LatticeMind Documentation

Release 0.3.1

Nanoacademic Technologies Inc.

Aug 25, 2026

CONTENTS

1 Introduction 2

2 Installation 4

3 Getting started 15

4 Basic tutorials 32

5 Advanced tutorials 44

6 Reference 50

7 Contact Us 69

LatticeMind is the AI agent for materials science from Nanoacademic. Describe the calculation you want in plain language, and LatticeMind plans the workflow, builds the crystal or device structure, generates and validates the solver inputs, runs the calculation locally or on your HPC cluster, and writes you a clear report — with the RESCU density functional theory solver at its core.

This manual walks you from installation to your first band structure, through every setting and interface, and on to advanced remote-execution and quantum transport workflows.

> **Note**
>
> LatticeMind's support for **RESCU** is mature and tested across the documented workflows. Support for **NanoDCAL** (quantum transport, effective mass, and more) is **experimental and in active development**, and support for **VASP** is in **preview (beta)**. This manual leads with RESCU and calls out solver-specific behaviour throughout.

CHAPTER

ONE

INTRODUCTION

LatticeMind is an AI agent for computational materials science. You describe the calculation you want in plain language — *"compute the band structure and density of states of silicon"* — and LatticeMind carries out the entire workflow for you: it plans the sequence of calculations, builds the crystal structure, chooses sensible numerical parameters, writes the solver input files, checks them for physical and syntactic correctness, runs the calculation (on your laptop or on a remote HPC cluster), analyses the raw output, and produces a readable report with figures.

At its core LatticeMind drives the RESCU density functional theory (DFT) solver, and it speaks RESCU fluently: self-consistent field, band structure, density of states, structural relaxation, DFT+U, density functional perturbation theory (DFPT) phonons, and more. It also has an experimental backend for the NanoDCAL quantum-transport solver and a backend for VASP.

What makes LatticeMind different from a template generator is that it *reasons about the science*. It maintains an explicit, validated plan of the work; it wires multi-step workflows together correctly (feeding the converged density of an SCF run into a subsequent band-structure run, for example); and it checks its own work at every stage, repairing mistakes before a single core-hour is spent.

1.1 How LatticeMind works

A LatticeMind project moves through a fixed sequence of stages: a planner turns your request into an explicit plan, a structure builder constructs the geometry, a parameterizer chooses the numerical settings, an input builder renders the solver decks, a validator gates execution, an execution backend runs the job (locally or on HPC), and a reporter writes up the results. Each stage inspects the output of the previous one and either proceeds or asks for a repair, so errors are caught early rather than surfacing as a failed calculation hours later.

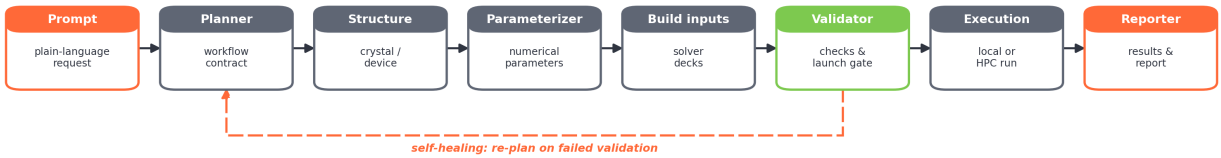

Fig. 1.1.1: The LatticeMind pipeline. A natural-language prompt is turned into a validated, executable project and finally a report. Validation can send the plan back for repair (self-healing) before anything is executed.

You stay in control throughout: solver execution is off until you enter `/execute on`. Use `/cost` to inspect the estimate and `/launch-preview` to inspect the exact local or remote launch before a job starts. For a stage-by-stage walkthrough — including the self-healing loop and the human-in-the-loop gates — see *How LatticeMind works*.

1.2 Supported solvers

LatticeMind binds one project to one solver. The three backends and their current maturity are:

Solver	Status	What it covers
RESCU	**Primary, stable**	SCF, band structure, density of states, projected/local DOS, structural relaxation, DFT+U, DFPT phonons and dielectric response, spin-polarized and non-collinear magnetism, alloys and defects, surface slabs and work functions.
NanoDCAL	**Experimental**	Quantum transport for two-probe devices (transmission, conductance, I–V, scattering states), effective mass, complex band structure, phonons, and ground-state properties. See *NanoDCAL support (experimental)*.
VASP	**Preview (beta)**	Multi-stage plane-wave workflows (SCF, relaxation, band structure, DOS).

> **Note**
>
> NanoDCAL support is under active development. Its supported calculation types have each been run end-to-end during internal testing, but the surface is smaller and evolving faster than the RESCU backend. This manual documents RESCU first and treats NanoDCAL as an advanced, experimental topic.

1.3 Key features

- **Plain-language interface.** No input-file syntax to memorize. Prompts can be as terse as *"SCF for fcc nickel"* or as detailed as you like.
- **Self-healing workflows.** Every plan is validated before execution and automatically re-planned when a check fails. Failed attempts are archived, not discarded, so you can see what changed.
- **Correct multi-step chaining.** Densities and wavefunctions are handed from one calculation to the next automatically, with unique result paths per step.
- **Three interfaces.** An interactive terminal **Dashboard** implemented with the Textual terminal-interface framework and Rich terminal-rendering Python library, a browser-based **Web UI**, and a plain **CLI** for servers and notebooks — all driving the same engine. See *The three interfaces*.
- **Bring your own AI model.** Works with OpenAI, Anthropic Claude, Google Gemini, Qwen, or a local Ollama model. See *Choosing a provider*.
- **Local or HPC execution.** Run on your machine, or offload to a remote workstation or Slurm cluster with a one-line profile. LatticeMind never sends your API key to the remote host. See *Remote & HPC execution*.
- **Multilingual.** Prompts in English, French, or Chinese are understood and reports can be produced in the language you asked in.
- **Self-describing projects.** Every project is a self-contained folder of inputs, results, plots, status files, an event log, and a report — portable and reproducible.

1.4 Who this manual is for

This manual is written for **users** — materials scientists, chemists, and engineers who want results, not build systems. You do not need to know RESCU's input syntax or how to compile anything. If you can describe a calculation, you can run it.

Ready? Continue to *Installation* to get LatticeMind installed, then to *Getting started* for your first project.

CHAPTER

TWO

INSTALLATION

LatticeMind is distributed as a platform-specific Python wheel. Installing version 0.3.1 takes three things, plus the solver files needed for the calculations you choose to execute:

1. **Python 3.10 through 3.14** and the wheel whose `cp310`, `cp311`, `cp312`, `cp313`, or `cp314` tag matches that interpreter. The release provides Linux x86-64, Windows AMD64, macOS Intel, and macOS Apple Silicon wheels.
2. **A valid RESCU license file for LatticeMind startup.** Put one `*.lic` or `*.license` file in `~/.nanoacademic/RESCU` or set `LATTICEMIND_LICENSE_FILE` to its exact path. LatticeMind checks this file before either the CLI or Web UI starts, including when NanoDCAL is selected.
3. **An AI provider API key** - OpenAI by default; Claude, Gemini, Qwen, or a local Ollama model also work.

The RESCU license configured for LatticeMind is also used for RESCU execution. To run RESCU calculations, configure the RESCU executable. To run NanoDCAL calculations, configure the NanoDCAL executable, a separate NanoDCAL license, and the NeutralAtomDatabase downloaded from the Nanoacademic Portal.

> **Tip**
>
> In a hurry? The *Quick start* gets you from the matching wheel and license file to your first validated project.

The pages below walk through installation in full: a fast path, the recommended environment setup for each operating system, connecting your AI model, setting up RESCU or NanoDCAL, cloud/notebook deployment, and how to update.

2.1 Quick start

This is the shortest path from a fresh machine to your first result.

1. **Install the matching wheel** you received from Nanoacademic. The `cp310` through `cp314` tag must match your interpreter, Python 3.10 through 3.14:

```
python -m pip install /path/to/lattice_mind-0.3.1-<matching-tags>.whl
```

2. **Install the RESCU license used by LatticeMind.** Put exactly one `*.lic` or `*.license` file in `~/.nanoacademic/RESCU`. On Windows that expands to `C:\Users\<username>\.nanoacademic\RESCU`. For any other location, set:

```
LATTICEMIND_LICENSE_FILE=/full/path/to/rescu.lic
```

3. **Add your AI provider API key.** OpenAI is the default:

```
export OPENAI_API_KEY="your_key_here"     # macOS / Linux
```

```
$env:OPENAI_API_KEY="your_key_here"       # Windows (PowerShell)
```

4. **Start LatticeMind:**

```
latticemind
```

Then just type what you want. For example:

```
Do a simple SCF and band structure calculation for silicon.
```

LatticeMind will plan the workflow, build the silicon structure, generate and validate the RESCU inputs, and show you the launch preview. Solver execution is controlled by the explicit `/execute on` and `/execute off` commands, not by wording in the calculation prompt. Execution is off by default.

> **Tip**
>
> Type `/examples` inside the app for ready-to-run prompts, or `/commands` to browse everything LatticeMind can do.

Prefer a browser? Launch the web interface instead:

```
latticemind-web
```

It opens at `http://127.0.0.1:7865` by default.

> **Note**
>
> Configure the RESCU license from step 2 before starting LatticeMind. The same license supports RESCU execution. To run a project, also configure the selected solver executable. NanoDCAL execution uses a separate NanoDCAL license and the NeutralAtomDatabase. See *Licenses and solver executables*.

2.1.1 Where do results go?

By default LatticeMind writes projects and outputs to a `latticemind_outputs/` folder in your current working directory. Each project is a self-contained folder with the generated inputs, results, plots, a report, and a full event log. See *Projects & the workspace* for the layout.

To store outputs elsewhere (recommended on shared machines and cloud), set:

```
export LATTICEMIND_ARTIFACTS_DIR="/path/to/outputs"
```

2.1.2 What next?

- New to LatticeMind? Continue to *Getting started* and build your first project step by step.
- Setting up on a cluster or notebook platform? See *Cloud & headless deployment*.
- Want the full configuration surface? See *Reference*.

2.2 Detailed installation

This page covers a clean, isolated installation on each operating system. We strongly recommend installing into a dedicated environment so LatticeMind's dependencies never collide with other Python projects.

2.2.1 Requirements

- **Python 3.10 through 3.14.** The 0.3.1 release provides `cp310`, `cp311`, `cp312`, `cp313`, and `cp314` wheels. Use the wheel matching both the Python tag and the operating-system architecture.
- **Internet access** for the AI model (the calculations themselves can run offline) and normally for `pip` to resolve Python dependencies.
- **A valid RESCU license file.** LatticeMind checks it before the CLI or Web UI starts, even when NanoDCAL is selected.
- **An AI provider API key** (see *Connecting your AI model*).
- *(To run calculations)* the selected solver and its licensed data. NanoDCAL requires its own solver license and NeutralAtomDatabase in addition to the RESCU license used by LatticeMind (see *Licenses and solver executables*).

2.2.2 Option A — Conda / Miniconda (recommended)

Conda gives you a clean, reproducible Python environment on any platform.

```
conda create -n latticemind python=3.10
conda activate latticemind
python -m pip install path/to/lattice_mind-0.3.1-<matching-tags>.whl
```

2.2.3 Option B — Python virtual environment

If you do not use Conda, a standard virtual environment works just as well.

Windows (PowerShell):

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
python -m pip install path\to\lattice_mind-0.3.1-<matching-tags>.whl
```

macOS / Linux:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install path/to/lattice_mind-0.3.1-<matching-tags>.whl
```

2.2.4 Verify the installation

After installation, these commands should be available on your `PATH`:

Command	Purpose
`latticemind`	Start the assistant (Dashboard / Web UI / CLI chooser).
`latticemind-web`	Launch the browser-based Web UI directly.

Confirm that the intended environment contains version 0.3.1:

```
python -c "from importlib.metadata import version; print(version('lattice-mind'))"
```

Then install the RESCU license as described in *Licenses and solver executables* before starting `latticemind` or `latticemind-web`.

2.2.5 What gets installed

The wheel installs everything LatticeMind needs to build, validate, and analyse calculations, including the post-processing libraries `h5py` (RESCU `.mat` and `.h5` outputs), `scipy` (legacy MATLAB files and numerical helpers), and `matplotlib` (Python plots when RESCU's own plotting does not cover a requested figure). It also bundles:

- the **RESCU documentation** used for retrieval — built in and offline, no setup required (see *Licenses and solver executables*);
- a **runtime pseudopotential archive** for turnkey RESCU execution;
- the **Web UI** assets.

> **Note**
>
> You do **not** need MATLAB, a compiler, or any build tools to *use* LatticeMind. Those are only relevant if you build the package from source, which is a developer task.

Continue to *Connecting your AI model* to connect your AI model, then *Licenses and solver executables* to configure the RESCU license used by LatticeMind and RESCU execution. If you plan to run NanoDCAL, that page also explains the separate NanoDCAL license and NeutralAtomDatabase setup.

2.3 Connecting your AI model

LatticeMind uses a large language model (LLM) to plan and reason about your calculations. You supply the model through an API key. **OpenAI is the default**, but Anthropic Claude, Google Gemini, Qwen, and a local Ollama model all work.

> **Important**
>
> The AI model is used only to *plan and reason*. Your calculations run on the RESCU/NanoDCAL/VASP solver, and your API key is never sent to a remote execution host.

2.3.1 What LatticeMind sends

To plan and reason about your work, LatticeMind sends requests to the LLM provider you configure (OpenAI by default). Those requests include your **prompts**, the **calculation context** it builds from them — structures, generated input decks, and short excerpts of solver logs or errors when it repairs a failed step — and any **files you attach**. If you turn on **web research** (off by default), your search queries are sent to the provider's web-search tool as well.

What is **not** sent to the AI provider: the calculations themselves. They run on the RESCU/NanoDCAL/VASP solver on your own machine or the remote host you choose, and your **API key is never sent to an execution host**. The bundled RESCU documentation is consulted locally and never leaves your machine. If you supply a Materials Project key, structure lookups are sent to Materials Project.

Data sent to a provider is handled under **that provider's data policy** — review it for retention and training terms. For fully local operation with no external LLM calls, use a local **Ollama** model (*Choosing a provider*).

2.3.2 Where LatticeMind looks for keys

LatticeMind reads a `.env` file from the folder where you start it, **or** plain environment variables. Either works; environment variables win.

Using a `.env` file (create it in your working folder):

```
OPENAI_API_KEY=your_key_here
```

Using an environment variable instead:

```
export OPENAI_API_KEY="your_key_here"      # macOS / Linux
```

```
$env:OPENAI_API_KEY="your_key_here"        # Windows (PowerShell)
```

2.3.3 Choosing a provider

OpenAI is used unless you set `RESCU_LLM_PROVIDER`. Set the provider and its matching API key:

Provider	`RESCU_LLM_PROVIDER`	API key variable
OpenAI *(default)*	`openai`	`OPENAI_API_KEY`
Anthropic Claude	`claude`	`CLAUDE_API_KEY` (or `ANTHROPIC_API_KEY`)
Google Gemini	`gemini`	`GEMINI_API_KEY` (or `GOOGLE_API_KEY`)
Qwen (DashScope)	`qwen`	`DASHSCOPE_API_KEY`
Local Ollama	`ollama`	none — set `OLLAMA_BASE_URL` if not on the default port

You can also switch providers at any time from inside the app with `/provider <name>`.

