cloud or VM fleets

Table 1. The three runner modes, from simplest and fastest to cleanest and most secure.



Figure 3. An ephemeral runner exists for exactly one job, so the return path from step 6 always starts the next job on a brand-new worker.

The ephemeral cycle is what makes autoscaling cheap: nothing runs while nothing is queued, so the bill falls to zero between bursts. The price is cold caches, since every worker starts empty. The usual answer is a registry mirror, a package proxy and a remote build cache on the local network or in the same cloud region.

The same idea under different names

Each CI platform names these parts differently, but the shapes match closely enough that one plan works across all of them.

PLATFORM	AGENT NAME	GROUPING	HOW A JOB PICKS IT	AUTOSCALING OPTION
GitHub Actions	Self-hosted runner	Runner groups (org/enterprise), labels	runs-on: [self-hosted, linux, x64, my-label] or a scale-set name	Actions Runner Controller (ARC), terraform-aws-github-runner, JIT API
GitLab CI	GitLab Runner with an executor (shell, docker, docker-windows, kubernetes, docker-autoscaler, instance, virtualbox, parallels, custom)	Instance, group and project runners; tags	tags: [windows, docker]	Kubernetes executor, docker-autoscaler and instance executors with fleeting plugins for AWS, GCP and Azure
Azure DevOps	Self-hosted agent	Agent pools, capabilities and demands	pool: { name: MyPool, demands: [...] }	Azure VM Scale Set agent pools, Managed DevOps Pools
Jenkins	Agent (node)	Labels	agent { label 'windows && docker' }	Kubernetes, EC2 / EC2 Fleet, Azure VM and Docker cloud plugins
Buildkite	buildkite-agent	Queues and tags	agents: { queue: "linux" }	Elastic CI Stack for AWS, agent-stack-k8s

PLATFORM	AGENT NAME	GROUPING	HOW A JOB PICKS IT	AUTOSCALING OPTION
CircleCI	Machine runner or container runner	Resource classes	resource_class: org/name	Container runner on Kubernetes
Bitbucket Pipelines	Runner (Linux Docker, Linux shell, Windows, macOS)	Labels	runs-on: [self.hosted, linux]	Your own orchestration
Forgejo / Gitea	forgejo-runner / act_runner (runs GitHub-Actions-style workflows)	Labels	runs-on: docker	Your own orchestration
TeamCity	Build agent	Agent pools	Agent requirements	Cloud profiles (AWS, Azure, GCP, Kubernetes, vSphere)

Table 2. The agent, its grouping and its autoscaling route on nine CI platforms.

Limits worth knowing on GitHub

- Self-hosted jobs can run for up to 5 days, while hosted jobs stop at 6 hours, which makes the self-hosted limit twenty times longer. Long builds, soak tests and big data jobs often fit only on self-hosted runners.
- A runner has to stay reasonably current. GitHub stops sending jobs to runner versions that fall too far behind, so let the runner update itself or rebuild its image on every runner release.
- Runner groups control which repositories may use which runners. Use them in organizations so an experimental repository can’t schedule work onto the release-signing machine.

What hosted CI costs

GitHub Actions

GitHub includes a monthly allowance of minutes on private repositories: 2,000 on the Free plan, 3,000 on Pro and Team, and 50,000 on Enterprise Cloud. After that you pay per minute. Public repositories on standard hosted runners are free, and self-hosted runners are never billed. GitHub announced a platform fee of $0.002 a minute for self-hosted runners, due to start in March 2026. It then postponed the fee, and it is not charged. Had it arrived, it would have been a third of the hosted Linux rate.

HOSTED RUNNER (STANDARD)	$ PER MINUTE	1,000 MIN	10,000 MIN	VS LINUX
Linux 2-core x64	0.006	$6	$60	1x
Linux 2-core arm64	0.005	$5	$50	0.8x
Windows 2-core x64	0.010	$10	$100	1.7x
macOS 3/4-core (M1 or Intel)	0.062	$62	$620	10x

Table 3. GitHub's standard hosted runner rates on private repositories, with the multiple of the Linux price that decides which jobs to move first.

Larger runners are billed even on public repositories. At January 2026 rates, a Windows 64-core runner costs $0.194 a minute, a macOS 5-core M2 Pro $0.102, a Linux 4-core GPU runner $0.052 and a Windows 4-core GPU runner $0.102. One hour on the 64-core Windows runner costs about $11.64, more than thirty-two times an hour on the 2-core Linux runner. A 16-core desktop already sitting under a desk pays for itself fast at that rate.

GitLab.com

GitLab counts compute minutes as job duration multiplied by a cost factor. On GitLab-hosted runners, Linux small carries a factor of 1x, medium 2x, large 3x, XLarge 6x and

2XLarge 12x. Windows medium is 1x, macOS M1 medium is 6x and M2 Pro large 12x, and GPU medium is 7x. The Free tier includes 400 compute minutes a month, and public open-source projects get a 0.5 factor, which halves what they use. Project and group runners you host yourself consume no compute minutes at all, which is why many GitLab.com users run their own runners even for small projects.

Azure DevOps

Azure Pipelines sells parallel jobs, meaning the number of jobs allowed to run at once, rather than minutes. On Microsoft-hosted agents, private projects get one free job, limited to 60 minutes per run and 1,800 minutes a month. Paid Microsoft-hosted jobs have no monthly cap and a 360-minute limit per job. On the self-hosted side you get one free parallel job, plus one more for each Visual Studio Enterprise subscriber, with no time limits. You can register any number of agents. Extra self-hosted parallel jobs cost considerably less than Microsoft-hosted ones; the Azure DevOps pricing page carries the current figures. On Azure DevOps Server, the on-premises edition, self-hosted concurrency is free. Public projects are being retired and convert to private in 2027, which removes their free hosted grant.

The others in brief

- Bitbucket Pipelines includes build minutes per plan, and steps that run on self-hosted runners don’t use build minutes.
- CircleCI is credit-based, and its self-hosted runners have their own plan terms, so check current pricing.
- Buildkite runs its control plane as a hosted service and the agents are always yours, so it carries no hosted per-minute charge for self-hosted agents.
- Jenkins, Forgejo, Gitea and an on-premises TeamCity leave both the controller and the agents with you. The costs there are hardware, licences (in TeamCity’s case) and operations.

Why it's cheaper, and the arithmetic that proves it

The mechanism is simple. A hosted runner bundles a machine, its electricity, its maintenance and the vendor's margin into a per-minute price. You pay that price every minute a job runs. A self-hosted runner unbundles them. The machine becomes a one-off cost, or no cost if you already own it. The electricity is metered by your utility at cents per hour; the maintenance is your time. Every additional minute of CI on owned hardware is close to free — the machine was paid for once — while every additional hosted minute costs the same as the first.

A self-hosted runner pays off when:

months to payback = hardware cost ÷ (hosted $ per month − power $ per month − upkeep $ per month)

If the hardware already exists (your PC, a NAS, an old laptop), the hardware cost is zero and the runner pays from day one, provided the upkeep stays small. That is the whole case for starting at home.

Power barely registers. A box drawing 60 W on average uses 43.8 kWh a month. At $0.17 per kWh that comes to about $7.45, the hosted price of roughly 745 Windows minutes. A 15 W mini PC uses about 11 kWh and costs under $2 a month.

Three examples show how fast the formula moves:

- A Mac mini against hosted macOS. A new Mac mini starts at about $600. At $0.062 per hosted macOS minute that is about 9,700 minutes, roughly one busy month for a small iOS team. After that, the Mac's minutes cost only its electricity.
- A small team's mixed fleet. The worked scenario later in this piece puts a small team's hosted bill at $526 a month. Against about $1,000 to $1,500 of new hardware and about $15 of power a month, plus a few hours of upkeep, the hardware pays back in about three months.
- A mid-size company on Kubernetes. This one starts at $4,860 a month in hosted minutes. A platform of Kubernetes on spot instances, a Windows VM pool and Mac minis should be planned at 30 to 60 percent of that bill in running costs, counting

spot capacity and operations time. Payback takes 6 to 12 months once engineering time is included.

Two things change the answer more than the rates do. Engineer time comes first: a runner platform that needs a day a week of someone's attention is expensive, and the formula's upkeep term is where that shows up. Build speed comes second, pushing the other way, because a faster pipeline on warm caches saves more developer time than the minutes ever cost. An honest model includes both.

Where in the stack the savings come from



Figure 4. Four layers save money outright, two save smaller amounts, and the dashed bottom row is the operations cost that self-hosting adds.

Compute minutes

This is the line item on the bill. The multiplier matters more than the volume: 10,000 macOS minutes cost as much as about 100,000 Linux minutes. Rank your jobs by minutes multiplied by rate and move them from the top. The typical first moves run in this order:

1. macOS and iOS builds go to a Mac mini.
2. Windows installers, .NET, C++ and game builds go to a Windows workstation.
3. The long tail of Linux test jobs goes to WSL2, Docker or a NAS.

Large and special runners

Hosted GPU runners, runners with 8 to 64 cores and macOS runners are billed even on public repositories. Their self-hosted equivalents are often hardware already bought for another reason: a gaming or machine-learning workstation, a render node, a developer's Mac. Tag them with labels such as gpu, cuda-12, xcode-16 or cores-32, and route only the jobs that need them.

