Ship While You Sleep

How to run coding agents overnight on Windows —
and trust what you find in the morning

White Bay · white-bay.com

Contents

1. The machine you need

Chapter 1 — The machine you need

Sample chapter from "Ship While You Sleep: running coding agents unattended." This is the full first chapter, not an excerpt. If the rest of the book reads like this, you'll know before you buy.

Most guides to running AI coding agents start with the exciting part: the agent working through the night, the pull request waiting for you in the morning. This one starts with a linker error, because that is what actually happens on day one.

The setup in this book runs on Windows. Everything here was executed on:

Component	Version
Host OS	Windows 10, build 19045
Rust toolchain	rustc 1.97.1
Node	24.19.0
Claude Code	2.1.229
Agent runtime	Orca 1.4.187

Write those numbers down somewhere. When something in this book stops working a year from now, the difference between your versions and these is the first place to look. I'll say more about why that matters at the end of the chapter.

What you're building toward

By the end of the book you'll have a machine that does this: you hand it a goal at night, it plans the work into tasks, it executes them inside an isolated copy of your repository, it stops and asks permission before doing anything that spends money or touches the outside world, and in the morning you review commits and test output rather than a status board that may or may not be telling you the truth.

The order matters. Chapter 2 is a single run you watch from start to finish. Chapter 3 is isolation. Chapter 4 is the approval gate. Only in Chapter 5 do you leave it alone overnight. If you skip to Chapter 5, you will eventually come downstairs to a main branch you don't recognize.

This chapter is the boring one. It's also the one where most people quit, because the failures on day one look like the tool is broken when in fact three dependencies are missing.

The four installs

You need four things: a Rust toolchain, a C++ linker, Node, and the agent runtime. Two of those will surprise you.

1. Rust

```
winget install Rustlang.Rustup
rustup --version
rustc --version
```

You want output like `rustc 1.97.1`. If `rustc` isn't found after installing, close the terminal and open a new one — the installer edits `PATH`, and an already-open shell doesn't see it. This is the single most common "it didn't install" that turns out to be a stale environment.

2. The C++ build tools (the first surprise)

Here is the failure that stops most people on their first build:

```
error: linker `link.exe` not found
note: the msvc targets depend on the msvc linker but `link.exe` was not found
```

Rust on Windows does not ship its own linker. It uses Microsoft's, and Microsoft's linker arrives with Visual Studio Build Tools — not with Rust, not with Node, not with anything else you have installed. `rustup` installs cleanly and tells you nothing about this until the first time you compile something that links.

The fix:

```
winget install Microsoft.VisualStudio.2022.BuildTools
```

Then launch the Visual Studio Installer, choose Modify, and enable the **Desktop development with C++** workload. Installing the Build Tools package alone is not sufficient — the base install can land without the MSVC toolset, and you get the identical error afterward. That double step is why this failure eats an evening: you install the thing the error message pointed at, try again, and get the same error.

Verify before moving on:

```
rustc --version
cargo new linkcheck; cd linkcheck; cargo build
```

If `cargo build` produces a binary, your linker is in place. If it repeats the `link.exe` error, reopen the installer and confirm the C++ workload is actually checked.

There is a second path here worth knowing: you can switch Rust to the GNU toolchain and avoid MSVC entirely. Don't, at least not for this book. Some crates in the desktop stack behave differently under GNU, and debugging a toolchain mismatch is a worse evening than a fifteen-minute installer.

3. Node

```
winget install OpenJS.NodeJS.LTS
node --version
npm --version
```

Node 24.19.0 is what everything here was tested against. Anything on the current LTS line should be fine. If you already have Node from an old installer plus a version manager plus something a tutorial told you to install two years ago, resolve that now rather than at 2 a.m. — `where.exe node` will tell you how many you have.

4. WebView2

If you're on Windows 10 or 11 you almost certainly already have this; it ships with the OS and with Edge. It's on the list because when it's missing, the failure is silent and confusing: the desktop shell launches, the window appears, and the contents are blank. Nothing errors. You sit there wondering whether the build succeeded.