2.3.4 Choosing models

Each provider has a **quick** (fast, default) model and a **thorough** (higher-effort) model. The defaults are sensible; override them only if you need to. See *Settings & configuration reference* for the complete list. A typical OpenAI setup:

```
RESCU_LLM_PROVIDER=openai
OPENAI_API_KEY=your_key_here
OPENAI_MODEL_QUICK=gpt-5.4-mini      # fast / default
OPENAI_MODEL_THOROUGH=gpt-5.4        # higher-effort
```

> **Tip**
>
> For the most capable planning, point the *thorough* model at your provider's strongest model. LatticeMind uses the quick model for routine turns and escalates to the thorough model for harder reasoning.

2.3.5 Optional: Materials Project

LatticeMind can seed crystal structures from the Materials Project[1] database, which improves structure sourcing for less common materials. Provide a key if you have one:

```
MATERIALS_PROJECT_API_KEY=your_key_here
```

2.3.6 Verifying it works

Start LatticeMind and type a trivial prompt:

[1] https://next-gen.materialsproject.org/

```
What can you do?
```

If the key is missing or wrong, LatticeMind reports *"Provider configuration required"* and tells you which variable to set. Otherwise you are ready to build your first project — continue to *Licenses and solver executables* if you also want to run calculations, or jump to *Getting started*.

2.4 Licenses and solver executables

The LatticeMind wheel supplies the assistant, validators, reports, and Python post-processing libraries. Configure a valid RESCU license before starting LatticeMind; the same license supports RESCU execution. If you plan to run NanoDCAL, configure a separate NanoDCAL license and the NeutralAtomDatabase.

2.4.1 The RESCU license used at LatticeMind startup

LatticeMind checks a valid RESCU license file before `latticemind` or `latticemind-web` starts. This check also runs when the selected solver is NanoDCAL. LatticeMind looks in the standard Nanoacademic directory:

- **Windows:** `C:\Users\<YourUsername>\.nanoacademic\RESCU`
- **Linux / macOS:** `$HOME/.nanoacademic/RESCU`

Put exactly one `*.lic` or `*.license` file there. If the file lives elsewhere, or the standard directory contains more than one candidate, select the exact file in `.env` or an environment variable:

```
LATTICEMIND_LICENSE_FILE=/path/to/rescu.lic
```

If LatticeMind cannot validate the selected RESCU license, it displays a `License check failed` message. Check the file path, expiry date, host eligibility, and whether the standard directory contains multiple candidates.

2.4.2 Running RESCU calculations

Input generation does not require a RESCU executable. RESCU execution does. By default LatticeMind runs `rescu` from `PATH`; to select a specific build, set one of:

```
LATTICEMIND_RESCU_CMD=/opt/esapp/rescu
RESCU_EXECUTABLE=/opt/esapp/rescu
```

Install RESCU from the Nanoacademic Portal[2] and test the command directly before the first LatticeMind execution.

The wheel ships a runtime RESCU pseudopotential bundle. To use a different extracted database, point at the directory containing `nano_tm_lda/` and `nano_tm_pbe/`:

```
LATTICEMIND_PSEUDO_ROOT=/path/to/pseudopotentials
RESCU_PSEUDO=/path/to/pseudopotentials
```

The bundled database covers 70 elements under LDA and 51 under PBE. Because LatticeMind uses PBE by default, a material containing one of the LDA-only elements — Ar, Ba, Bi, Br, Cs, F, He, Hg, I, K, Kr, Mn, Ne, Po, Rb, Rn, Sr, Te, Xe — cannot run with the default functional from the bundle alone. LatticeMind reports this before launching rather than silently substituting an LDA potential, which would answer a different question. Supply the missing potential through `LATTICEMIND_PSEUDO_ROOT`, or request LDA explicitly.

[2] https://portal.nanoacademic.com/

2.4.3 Running NanoDCAL calculations

The RESCU license configured for LatticeMind remains in place. To execute a NanoDCAL project, also configure these NanoDCAL components:

1. The NanoDCAL executable.
2. A separate NanoDCAL license on the local or remote computer that runs the calculation, following the NanoDCAL installation instructions.
3. The extracted NeutralAtomDatabase from the Nanoacademic Portal[3].

Set the executable and extracted database root in `.env`:

```
LATTICEMIND_NANODCAL_CMD=/full/path/to/nanodcal
LATTICEMIND_NANODCAL_DATA_PATH=/full/path/to/NeutralAtomDatabase
```

The Web UI exposes the same values under **Settings -> Solver -> NanoDCAL** in the fields **Local NanoDCAL command** and **NanoDCAL neutral atom data**. Point the data field at the extracted directory, not the downloaded archive. LatticeMind searches that directory recursively for each requested element's `*.nad` or `*.mat` basis file. Missing element coverage blocks launch with the missing symbols listed.

LatticeMind stages the selected basis files under the generated project's `nanodcal_basis/` directory so the exact decks can run locally or in a remote bundle. When sharing a project, please keep `nanodcal_basis/` within the scope of your Nanoacademic license.

For remote execution, the selected entry in `~/.latticemind/remote_profiles.toml` must define `nanodcal_command`. The remote computer must have NanoDCAL, its runtime dependencies, and an active NanoDCAL license. See *Remote & HPC execution* and *NanoDCAL support (experimental)*.

2.4.4 The built-in RESCU documentation

While it builds your calculations, LatticeMind looks things up in the RESCU documentation — **and that documentation already ships inside LatticeMind**. It works out of the box, offline, the moment you install, so for almost all users **there is nothing to set up here**.

If you ever want LatticeMind to read the docs from a different local folder (for example a newer RESCU release), add one line to your `.env`:

```
RESCU_DOCS_DIR=/path/to/your/rescu_docs
```

The built-in documentation is fully capable on its own and needs no vector store. If you specifically want retrieval to come from an OpenAI vector store (for example to add your own documents later):

1. Locate the shipped RESCU documents. Run `pip show lattice-mind` and find the **Location:** line; the documents are under `<Location>/rescu_agent/database/rescu`.
2. On https://platform.openai.com, open **Storage → Vector stores**, click **Create**, and name it (e.g. `RESCU docs`).
3. Open the store, click **Add files**, and upload *all* files from the `rescu_agent/database/rescu` folder. Wait until each shows as ready.
4. Copy the store **ID** (it begins with `vs_`). Use *your own* account's store.
5. Add to your `.env`:

   ```
   RESCU_RAG_BACKEND=openai
   OPENAI_VECTOR_STORE_ID=vs_your_store_id_here
   # if your account uses org/project scoping:
   ```

(continues on next page)

[3] https://portal.nanoacademic.com/

(continued from previous page)

```
OPENAI_ORG_ID=org_...
OPENAI_PROJECT_ID=proj_...
```

2.4.5 Do I have everything?

To do this...	You need	Covered by
Start LatticeMind	Wheel + RESCU license + AI provider key	this page and *Connecting your AI model*
Generate & validate inputs	Running LatticeMind; no solver executable required	*Getting started*
Run RESCU locally	RESCU executable + startup license	this page
Run NanoDCAL locally	NanoDCAL executable + NanoDCAL license + NeutralAtom-Database	this page and *NanoDCAL support (experimental)*
Run calculations on HPC	Remote profile + solver and solver license on the remote computer	*Remote & HPC execution*

With the startup license, AI provider, and selected solver files in place, continue to *Getting started*.

2.5 Cloud & headless deployment

LatticeMind is designed to run on cloud platforms, HPC login nodes, and notebook services such as qBraid[4] and JupyterHub.

2.5.1 Running the Web UI on a server

Bind the Web UI to all interfaces and do not try to open a local browser:

```
latticemind-web --port 7865 --no-open
```

The equivalent environment configuration:

```
export LATTICEMIND_CLOUD_MODE=1
   export LATTICEMIND_WEB_PORT=7865
export LATTICEMIND_WEB_OPEN_BROWSER=0
export LATTICEMIND_ARTIFACTS_DIR=/persistent/latticemind_outputs
latticemind-web
```

[4] https://www.qbraid.com/

2.5.2 Cloud checklist

Concern	Recommendation
Secrets	Prefer real environment variables or your platform's secret store over a committed `.env`.
Persistent results	Set `LATTICEMIND_ARTIFACTS_DIR` to a persistent volume so projects survive container restarts.
No browser	Use the plain CLI: `LATTICEMIND_UI_MODE=cli latticemind`.
Behind a proxy	The UI uses relative asset and API URLs by default, so most proxies work as-is (see below).

2.5.3 qBraid / Jupyter proxy quickstart

1. **Set credentials** in a notebook cell or terminal:

```
export OPENAI_API_KEY=...
export LATTICEMIND_CLOUD_MODE=1
```

2. **Start the Web UI** bound to all interfaces:

```
latticemind-web --port 7865 --no-open
```

3. **Open the proxied URL** for port `7865`. Use the form `<notebook-base>/proxy/7865/` — the **trailing slash is required** so that relative paths like `static/styles.css` and `api/config` resolve under the proxy prefix.
4. **If HTML loads but CSS or live updates do not**, the proxy needs an explicit base path:

```
export LATTICEMIND_WEB_BASE_PATH=/proxy/7865
latticemind-web --port 7865 --no-open
```

2.5.4 Working inside an existing folder

On cloud and notebook platforms you often already have a working directory. Use these CLI commands to work *in place* rather than creating a surprise new project folder:

```
/workspace          # inspect the current folder first
/attach .           # adopt the current folder as the active project
/resume last        # continue the latest indexed project
/whereami           # show active project, backend, execution, next action
```

If `/workspace` detects RESCU input files or a prior LatticeMind project, use `/attach .` so LatticeMind adopts that folder.

2.5.5 Choosing where calculations run

Small validation runs are fine in the notebook container. For anything expensive — dense k/q grids, relaxations, DFPT, phonons, optics — run on an HPC cluster via a remote profile (see *Remote & HPC execution*) and gate launches with `/cost` and `/launch-preview`.

2.5.6 A safe first cloud calculation

Keep solver execution off while LatticeMind prepares and validates the inputs, then enter the calculation request:

```
/execute off
Do a simple SCF and band structure calculation for silicon.
```

Once the launch preview looks correct, enable execution with the explicit command, then ask LatticeMind to continue the project:

```
/execute on
Continue this project, run the SCF and band structure, post-process the
bands, and update the report.
```

2.6 Updating & troubleshooting

2.6.1 Updating LatticeMind

When an interactive CLI or Web startup finds a newer release on the public Nanoacademic package index, it prints one line containing the exact upgrade command. LatticeMind checks at most once per 24 hours and stores the last result in `~/.latticemind/update_check.json`. A failed check is silent and is not retried for one hour. Interactive startup waits at most 1.5 seconds for the notice.

```
python -m pip install --upgrade \
  --index-url https://pypi.nanoacademic.ca/simple/ \
  --extra-index-url https://pypi.org/simple/ \
  lattice-mind
```

Set `LATTICEMIND_UPDATE_CHECK=0` to disable the check. It reads only the public `lattice-mind` project page; it does not send prompts, project paths, API keys, or solver files. `latticemind --version` and other shell subcommands never make the check.

To install a wheel file supplied directly by Nanoacademic instead, open a terminal in the directory containing it and run:

```
python -m pip install --upgrade --force-reinstall <matching-wheel>.whl
```

The wheel installs the post-processing libraries (`h5py`, `scipy`, `matplotlib`) by default. Your projects in `latticemind_outputs/` (or `LATTICEMIND_ARTIFACTS_DIR`) are untouched by an upgrade.

To confirm which version you are running, use `latticemind --version`; the interactive welcome banner and the Web UI header and footer also state it.

2.6.2 Troubleshooting

Symptom	Fix
"Provider configuration required" / missing API key	Set the key for your selected provider (see *Choosing a provider*) in `.env` in the folder you launch from, or as an environment variable.
Web UI port already in use	LatticeMind picks another free port automatically, or set `LATTICEMIND_WEB_PORT=7866`.
Web UI loads but CSS / live updates don't (behind a proxy)	Set `LATTICEMIND_WEB_BASE_PATH` to your proxy prefix, e.g. `/proxy/7865`.
No browser / running on a server	Use the CLI: `LATTICEMIND_UI_MODE=cli latticemind`.
LatticeMind reports "License check failed"	Put exactly one valid RESCU `*.lic` or `*.license` file under `~/.nanoacademic/RESCU`, or set `LATTICEMIND_LICENSE_FILE` to the exact file. This startup check applies to every solver selection.
RESCU command needs configuration	Test `rescu` directly, or set `LATTICEMIND_RESCU_CMD` to the exact executable. The RESCU license configured for LatticeMind is also used for RESCU execution.
NanoDCAL command needs configuration	Test NanoDCAL directly, then set `LATTICEMIND_NANODCAL_CMD` or the Web UI's **Local NanoDCAL command** field. NanoDCAL needs its own active solver license on the computer that executes it.
NanoDCAL neutral-atom data path needs configuration	Extract the NeutralAtomDatabase from the Nanoacademic Portal and set `LATTICEMIND_NANODCAL_DATA_PATH` or the Web UI's **NanoDCAL neutral atom data** field to its root directory, not to the archive file.
A DFPT/phonon post-processing step fails in parallel	Some RESCU post-analysis stages are serial-only; LatticeMind strips MPI launchers for them automatically. See *Settings & configuration reference*.

2.6.3 Before launching expensive jobs

Use these guards to avoid surprises:

```
/cost              # estimate resource risk before launch
/launch-preview    # preview commands, inputs, readiness, serial guards, cost
```

RESCU and NanoDCAL feasibility depends on atom count, basis, k/q grids, functional, spin, wall time, memory, and scheduler limits. Start with a cheap validation run before launching dense DFPT, phonon, optics, Raman, relaxation, transport, or large-cell workflows.

2.6.4 Getting help

See *Contact Us* for the Nanoacademic forum and support email, and what to include in a report.

CHAPTER

THREE

GETTING STARTED

This chapter takes you from a fresh install to a finished calculation. You will choose an interface, run your first project step by step, and learn the few concepts — projects, the pipeline, and the interactive commands — that make the rest of LatticeMind easy.

If you have not installed LatticeMind yet, start with *Installation*.

3.1 The three interfaces

LatticeMind offers three ways to work, all driving the same engine. Pick the one that fits where you are working; you can switch at any time, and projects created in one interface open in the others.

Interface	Best for	How to start
Dashboard	Interactive work in a terminal, with live panels	`latticemind` → choose *Dashboard*
Web UI	A visual, browser-based experience; cloud/notebooks	`latticemind-web`
CLI	Servers, HPC login nodes, headless/scripted use	`LATTICEMIND_UI_MODE=cli latticemind`

3.1.1 Starting LatticeMind

Running `latticemind` launches the assistant and lets you choose Dashboard, Web UI, or plain CLI:

```
latticemind
```

To go straight to the browser interface:

```
latticemind-web        # opens http://127.0.0.1:7865
```

To force the plain CLI (recommended on servers and headless terminals):

```
LATTICEMIND_UI_MODE=cli latticemind
```

3.1.2 The Dashboard

The Dashboard is an interactive terminal interface implemented with the Textual terminal-interface framework and the Rich terminal-rendering Python library. You type your request in a prompt box and watch LatticeMind work in live panels — the plan, the current stage, generated files, validation findings, and results update in place. It is the most visual in-terminal interface for interactive work.

3.1.3 The Web UI

The Web UI runs a local web server and opens in your browser. It presents the same workflow visually: a chat/prompt area, a live activity log, and panels for generated inputs, results, plots, and the report. It is ideal for cloud and notebook platforms — see *The Web UI tour* for a full tour and *Cloud & headless deployment* for deployment behind a proxy.

```
latticemind-web                                  # local
latticemind-web --port 7865 --no-open   # server / cloud
```

3.1.4 The CLI

The plain CLI is a straightforward read–eval–print loop: you type prompts and commands, and output streams to the terminal. It has no live panels or browser dependency, which makes it the most robust choice on HPC login nodes, inside containers, and over SSH. All interactive commands (see *Interactive commands*) work here.