Parallelism, seats and plan tiers

Where concurrency is sold, as with Azure DevOps parallel jobs and some CircleCI plans, self-hosted concurrency is cheaper or free. Where minutes are pooled per plan, as on GitHub and GitLab, moving the heavy jobs off hosted runners can keep you inside the included allowance. That avoids a plan upgrade you were only buying for minutes.

Build time and developer wait

The largest saving is often invisible on the invoice. Hosted runners start clean, so every job downloads its dependencies, rebuilds its Docker layers and recompiles from scratch. A persistent self-hosted runner with a big disk keeps node_modules, ~/.m2, ~/.gradle, ~/.nuget, ~/.cargo, ccache or sccache (compiler output caches) and Docker layers between runs. A 12-minute pipeline that becomes 4 minutes saves 8 minutes of

every developer's wait on every push, a cut of two thirds. On a team of ten or more, that engineer time dwarfs the minute bill.

A 16-core desktop also beats a 2-core hosted runner on anything parallel (compilation, test sharding, image builds), since it has eight times the cores to spread the work across.

Storage

Hosted CI bills artifact, cache and package storage above plan quotas, and caches are capped; GitHub's default is 10 GB per repository. A NAS gives you terabytes for artifacts, test reports, build caches and container images, hundreds of times that cap. Keep release artifacts in the platform, since they're small and need to be public, and keep bulk intermediates on the local network.

Network: pulls, rate limits and egress

- Registries rate-limit anonymous pulls (Docker Hub does), so a busy CI fleet starts failing with HTTP 429 "Too Many Requests" errors. A pull-through cache on the local network, which fetches each image once and serves it locally afterwards, removes the problem.
- Cloud providers bill egress, the data that leaves their network. Runners in the same region or virtual private cloud as your deploy targets, artifact buckets and databases avoid it, and they can reach private endpoints without a VPN.
- Package mirrors for npm, PyPI, Maven, NuGet and apt cut both bandwidth and install time.

The cost you add: operations

Budget for it honestly. It covers OS patching, runner updates, disk cleanup, toolchain drift, security isolation, monitoring, and someone's time when a runner is wedged at 2 a.m. The fleet-management section shows how to keep this to minutes a month with images, infrastructure as code and ephemeral runners. If you can't automate it, keep the runner count small and keep the hosted fallback.

The operating system decides the host

OS / ARCHITECTURE	HOSTS THAT CAN PROVIDE IT	TYPICAL JOBS	SETUP NOTES
Linux x64	Any PC via WSL2, Docker, a VM, a NAS, Kubernetes, any cloud	Web apps, backend tests, containers, most CI	The easiest and cheapest; containers give clean state
Linux arm64	Raspberry Pi 5 (light jobs), Ampere/Graviton cloud VMs, Oracle Always Free, Apple silicon via Linux VMs	arm64 images, embedded Linux, cost-efficient cloud CI	Native multi-arch builds without QEMU emulation, which is many times slower
Windows x64	Windows PC or server, Hyper-V/Proxmox/ESXi VMs, cloud Windows VMs	MSI/NSIS installers, .NET Framework, MSVC C++, Electron, code signing, games	Needs a Windows licence per VM; long paths; Defender exclusions
Windows arm64	Snapdragon laptops, Azure arm64 VMs	Native arm64 Windows builds and tests	Toolchain support is still uneven; test what you need
macOS arm64	Mac mini/Studio/Pro, rented cloud Macs	Xcode, iOS, macOS apps, notarization, Swift	Apple hardware required; licence allows at most two extra macOS VMs per Mac
Android	Linux x64 with KVM (bare metal or nested virtualization)	Emulator tests	Emulators need hardware virtualization; many cloud VMs lack nested virt
FreeBSD / other	VMs	Portability testing	GitHub's runner has no official FreeBSD build; use VMs driven from a Linux runner, or a

OS / ARCHITECTURE	HOSTS THAT CAN PROVIDE IT	TYPICAL JOBS	SETUP NOTES
			platform with a native agent

Table 4. Which hosts can supply each operating system and architecture, and the setup catch for each.

QEMU, named in the arm64 row, is the emulator that runs one processor's code on another. KVM is the Linux kernel's built-in hypervisor, the layer that runs virtual machines at near-native speed. The rule of thumb follows from the table: Linux jobs can go almost anywhere, Windows jobs need Windows, and macOS jobs need Apple hardware. Pick the host by the operating system first, then by the isolation the job needs, then by cost.

Where to run them: the hosting options



Figure 5. A small fleet keeps most jobs on three owned machines, with every one of them dialling out on port 443, and calls on the cloud only for bursts.

Each option below follows the same pattern: when it fits, how to set it up, how a pipeline uses it, and what it gains and costs. The setup commands use GitHub Actions; the platform section that follows gives the GitLab, Azure DevOps, Jenkins and other equivalents.

Windows workstation or server, as a native service

When it fits. Use it for Windows-only builds: installers, MSVC (Microsoft's C++ compiler), .NET Framework, WPF desktop apps, code signing with a hardware token, and Unreal or Unity Windows targets. It also fits whenever the PC is already on all day. It's the cheapest place to start.

Setup. Get a registration token from Settings > Actions > Runners > New self-hosted runner, then run this in an elevated PowerShell:

```
mkdir C:\actions-runner; cd C:\actions-runner
# download and unzip actions-runner-win-x64-<version>.zip from the settings page
./config.cmd --url https://github.com/OWNER/REPO --token <TOKEN> `
  --name pc-win --labels windows-build --runasservice `
  --windowslogonaccount .\ci-runner --windowslogonpassword <PASSWORD> --unattended
```

Hardening and tuning. Six changes make the service safer and faster.

- Run the service as a dedicated local account (ci-runner) rather than your admin account. The default NETWORK SERVICE account works for simple jobs, but it has no user profile for tools that expect one.
- Use a short work path (C:\actions-runner_work) and enable long paths with git config --system core.longpaths true plus the LongPathsEnabled registry value.
- Add a Microsoft Defender exclusion for the work directory and the package caches. Real-time scanning of node_modules can double build time — exclude it, and keep scanning on for downloads.
- Install toolchains with winget or Chocolatey from a script kept in the repository, so a rebuilt machine gets the same tools.
- Set the power options so the PC doesn't sleep while the runner is expected online, or accept sleep and rely on the hosted fallback described in the pipelines section.
- For code signing, keep the certificate in the runner account's store or on a hardware token, and restrict the signing runner to release workflows with a runner group or a dedicated label.

Usage. A workflow job asks for the runner by its labels.

```
jobs:
  installer:
    runs-on: [self-hosted, windows, windows-build]
    timeout-minutes: 30
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm run dist
```

PROS	CONS
Replaces the most expensive non-Mac standard minutes, $0.010 each, at zero per-minute cost	The PC sleeps, reboots for updates and gets used for other work
Runs the Windows-only toolchains that Linux hosts can't	Persistent state drifts between jobs unless it is cleaned
Can hold a signing certificate or hardware token that never leaves the building	A job runs with the service account's rights on your own machine
Warm caches and many cores make builds faster than on a 2-core hosted runner	Defender, long paths and missing user profiles need tuning first

Table 5. What a native Windows service runner saves, set against what it asks of you.

Linux bare metal: a desktop, mini PC or server

When it fits. Use it for always-on Linux capacity. A retired desktop or a 15 W mini PC is ideal, since either one can sit in a cupboard drawing a few watts and take jobs around the clock without anyone noticing it's there.

Setup. Create a dedicated user, register the runner as that user, and install it as a systemd service.

```
sudo useradd -m -s /bin/bash runner
sudo -iu runner bash -c '
```

```
  mkdir -p ~/actions-runner && cd ~/actions-runner
  # download and extract actions-runner-linux-x64-<version>.tar.gz
  ./config.sh --url https://github.com/OWNER/REPO --token <TOKEN> \
    --name mini-1 --labels linux-fast --unattended'
cd /home/runner/actions-runner && sudo ./svc.sh install runner && sudo ./svc.sh
start
```

- Give the runner user passwordless sudo or membership of the docker group only when jobs truly need it, because both are effectively root.
- If jobs need Docker, prefer rootless Docker or Podman for that user, or run the runner itself in a container as the Docker section describes.
- Put the work directory on an SSD with room to spare. A full disk is the most common failure on persistent runners.

PROS	CONS
Cheapest always-on capacity; a 15 W mini PC costs under $2 a month in power	Persistent state drifts unless jobs run in containers
Native Linux speed with no VM or WSL layer in the way	Sudo or docker-group access hands jobs root on the host
Old hardware gets a second life at no purchase cost	The disk fills up if nobody prunes it

Table 6. Bare-metal Linux trades a little upkeep for the cheapest always-on capacity.

macOS: Mac mini, Mac Studio or a cloud Mac

When it fits. Use it for anything that needs Xcode: iOS and macOS apps, Swift packages, notarization and Safari tests.