Check quickly:

```
Get-ChildItem "HKLM:\SOFTWARE\WOW6432Node\Microsoft\EdgeUpdate\Clients" -ErrorAction
SilentlyContinue |
  ForEach-Object { (Get-ItemProperty $_.PSPath).name }
```

If you see "Microsoft Edge WebView2 Runtime" in the output, you're set. If not, install the Evergreen runtime from Microsoft before you build anything with a window.

5. The agent runtime

Run the installer, then confirm it's actually there:

```
orca status --json
```

Two notes from doing this myself. First, `orca --version` does not necessarily print a version — on the build I installed it printed help text instead, which is not the same as the command being missing. `orca status --json` is the reliable check. Second, **do not assume the documentation you're reading matches the build you installed.** This runtime ships frequently, sometimes multiple times a day. Take the JSON output from `orca status` and save it in your project notes as a capability snapshot. When a later chapter tells you to use a feature and your runtime doesn't have it, that snapshot is how you find out in ten seconds instead of an hour.

The second surprise: things that hang instead of failing

The `link.exe` error is loud. The failures in this section are quiet, and quiet failures are the ones that cost you a night.

A terminal that stops responding. When an agent runs inside a terminal you've built or embedded rather than the one Windows gives you, one specific interaction bites: the program inside asks the terminal where the cursor is, and waits for an answer. On Windows this arrives through ConPTY, and if whatever is hosting the session doesn't answer, the program simply stops. No error, no exit, no output. It looks exactly like an agent thinking hard about a difficult problem.

You will meet this in Chapter 2 the first time you run something with a progress bar or an interactive prompt. The lesson to carry forward: **an agent that has produced no output for several minutes is not necessarily working.** Chapter 6 gives you the checks that distinguish thinking from hanging, and they are not "look at the status indicator."

A process that won't die. The second one: you tell a session to stop, the stop call reports failure with an error code that translates to "the operation completed successfully." Error zero. What it usually means is that the process already exited and something tried to kill it again.

This matters more than it sounds. If your stop path errors, your cleanup doesn't run, and stale sessions accumulate all night. By morning you have a dozen orphans holding file locks in worktrees you're trying to delete. The fix is to treat "already gone" as success rather than as an error — but you need to know the symptom to recognize it, because "os error 0" reads like a bug in your own code.

I'm flagging both here, in the setup chapter, because they present as *the agent is broken* when they are really *the plumbing under the agent is incomplete*. Knowing they exist changes what you look at.

Git, and one setting

You need git, and you need it configured with an identity, because the agent is going to make commits:

```
winget install Git.Git
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

One setting worth changing now:

```
git config --global core.autocrlf input
```

Line-ending translation is a small thing until an agent rewrites a file, git reports every line as changed, and your morning review is a diff of the entire repository with the actual change buried somewhere inside it. Review quality is the whole point of this method, and this setting protects it.

We're not touching worktrees yet — that's Chapter 3, and it deserves its own treatment. But this is the moment to confirm your repository is a real git repository with a clean status, because everything from here on assumes that.

Your setup is done when

Don't move to Chapter 2 until all five of these pass:

- [ ] `rustc --version` prints a version
- [ ] `cargo build` succeeds on a fresh `cargo new` project — this is your real linker test
- [ ] `node --version` prints a version
- [ ] `orca status --json` returns JSON, and you've saved that output as your capability snapshot
- [ ] `git status` runs cleanly in the repository you plan to use

Five checks, and each one has a verifiable output. That's the pattern for the rest of the book: every chapter ends in a state you can confirm before continuing. Unattended execution is only safe when each layer underneath it is known-good, and "I think I installed it" is not known-good.

Why the versions are pinned

Every guide in this category rots, and it rots for the same reason: the author writes "install the agent runtime," ships it, and eight weeks later a release changes a flag name. The reader follows the instructions, hits an error the guide doesn't mention, and concludes the whole approach doesn't work.

The table at the top of this chapter is my defense against that. Every command in this book was run against those exact versions. When your versions differ and something behaves unexpectedly, you have a place to start instead of a guess. Buyers get revisions as the pinned versions move — that's part of what you're paying for, and it's the part a free blog post structurally cannot offer, because nobody maintains a free blog post for two years.

Chapter 2 is a single run, start to finish, watched. Small goal, nothing clever, no autonomy. You need to see what normal looks like before you can recognize abnormal at 3 a.m.