> **Tip**
>
> Whichever interface you use, the same interactive commands (`/help`, `/examples`, `/cost`, `/remote` …) and the same project folders apply. Start a project in the Web UI and resume it later from the CLI with `/resume last`.

3.1.5 Which should I use?

- **On your laptop, interactively:** the Dashboard or the Web UI.
- **On a cloud/notebook platform:** the Web UI (see *qBraid / Jupyter proxy quickstart*).
- **On an HPC login node or in a script:** the CLI.
- **From Codex, Claude, or another shell agent:** use the commands in *Using LatticeMind from Codex, Claude, and shell agents*; add `--json` so the agent can read exact fields and exit codes.

3.2 Using LatticeMind from Codex, Claude, and shell agents

Codex, Claude, and other agents that can run shell commands can discover and operate LatticeMind without learning its interactive screen layout. The commands on this page print plain text by default. Add `--json` when another program will read the result.

The discovery commands do not require a solver license or an AI-provider key. They are designed to work while an agent is first examining a computer and deciding how LatticeMind should be used.

3.2.1 Let an agent discover the installation

Ask the agent to run these commands in order:

```
latticemind --version
latticemind capabilities --json
latticemind docs "how do I configure a solver and start a project?"
latticemind doctor --json
```

`capabilities` returns the shell commands, the interactive slash commands, supported solvers, JSON field names, exit codes, and the solver-execution approval rule. `docs` searches LatticeMind, RESCU, and NanoDCAL reference text and infers the corpus from the question. `doctor` checks the selected solver command, project output directory, Web UI assets, and remote-profile configuration without launching a solver or opening an SSH connection.

3.2.2 Search LatticeMind and solver documentation

Use a natural-language question. The default response contains one concise answer and its source:

```
latticemind docs "How do I configure a NanoDCAL NeutralAtomDatabase?"
latticemind docs "What does info.savepath control?"
latticemind docs "What values can calculation.SCF.mixMethod use?"
latticemind docs "How do I resume a failed Slurm calculation?"
```

`docs` recognizes NanoDCAL markers such as `calculation.SCF` and `NeutralAtomDatabase`, RESCU markers such as `info.savepath` and `kpoint.gridn`, and LatticeMind command/interface names. If those markers do not identify one corpus, it searches all three. `--corpus auto` is the default; `--corpus latticemind`, `rescu`, `nanodcal`, or `all` overrides the choice.

Add `--json` for a compact JSON envelope containing the inferred corpus and up to three excerpts. Add `--verbose` to show full fields for those excerpts, including scores, distribution names, and corpus counts. Use `--limit N` to change the maximum number of returned excerpts:

```
latticemind docs "What does info.savepath control?" --json
latticemind docs "What does info.savepath control?" --verbose
latticemind docs "What does info.savepath control?" --verbose --limit 6
```

An agent can read an exact LatticeMind page when it needs the full wording:

```
latticemind docs --list
latticemind docs --source installation/solver_setup.rst
```

`--source` is limited to this LatticeMind user manual. It does not return a complete RESCU or NanoDCAL page. A source checkout can search approved `.rst` or `.txt` excerpts from its local solver mirrors. A wheel instead uses the sanitized RESCU text under `rescu_agent/database/rescu` and the hand-reviewed NanoDCAL keyword/calculation-name schema. The raw RESCU, NanoDCAL, and VASP documentation mirrors are not included in the wheel.

The solver search reads documentation text only. It does not index Python, shell, C/C++, MATLAB, notebook, raw HTML, `raw/`, `generated/`, or `extracted/` files. A document containing a private-key or recognized API-token signature is skipped, and a Windows, Linux, or macOS user-home path in an excerpt is replaced with `<user-home>`. `latticemind docs --list --json` reports corpus counts and these exclusions without returning local solver-mirror paths.

3.2.3 Inspect an existing project without changing it

The inspection command reads the project path, `project_state.json`, `calculation_status.json`, `artifact_registry.json`, report excerpt, and indexed project files:

```
latticemind project inspect ./my-silicon-project --json
```

For a specific question, combine those project files with relevant LatticeMind, RESCU, and NanoDCAL documentation excerpts:

```
latticemind project ask \
  "Which calculation failed, which output is stale, and how should I continue it?" \
  --project ./my-silicon-project --json
```

Both commands are read-only. They do not append `conversation.jsonl` or `project_events.jsonl`, and they do not rewrite solver inputs or reports.

3.2.4 Run one prompt without opening an interactive interface

`latticemind run` creates or continues a normal LatticeMind project, runs the planner, writes solver inputs, validates them, and returns the report and exact project path in one response:

```
latticemind run \
  "Do a PBE SCF and band structure calculation for silicon." \
  --solver rescu --json
```

The one-shot command disables solver execution by default. It may write the same project files as an interactive planning turn, including `project_state.json`, `project_events.jsonl`, `project_root/inputs/*`, helper scripts, validation diagnostics, and `report.md`.

Continue the returned project path with a second prompt:

```
latticemind run \
  "Increase the SCF k-point mesh to 10 by 10 by 10." \
  --project ./latticemind_outputs/<returned-project> --solver rescu --json
```

Permit a solver launch only when that is the intended action:

```
latticemind run \
  "Run the validated silicon workflow and prepare the final figures." \
  --project ./latticemind_outputs/<returned-project> \
  --solver rescu --execute --approve-execution --json
```

Both execution flags are required. With neither flag, LatticeMind plans and validates without launching RESCU, NanoDCAL, or VASP. With exactly one flag, the command exits before changing a project. With both flags, the normal validator and launch-readiness checks still run before the solver command.

3.2.5 Licenses and provider settings for one-shot work

`latticemind run` uses the same setup as the Dashboard and Web UI. Configure a valid RESCU license for LatticeMind and RESCU execution. Configure a separate NanoDCAL license when the selected solver is NanoDCAL, and provide the NanoDCAL `NeutralAtomDatabase` obtained from the Nanoacademic portal. Select an AI provider with `RESCU_LLM_PROVIDER` and configure that provider's key, or use a configured local Ollama endpoint.

If the license or provider check fails, the JSON response uses `ok: false`, places the explanation in `diagnostics`, and exits with code 3. Existing project files are preserved.

3.2.6 Read the JSON response and exit code

Machine-readable responses use the same six top-level fields:

```
{
  "schema_version": 1,
  "command": "project.inspect",
  "ok": true,
  "data": {},
  "diagnostics": [],
  "next_actions": []
}
```

An agent should check `ok` first, read the command-specific `data` object, report every `diagnostics` entry, and use `next_actions` only after it has checked that the suggested path and action match the user's request.

Exit code	Meaning
0	The command completed successfully.
2	A command name, argument, or execution-approval flag was invalid.
3	The license or selected AI provider needs configuration.
4	The requested project path or manual page was not found.
5	The one-shot graph run failed; inspect `diagnostics` and the preserved project path.

3.3 Your first project

Let's run a real calculation end to end. We'll compute the band structure and density of states of silicon — the same example used in *Silicon: SCF, band structure & DOS*, but here we focus on the *experience* of driving LatticeMind for the first time.

Before you begin

You need LatticeMind installed with an API key (*Connecting your AI model*) and the RESCU license required at startup (*Licenses and solver executables*). To execute this calculation you also need the RESCU executable. Without the executable, you can still plan, generate, validate, and inspect the inputs safely.

3.3.1 Step 1 — Start LatticeMind

Pick whichever interface you like (*The three interfaces*). We'll show the CLI; the Web UI is identical in spirit.

```
latticemind
```

3.3.2 Step 2 — Describe what you want

Type your request in plain language:

```
Do an SCF, band structure, and DOS calculation for silicon, and plot the
band structure and DOS.
```

LatticeMind plans the workflow, builds the silicon structure, chooses parameters, and renders the RESCU inputs — narrating each stage as it goes.

3.3.3 Step 3 — Look before you leap

This is the habit worth forming. Before running anything, inspect the plan:

```
/status            # the planned calculations and their state
/inputs            # the input decks LatticeMind generated
/launch-preview    # exactly what will run, plus a cost estimate
```

If anything looks off, just say so — *"use an 8×8×8 k-grid"*, *"add a relaxation first"* — and LatticeMind re-plans. You are always in control.

3.3.4 Step 4 — Run it

When you're happy, enable execution and go:

```
/execute on
run it
```

LatticeMind runs the SCF, hands its converged density to the band-structure and DOS steps, and post-processes the results. In the Web UI, the **Live run** panel shows this happening in real time:

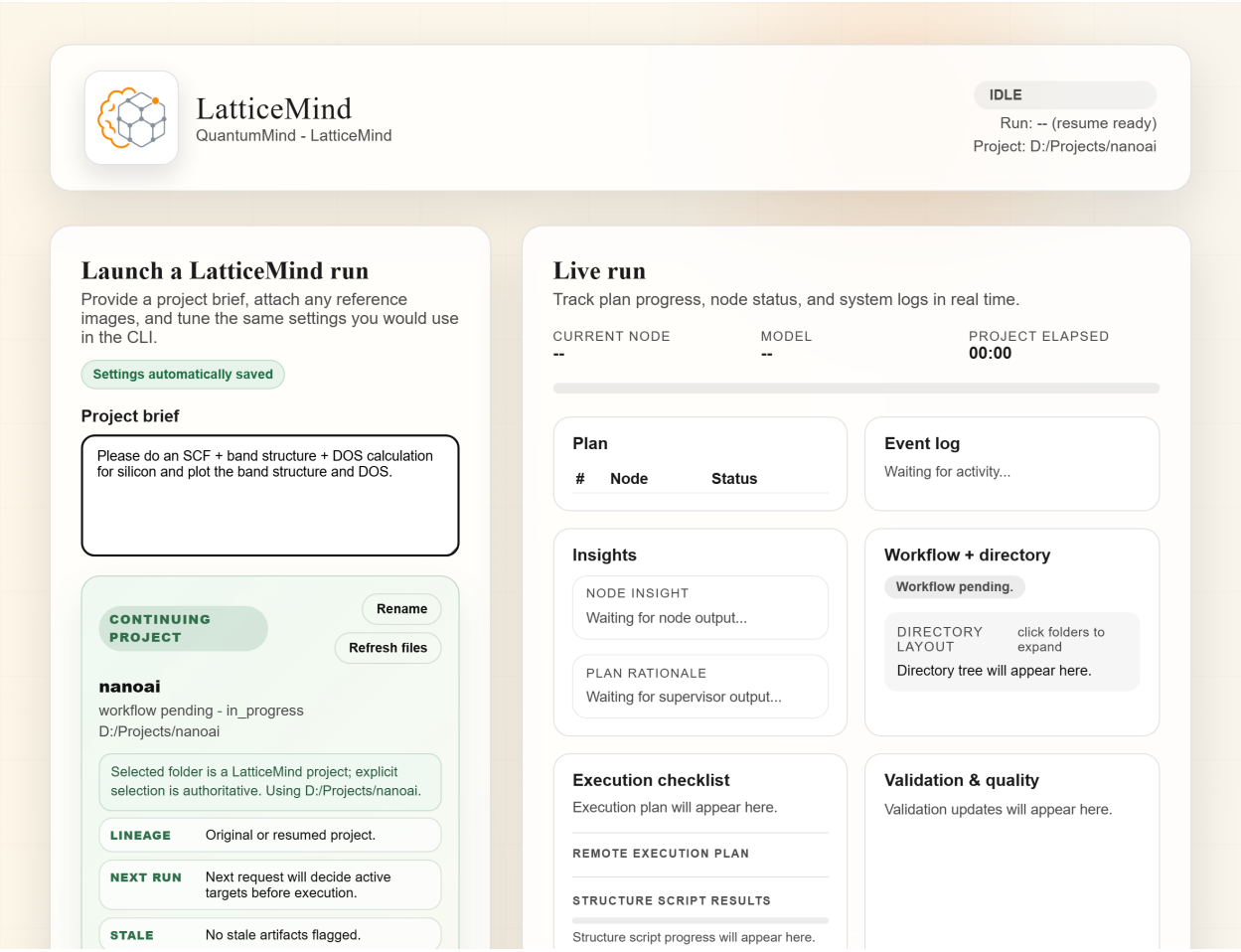

Fig. 3.3.1: The same first project in the Web UI: the brief on the left, live progress on the right.

3.3.5 Step 5 — See your results

When it finishes, LatticeMind has produced plots, extracted quantities, and a report:

```
/analysis     # extracted values: total energy, band gap, electron count
/plots        # the generated figures
/outputs      # everything, including the report
```

Everything lives in a self-contained project folder you can revisit, share, or resume — see *Projects & the workspace*.

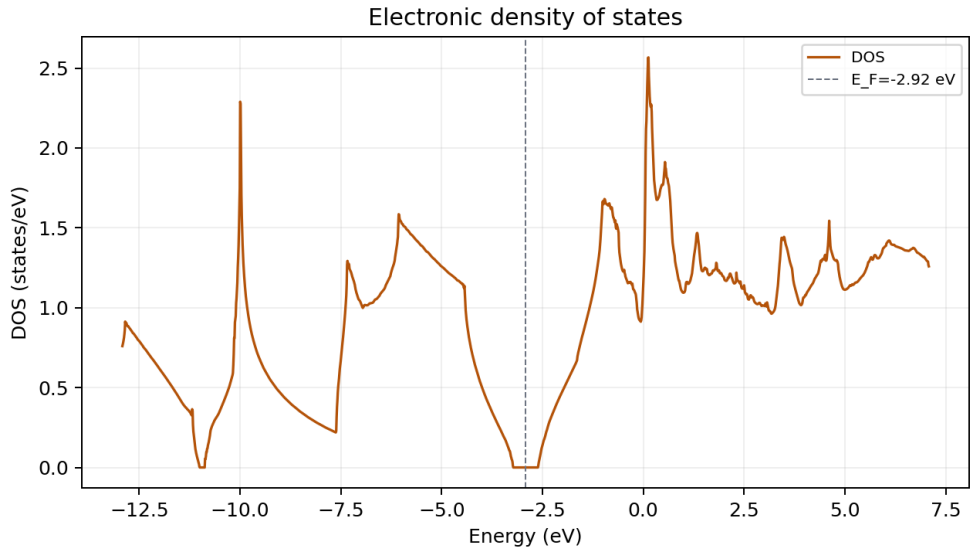

Fig. 3.3.2: Your first result: the silicon density of states, with the Fermi level in the gap. (See *Silicon: SCF, band structure & DOS* for the band structure and the numbers.)

3.3.6 What next?

- Learn what happened under the hood: *How LatticeMind works*.
- Try more materials and calculation types: *Basic tutorials*.
- Scale up to a cluster: *Running on an HPC cluster*.

Tip

Stuck at any point? Type `/why` for an explanation of the current state, or `/tips` for suggested next actions.

3.4 How LatticeMind works

You do not need to understand LatticeMind's internals to use it — but a mental model of what happens between your prompt and your results will help you write better prompts, read the live progress, and understand why a step was repaired or blocked.

3.4.1 The pipeline

Every project flows through the same stages, shown in Fig. 3.4.1. Each stage reads the output of the previous one and either proceeds or requests a repair.

Planner
: Reads your request and writes an explicit **plan**: the ordered list of calculations, each with a type (SCF, DOS, band structure, …) and its dependencies. This is what everything else is built from.