Setup. Register the runner and install it as a launchd agent.

mkdir ~/actions-runner && cd ~/actions-runner
download actions-runner-osx-arm64-<version>.tar.gz

./config.sh --url https://github.com/OWNER/REPO --token <TOKEN> --name mini-m4 --labels xcode-16 --unattended
./svc.sh install && ./svc.sh start # installs a launchd agent for the logged-in user

- Enable automatic login for the runner user if jobs need a GUI session, as UI tests, simulators and keychain prompts do.
- Keep several Xcode versions side by side (/Applications/Xcode_16.app) and select one per job with xcode-select or DEVELOPER_DIR.
- Use a separate keychain for signing identities, unlock it in the job from a secret, and delete it in cleanup.
- For clean state, run jobs in macOS VMs with Tart or Anka, two tools that manage macOS virtual machines on a Mac. Apple's licence allows up to two additional macOS VMs per Mac, so one Mac mini gives you up to two clean parallel runners.
- A new Mac mini starts at about $600. At GitHub's $0.062 per hosted macOS minute that is about 9,700 minutes, roughly one busy month for a small iOS team.

PROS	CONS
Fastest payback of any runner, because macOS minutes cost ten times Linux	Apple hardware is required to run Xcode
One Mac mini can host two clean VM runners under Apple's licence	GUI sessions, keychains and permission prompts need care
Several Xcode versions can live side by side on one machine	The licence caps VMs at two extra per Mac, which limits density

Table 7. A self-hosted Mac pays back fastest and carries the tightest licence limit.

WSL2: Linux jobs on a Windows PC

When it fits. WSL2, the Windows Subsystem for Linux, runs a real Linux kernel in a lightweight VM inside Windows. Use it when you have a Windows PC and want Linux CI on it without a separate machine or hypervisor.

Setup. Turn on systemd inside Ubuntu, then register the runner as a service and cap what WSL may use.

```
# inside Ubuntu on WSL2
sudo tee /etc/wsl.conf >/dev/null <<'EOF'
[boot]
systemd=true
EOF
# from Windows: wsl --shutdown, then reopen Ubuntu
mkdir ~/actions-runner && cd ~/actions-runner
# download and extract actions-runner-linux-x64
./config.sh --url https://github.com/OWNER/REPO --token <TOKEN> --name pc-wsl --labels wsl --unattended
sudo ./svc.sh install && sudo ./svc.sh start
```

```
# %UserProfile%\.wslconfig on Windows
[wsl2]
memory=12GB
processors=8
vmIdleTimeout=-1
```

- WSL shuts down when no Windows process holds it open. Start it at logon with a scheduled task that runs wsl -d Ubuntu --exec sleep infinity, or disable the idle timeout as above; support for vmIdleTimeout depends on the WSL version.
- Keep repositories inside the Linux filesystem (~/). Builds under /mnt/c run several times slower because every file access crosses the VM boundary.
- Cap memory and CPU so a runaway build can't starve the Windows desktop.
- WSL2 has no GUI session by default, so headless browser and Electron tests need xvfb-run, which supplies a virtual display.

PROS	CONS
Linux CI on a PC you already own, with no new hardware	WSL stops when idle unless something holds it open
Linux and Windows minutes can both come off one machine	Builds under /mnt/c run several times slower
Memory and CPU caps protect the desktop from runaway builds	No GUI session by default, so headless tests need xvfb

Table 8. WSL2 adds Linux CI to a PC you own, at the price of keeping WSL awake.

Docker and Podman containers

When it fits. A container packages a program with everything it needs and runs it isolated from the host. Use containers for several isolated Linux runners on one host, with clean state on every restart and easy scaling. One definition works the same on a PC, a NAS or a server.

Setup. One compose file defines an ephemeral runner container.

```
# compose.yaml
services:
  runner:
    image: ghcr.io/actions/actions-runner:latest
    restart: always
    environment:
      REPO_URL: https://github.com/OWNER/REPO
      RUNNER_TOKEN: ${RUNNER_TOKEN}
    command: >
      bash -c "./config.sh --url $$REPO_URL --token $$RUNNER_TOKEN
               --name docker-$$HOSTNAME --labels docker --ephemeral --unattended
               && ./run.sh"
    deploy:
      resources:
        limits: { cpus: "4", memory: 8g }
```

Run docker compose up -d --scale runner=3 for three parallel ephemeral runners.

Token handling. Registration tokens expire after an hour. For unattended fleets, have the container fetch a fresh registration token (or a JIT config) from the API at start-up, authenticating with a GitHub App or a fine-grained token. Community images such as myoung34/github-runner wrap this for you.

Docker inside CI jobs. There are four ways to give a job Docker, listed here from safest to riskiest.

OPTION	HOW	RISK
Rootless build tools	Buildah, Kaniko-style builders or BuildKit rootless inside the job	Lowest; no daemon access
Docker-in-Docker with Sysbox	Sysbox runtime runs an inner Docker without --privileged	Low-medium
Privileged DinD	docker:dind sidecar with --privileged	High: container escape is easy
Host socket mount	-v /var/run/docker.sock:/var/run/docker.sock	Highest: every job is root on the host

Table 9. Four ways to give a CI job Docker, ranked by how much of the host a hostile job could reach.

Podman runs rootless by default and works well for runner containers on Linux hosts. On Windows, Docker Desktop and Podman Desktop both run Linux containers inside WSL2, so a containerised runner on Windows is still a Linux runner.

Windows containers exist (GitLab's docker-windows executor uses them), but the images are large and must match the host's Windows build. For most teams, a Windows VM or a native Windows service is simpler.

PROS	CONS
Clean state per job with --ephemeral and a restart policy	Ephemeral containers start with cold caches
Several runners on one host, scaled with one command	Registration tokens expire hourly, so unattended fleets need automation
The same compose file runs on a PC, a NAS or a server	Docker inside jobs ranges from safe to root-on-host, depending on the method

Table 10. Container runners buy clean state per job and pay for it in cold caches.

NAS: Synology, QNAP, TrueNAS SCALE or Unraid

A NAS, or network-attached storage box, is on around the clock, uses little power, carries a lot of disk and usually runs containers. That makes it a good always-on Linux runner host — and an even better cache and artifact server for the rest of the fleet.

NAS PLATFORM	CONTAINER SUPPORT	RUNNER NOTES
Synology DSM 7.2+	Container Manager (Docker and Compose projects) on supported models, mostly x86 Plus series	Use a compose project like the one in the Docker section; put volumes on SSD cache if present
QNAP QTS / QuTS hero	Container Station (Docker, Compose, LXD)	Same compose file; watch RAM on low-end models
TrueNAS SCALE 24.10+	Apps are Docker-based; custom apps from compose YAML	Good for runners plus MinIO or Garage for S3-compatible storage
Unraid	Docker templates and Compose plugin; VMs via KVM	Community runner templates exist; VMs can host Windows runners
Any NAS without containers	Use it as SMB/NFS or S3 storage only	Runners elsewhere, caches here

Table 11. Container support and runner notes for the common NAS platforms.

Hardware. Choose x86 with 16 GB of RAM or more for runners. ARM NAS units work for arm64-compatible jobs, but they're slow for builds. Put runner work directories on SSD and keep them off a parity-protected spinning array, since builds are heavy on random I/O.

Four services are worth running on the NAS for the whole fleet:

- A container registry pull-through cache, using the CNCF Distribution registry (registry:2) with a proxy remoteurl, or Harbor with proxy-cache projects.
- Package proxies, meaning Verdaccio for npm, devpi for PyPI, apt-cacher-ng for Debian and Ubuntu, or Nexus Repository or Artifactory for many formats at once.
- S3-compatible object storage for build caches and artifacts, using MinIO, Garage or SeaweedFS. S3 is Amazon's storage interface, and these three speak it on your own disks.
- Remote build caches, such as a Bazel remote cache (bazel-remote), a Gradle build cache node, an sccache S3 bucket, or a Turborepo or Nx remote cache.

PROS	CONS
Always on at low power, with no sleeping-PC problem	Low-end models run short of RAM and CPU for builds
Terabytes of disk for caches, artifacts and images	ARM units are slow for builds and limited to arm64 jobs
Serves caches and mirrors that speed up every other runner	Spinning parity arrays make poor work directories

Table 12. A NAS earns its place as much for the caches it serves as for the jobs it runs.

Kubernetes

When it fits. Kubernetes is a system that schedules containers across a cluster of machines and restarts them when they fail. Use it for many parallel Linux jobs, bursty load, teams that already run Kubernetes, or a homelab running k3s, a lightweight Kubernetes distribution. It gives you an ephemeral pod per job, bin-packing of jobs onto nodes, autoscaling and one control surface for the whole fleet. It is also the heaviest option here.

GitHub: Actions Runner Controller (ARC) with runner scale sets. ARC is GitHub's Kubernetes controller for runners, and a scale set is the autoscaling group of runners it manages.

```
# 1. The controller
helm install arc --namespace arc-systems --create-namespace \
  oci://ghcr.io/actions/actions-runner-controller-charts/gha-runner-scale-set-
controller

# 2. A runner scale set for one repository or organization
helm install arc-runner-set --namespace arc-runners --create-namespace \
  --set githubConfigUrl="https://github.com/OWNER/REPO" \
  --set githubConfigSecret.github_token="$GITHUB_PAT" \
  --set minRunners=0 --set maxRunners=10 \
  oci://ghcr.io/actions/actions-runner-controller-charts/gha-runner-scale-set
```

```
jobs:
  test:
    runs-on: arc-runner-set     # the scale set's installation name
```

- Use a GitHub App instead of a PAT, a personal access token, in production.
- containerMode: dind gives jobs a Docker daemon in a privileged sidecar container, while containerMode: kubernetes runs job containers as separate pods with no privileged sidecar.
- ARC's listener and controller expose Prometheus metrics (jobs queued, running and completed, and runner counts) for capacity planning.