Structure builder
: Builds the crystal or two-probe device geometry — seeding from known lattice parameters, constructing supercells, alloys, or defects as requested — and verifies the result is physically reasonable before continuing.

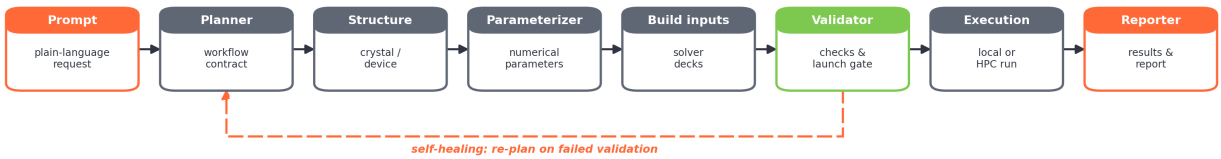

Fig. 3.4.1: The stages of a LatticeMind project, from prompt to report.

Parameterizer
: Selects the numerical parameters for each calculation from a curated, solver-specific vocabulary: k-point grids, real-space resolution, smearing, exchange–correlation functional, Hubbard U, electric fields, q-paths, and so on.

Build inputs
: Renders the concrete solver input decks (e.g. `inputs/scf_main.input`) and automatically wires the hand-offs between steps — pointing a band-structure run at the converged density of its SCF parent, for example.

Validator
: Runs deterministic and AI checks on the rendered inputs. A blocking failure names the exact file, field, or keyword at fault (a missing structure, a duplicate output path, a missing density hand-off). If the plan cannot be executed as written, the project is sent back to be re-planned — this is the **self-healing** loop.

Execution
: Runs the validated inputs, locally or by bundling a self-contained job and submitting it to a remote workstation or Slurm cluster, then collecting the results.

Reporter
: Post-processes the raw output — extracting band gaps, generating plots, integrating densities — and writes a readable report. Band structures are plotted along the high-symmetry path your deck names, with the standard labels (Γ, X, W, …) at their exact positions; a convergence study (for example, comparing k-point meshes) concludes with a total-energy convergence figure rather than a list of numbers.

3.4.2 Self-healing and human-in-the-loop

Two design choices make LatticeMind trustworthy:

It validates before it runs. Nothing is executed until the validator agrees the inputs are launch-ready. When validation fails, LatticeMind archives the current attempt, revises the plan, and tries again — rather than launching a doomed calculation and failing hours later.

It asks before expensive or irreversible actions. LatticeMind previews the exact commands, inputs, and (for remote runs) the job script and resources before launching. You can inspect cost with `/cost` and the full plan with `/launch-preview`. Execution is off until you turn it on.

> **Tip**
>
> Because the plan is explicit, you can steer it. If LatticeMind's plan is not what you wanted, just say so in plain language — *"use a denser k-grid"*, *"add a DOS step"*, *"relax the structure first"* — and it re-plans.

3.4.3 One project, one solver

A single project is bound to one solver (RESCU, NanoDCAL, or VASP). LatticeMind does not mix solver vocabularies within a project. To compare solvers, create separate projects. The solver is inferred from your prompt (for example, *"Using NanoDCAL, …"*) or set explicitly.

3.4.4 What you can rely on

- **Correct multi-step chaining.** Densities and wavefunctions flow from one step to the next automatically, each with a unique result path.
- **Reproducibility.** Every decision is recorded in the project's event log and diagnostics, so a project can be resumed, inspected, or shared.
- **Restart safety.** Continuing an interrupted calculation reuses the same inputs and save paths; changing an input marks downstream steps as stale so they are recomputed correctly. A calculation released with `--detach` can be picked up in a later session — see *Releasing a long calculation*.
- **The structure you asked for.** Naming a Materials Project identifier — for example `scf + bs + dos for Si: mp-149` — builds from that exact published cell. A request that names no identifier uses LatticeMind's curated crystal reference table instead, so the two can give slightly different numbers for the same material; that difference is the difference between the two cells.
- **Deterministic band paths.** High-symmetry k-point paths are chosen from the lattice by a fixed table rather than being improvised per run, so the same structure always yields the same path.
- **Honest classification.** A step is described as metallic on the evidence, but never for a material known experimentally to be a semiconductor or insulator whose gap merely closed under PBE. Such a step is reported as unverified, keeps its computed gap as a lower bound, and notes that a hybrid functional is needed for a quantitative answer.
- **Preflight on the environment.** Missing pseudopotential coverage for the requested functional is reported before launch rather than surfacing as a solver error part-way through.
- **Continuation that always works.** A finished run never blocks its own project: you can keep adding steps — *"add a DOS calculation"* — turn after turn, and each one reuses the converged results already on disk. Questions and approval prompts belong to the turn that raised them; they never resurface on a later request or in a new project.
- **The same figures on every platform.** Where RESCU's built-in plotting is unavailable (for example, managed cloud environments without a MATLAB runtime), LatticeMind renders the same canonical figures itself and the report says so in a note — the results and plots are complete either way. On such machines, turn the built-in plotter off with `/rescu-plotting off` (saved with your settings) so it is never attempted at all.

Next, see where all of this lands on disk in *Projects & the workspace*.

3.5 Projects & the workspace

Everything LatticeMind does lives in a **project** — a self-contained folder holding the inputs it generated, the results it produced, the plots and report, and a complete log of what happened. Projects are portable and reproducible: you can archive one, share it, or resume it later.

3.5.1 Where projects are stored

By default, projects are created under a `latticemind_outputs/` folder in your current working directory, each in its own timestamped subfolder. Override the root with:

```
export LATTICEMIND_ARTIFACTS_DIR=/path/to/outputs
```

On shared machines, HPC, and cloud, set this to persistent storage so projects survive restarts.

3.5.2 Managed vs. attached projects

Mode	Behaviour
Managed	LatticeMind creates a new project folder under `latticemind_outputs/`. This is the default when you start a new calculation.
Attached	LatticeMind works *in place* inside a folder you choose. Use `/attach .` to adopt the current folder — ideal when you already have RESCU inputs or a prior project. LatticeMind writes only lightweight metadata (workspace file, diagnostics, reports, logs) into it.

Use `/workspace` to inspect a folder before deciding, and `/whereami` to see which project is active.

3.5.3 Anatomy of a project folder

A completed RESCU project typically contains:

```
my_project/
├── inputs/                    # generated solver input decks
│   ├── scf_main.input
│   ├── dos_main.input
│   └── bs_main.input
├── results/                   # solver output
│   ├── scf_main.mat           #   raw results (.mat / .h5)
│   ├── figures/               #   band structure & DOS plots (.png)
│   └── analysis_summary.json  #   extracted quantities (band gap, …)
├── scripts/                   # generated structure & post-processing helpers
├── report.html                # the human-readable report
└── (status & log files)       # bookkeeping LatticeMind manages for you
```

File / folder	What it is for
`inputs/*.input`	The solver input decks LatticeMind generated. These are exactly what is executed.
`results/`	Raw solver output (`.mat`/`.h5`), extracted quantities (`analysis_summary.json`), and plots under `figures/`.
`scripts/`	Generated structure-building and post-processing helper scripts.
`report.html`	The readable summary of the project: what was run, key results, and figures. (Set `RESCU_REPORT_FORMAT=off` to disable.)
`.latticemind/`	A hidden directory holding the status, event-log and provenance records LatticeMind uses to resume a project, track progress and self-heal. It travels with the project when you copy, archive or share it, so the history of every number stays attached to the results. You do not need to open or edit anything inside it, and the project directory itself shows only the files you asked for.

3.5.4 Where projects may be created

LatticeMind will not treat your home directory, the filesystem root, or a system directory as a project, so a mistyped path cannot scatter project files across your machine. Work in a subdirectory instead — a folder of your own, or the `latticemind_outputs/` directory LatticeMind creates for you.

When you work in a directory LatticeMind has not seen before, it may ask whether the files there are yours to read, because it can pick up structure files sitting alongside your work. Record an answer once and it is remembered, including for projects created beneath that folder:

```
/trust              # show the current answer for this directory
/trust yes          # allow LatticeMind to use structure files found here
/trust no           # attach files explicitly instead
/trust forget       # clear the recorded answer
```

Files you attach by name are always used; the question is only about files LatticeMind finds on its own.

3.5.5 Resuming and organizing work

```
/resume last           # continue the most recent project
/title "Si phonons"    # give the active project a readable display title
/whereami              # show active project, backend, execution, next action
```

Because a project is just a folder, you can copy it to another machine and resume it there — the generated inputs and (for NanoDCAL) the basis files travel with the project.

> **Note**
>
> LatticeMind preserves `input_filename` and `info.savepath` when you continue an interrupted calculation, so a restart reuses the same inputs and result paths. If you change an input, downstream steps are marked *stale* and recomputed.

3.6 Interactive commands

Most of the time you just talk to LatticeMind in plain language. But a set of short **slash commands** give you precise control over projects, execution, and inspection. Type them into any interface; they all begin with `/`.

> **Tip**
>
> Two commands to remember on day one: `/commands` lists everything, and `/why` explains the current pause or blocker in plain language.

3.6.1 The essentials

Command	What it does
`/commands`	Show the full command palette (alias `/help`).
`/examples`	Show short, ready-to-run example prompts.
`/why`	Explain the current pause, blocker, or suggested next action.
`/tips`	Suggest sensible next actions for where you are.
`/whereami`	Show the active project, solver, execution mode, and settings.
`/exit`	Leave LatticeMind.

3.6.2 Working with projects

Command	What it does
`/new`	Start a fresh managed project.
`/projects`	List recent projects.
`/resume last`	Continue the most recent project (or name one).
`/attach .`	Adopt the current folder as the active project (work in place).
`/workspace`	Inspect the current folder before deciding new/resume/attach.
`/trust [yes|no|forget]`	Show or set whether LatticeMind may use structure files it finds in the current directory. Files you attach by name are always used.
`/title <name>`	Give the active project a readable display title.
`/fork last`	Copy a project before experimenting, leaving the original intact.

3.6.3 Controlling execution

Execution is **off** until you turn it on, and LatticeMind always lets you preview first.

Command	What it does
`/execute on`	Allow the solver to run after validation passes.
`/launch-preview`	Preview the exact commands, inputs, readiness, and cost before launch.
`/cost`	Estimate the resource risk of the planned run.
`/compute <local|remote|slurm>`	Choose where calculations run.
`/rescu <command>`	Point LatticeMind at your RESCU executable.
`/lease status`	Show whether a released calculation still holds the project's execution lease; `/lease clear` removes a stale one.
`/provider` / `/model`	Show (or switch) the AI model provider and model in use.

3.6.4 Checking progress & results

Command	What it does
`/status`	Show workflow and per-calculation status.
`/inputs` / `/outputs`	List generated input decks / results, plots, logs, and reports.
`/plots`	List generated plots.
`/analysis`	Show the extracted analysis summary (band gaps, energies, …).
`/failed` / `/stale`	List calculations that failed or became stale, with suggested reruns.
`/continue <target>`	Resume a failed or stale step.

3.6.5 Steering the plan

You do not need a command to change the plan — just say what you want:

```
Use a denser 8x8x8 k-grid for the SCF.
Add a DOS step after the SCF.
Relax the structure before the band structure.
Redo the band structure along L-Γ-X-W-K.
```

LatticeMind updates the plan and re-validates.

3.6.6 Where to go deeper

- The **complete** command palette, with every command and alias, is in *Interactive command palette*.
- Remote/HPC commands (`/remote …`) are covered in *Remote & HPC execution*.
- Cloud/proxy commands (`/qbraid`, `/cloud status`) are in *Cloud & headless deployment*.

3.7 The Web UI tour

The Web UI gives you the full LatticeMind workflow in your browser: describe a project, tune settings, launch, and watch it run in real time. Launch it with:

```
latticemind-web        # opens http://127.0.0.1:7865
```

3.7.1 The dashboard at a glance

Across the top of the run panel you always see the current **status** (`IDLE`, running, etc.), the **Run** id, and the active **Project** path.

3.7.2 Launching a run

1. Type your request in the **Project brief** box — the same plain-language prompt you would use in the CLI (*"Do an SCF, band structure, and DOS calculation for silicon, and plot the band structure and DOS."*).
2. Optionally attach reference **images** or **documents**.
3. Tune settings (below) if you need to.
4. Click Run.

The **active project** card shows whether you are starting fresh or *Continuing* an existing project, with **Rename** and **Refresh files** controls.

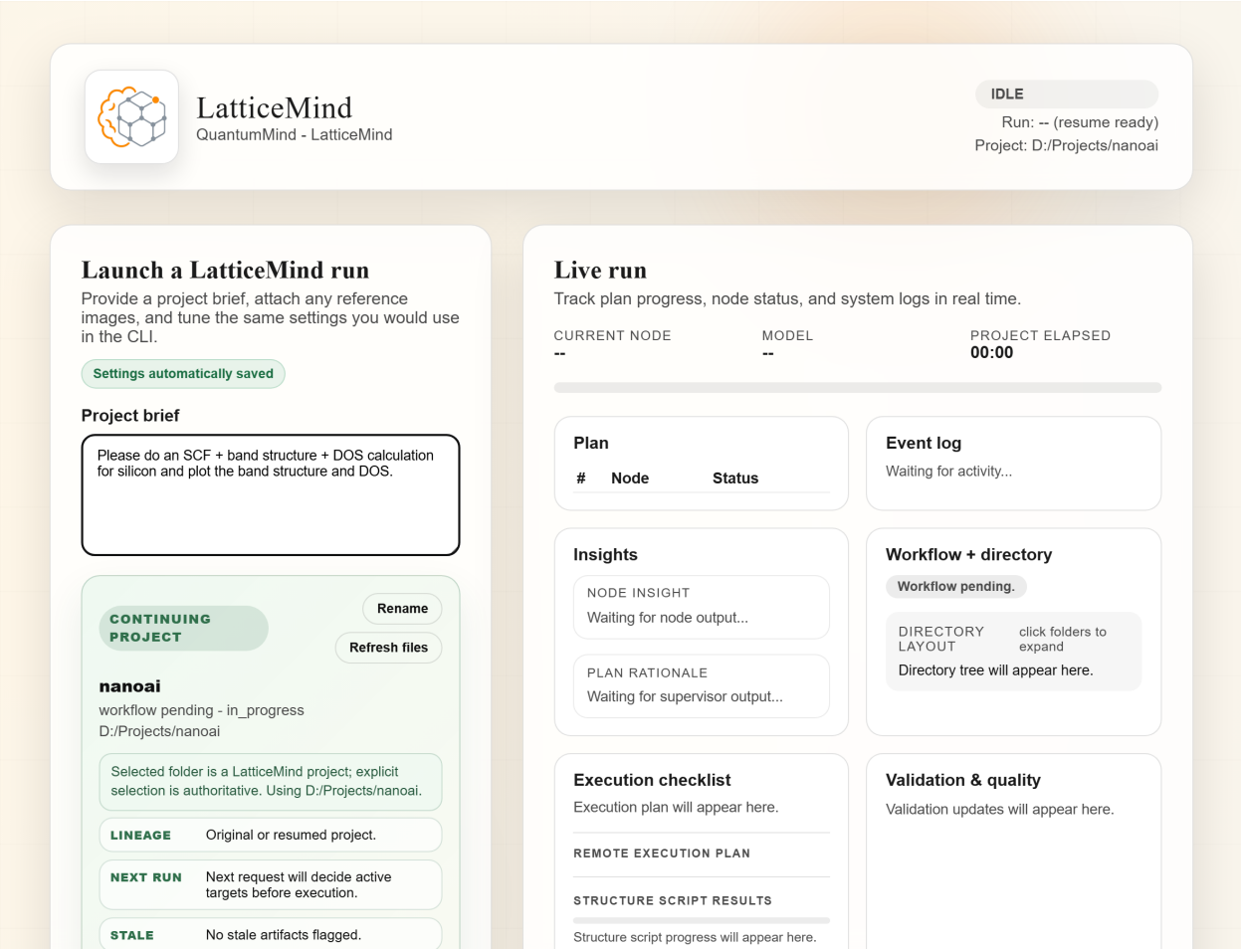

Fig. 3.7.1: The LatticeMind dashboard. **Left:** the run panel — status, project brief, and the active project. **Right:** the *Live run* panel — plan progress, event log, insights, workflow/directory tree, execution checklist, and validation.