- Pair it with a node autoscaler, Cluster Autoscaler or Karpenter, on spot nodes so the cluster itself scales to zero or near zero.

Other platforms. On GitLab, install the gitlab-runner Helm chart with the Kubernetes executor, and each job becomes a pod. On Jenkins, the Kubernetes plugin launches agent pods per build. Buildkite uses agent-stack-k8s, and CircleCI uses its container runner on Kubernetes.

Homelab Kubernetes. k3s on a mini PC or a NAS VM is enough for ARC. A three-node k3s cluster of used mini PCs gives redundancy for less than a few months of hosted Windows minutes would cost.

PROS	CONS
Ephemeral pod per job, so every job starts clean	A cluster is a system to run, patch and understand
Scales from zero to maxRunners and back with demand	Cold caches unless a registry and build cache sit nearby
One control surface and Prometheus metrics for the fleet	Docker-in-Docker mode needs a privileged sidecar
Spot node pools push the idle cost toward zero	Pays off only once there are enough parallel jobs

Table 13. Kubernetes runners scale to zero, provided someone is willing to run the cluster.

On-premises virtual machines: Proxmox, Hyper-V, VMware ESXi or KVM

When it fits. A virtual machine is a whole computer simulated in software on a host, and a hypervisor is the software that runs it. Use VMs for Windows runners with clean state, mixed-OS fleets, stronger isolation than containers give, or to reuse an existing virtualization cluster.

The golden-image pattern. A golden image is a prebuilt template with the OS, the tools and the runner already installed.

1. Build a template VM with the OS, the toolchains and the runner program installed but not registered. On Windows, generalize it with Sysprep; on Linux, install cloud-init, the standard first-boot configuration tool.
2. Per job (or per day), clone the template, using linked clones on Proxmox, differencing disks on Hyper-V or instant clones on vSphere.
3. On first boot, a startup script fetches a JIT config or a registration token and starts the runner with --ephemeral.
4. After the job, the VM shuts down and the orchestrator deletes it.

A small orchestrator can be a scheduled script. It polls the GitHub API (or receives the workflow_job webhook) for queued jobs carrying your label, then clones a VM. It passes the JIT config in through cloud-init or a Hyper-V integration service, and it deletes the VM when the job completes. Rebuild the template monthly with Packer, HashiCorp's image-building tool, so patches and toolchains stay current.

Hyper-V specifics. Use Generation 2 VMs, dynamic memory, and checkpoints for quick revert; New-VHD -Differencing makes per-job disks in seconds. Proxmox specifics. Run qm clone <template> <newid> --full 0 for linked clones, use cloud-init drives for Linux, and drive it from anywhere through the Proxmox API.

Licensing. Budget for it. Every Windows VM needs a licence. Windows Server Datacenter covers unlimited VMs on the licensed host, which is what enterprises use for Windows runner pools.

PROS	CONS
Clean Windows state per job, which containers give only awkwardly	Every Windows VM needs a licence
Stronger isolation than containers for less-trusted code	Templates need monthly rebuilds to stay patched
Reuses a virtualization cluster you may already own	Needs an orchestrator script or tool to clone and delete VMs

Table 14. On-premises VMs give clean Windows state per job, with a licence for every VM.

Cloud instances: on-demand, spot and reserved

When it fits. Use cloud instances for always-on capacity without owning hardware, and for bursts beyond what the local fleet can take. They also cover architectures you lack (arm64, GPU) and runners that should sit close to cloud-hosted deploy targets.

PRICING MODEL	DISCOUNT VS ON-DEMAND	FIT FOR CI
On-demand	none	Short-lived burst runners; simplest
Spot / preemptible (AWS Spot, Azure Spot, GCP Spot VMs)	Providers advertise up to about 90%	Ephemeral runners; jobs must tolerate interruption and retry
Reserved instances / savings plans / committed use	Up to roughly 70% for 1-3 year commitments	Steady always-on runner baseline
Free tiers	100% within limits	Oracle Always Free Ampere A1 (1,500 OCPU-hours and 9,000 GB-hours a month; idle instances reclaimed)
Budget VPS (Hetzner, OVHcloud, Scaleway, Contabo)	Flat monthly	Always-on small Linux runners; check current price lists

Table 15. Cloud pricing models and the kind of runner each one suits.

Spot is the cheap end. It is spare cloud capacity sold at a discount — and the provider can take it back at short notice, which is why only interruption-tolerant jobs belong on it.

Each platform has its own route to autoscaled cloud runners:

- GitHub on AWS. terraform-aws-github-runner (github-aws-runners) listens to workflow_job webhooks and launches EC2 runners, spot or on-demand, one per job, scaling to zero.
- GitHub on any cloud. ARC runs on EKS, AKS or GKE, the managed Kubernetes services of AWS, Azure and Google, with spot node pools.
- GitLab. The docker-autoscaler and instance executors use fleeting plugins for AWS, Google Cloud and Azure, replacing the deprecated Docker Machine executor.

- Azure DevOps. Azure VM Scale Set agent pools or Managed DevOps Pools create and recycle agents for you.
- Jenkins. The EC2 Fleet, Azure VM Agents and Google Compute Engine plugins add cloud capacity.
- Buildkite. The Elastic CI Stack for AWS does the same job there.

Setup pattern on any cloud. Bake an image with Packer and launch from a launch template or instance template. Pass a JIT config or a short-lived token through user-data or instance metadata, run the job ephemerally, and terminate. Keep instances in private subnets with NAT or egress controls, since the runner needs only outbound port 443.

Spot hygiene. Set timeout-minutes, keep jobs short, and make them idempotent, meaning safe to run twice. Allow automatic re-runs, and use several instance types and zones so a capacity shortfall in one pool doesn't block CI.

PROS	CONS
No hardware to buy, and capacity in minutes	Billed while running, so idle instances still cost money
Spot prices advertised up to about 90% below on-demand	Spot instances can vanish mid-job
arm64, GPU and regions you don't own	Egress and storage charges apply when data leaves the region
Sits next to cloud deploy targets, avoiding egress and VPNs	Needs images, templates and an autoscaler to run well

Table 16. Cloud instances need no hardware and charge for every hour they stay up.

Cloud Macs

AWS EC2 Mac is a dedicated host billed per second, with a 24-hour minimum allocation to comply with Apple's licence. Specialist Mac hosts such as MacStadium and

Scaleway Apple silicon, among others, offer monthly or daily pricing. Because of the 24-hour minimum and the dedicated-host model, cloud Macs suit steady load rather than per-job bursts. For most small teams, one or two owned Mac minis come out cheaper within a month or two.

PROS	CONS
macOS capacity with no hardware on site	A 24-hour minimum allocation on EC2 Mac makes bursts expensive
Suits steady, predictable iOS load	Owned Mac minis are usually cheaper within a month or two

Table 17. Cloud Macs suit steady iOS load and punish bursts with a 24-hour minimum.

GPU runners

GPU jobs (machine-learning training smoke tests, CUDA builds, rendering, game cooking) are the most expensive hosted minutes after macOS. There are three self-hosted routes. A workstation GPU can run containerised jobs with the NVIDIA driver and the NVIDIA Container Toolkit. A Kubernetes node pool can carry the NVIDIA device plugin, where jobs request nvidia.com/gpu: 1. Or spot GPU instances can be launched per job. Label precisely (gpu, cuda-12, vram-24g) and keep non-GPU jobs off these machines.

PROS	CONS
Replaces hosted GPU minutes that are billed even on public repos	GPU drivers and container toolkits need upkeep
Often uses a workstation GPU already bought for other work	Non-GPU jobs will squat on the machine unless labels keep them off

Table 18. GPU runners replace the second most expensive hosted minutes, given careful labels.

MicroVMs and sandboxed runtimes

Firecracker microVMs boot in well under a second and give VM-level isolation at container-like density. Some runner products and self-built fleets use them for ephemeral runners that are both clean and fast. gVisor and Kata Containers are the Kubernetes-native equivalents, applied as a runtime class per pod. Use them when you run code you don't fully trust and still want self-hosted economics.

PROS	CONS
VM-level isolation that boots in under a second	More moving parts than plain containers
Safe enough for less-trusted code	Fewer ready-made runner images and guides

Table 19. MicroVMs add VM isolation at near-container speed, with more parts to look after.

Third-party hosted runners, the middle option

A group of vendors sell drop-in replacements for GitHub-hosted runners, usually switched on by changing the runs-on label. They advertise lower per-minute prices and faster hardware than GitHub's own. Examples at the time of writing include Blacksmith, Namespace, WarpBuild, Ubicloud, BuildJet, Depot (for container builds) and RunsOn, which runs in your own AWS account. They remove the operations burden — and they still bill per minute. Compare them against both GitHub's rates and your own hardware, and verify pricing, data handling and supported operating systems directly with each vendor.