3.7.3 Settings

All the settings you would set in the CLI or `.env` are here, saved automatically as you change them (*"Settings automatically saved"*):

Control	What it sets
Reasoning effort	How hard the AI thinks (`Auto` by default).
LLM provider / Model override	The AI backend and an optional specific model id.
Report format / Report language / Interface language	HTML or off; report and UI language (auto/en/fr/zh).
Solver	RESCU, NanoDCAL (experimental), or VASP (preview).
Local RESCU command	Path to your solver executable (e.g. `D:\rescu-2.9.0\bin\rescu.exe`).
Parallel launcher / MPI ranks / Threads/rank	Local and remote parallel execution.
RESCU pseudopotential library	Optional override for the bundled pseudopotentials.
Compute target / Remote profile / Scheduler	Local, remote SSH, or Slurm — with **Set Up SSH Reuse**, **Prepare SSH Session**, and **Test Remote Profile** buttons (see *Remote & HPC execution*).

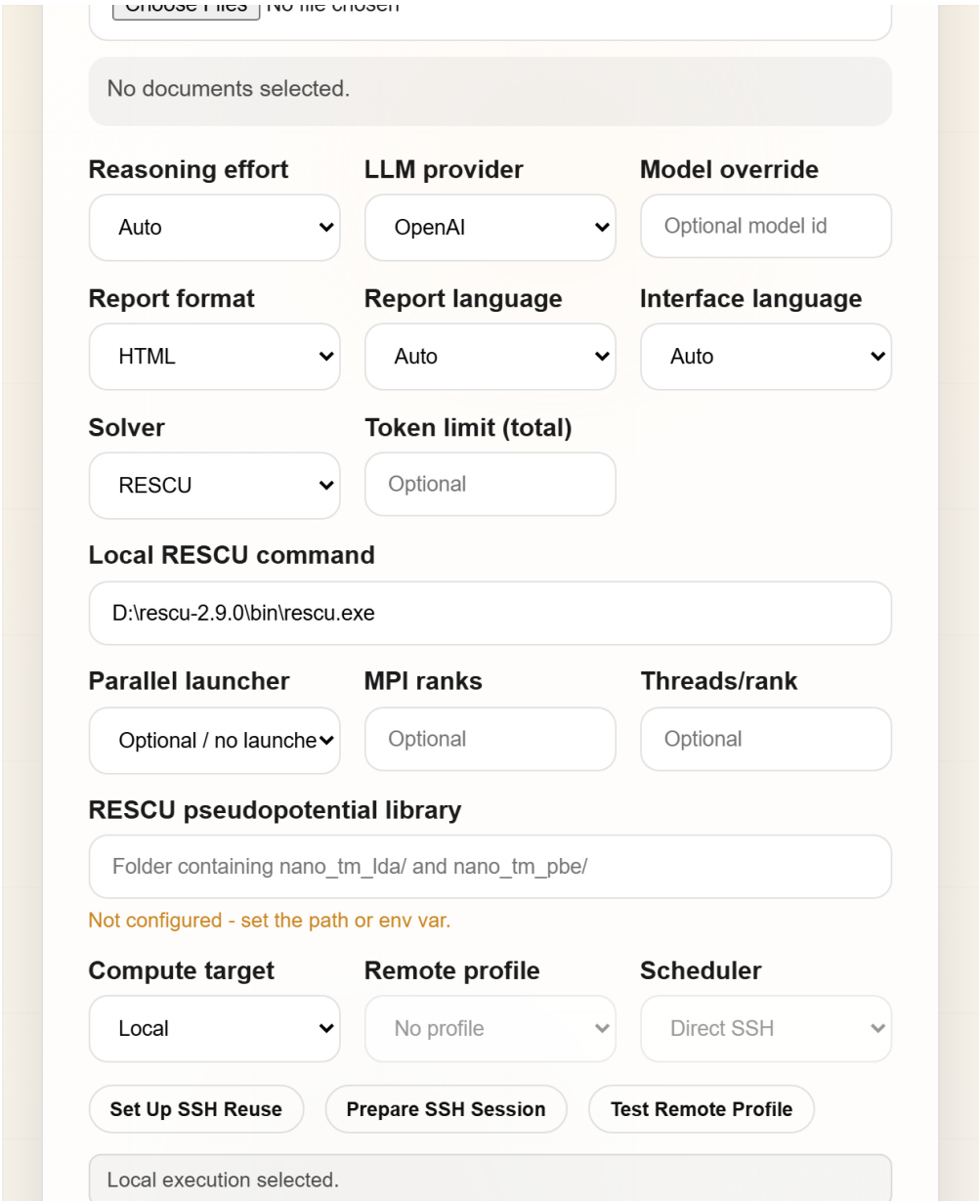

Fig. 3.7.2: The settings panel, with RESCU selected as the solver.

3.7.4 Execution controls

Below the settings are the execution toggles and the action buttons:

- **Allow clarifications** lets LatticeMind ask you a question when a request is ambiguous (on by default).
- **Enable web research** lets the model run web searches through your LLM provider's built-in search tool when it needs outside information. It is **off by default**; when on, your search queries are sent to your provider (see *What LatticeMind sends*).
- **Skip validation** turns off the launch gate (not recommended).
- **Enable solver execution** allows the calculation to actually run; leave it off to only generate and validate inputs.
- **Autopilot execution mode** auto-approves launch gates when the plan is ready.

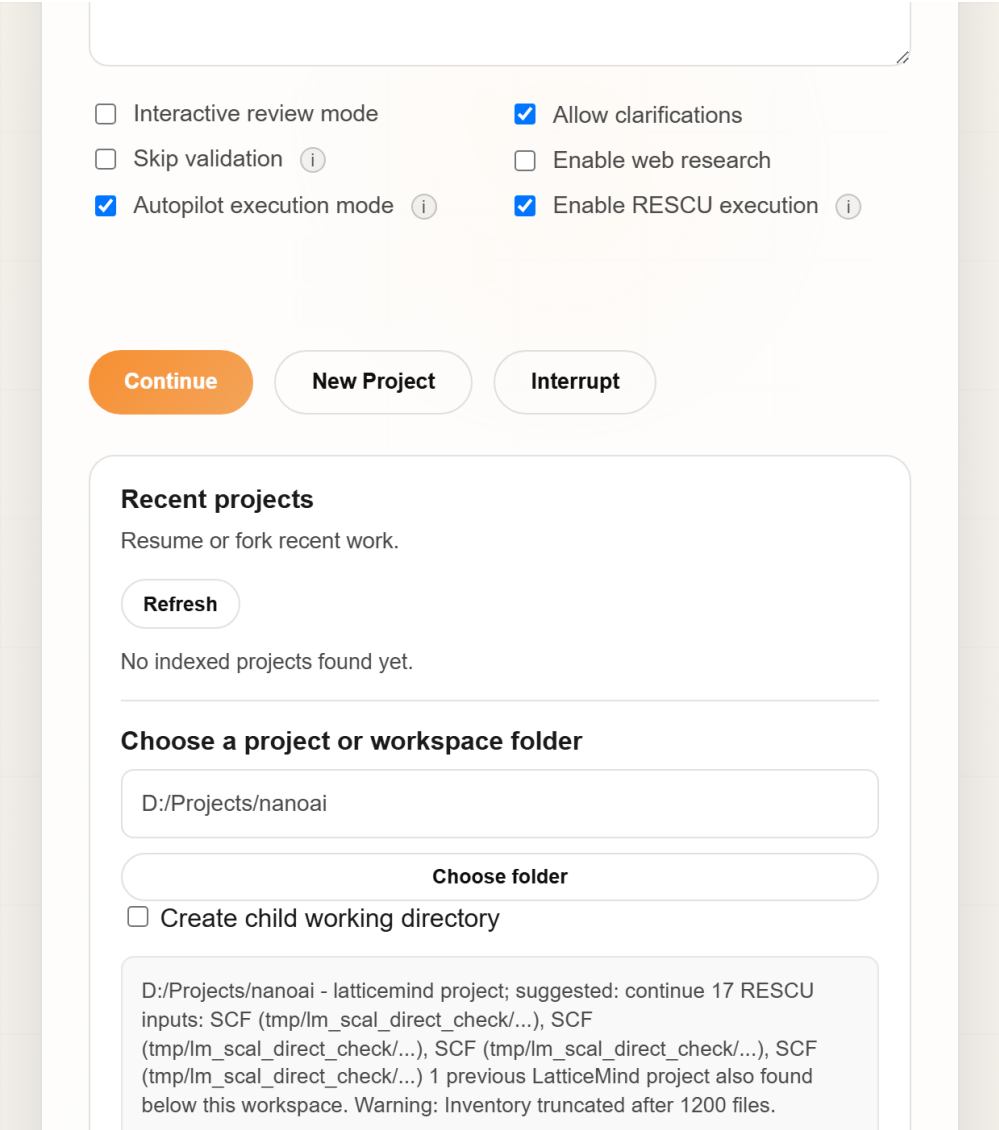

Fig. 3.7.3: Execution toggles and the run controls. **Enable RESCU execution** and **Autopilot execution mode** are the two you will use most.

- The buttons — Run / **Continue**, **New Project**, and **Interrupt** — start, restart, or stop a run.

3.7.5 Watching a run

The **Live run** panel (right side of Fig. 3.7.1) updates in real time as LatticeMind works:

- **Current node**, **Model**, and **Project elapsed** time.
- A **Plan** table showing each calculation and its status.
- An **Event log** of everything happening.
- **Insights** (node and plan rationale), the **Workflow + directory** tree, the **Execution checklist**, and **Validation & quality** updates.
- **Tokens & budget** and **Computation cost** panels.

3.7.6 Recent projects & folders

The **Recent projects** panel lists indexed projects to *Resume or fork*. The **Choose a project or workspace folder** control lets you point LatticeMind at an existing folder and adopt it as the active project (the browser equivalent of `/attach`). See *Projects & the workspace*.

3.7.7 Results

When a run finishes, the generated plots and the report appear in the project, and the raw files are on disk in the project folder. See *Projects & the workspace* for the layout, and *Silicon: SCF, band structure & DOS* for a full worked example.

CHAPTER

FOUR

BASIC TUTORIALS

These worked examples use the RESCU solver — LatticeMind's primary, well-tested backend. Each tutorial shows the exact prompt, what LatticeMind does with it, and the results you get. Every one is short and reproducible; the projects they are based on ship in `latticemind_outputs/`.

4.1 Silicon: SCF, band structure & DOS

This is the "hello world" of LatticeMind. In one prompt you will compute the electronic ground state of bulk silicon, its band structure, and its density of states — and get plots and a report. It is the ideal first project.

> **What you need**
>
> LatticeMind installed with an API key (*Connecting your AI model*). To *run* the calculation you also need the RESCU solver and a license (*Licenses and solver executables*); without them you can still follow every step up to execution.

4.1.1 The prompt

Start LatticeMind (`latticemind` or `latticemind-web`) and type:

```
Do an SCF, band structure, and DOS calculation for silicon, and plot the
band structure and DOS.
```

That is the entire input. You do not specify the lattice constant, the k-point grid, the functional, or any RESCU keyword — LatticeMind fills those in.

4.1.2 What LatticeMind does

LatticeMind moves through its pipeline (*How LatticeMind works*) and reports each stage as it goes:

1. **Plans** a three-step workflow: `SCF` → `BAND-STRUCTURE` → `DOS`, with both post-SCF steps depending on the converged SCF density.
2. **Builds** the diamond-cubic silicon cell (two atoms, lattice constant 5.43 Å).
3. **Chooses parameters**: the PBE (GGA) functional, a `12×12×12` k-point grid, and a double-zeta-polarized pseudopotential basis.
4. **Renders** three input decks and wires the density hand-off from SCF to the band-structure and DOS steps.
5. **Validates** the inputs and, when execution is enabled, **runs** them.
6. **Post-processes** the outputs into plots and a report.

4.1.3 The generated SCF input

LatticeMind writes standard RESCU decks to `project_root/inputs/`. The SCF deck it produced for this project — which you can open, keep, and re-run — is:

```
info.calculationType = 'self-consistent';
info.savepath = './results/scf';
units.length = 'Angstrom';
domain.latvec = [...
    0.0 2.715 2.715;
    2.715 0.0 2.715;
    2.715 2.715 0.0;
];
domain.lowres = 0.5;
atom.fracxyz = [...
    0.0 0.0 0.0;
    0.25 0.25 0.25;
];
atom.element = [1 1];
element(1).species = 'Si';
element(1).path = './pseudopotentials/Si_PBE_TM_DZP.mat';
functional.libxc = true;
functional.list = {'XC_GGA_X_PBE' 'XC_GGA_C_PBE'};
kpoint.gridn = [12 12 12];
option.maxSCFiteration = 100;
```

Tip

Not happy with a choice? Just say so — *"use an 8×8×8 k-grid"* or *"switch to LDA"* — and LatticeMind re-plans and re-renders. You never edit the deck by hand (though you can).

4.1.4 Running it

Preview and launch:

```
/launch-preview     # see exactly what will run, and the cost
/execute on         # allow execution
run it
```

LatticeMind runs the SCF, feeds its density into the band-structure and DOS steps, and then post-processes the results.

4.1.5 The results

For this project LatticeMind extracted:

Quantity	Value
SCF total energy	−7.8827 Ha
Integrated electron count	8.000 e (exactly, an 8-electron cell)
Band gap (VBM → CBM)	0.585 eV

> **Note**
>
> PBE, like all semi-local functionals, underestimates the silicon band gap (experiment ≈ 1.1 eV). The ~0.6 eV result is the expected PBE value — a good sanity check, not an error. For an accurate gap you would use a hybrid functional; just ask LatticeMind for one.