PROS	CONS
No machines to patch, clean or secure	Still billed per minute
Usually a one-line runs-on change	Your code and secrets run on another vendor's machines
Advertised as cheaper and faster than GitHub-hosted	Pricing, data handling and OS support vary by vendor

Table 20. Third-party runners remove the operations work and keep the per-minute bill.

Setting up runners on each CI platform

GitHub Actions

The hosting options above already walk through GitHub's runner on every host. The key commands are config.sh on Linux and macOS, or config.cmd on Windows, with --url, --token, --labels, --runnergroup, --ephemeral and, on Windows, --runasservice. On Linux and macOS, svc.sh install|start|status|stop|uninstall manages the service. Remove a runner with config.sh remove --token <REMOVAL_TOKEN>. For automation, three REST endpoints do most of the work: POST /repos/{owner}/{repo}/actions/runners/registration-token, POST /orgs/{org}/actions/runners/generate-jitconfig and GET /repos/{owner}/{repo}/actions/runners.

GitLab Runner

```
# Linux: install the package from packages.gitlab.com, then register with a runner authentication token
sudo gitlab-runner register --non-interactive \
  --url https://gitlab.com --token glrt-XXXXXXXX \
  --executor docker --docker-image alpine:latest --description nas-docker
```

```
# /etc/gitlab-runner/config.toml (excerpt)
concurrent = 4
[[runners]]
  name = "nas-docker"
  executor = "docker"
  [runners.docker]
    image = "alpine:latest"
    privileged = false
    volumes = ["/cache"]
  [runners.cache]
    Type = "s3"
    [runners.cache.s3]
      ServerAddress = "nas.lan:9000"
      BucketName = "gitlab-cache"
      Insecure = true
```

Create the runner in the GitLab UI, at project, group or instance level, to get the glrt-token, and set its tags there. On Windows, install gitlab-runner.exe, run gitlab-runner install and gitlab-runner start, and use either the shell executor with PowerShell or the docker-windows executor. The [runners.cache] block pointed at MinIO on a NAS gives the whole fleet one shared cache.

Azure DevOps agents

Windows
.\config.cmd --unattended --url https://dev.azure.com/ORG --auth pat --token <PAT> `
--pool SelfHosted --agent PC-WIN --runAsService --work _work

Linux / macOS
./config.sh --unattended --url https://dev.azure.com/ORG --auth pat --token <PAT> \
--pool SelfHosted --agent nas-1 --acceptTeeEula
sudo ./svc.sh install && sudo ./svc.sh start

Agents advertise capabilities, such as installed tools and environment variables, and pipelines select them with demands. For autoscaling, create a VM Scale Set agent pool or a Managed DevOps Pool, and Azure DevOps adds and removes agents to match the queue.

Jenkins agents

Create a node with labels under Manage Jenkins > Nodes, then start the agent on the machine:

java -jar agent.jar -url https://jenkins.example.com/ -secret <SECRET> -name win-builder -workDir C:\jenkins

The controller can also connect over SSH to Linux and macOS agents. For elastic capacity, use the Kubernetes plugin with a pod template per label, or the EC2 Fleet, Azure VM or Docker cloud plugins. Jenkins has no per-minute billing at all, so its whole bill is infrastructure, and the same caching and ephemeral advice applies to it.

Buildkite, Bitbucket, CircleCI, Forgejo and Gitea

- Buildkite. Start the agent with buildkite-agent start --token <AGENT_TOKEN> --tags "queue=linux,os=linux", and pipelines target it with agents: { queue: "linux" }.
- Bitbucket Pipelines. Add a runner under Repository or Workspace settings > Runners, run the Docker command it provides (Linux) or its PowerShell script (Windows), then put runs-on: [self.hosted, linux] in a step.
- CircleCI. Create a resource class, install the machine runner (Linux, macOS, Windows) or the container runner (a Kubernetes Helm chart), and set machine: true with resource_class: <namespace>/<name>.
- Forgejo and Gitea. Register with forgejo-runner register or act_runner register, giving the instance URL and token. Labels map to Docker images or to the host itself (ubuntu-latest:docker://node:20-bookworm, windows:host). Workflows use GitHub Actions syntax, which makes this a popular fully self-hosted stack for homelabs and air-gapped sites.

Using runners in pipelines

The basic pattern

```
name: ci
on: [push, pull_request]
jobs:
  test:
    runs-on: [self-hosted, linux, x64]
    timeout-minutes: 20
    concurrency:
      group: ci-${{ github.ref }}
      cancel-in-progress: true
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 22, cache: npm }
      - run: npm ci
      - run: npm test
```

Always set timeout-minutes. Without it a lost runner holds the job until the platform's own limit — five days, on a self-hosted runner. Set a concurrency group as well, so a newer push cancels the run it superseded.

Self-hosted when online, hosted otherwise

This is the fix for the sleeping PC. Hard-coding self-hosted makes jobs queue for up to a day when the machine is off. Put the runner labels in a repository variable instead, and fall back to hosted runners when the variable is empty:

```
jobs:
  check:
    runs-on: ${{ (github.event_name != 'pull_request' ||
github.event.pull_request.head.repo.full_name == github.repository) && vars.RUNNER
&& fromJSON(vars.RUNNER) || 'ubuntu-latest' }}
```

The expression does two jobs at once. Its first half is a fork gate: a pull request whose code comes from a fork — someone else's copy of the repository — always goes to a hosted runner, so a stranger's code never reaches your machine. Its second half reads the variable, sending the job to your runner's labels when RUNNER is set and to ubuntu-latest when it isn't.



Figure 6. A fork pull request always takes the hosted path, and any other job reaches the PC only while the RUNNER variable says the PC is online.

A small script or app keeps the variable honest. Every few minutes it reads GET /repos/{owner}/{repo}/actions/runners. If a matching runner is online, it sets RUNNER to ["self-hosted","linux","my-label"], using PATCH /repos/{owner}/{repo}/actions/variables/RUNNER (or a POST to create it), and if not, it deletes the variable. Use the same pattern with WINDOWS_RUNNER and windows-latest for Windows jobs.

One matrix across operating systems and runner types

```
jobs:
  build:
    strategy:
      fail-fast: false
      matrix:
        include:
          - name: linux
            runs-on: '["self-hosted","linux","docker"]'
          - name: windows
            runs-on: '["self-hosted","windows","windows-build"]'
          - name: macos
            runs-on: '"macos-latest"'          # hosted: no Mac in the fleet yet
    name: build-${{ matrix.name }}
    runs-on: ${{ fromJSON(matrix.runs-on) }}
    steps:
      - uses: actions/checkout@v4
      - run: ./build.sh
```

Each leg of the matrix goes to the cheapest machine that can run it. Move the macOS leg to [”self-hosted”,”macOS”,”xcode-16”] when a Mac joins the fleet.

Container jobs on a self-hosted runner

```
jobs:
  test:
    runs-on: [self-hosted, linux, docker]
    container:
      image: registry.lan:5000/library/node:22-bookworm   # pulled through the NAS
mirror
    services:
      postgres:
        image: registry.lan:5000/library/postgres:16
        env: { POSTGRES_PASSWORD: ci }
    steps:
```

```
      - uses: actions/checkout@v4
      - run: npm ci && npm test
```

Container jobs give clean state even on a persistent runner; the service container gives each job its own database.

Docker image builds with a shared cache

```
- uses: docker/setup-buildx-action@v3
- uses: docker/build-push-action@v6
  with:
    push: true
    tags: registry.lan:5000/app:${{ github.sha }}
    cache-from: type=registry,ref=registry.lan:5000/app:buildcache
    cache-to: type=registry,ref=registry.lan:5000/app:buildcache,mode=max
```

BuildKit is Docker’s build engine, and a registry-backed BuildKit cache on the local network makes image builds incremental even across ephemeral runners that start empty.

Scale-set, GPU and release jobs

```
jobs:
  unit:
    runs-on: arc-runner-set             # Kubernetes, scales 0..N
  gpu-smoke:
    runs-on: [self-hosted, linux, gpu, cuda-12]
    timeout-minutes: 60
  release:
    if: startsWith(github.ref, 'refs/tags/v')
    runs-on: [self-hosted, windows, signing]   # restricted by a runner group
    environment: production                     # requires approval
```

Cleanup and hygiene steps

```
- name: Free disk (persistent runners)
  if: always()
  run: |
    docker system prune -af --filter "until=72h" || true
    rm -rf "$RUNNER_TEMP"/* || true
```

Hooks do this better. Set the runner's ACTIONS_RUNNER_HOOK_JOB_STARTED and ACTIONS_RUNNER_HOOK_JOB_COMPLETED environment variables to scripts that check free disk before a job and clean up after it. Every workflow then gets the same hygiene without anyone editing YAML.

GitLab, Azure DevOps and Jenkins equivalents

```
# .gitlab-ci.yml
test:
  tags: [nas-docker]          # your runner
  image: node:22
  script: [npm ci, npm test]

windows-build:
  tags: [windows]
  script: [npm ci, npm run dist]
```

```
# azure-pipelines.yml
pool:
  name: SelfHosted
  demands:
    - Agent.OS -equals Windows_NT
steps:
  - script: npm ci && npm run dist
```

```
// Jenkinsfile
pipeline {
  agent { label 'windows && docker' }
```

```
  options { timeout(time: 30, unit: 'MINUTES') }
  stages { stage('Build') { steps { bat 'npm ci && npm run dist' } } }
}
```

Caching and artifacts, the multiplier

Self-hosted runners pay twice when the caches sit next to them: once in minutes no longer billed, and again in minutes no longer spent downloading.

CACHE	TOOL	WHERE TO HOST	EFFECT
Container images	Registry pull-through cache (Distribution, Harbor, Nexus, Artifactory)	NAS or cluster	No rate limits, fast pulls, less bandwidth
npm / Yarn / pnpm	Verdaccio, Nexus	NAS	Installs from the LAN
PyPI	devpi, Nexus	NAS	Same
Maven / Gradle	Nexus, Artifactory; Gradle build cache node	NAS	Dependency and task-output reuse
NuGet	Nexus, Artifactory, BaGet	NAS or Windows host	Faster .NET restores
apt / dnf	apt-cacher-ng, a local mirror	NAS	Faster image builds
Compiler outputs	ccache, sccache (S3 backend)	Persistent disk or MinIO	Large C/C++/Rust speed-ups
Monorepo tasks	Bazel remote cache, Nx and Turborepo remote caches	NAS or cluster	Skip unchanged work across all runners
Docker layers	BuildKit registry cache (see the shared-cache build above)	Local registry	Incremental image builds

Table 21. Which cache to run for each kind of dependency, where it lives, and what it saves.

GitHub's cache action on self-hosted runners still stores caches in GitHub's own cache service, subject to its size limits and eviction. On persistent runners you can often skip it and rely on the local disk, or use an S3-compatible cache action pointed at your NAS. The runner tool cache is the other quiet win. Setup-node, setup-python and similar actions keep downloaded toolchains in RUNNER_TOOL_CACHE, and on a persistent runner that directory makes setup near-instant.

Running the fleet



Figure 7. Fleet management runs as a loop, and the feedback path from Observe back to Build images is where queue time and disk alerts turn into new capacity and patched images.

Images: build once, run anywhere

- Keep a Packer template (for VMs and cloud) or a Dockerfile (for containers) for each runner flavour in a repository.
- Pin toolchain versions, and update them with a pull request, the same way code changes.

- Rebuild monthly and whenever the runner program releases, so patches and runner updates arrive without hand work.
- For persistent Windows and macOS machines, keep a setup script (winget or Chocolatey, Homebrew) that can rebuild the machine from scratch.

Provisioning and configuration

Infrastructure as code, usually shortened to IaC, means describing machines and networks in files that a tool applies, so the fleet can be rebuilt from a repository. These are the usual tools for each job:

JOB	TOOLS
Cloud infrastructure	Terraform or OpenTofu, Pulumi, CloudFormation, Bicep
Machine configuration	Ansible, PowerShell DSC, cloud-init
Kubernetes	Helm, Argo CD or Flux (GitOps)
Hypervisors	Proxmox API or Terraform provider, Hyper-V PowerShell, vSphere provider
Secrets	GitHub App credentials in a secret store (Vault, AWS Secrets Manager, Azure Key Vault)

Table 22. Provisioning tools by job.

Labels, groups and routing

- Label by capability rather than by machine name: windows, gpu, cuda-12, xcode-16, signing, cores-16, and lan for jobs that need local network access.
- Use runner groups (GitHub organizations and enterprises), project and group runners (GitLab) or pools (Azure DevOps) to control which repositories may use which runners.

- Keep a hosted fallback for every label that isn't strictly required, using the switch from the pipelines section.

Updates

The runner program updates itself by default. Turning that off is common with images, and then the images need a rebuild on every runner release. GitHub stops routing jobs to runners that fall too far behind. Patch host operating systems on a schedule, and drain a persistent runner (stop the service between jobs) before rebooting it.

Monitoring and alerting

Watch these in order of importance:

1. Queue time is how long jobs wait for a runner. A growing queue time means you need capacity.
2. Runner online or offline, busy or idle comes from the platform API. Alert when a required label has no online runner.
3. Free disk and inode count matter on persistent runners. Alert at 20 percent free.
4. Job failure rate by runner shows drift. A runner that fails more often than its peers has drifted from its image, so rebuild it.
5. Cost covers the hosted minutes still billed, per repository and per SKU (the billing line for each runner type), plus the self-hosted minutes and the dollars they avoided at list price.

The data comes from the platform REST APIs and from workflow_job webhooks, whose queued, in_progress and completed timestamps give exact queue and run times. ARC's Prometheus metrics, GitLab Runner's Prometheus endpoint, Azure DevOps pool consumption reports and node exporters on the hosts fill in the rest. Grafana dashboards on the NAS or the cluster tie it together.

Capacity planning

Start with the concurrency you actually need, which is the peak number of simultaneous jobs during working hours. A rough rule from Azure DevOps' guidance is one parallel job for every four or five active developers, so a team of twenty would start near four or five. Measure your own queue times rather than guessing. Size persistent runners for the median job, and use ephemeral cloud capacity for the peaks.

Cost accounting

Count three numbers each month: hosted spend still billed, self-hosted minutes, and the list-price value of those minutes. The difference, minus power and hardware amortisation, is the real saving. Report it per repository so teams can see which pipelines are expensive.

Runbooks

Write short runbooks, step-by-step notes for the cases that recur: runner offline, disk full, stuck job, token rotation, adding a label, retiring a machine. Keep them next to the image definitions so they change when the images do.

Security

A self-hosted runner executes whatever code the workflow checks out, with the runner account's permissions and network access. Plan for that. Every rule below follows from it, and most of them cost nothing more than a setting changed once and a habit kept afterwards.

- Never attach self-hosted runners to public repositories. Anyone can open a pull request from a fork and get code running on your machine; public repositories get free hosted minutes anyway.
- Gate forks on private repositories too if forking is enabled. The expression in the fallback section does this.

- Apply least privilege. Use a dedicated low-privilege account with no passwordless sudo and no Docker socket unless it's essential. Keep personal files, browser profiles, SSH keys and cloud credentials off the runner account.
- Isolate by trust level. Containers suit trusted code, VMs or microVMs suit less-trusted code, and release signing and production deploys get separate machines and runner groups.
- Make runners ephemeral by default wherever you can afford cold starts. A persistent runner lets one compromised job plant something for the next.
- Handle credentials carefully. Use a GitHub App or short-lived tokens for registration, prefer JIT configs, and rotate anything long-lived. Use OIDC federation, where the job proves its identity to the cloud provider and receives a short-lived credential, instead of storing cloud keys.
- Fence the network. Outbound port 443 to the platform is all the runner needs. Put runners in their own VLAN or subnet, and allow only what jobs need: registries, package mirrors and deploy targets.
- Protect the supply chain. Pin third-party actions to commit SHAs, the exact hash of a commit, which can't be moved the way a tag can. Use your mirrors, and scan the images you build.
- Clear secrets from disk. Clean _work, temp folders and keychains after each job, and never cache directories that may contain secrets.

How different industries use self-hosted runners

The patterns below are common across companies of each type, and the details vary from one company to the next.

SECTOR	TYPICAL SETUP	MAIN REASON	SAVINGS LEVER
Startups and small teams	Developer PCs or one mini PC plus hosted fallback; a Mac mini for iOS	Stay inside free minutes, speed up builds	Compute minutes, build time
Mid-size SaaS	Kubernetes (ARC or GitLab k8s executor) on spot nodes; NAS or S3 caches	Parallel test suites, cost control at scale	Minutes, caching, autoscaling to zero
Large enterprise	Runner groups per business unit, golden images, IaC, VM scale sets, internal registries	Governance, network access to internal systems, audit	Parallelism fees, egress, plan tiers
Finance, healthcare, government, defense	Air-gapped or private-network runners, GitHub Enterprise Server, GitLab self-managed or Azure DevOps Server; hardened VMs	Compliance and data residency; code and secrets never leave the network	Mostly compliance, then cost
Mobile app companies	Mac mini farms (bare metal or Tart/Anka VMs), Android emulators on KVM Linux	macOS minutes are the most expensive; simulators need real hardware	macOS compute
Game studios	Windows workstations and servers with GPUs, huge disks, Perforce or Git LFS caches; console SDK machines on isolated networks	Builds take hours and need licensed SDKs and GPUs; jobs exceed hosted limits	Large runners, build time, storage

SECTOR	TYPICAL SETUP	MAIN REASON	SAVINGS LEVER
AI and ML	GPU runners (workstations, on-prem GPU nodes, spot GPU instances); model and dataset caches on NAS	GPU minutes are expensive; data is large	GPU compute, storage, egress
Embedded, IoT, automotive	Runners attached to physical boards (hardware-in-the-loop), USB debuggers, test rigs; arm64 builders	Tests need real hardware on the bench	Enables testing that cloud cannot do at all
Semiconductors and EDA	On-prem compute farms with licensed tools and license servers	Tool licences are tied to on-prem hosts	Licences and compute
Open-source projects	Mostly hosted (free); sponsored or donated hardware for arm64, GPU or exotic platforms, gated to maintainers	Platform coverage	Special hardware
Education and homelabs	Forgejo/Gitea plus act_runner, k3s, old PCs, Oracle Always Free	Learning, zero budget	Everything

Table 23. How eleven sectors set up their runners, why, and which cost each setup mainly cuts.