The band structure and density of states are plotted automatically:

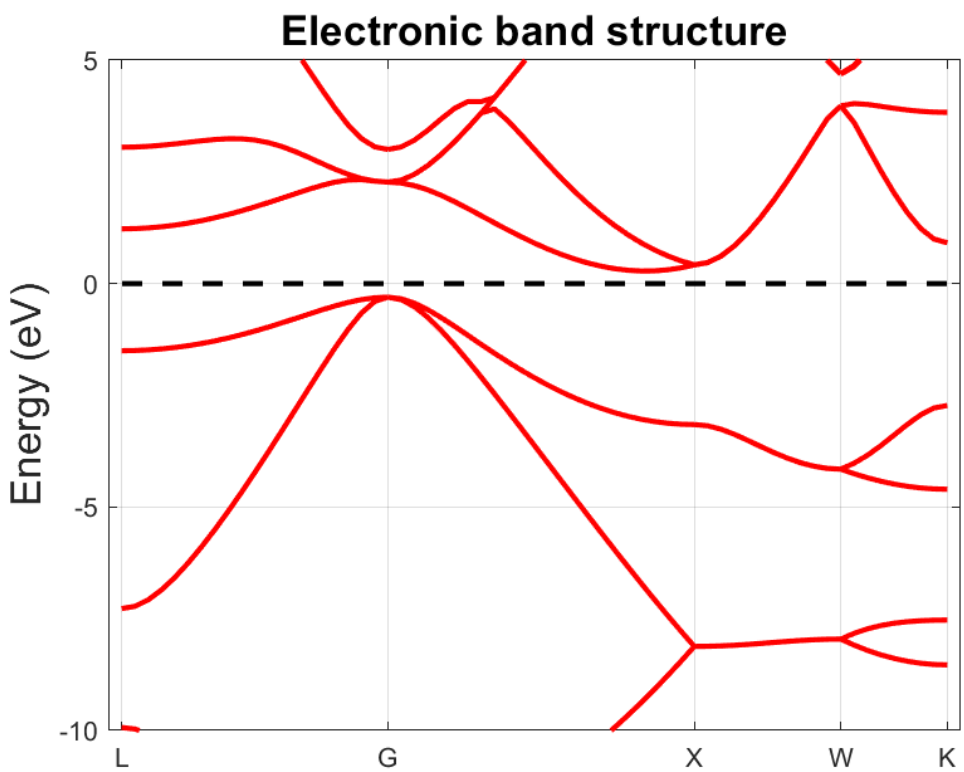

Fig. 4.1.1: Silicon band structure along the high-symmetry k-path, generated by LatticeMind from the one-line prompt.

4.1.6 Growing the project, turn by turn

You do not have to ask for everything up front. The same project could have started with just the SCF and band structure, and gained its DOS later:

```
can you add a dos calculation and plot the dos for the converged density
```

A continuation reuses the project's existing plan and whatever is already converged on disk — here, only the DOS actually runs, in well under a minute, and its plot lands beside the band structure. This is the intended way to work: each request builds on the last, completed steps are never recomputed, and a finished run never blocks its own project from continuing.

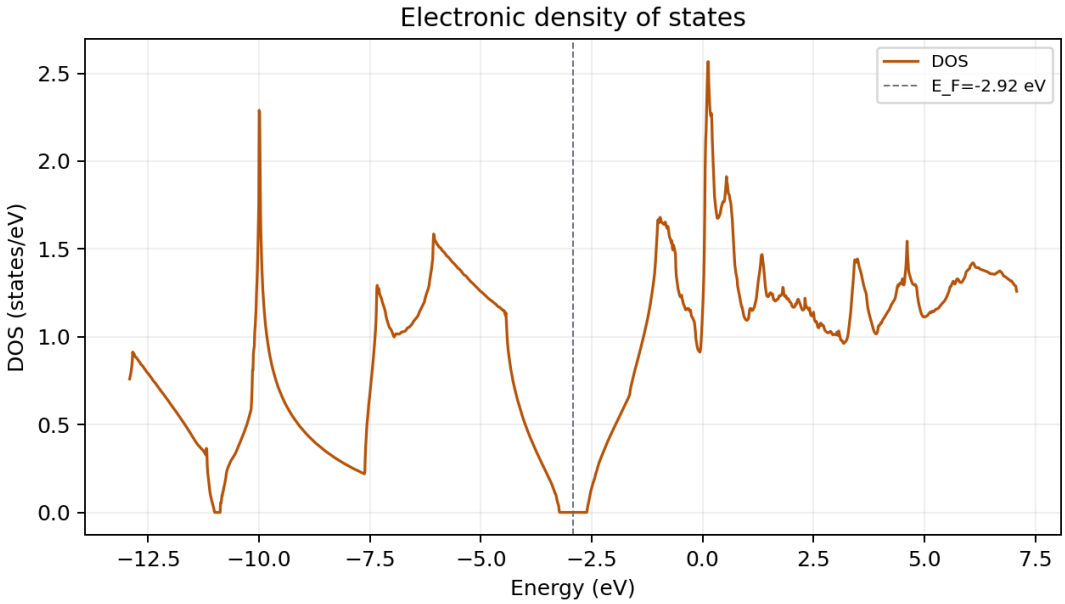

Fig. 4.1.2: Silicon density of states. The Fermi level sits in the gap, as expected for a semiconductor.

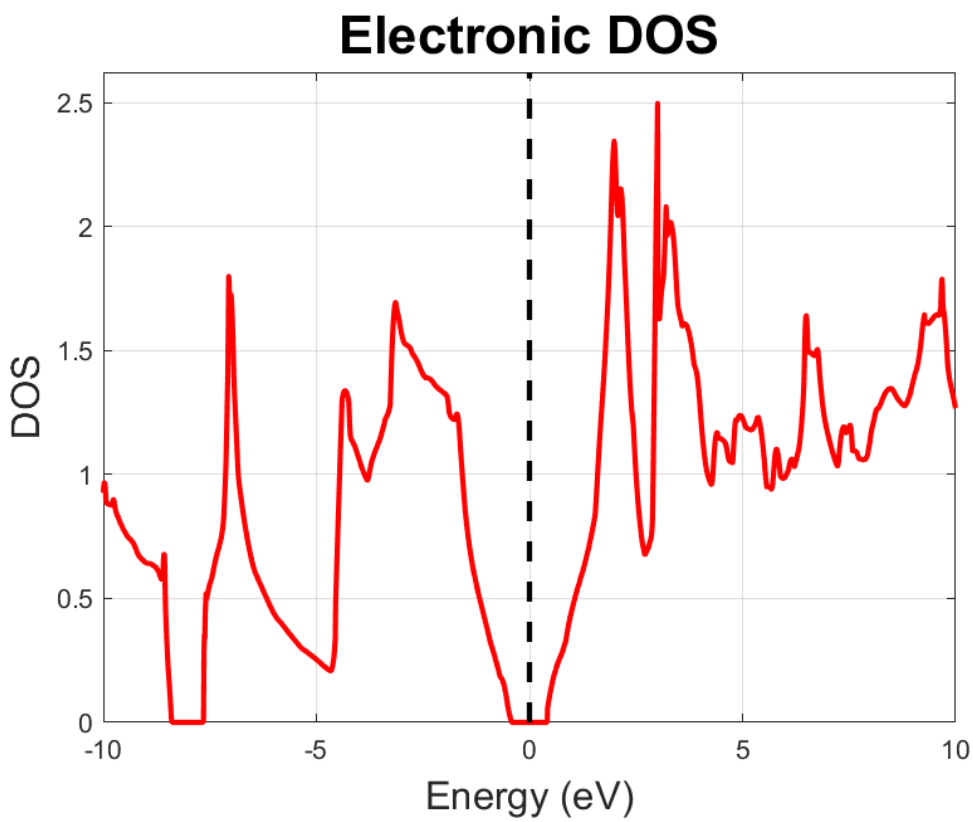

Fig. 4.1.3: The DOS added by the one-line continuation above, computed from the density the SCF step had already saved. The Fermi level sits in the gap.

4.1.7 Inspecting the project

Everything is on disk in the project folder (*Projects & the workspace*):

```
/outputs      # list results, plots, logs, and the report
/plots        # list generated plots
/analysis     # show extracted quantities (energy, gap, electron count)
```

The generated files include the RESCU decks (`inputs/scf.input`, `band.input`, `dos.input`), the raw output (`results/scf.mat/.h5` and friends), the plots, and `report.html`.

4.1.8 What you learned

- A single plain-language prompt produced a correct, multi-step DFT workflow.
- LatticeMind chose the structure, functional, k-grid, and basis for you.
- The density hand-off between SCF and its dependents was wired automatically.
- You got validated inputs, real results, plots, and a report.

Next, plot a III–V band structure in *GaAs: band structure along the FCC path*, run a k-point convergence study in *Aluminium: a k-point convergence study*, try a spin-polarized metal in *Ferromagnetic nickel: a two-step spin DOS*, or an alloy in *Building alloys & supercells (SiGe)*.

4.2 GaAs: band structure along the FCC path

Gallium arsenide is the classic III–V semiconductor, and its band structure is a two-step workflow: a self-consistent ground state, then eigenvalues along the standard high-symmetry path of the FCC Brillouin zone. This tutorial also demonstrates something subtler — how LatticeMind reports a result **honestly** when the level of theory is known to distort it.

> **What you need**
>
> The same setup as *Silicon: SCF, band structure & DOS*. Start to finish, this project runs in about three minutes on a laptop.

4.2.1 The prompt

This is one of the ready-to-run prompts shipped with `/examples`:

```
Create a two-step zinc-blende GaAs workflow: SCF with a saved density, then
a band structure along the standard FCC path.
```

4.2.2 What LatticeMind does

1. **Plans** `SCF` → `BAND-STRUCTURE`, with the band step reading the converged density through `rho.in`.
2. **Builds** the two-atom zinc-blende cell (Ga at the origin, As at ¼¼¼).
3. **Renders and validates** both decks, then **runs** them.
4. **Plots** the bands along Γ–X–W–K–Γ–L–U–W–L–K — the standard FCC path, chosen from a fixed table for the detected lattice, with each label placed at its true position along the path.

4.2.3 The results

Quantity	Value
SCF total energy	−78.4265 Ha
Integrated electron count	18.0 e
Band gap (PBE)	0.00 eV — *flagged, see below*

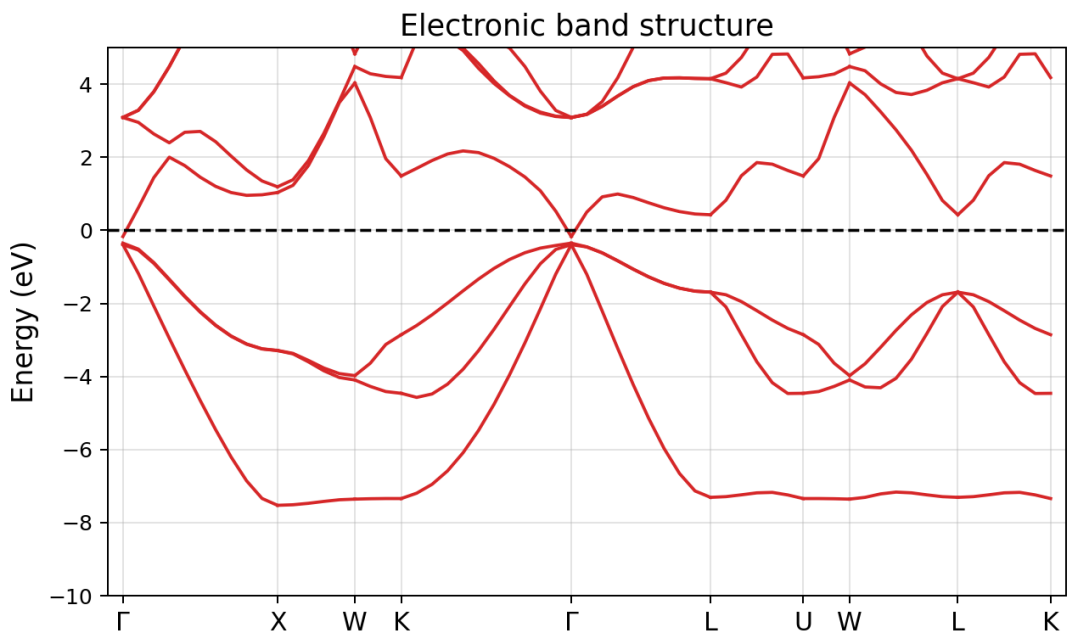

Fig. 4.2.1: Zinc-blende GaAs bands along Γ–X–W–K–Γ–L–U–W–L–K, generated from the converged SCF density. Note the unequal label spacing: each segment occupies its true share of the reciprocal-space path length.

4.2.4 An honest zero

Experimentally, GaAs has a direct gap of about 1.42 eV. Within PBE the gap **collapses to zero** — a well-known failure of semi-local functionals for small-gap semiconductors, more severe here than the ~50 % underestimate you saw for silicon.

What matters is how this is reported. LatticeMind does **not** call GaAs a metal. The analysis records a structured warning naming the collapse as an expected artifact of the functional, the report says a hybrid functional or many-body correction is needed for a quantitative gap, and the extracted `band_gap = 0.0` is presented with that caveat attached. The honest answer to "what is the PBE gap of GaAs?" *is* zero — and the report makes sure the number cannot be mistaken for a physical prediction.

4.2.5 What you learned

- The shipped `/examples` prompts are verified, copy-paste starting points.
- Band paths are deterministic: the FCC path and its labels come from the detected lattice, and the plot places every label at its exact vertex.
- A known limitation of the level of theory is reported as such — never silently, and never dressed up as a physical result.

For the k-point convergence companion study, continue to *Aluminium: a k-point convergence study*.

4.3 Aluminium: a k-point convergence study

Before trusting any production number, a DFT practitioner checks that it is converged with respect to the sampling parameters. This tutorial runs the most common such check — total energy against the k-point mesh — as a single prompt, and shows how LatticeMind turns the study's three numbers into a conclusion.

Metals like aluminium are the demanding case: their partially filled bands make the total energy converge slowly with Brillouin-zone sampling, which is exactly why this check matters.

What you need

The same setup as *Silicon: SCF, band structure & DOS*. The three SCF runs complete in about three minutes altogether.

4.3.1 The prompt

Another of the shipped `/examples` prompts:

```
Set up a k-point convergence study for fcc aluminum comparing 6x6x6, 8x8x8,
and 10x10x10 meshes at fixed geometry.
```

4.3.2 What LatticeMind does

1. **Plans three SCF instances** of the same fcc aluminium cell, identical in every respect except `kpoint.gridn`. The validator confirms the three decks really do differ only in the mesh — a sweep that accidentally collapses to one value is refused, not run.
2. **Runs** the three calculations and extracts each total energy.
3. **Recognises the sweep** in post-processing: several SCF steps whose decks differ in one sampling knob are a convergence study, so the reporter plots the energy against the configurations and leads with that figure.

4.3.3 The results

Mesh	Total energy (Ha)	ΔE to finest (meV)
6×6×6	−2.076958	−35.11
8×8×8	−2.075994	−8.89
10×10×10	−2.075667	0 (reference)

Reading the curve: refining from 6^3 to 8^3 moves the energy by ~26 meV, but the last refinement to 10^3 changes it by under 9 meV. If your target accuracy is ~10 meV, the 8×8×8 mesh is the economical choice; for tighter work you would extend the series — *"add a 12x12x12 mesh to the study"* — and LatticeMind adds the fourth point to the same curve.

The same numbers are recorded machine-readably in the project's `analysis_summary.json` under `convergence_study`, with each configuration's energy and its delta to the finest run — ready for scripts or notebooks that consume the study programmatically.

4.3.4 What you learned

- A convergence study is one prompt, and its conclusion is a figure, not three disconnected numbers.
- The sweep is validated as a sweep: three identical decks with different labels would be refused before anything runs.
- The study extends naturally — ask for another mesh and the curve grows.