EDA in the semiconductor row stands for electronic design automation, the software used to design chips, and its licence servers are tied to machines on site.

Notes by pattern

Hardware in the loop. A runner sits on a small Linux box, often a Raspberry Pi or an Intel NUC. It connects to target boards by USB, serial and GPIO, the general-purpose pins on a board. Jobs flash firmware, run tests and collect logs. Label runners by the board

attached to them (board-nrf52, ecu-rig-3), and use concurrency groups so two jobs never share a board.

Air-gapped. The control plane (GitHub Enterprise Server, GitLab self-managed, Azure DevOps Server, Jenkins or Forgejo) runs inside the network. Runners, registries and package mirrors are all internal, and actions and images are synced in through a controlled transfer process.

Big monorepos. A monorepo keeps many projects in one repository. There, remote build caching and remote execution (Bazel with BuildBuddy, EngFlow or Buildbarn; Gradle Enterprise-style caches) matter more than the runner itself. Self-hosted runners placed next to the cache give the biggest gains.

Regulated deploys. Separate deploy runners sit inside production networks, reachable only from protected branches and environments with required reviewers, while build and test jobs run elsewhere.

Reference architectures

Solo developer or freelancer

- One Windows PC runs a native Windows runner plus a WSL2 runner.
- A hosted fallback switch covers both, as the pipelines section describes.
- Public repositories stay on hosted runners.
- The cost is electricity, and the upkeep takes minutes a month.

Small team, 2 to 15 developers

- Workstation runners as above, plus a NAS or mini PC running three ephemeral Docker runners for always-on Linux work.
- The NAS also runs a registry pull-through cache, an npm and PyPI proxy, and MinIO for caches and artifacts.
- One Mac mini handles iOS and macOS jobs, with two Tart VMs for clean parallel runs.

- A hosted fallback covers everything except the Mac jobs.
- Monitoring is a scheduled script or a small dashboard showing queue time, runner status and hosted spend.

Mid-size, 15 to 200 developers

- Kubernetes, managed or on-premises, runs ARC or the GitLab Kubernetes executor on spot node pools, autoscaling to near zero at night.
- A VM pool for Windows (Hyper-V, Proxmox or cloud scale sets) runs golden images rebuilt monthly.
- A rack of Mac minis or a cloud Mac provider handles iOS.
- Central caches (registry, package proxies, build caches) sit in the same network as the runners.
- Terraform, Helm and Packer live in a runner-infrastructure repository, with dashboards and alerts on queue time and disk.

Enterprise

- Runner groups are split per business unit and environment, with separate signing and deploy runners.
- Multi-region pools sit near the source code and the deploy targets.
- JIT runners come from an internal orchestrator or from managed pools (ARC, Managed DevOps Pools, VM scale sets).
- A central platform team owns images, security baselines, cost reporting and chargeback.
- Hosted runners are kept for bursts, public projects and platforms not worth owning.

Cost modelling: worked scenarios

The numbers below are illustrative, priced at GitHub's January 2026 list rates and ignoring the minutes included in each plan. Use your own billing export for real

decisions.

SCENARIO	MONTHLY HOSTED USAGE	HOSTED COST / MONTH	SELF-HOSTED SETUP	ONE-OFF HARDWARE	RUNNING COST / MONTH	PAYBACK
Solo developer	3,000 Linux + 1,500 Windows min	$33	Existing PC: Windows service + WSL2	$0	~$3 power	Immediate
Small team	40,000 Linux + 10,000 Windows + 3,000 macOS min	$526	Mini PC + NAS containers + Mac mini	~$1,000-1,500 (NAS already owned)	~$15 power + a few hours upkeep	About 3 months
Mid-size	400,000 Linux + 60,000 Windows + 30,000 macOS min	$4,860	Kubernetes on spot + Windows VM pool + Mac minis	Varies	Cloud spot + ops time; plan for 30-60% of hosted	6-12 months incl. engineering time

Table 24. Three worked cost scenarios at GitHub's January 2026 list rates, from immediate payback to a year.

Each row is computed as Linux minutes × $0.006 + Windows minutes × $0.010 + macOS minutes × $0.062. The solo developer's bill is $18 + $15 = $33. The small team's is $240 + $100 + $186 = $526, and its macOS line is 35 percent of that bill from under 6 percent of the minutes. The mid-size bill is $2,400 + $600 + $1,860 = $4,860, or about $58,000 a year.

The solo row is the one that pays from day one, since its hardware already exists. The small team's $526 a month repays up to $1,500 of new hardware in about three months once power and upkeep are counted. The mid-size case takes longest because it buys engineering time as well as machines. Engineer time and build speed still move every row more than the rates do, as the arithmetic section set out, and an honest model carries both.

Pros and cons, side by side

The per-host tables above weigh each option on its own. This one sets the three broad routes against each other.

QUESTION	GITHUB-HOSTED RUNNERS	SELF-HOSTED RUNNERS	THIRD-PARTY HOSTED RUNNERS
Per-minute cost	$0.006 Linux, $0.010 Windows, $0.062 macOS on private repos; free on public repos	$0 per minute; you pay hardware, power and upkeep	Per minute, advertised below GitHub's rates
Upfront cost	None	Zero if you own the hardware; about $600 for a Mac mini	None
Operations	None	Patching, updates, disk, security, monitoring	None
Build speed	Clean 2-core machines; cold caches every job	Warm caches and many cores on persistent runners	Advertised as faster hardware
Job length limit	6 hours	Up to 5 days	Varies by vendor
Security exposure	Vendor's isolated machine per job	Your machine runs the workflow's code; needs fork gating and isolation	Vendor's machines hold your code and secrets
Availability	Always there	Only while your machine is on, so a hosted fallback is needed	Always there
Special hardware	GPUs, big machines and Macs billed even on public repos	Use the GPU, cores and Macs you already own	Varies by vendor
Network reach	Public internet only	Reaches LAN, VPN-only and internal systems	Varies; some run in your own cloud account

QUESTION	GITHUB-HOSTED RUNNERS	SELF-HOSTED RUNNERS	THIRD-PARTY HOSTED RUNNERS
Best for	Public repos, low volume, bursts	Windows, macOS, GPU and high-volume private work	Teams that want lower minutes without operations

Table 25. GitHub-hosted, self-hosted and third-party hosted runners compared on the ten questions that decide the choice.

Decision guide and checklist



Figure 8. Walk down the questions on the left and the first yes decides where a job runs, with every path keeping a hosted fallback.

Work through the list once before the pilot and again at each monthly review:

- Export billing by repository and SKU, and rank jobs by minutes × rate.

- Keep public repositories on hosted runners.
- Move macOS and Windows jobs first, to hardware you own.
- Move Linux jobs to WSL2, Docker or a NAS.
- Use the fork-gated switch with a hosted fallback on every self-hosted job.
- Automate the switch from runner status.
- Run services under low-privilege accounts, and prefer ephemeral runners inside containers or VMs.
- Put registry, package and build caches next to the runners.
- Set timeout-minutes and concurrency on every job.
- Monitor queue time, runner status, disk and hosted spend.
- Rebuild images monthly and on runner releases.
- Add Kubernetes, spot capacity or cloud Macs only when demand justifies them.

When to do it: the rollout plan

The move works best in stages, each one paid for by the savings of the one before it.

1. Measure, in week 1. Export billing usage by repository and SKU. List the top ten jobs by minutes × rate, and note queue times and build durations.
2. Pilot, in week 2. Put one runner on an existing machine for the single most expensive job family, which is usually Windows or macOS. Add the fork-gated fallback, then compare duration and failure rate for two weeks.
3. Expand, in weeks 3 to 6. Add always-on capacity (a NAS or mini PC), caches and package proxies. Move the long tail of Linux jobs, and add labels and runner groups.
4. Automate, in weeks 6 to 10. Bring in images (Packer or a Dockerfile), IaC, ephemeral runners, the runner-status switch, monitoring and disk alerts.
5. Optimize, from then on. Review queue time and cost monthly. Add Kubernetes or spot autoscaling when parallel demand justifies it, and retire idle hardware.

Troubleshooting

SYMPTOM	LIKELY CAUSE	FIX
Job waits "for a runner to pick up this job"	No online runner has every requested label	Check labels; add the hosted fallback; check the service is running
Runner shows offline after reboot	Service not set to start, WSL not started, PC asleep	svc.sh status; start WSL at logon; power settings
"No space left on device"	Docker images, _work, caches	Prune in a job-completed hook; alert at 20% free; bigger SSD
Builds slower than hosted	Work dir on HDD or /mnt/c, Defender scanning, cold ephemeral caches	Move to SSD / Linux FS; Defender exclusion; LAN caches
429 Too Many Requests from registries	Anonymous pull limits	Pull-through cache; authenticate pulls
Jobs fail only on one runner	State drift	Rebuild from image; go ephemeral
Runner stops getting jobs after months	Runner version too old	Enable auto-update or rebuild images each release
Spot runner vanishes mid-job	Spot reclaim	Retries, shorter jobs, multiple instance types and zones
macOS job hangs on a dialog	Keychain or permission prompt without a GUI session	Auto-login, unlock a dedicated keychain in the job, pre-grant permissions
Windows path errors	Path length limit	core.longpaths, LongPathsEnabled, short work dir

Table 26. The ten failures self-hosted runners hit most, with the usual cause and fix for each.