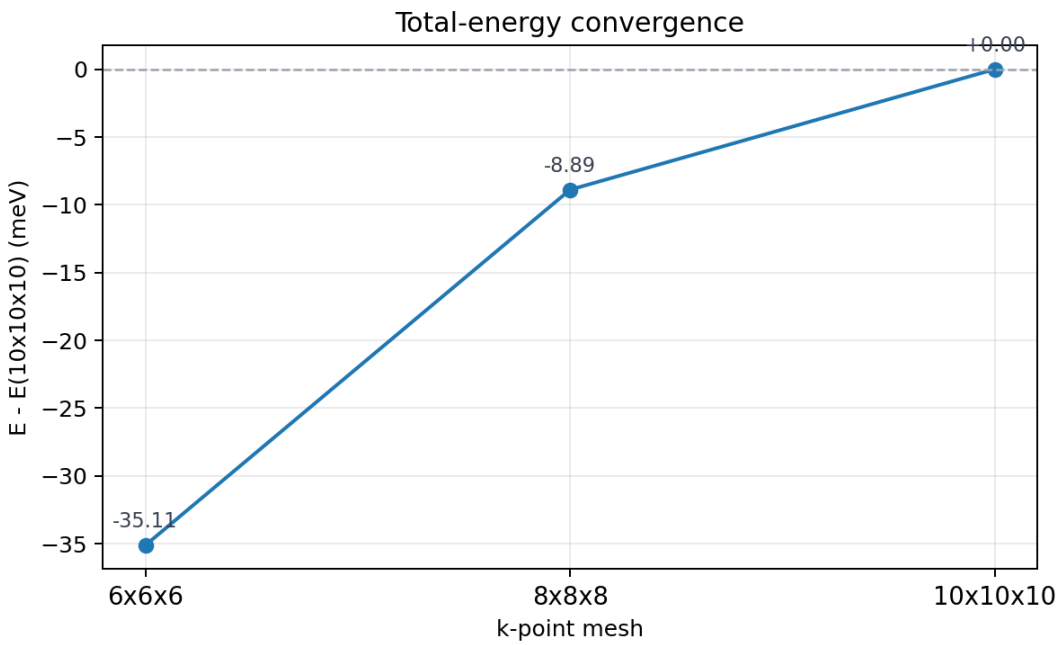

Fig. 4.3.1: Total-energy convergence of fcc Al. Each point is annotated with its distance to the finest mesh in meV; the dashed line is the 10×10×10 reference.

The same pattern applies to other sampling knobs: a real-space resolution (`domain.lowres`) sweep is recognised and plotted the same way.

4.4 Ferromagnetic nickel: a two-step spin DOS

This tutorial shows two things at once: how LatticeMind handles a **metal** with **collinear spin polarization**, and how it chains a **two-step workflow** where a DOS run reuses the converged density of a prior SCF run.

4.4.1 The prompt

```
Create a two-step workflow for fcc nickel: first an SCF calculation, then a
DOS calculation using the saved density. Use PBE and collinear spin, set an
initial magnetic moment, and keep the calculation small but physically
reasonable for a metal.
```

4.4.2 What LatticeMind does

- Builds the fcc nickel primitive cell (one atom, lattice constant 3.52 Å).
- Turns on **collinear spin** and seeds an initial magnetic moment on the nickel atom.
- Uses a metal-appropriate Brillouin-zone sampling (the tetrahedron method with Blöchl corrections) on a `12×12×12` k-grid.
- Runs the SCF, then a DOS step that reads the saved SCF density.

The spin- and metal-specific settings show up in the generated SCF deck:

```
atom.magmom = [0.6];
element(1).species = 'Ni';
functional.list = {'XC_GGA_X_PBE' 'XC_GGA_C_PBE'};
spin.type = 'collinear';
kpoint.gridn = [12 12 12];
kpoint.sampling = 'tetrahedron+blochl';
```

Notice `spin.type = 'collinear'`, the initial `atom.magmom`, and the tetrahedron sampling — all chosen by LatticeMind because you asked for a spin-polarized metal.

4.4.3 The result

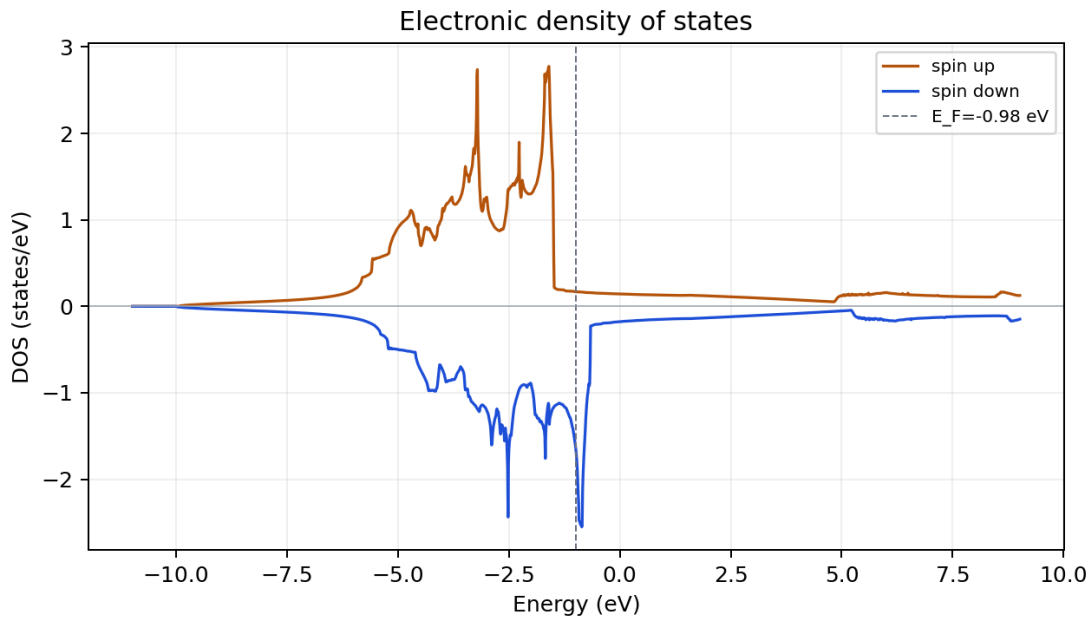

Fig. 4.4.1: Spin-resolved density of states of ferromagnetic nickel. The **spin-up** (majority) channel is plotted upward and **spin-down** (minority) downward. The exchange-split *d*-band and the imbalance of states at the Fermi level are the fingerprints of itinerant ferromagnetism.

The majority *d*-band is almost filled while the minority *d*-band still has weight at the Fermi level — the origin of nickel's magnetic moment. This is exactly the physics you expect, produced from a one-paragraph prompt.

4.4.4 What you learned

- LatticeMind recognizes metals and magnetism and adjusts sampling and spin settings accordingly.
- A two-step workflow (SCF → DOS) reuses the converged density automatically — you asked for it in words, and the hand-off was wired for you.

Next: build an alloy in *Building alloys & supercells (SiGe)*.

4.5 Building alloys & supercells (SiGe)

The previous tutorials used tiny primitive cells. This one shows how LatticeMind **builds larger and more complex structures** for you — here, a random silicon-germanium alloy supercell — straight from a plain-language description.

4.5.1 The prompt

```
Do an SCF and band structure calculation for a 32-atom SiGe alloy with a
Ge fraction of 0.5, and place the Ge atoms randomly.
```

4.5.2 What LatticeMind does

From that one sentence, LatticeMind's structure builder:

- starts from the diamond-cubic silicon lattice and constructs a **2×2×2 supercell** — 32 atoms in a cubic cell of side ≈ 10.86 Å;
- randomly assigns **half the sites to germanium** (Ge fraction 0.5, so 16 Si and 16 Ge);
- defines both species with their PBE pseudopotentials; and
- verifies the structure is physically sane (composition, site count, bond lengths) before any calculation runs.

The generated cell carries both elements, for example:

```
domain.latvec = [...
    10.86 0.0 0.0;
    0.0 10.86 0.0;
    0.0 0.0 10.86;
];
element(1).species = 'Si';
element(1).path = './pseudopotentials/Si_PBE_TM_DZP.mat';
element(2).species = 'Ge';
element(2).path = './pseudopotentials/Ge_PBE_TM_DZP.mat';
```

LatticeMind then plans the SCF and band-structure steps on this 32-atom cell, using a k-point grid appropriate for the larger cell (a supercell needs fewer k-points than a primitive cell).

4.5.3 Varying the alloy

Everything about the structure is controllable in words. Try:

```
Make it a 64-atom cell with Ge fraction 0.25.
Use an ordered SiGe arrangement instead of random.
Add 1% biaxial strain in the xy-plane.
```

LatticeMind rebuilds the structure and re-plans accordingly.

> **Tip**
>
> The same builder handles supercells, alloys, point defects, surfaces, and two-probe device geometries. If you can describe the structure, LatticeMind can usually build it — and it will tell you when a request is ambiguous and ask for the missing detail rather than guessing.

4.5.4 What you learned

- LatticeMind builds non-trivial structures — supercells and random alloys — from a description, and validates them before running.
- Structural parameters (size, composition, ordering, strain) are all adjustable through natural language.

That completes the basic tutorials. Continue to *Advanced tutorials* for remote/HPC execution and the experimental NanoDCAL backend.

4.6 Aluminium (111): a work function converged with slab thickness

This tutorial shows how LatticeMind builds a **surface slab** from a bulk crystal, reads a **work function** from the electrostatic potential, and runs **several slab thicknesses in one request** so you can see the value converge.

4.6.1 The prompt

```
Calculate the work function of the aluminium (111) surface and check how it
converges with slab thickness. Build symmetric Al(111) slabs from fcc
aluminium with lattice constant 4.05 A, using a 2x2 surface cell, 18 A of
vacuum, and 7, 8 and 9 atomic layers. For each thickness run a PBE SCF with
Gaussian smearing and a 6x6x1 k-point grid, then a potential calculation on
the saved density, and compare the three work functions.
```

4.6.2 What LatticeMind does

- Builds a symmetric Al(111) slab for each requested thickness, holding the surface cell and the vacuum fixed so thickness is the only thing that changes.
- Centres each slab in its cell, giving the same vacuum on both faces — the potential must reach a flat plateau on either side for a vacuum level to be well defined.
- Samples the surface plane densely and the vacuum direction at Γ only, and applies Gaussian smearing because aluminium is a metal.
- Runs each SCF, then a potential calculation that reads that SCF's density.
- Reads the vacuum level from the plateau of the plane-averaged electrostatic potential, subtracts the Fermi energy, and reports the work function for each thickness with the measured value alongside.

The 9-layer cell comes out as 36 atoms in a 5.73 × 5.73 × 36.71 Å cell, with 9 Å of vacuum above and below the slab:

```
domain.latvec = [...
    5.7276 0.0 0.0;
    2.8638 4.9602 0.0;
    0.0 0.0 36.7061;
];
kpoint.gridn = [6 6 1];
kpoint.sampling = 'gauss';
```

The nearest-neighbour distance in every slab is 2.864 Å and the interlayer spacing is 2.338 Å — the values fcc aluminium at a = 4.05 Å requires, so the surface is built from the bulk crystal rather than approximated.

4.6.3 Results

Layers	Atoms	Work function	Change
7	28	3.96 eV	—
8	32	4.11 eV	+0.15 eV
9	36	4.07 eV	−0.04 eV

The converged value of **4.07 eV** compares with a measured **4.24 eV** for Al(111). A few tenths of an electronvolt is the expected agreement for a semi-local functional on a simple metal.

The alternation between odd and even layer counts is physical: it is the quantum-size effect in thin metal films, and it damps as the slab thickens — from +0.15 eV between 7 and 8 layers to −0.04 eV between 8 and 9. The energy per atom tells the same story: it moves 425 meV between 7 and 8 layers and only 11 meV between 8 and 9, so the interior of the slab has become bulk-like. This is why the thickness study is worth running rather than trusting a single slab.

> **Note**
>
> The work function is read from the plateau of the plane-averaged potential far from the surface, not from an average over the cell — inside the slab the potential is dominated by the ionic cores. LatticeMind reports the plateau it used, so you can confirm the vacuum region was flat enough for the value to mean something.

4.6.4 Try next

- Add a fourth thickness to the same request to tighten the convergence.
- Ask for a wider surface cell (`3x3`) at fixed thickness to check the in-plane sampling.
- Run the same study for another close-packed metal surface and compare the trend.

CHAPTER

FIVE

ADVANCED TUTORIALS

These tutorials go beyond a single local RESCU run: offloading calculations to an HPC cluster, and using the experimental NanoDCAL backend for electronic structure and quantum transport.

5.1 Running on an HPC cluster

LatticeMind builds and validates inputs on your machine, then offloads the heavy calculation to a remote workstation or Slurm cluster and fetches the results back. This tutorial walks through the full cycle. For the settings reference, see *Remote & HPC execution*.

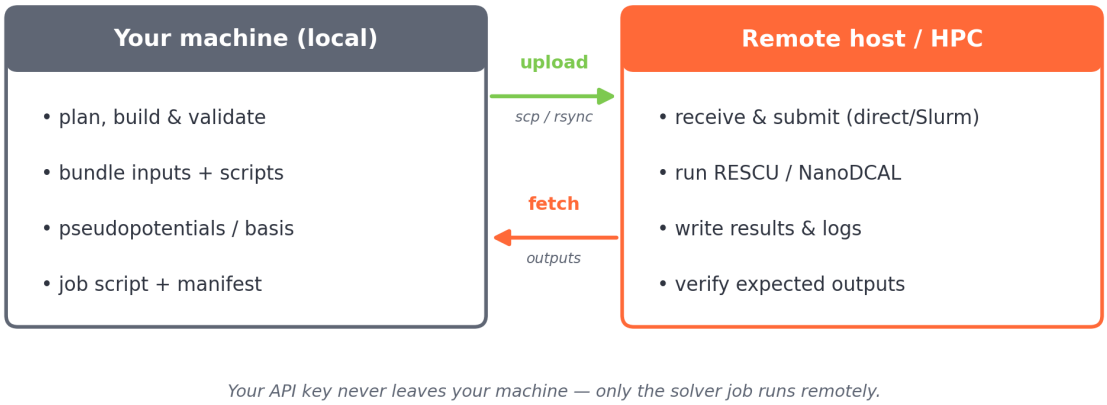

Fig. 5.1.1: The local-build / remote-run / fetch-back cycle.

5.1.1 Before you start

Confirm plain SSH already works to your cluster — LatticeMind uses your existing SSH configuration:

```
ssh my-cluster-login true
```

5.1.2 Step 1 — Define the cluster profile

Add a Slurm profile to `~/.latticemind/remote_profiles.toml`:

```
[profiles.cluster]
host = "my-cluster-login"
```

(continues on next page)

(continued from previous page)

```
scheduler = "slurm"
remote_root = "/scratch/myuser/latticemind_runs"
transfer = "rsync"
rescu_command = "rescu"
shell_preamble = "module load rescu"
account = "my-account"
partition = "short"
time = "02:00:00"
nodes = 1
ntasks = 4
cpus_per_task = 1
mem = "8G"
```

5.1.3 Step 2 — Select and test it

Inside LatticeMind:

```
/remote profile cluster
/remote setup cluster      # install SSH session-reuse settings
/remote warm cluster       # approve 2FA once, if your cluster uses it
/remote test cluster       # preflight: SSH, transfer, scheduler, solver
/compute slurm
```

`/remote test` verifies the connection, the transfer tool, the scheduler, and that the solver command exists on the remote host — catching problems before you submit.

5.1.4 Step 3 — Build, preview, and submit

Describe your calculation as usual, for example:

```
Do an SCF and band structure calculation for silicon.
```

LatticeMind plans and validates locally. Before it submits anything, preview the job:

```
/launch-preview
```

This shows the exact remote directory, upload method, the generated Slurm job script, the input files, and the requested resources. When it looks right:

```
/execute on
run it
```

LatticeMind bundles the validated job (inputs, scripts, pseudopotentials, a manifest, and the job script), uploads it, and submits it.