Worked example: one Windows PC with automatic fallback

The AlienInterface repository is the home of the M.U/T.H/E.R desktop app. It runs its CI on the owner's Windows PC when the PC is on, and on GitHub-hosted runners when it is off.

RUNNER	HOST	LABELS	TAKES	HOSTED RATE IT REPLACES
WSL	Ubuntu in WSL2	self-hosted, linux, mother	ci.yml: typecheck, tests, build, headless Electron smoke tests	$0.006/min
Windows	Windows service	self-hosted, windows, mother	release.yml: NSIS installer, GitHub release	$0.010/min

Table 27. The two runners on the AlienInterface PC and the hosted rate each one replaces.

Both workflows use the fork-gated expression from the pipelines section, with vars.RUNNER falling back to ubuntu-latest for Linux jobs and vars.WINDOWS_RUNNER falling back to windows-latest for the release job. The desktop app acts as the switch. Every five minutes it reads the repository's runners, sets the variable while its runner is online, and deletes it when the runner goes offline. Its RUNNERS panel shows each runner's status, its current job and the queue. It also sets this month's hosted spend against the minutes run locally and the dollars those minutes saved at list price. It refuses to mark a repository as wired if any workflow uses an ungated vars.RUNNER, because that would let a fork's pull request run code on the PC.

Running it taught three lessons.

- Self-hosted machines lack tools that hosted images include, such as jq, xvfb and passwordless sudo. Steps now check for a tool and install it into $RUNNER_TEMP when it's missing.
- A job can start on the PC seconds before the PC sleeps. Every job carries timeout-minutes, so a lost runner fails fast instead of holding the job for days.
- "Could not look" is different from "zero." When the GitHub API call fails, the panel says UNKNOWN rather than showing no runners or $0. A dashboard that reported a failed check as a quiet, cheap month would hide exactly the sleeping PC this whole setup exists to catch — and it would look like good news.

Glossary

TERM	DEFINITION
Actions Runner Controller (ARC)	GitHub's Kubernetes controller that creates and removes runner pods to match the job queue. It is the usual way to autoscale GitHub runners on a cluster.
Agent	The program that takes jobs from the CI service and runs them, called a runner on GitHub and GitLab and a node on Jenkins. It is the same program on hosted and self-hosted machines.
Air-gapped	A network with no connection to the public internet. CI there runs its control plane, runners and mirrors inside the network and brings actions and images in through a controlled transfer.
Autoscaler	A process that watches the job queue and creates or deletes runners to match it. It is what lets an ephemeral fleet cost nothing while idle.
BuildKit	Docker's build engine, which can store its layer cache in a registry. A cache on the local network makes image builds incremental across runners that start empty.
Cloud-init	The standard tool that configures a Linux machine on its first boot from data passed in at launch. Runner images use it to receive a JIT config or registration token.
Concurrency group	A workflow setting that lets only one run with the same group name proceed and can cancel the older one. It stops superseded pushes from wasting runner time.
Container	A packaged program that runs isolated from the host while sharing the host's kernel. Containers give each job clean state at little cost.
Continuous integration (CI)	The practice, and the service, that builds and tests every change pushed to a repository. Its machine time is what hosted vendors bill by the minute.
Control plane	The CI service that stores pipelines, queues jobs, holds secrets and collects logs. Runners connect out to it and never the other way

TERM	DEFINITION
	round.
DinD (Docker-in-Docker)	A Docker daemon running inside a container so a job can build images. The privileged form lets a hostile job escape to the host, so safer methods come first.
Egress	Data leaving a cloud provider's network, which the provider bills. Runners placed in the same region as their targets avoid it.
Ephemeral runner	A runner that takes one job, then unregisters and exits. It trades warm caches for a clean, safe start every time.
Executor	GitLab Runner's method of running a job, such as shell, docker or kubernetes. The executor decides how isolated each job is.
Fork gate	A workflow condition that keeps pull requests from forks off self-hosted runners. Without it, a stranger's code can run on your machine.
Golden image	A prebuilt VM or container image with the OS, the tools and the runner installed. Rebuilding it monthly is how a fleet stays patched without hand work.
Hardware in the loop	Tests that run against physical devices attached to the runner, such as boards on a test bench. It is CI that no cloud runner can do.
Hypervisor	The software that runs virtual machines, such as Hyper-V, Proxmox, VMware ESXi or KVM. On-premises VM runners depend on one.
Infrastructure as code (IaC)	Describing machines, networks and clusters in files that a tool applies, such as Terraform or Ansible. It lets the whole fleet be rebuilt from a repository.
JIT runner	A runner started with a single-use configuration minted by the API for one job. No long-lived registration token exists, which makes it the most secure mode.
k3s	A lightweight Kubernetes distribution that runs on a mini PC or a NAS VM. It brings ARC and autoscaling to a homelab.

TERM	DEFINITION
Kubernetes	A system that schedules containers across a cluster of machines and restarts them when they fail. It suits fleets with many parallel jobs.
KVM	The hypervisor built into the Linux kernel. Android emulator tests need it, or a cloud VM with nested virtualization.
Label	A string a runner advertises and a job requests, such as windows or gpu, called a tag on GitLab. Labels route each job to hardware that can run it.
Long-polling	Holding a request open until the server has something to send. It is how a runner waits for jobs without any inbound connection.
M.U/T.H/E.R	Model Unification / Tool Harness / Execution Runtime, the desktop app whose repository, AlienInterface, is this article's worked example. It runs that repository's CI on the owner's PC when the PC is on.
MicroVM	A minimal virtual machine, such as Firecracker, that boots in well under a second. It gives VM isolation at near-container density for less-trusted code.
NAS (network-attached storage)	A storage box on the network, often able to run containers. It makes a good always-on runner host and cache server.
NAT (network address translation)	The address sharing a home or office router does, which blocks unsolicited inbound connections. Runners work behind it because they only dial out.
OIDC federation	A job proving its identity to a cloud provider through OpenID Connect and receiving a short-lived credential in return. It removes stored cloud keys from CI.
Packer	HashiCorp's tool for building machine images from a template. It produces the golden images that VM and cloud runners boot from.
PAT (personal access token)	A token tied to a person's account that grants API access. A GitHub App is safer for production runner fleets.

TERM	DEFINITION
Persistent runner	A runner that stays registered and takes job after job on the same disk. It is the fastest mode and the one most exposed to drift and tampering.
Pull-through cache	A local registry or proxy that fetches each item from upstream once and serves it locally afterwards. It removes rate limits and saves bandwidth.
QEMU	An emulator that runs code built for one processor on another. It makes cross-architecture builds possible and many times slower than native ones.
Runner group	A set of runners with rules about which repositories may use them, called a pool on Azure DevOps. It keeps experimental repositories off sensitive machines.
Scale set	ARC's autoscaling group of runners on Kubernetes, addressed in a workflow by its installation name. It scales from zero to a set maximum.
SKU	A billing line item for one product or runner type. Exporting usage by SKU shows which runner types cost the most.
Spot instance	Discounted spare cloud capacity, also called preemptible, that the provider can reclaim at short notice. It suits ephemeral jobs that tolerate retries.
Virtual machine (VM)	A whole computer simulated in software on a host. VMs give stronger isolation than containers and clean Windows state per job.
VLAN	A separately fenced segment of a network. Putting runners on their own VLAN limits what a compromised job can reach.
WSL2 (Windows Subsystem for Linux)	A real Linux kernel running in a lightweight VM inside Windows. It puts Linux CI on a Windows PC with no extra hardware.

Sources

- About billing for GitHub Actions
  - docs.github.com · github actions
- GitHub Actions runner pricing
  - docs.github.com · actions runner pricing
- IT Brief: GitHub cuts Actions runner prices, adds new usage fee
  - itbrief.news · github cuts actions runner prices adds new usage fee
- Adding self-hosted runners (GitHub)
  - docs.github.com · adding self hosted runners
- Security hardening for self-hosted runners (GitHub)
  - docs.github.com · security hardening for github actions
- About Actions Runner Controller (GitHub)
  - docs.github.com · about actions runner controller
- terraform-aws-github-runner
  - github.com · terraform aws github runner
- GitLab compute minutes
  - docs.gitlab.com · compute minutes
- Azure DevOps parallel jobs
  - learn.microsoft.com · concurrent jobs
- Oracle Cloud Always Free resources
  - docs.oracle.com · freetier topic Always Free Resources
- Amazon EC2 Mac instances FAQs
  - aws.amazon.com · faqs
- AlienInterface repository: .github/workflows/ci.yml, .github/workflows/release.yml, src/main/runners.ts, src/main/runner-cost.ts, and the README section "Self-hosted runners".

Appendix J, tables

Every body table, Tables 1 through 27, is already the complete data set it presents, so none takes an appendix of its own. Table 24's computed hosted costs are shown with their arithmetic in the paragraph that follows it.

CLOUD / DEVOPS WPR-CLD-003

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn → ORCID →