5.1.5 Step 4 — Track and fetch

```
/remote status         # queued / running / complete
/remote logs 80        # tail the remote log
/remote fetch          # bring verified outputs back into the project
/remote workdir        # show the remote and local bundle paths
```

Once fetched, the results land in your local project exactly as if they had run locally — post-processing, plots, and the report follow.

5.1.6 Recovering an interrupted job

If a job is interrupted (wall-time limit, node failure), you do not start over. LatticeMind preserves the inputs and save paths, so you can:

```
/remote status
/failed                # see which steps need attention
/continue <target>     # resume the interrupted or downstream steps
```

Because inputs are unchanged, RESCU restarts cleanly, and only the affected and downstream steps are recomputed.

> **Note**
>
> LatticeMind automatically applies serial-only guardrails to certain RESCU post-analysis stages (phonon band structure/DOS, optics, Raman), stripping MPI launchers for them. See *Remote & HPC execution*.

5.1.7 Safety and cost

- LatticeMind **never** sends your API key to the cluster and **never** runs the AI agent remotely — only the solver job.
- Use `/cost` before launching dense k/q grids, relaxations, DFPT, or large-cell workflows, and start with a cheap validation run.

5.2 NanoDCAL: electronic structure and transport (experimental)

> **Note**
>
> NanoDCAL support is **experimental and in active development**. The calculation types below have been tested end-to-end, but the backend is evolving. For the full support matrix, see *NanoDCAL support (experimental)*.

LatticeMind's NanoDCAL backend brings quantum-transport-oriented calculations into the same plain-language workflow. You select it simply by saying *"Using NanoDCAL, …"* in your prompt. This tutorial walks through a silicon band structure, then points to the full two-probe transport workflow.

5.2.1 Silicon band structure

```
Using NanoDCAL, run an SCF and band structure calculation for silicon.
```

LatticeMind builds the silicon crystal, stages the LCAO basis files with the project, runs the SCF, then computes the band structure along the standard high-symmetry path — and plots it referenced to the Fermi level:

5.2.2 Validate the band-gap interpretation

The plotted zero is the NanoDCAL Fermi energy. The reported silicon gap is obtained from the occupied and unoccupied bands sampled along the full L–Γ–X–W–K path. It is therefore an indirect path-sampled gap, not the energy separation between two states selected at one k-point.

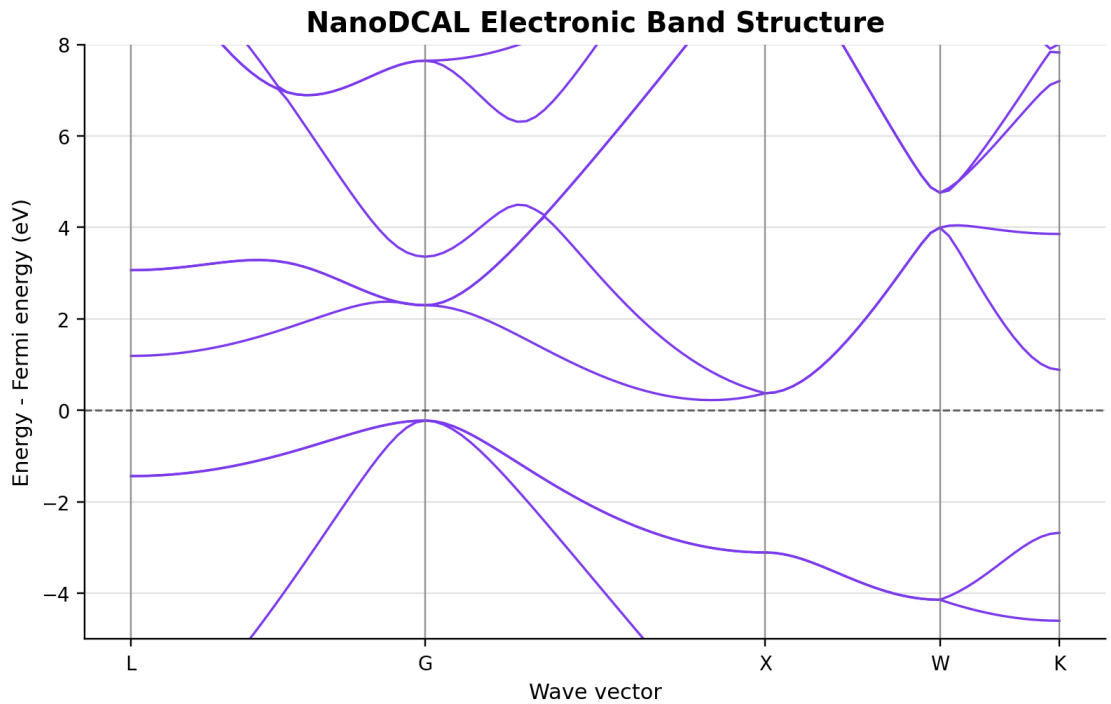

Fig. 5.2.1: Silicon band structure from NanoDCAL, along L–Γ–X–W–K with energies referenced to the Fermi level (dashed line). The indirect gap between the valence-band maximum at Γ and the conduction-band minimum near X is clearly resolved.

This distinction matters when comparing `BandStructure.mat` with `EigenStates.mat`. For example, requesting one state below and one state above the Fermi level on a `[1, 1, 1]` grid samples two Γ-point states. Their direct separation can be much larger than silicon's indirect band gap and should not be reported as the PBE fundamental gap.

Before using the value in a manuscript:

- repeat the SCF calculation with a denser k-grid and keep the plotted high-symmetry path unchanged;
- record the exchange-correlation functional and NeutralAtomDatabase basis choice, because both affect the numerical gap;
- confirm that `results/analysis_summary.json` reports the same band edges visible in the figure; and
- remember that a converged PBE gap is a method-specific result, not an experimental-gap correction.

Tip

Because the LCAO basis travels with the project (`nanodcal_basis/`), the generated project runs unchanged on another machine — locally or on a remote NanoDCAL server (see *Remote & HPC execution*). When sharing the generated project, please keep the staged NeutralAtomDatabase files within the scope of your Nanoacademic license.

5.2.3 Spin-resolved transmission channels

This prompt exercises the transverse-k and spin dimensions in `TransmissionChannel.mat`:

```
Using NanoDCAL, run SCF and transmission-channel analysis for
ferromagnetic bcc iron.
```

LatticeMind plots the channel count on the regular transverse-k mesh and keeps the spin-up and spin-down maps on the same colour scale:

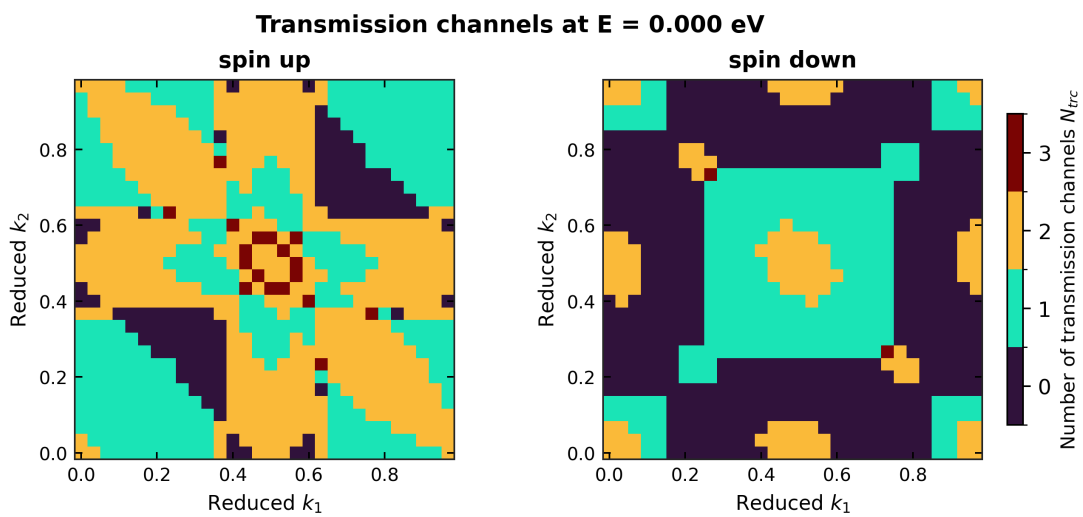

Fig. 5.2.2: NanoDCAL transmission-channel count at the Fermi level for 900 transverse k-points. Separate spin panels preserve the two-dimensional k-space structure instead of flattening it into a sample-index curve.

5.2.4 How to read and validate the channel map

Each pixel represents one transverse `(k1, k2)` point at `E - EF = 0 eV`; the colour gives the number of open transmission channels at that point. Both spin panels use the same colour limits, so their different k-space patterns can be compared directly. The map is not, by itself, a total conductance: a scalar transport value also requires the transverse-k weights and the appropriate energy or bias integration.

Before interpreting the spin contrast, check that:

- the bcc Fe SCF calculation converged to the intended magnetic state;
- both panels use the same energy, transverse mesh, and colour scale;
- the channel pattern and any integrated transport quantity are stable when the transverse mesh is refined beyond the illustrated 30 by 30 grid; and
- the lead and device calculations use compatible cells, basis data, and spin settings.

Use `figures/transmission_channels.png` in the generated HTML report and the SVG or PDF export when editing a manuscript. The matching `results/figure_manifest.json` entry records `TransmissionChannel.mat` as the source and lists the PNG, SVG, and PDF paths, dimensions, and DPI.

5.2.5 Going further: quantum transport

The same one-line pattern drives full two-probe transport calculations — LatticeMind builds the lead-SCF → device-SCF → property chain automatically:

```
Using NanoDCAL, run a transmission calculation along z for a homogeneous
aluminum two-probe device.
```

Start with the zero-bias transmission and inspect `Transmission.mat` before requesting a bias sweep. A useful follow-up is:

```
Using NanoDCAL, calculate an I-V curve along z for a homogeneous aluminum
two-probe device from -0.2 V to 0.2 V.
```

For every transport comparison, keep the device geometry, lead construction, NeutralAtomDatabase basis, transverse k-grid, energy grid, and electronic temperature fixed unless the prompt explicitly studies one of those choices.

See *NanoDCAL support (experimental)* for the complete list of supported transport, electronic-structure, phonon, and mechanical calculations.

5.2.6 Three useful follow-up checks

The following prompts exercise different NanoDCAL output families without changing the installation:

1. `Using NanoDCAL, calculate total energy, effective potential, electron density, and charge analysis for rocksalt MgO.` This produces `TotalEnergy.mat`, `EffectivePotential.mat`, `TotalElectronDensity.mat`, and `Charge.mat` after one SCF calculation.
2. `Using NanoDCAL, calculate the effective mass for GaAs.` This produces `EffectiveMass.mat` and tests the electron/hole dispersion fit and directional masses in units of m_e.
3. `Using NanoDCAL, run a conductance calculation along z for a homogeneous aluminum two-probe device.` This exercises the lead-SCF, device-SCF, and conductance chain and produces `ConductanceAndCurrent.mat`.

For each project, inspect `/launch-preview` before execution. After execution, confirm that `calculation_status.json.instances` marks every requested deck as `succeeded`, that the expected `.mat` files exist, and that `results/analysis_summary.json.warnings` contains no unexplained parser or physics warning. To test only the updated plotting path, ask LatticeMind to reprocess the existing NanoDCAL outputs without rerunning the solver.

5.2.7 What you learned

- Selecting NanoDCAL is as simple as saying so in your prompt.
- LatticeMind handles NanoDCAL's basis staging, multi-step dependencies, and device chains for you.
- Plot-capable results produce a 360 dpi PNG for `report.html` plus SVG and PDF manuscript exports under `figures/`.
- `results/analysis_summary.json` stores numeric summaries and `results/figure_manifest.json` links each figure to its NanoDCAL `.mat` source file.

CHAPTER

SIX

REFERENCE

This chapter is the complete reference for LatticeMind: every configuration setting, the full command-line and interactive command surface, the remote/HPC execution model, the solver support matrix, and the experimental NanoDCAL backend.

6.1 Settings & configuration reference

This page lists defaults and explicit overrides. LatticeMind still requires an AI provider configuration and the RESCU license described in *Licenses and solver executables*. A solver command and its licensed data are required only when that solver executes.

6.1.1 Where settings come from

LatticeMind reads configuration from several places, in order of precedence (later sources override earlier ones):

1. **Defaults** baked into LatticeMind.
2. **Config file —** `~/.latticemind/config.json` (or a file named by `LATTICEMIND_CONFIG_FILE`) and the Web UI's `~/.latticemind/dashboard_settings.json`.
3. **The** `.env` **file** in the folder you launch from.
4. **Environment variables.**
5. **Interactive commands** (`/rescu`, `/compute`, `/language`, …) and `latticemind-web` flags, for the current session.

> **Tip**
>
> The Web UI persists your choices to `~/.latticemind/dashboard_settings.json` automatically. For servers and reproducibility, prefer `.env` or real environment variables.

6.1.2 AI provider & models

Variable	Purpose
`RESCU_LLM_PROVIDER`	`openai` (default) | `claude` | `gemini` | `qwen` | `ollama`.
`OPENAI_API_KEY`	OpenAI API key.
`OPENAI_MODEL_QUICK` / `OPENAI_MODEL_THOROUGH`	Fast/default and higher-effort OpenAI models (e.g. `gpt-5.4-mini` / `gpt-5.4`).
`OPENAI_ORG_ID` / `OPENAI_PROJECT_ID`	Only if your account uses org/project scoping.
`CLAUDE_API_KEY` (or `ANTHROPIC_API_KEY`)	Anthropic Claude key.
`CLAUDE_MODEL_QUICK` / `CLAUDE_MODEL_THOROUGH`	Claude models (e.g. `claude-sonnet-4-6` / a stronger Claude model).
`GEMINI_API_KEY` (or `GOOGLE_API_KEY`)	Google Gemini key.
`GEMINI_MODEL_QUICK` / `GEMINI_MODEL_THOROUGH`	Gemini models (e.g. `gemini-3-flash-preview` / `gemini-3.1-pro-preview`).
`DASHSCOPE_API_KEY`	Qwen (DashScope) key.
`DASHSCOPE_MODEL_QUICK` / `DASHSCOPE_MODEL_THOROUGH`	Qwen models (e.g. `qwen3-max`).
`OLLAMA_BASE_URL`	Local Ollama endpoint (default `http://127.0.0.1:11434`).
`OLLAMA_MODEL_QUICK` / `OLLAMA_MODEL_THOROUGH`	Local Ollama models (e.g. `qwen3:8b`).

6.1.3 Reports & output location

Variable	Purpose
`LATTICEMIND_ARTIFACTS_DIR`	Root folder for projects and outputs (default `./latticemind_outputs`). Point at persistent storage on shared machines and cloud.
`RESCU_REPORT_FORMAT`	`html` (default) or `off` to disable report generation.
`LATTICEMIND_REPORT_LANGUAGE`	`auto` (default) | `en` | `fr` | `zh`. `auto` follows the prompt language.

6.1.4 Release update notice

Variable or file	Purpose
`LATTICEMIND_UPDATE_CHECK`	`1` (default) checks the public Nanoacademic package index during interactive CLI and Web startup; `0` disables the check.
`~/.latticemind/update_check.json`	Atomic cache written by `version_check.py`. A successful index result is reused for 24 hours; an unavailable-index result is reused for one hour. When `LATTICEMIND_CONFIG_FILE` names another config file, this cache is written beside that file.

6.1.5 Web UI

Variable	Purpose
`LATTICEMIND_WEB_HOST`	Bind host (default `127.0.0.1`). The dashboard has no authentication of its own, so it is reached through the platform's proxy rather than published.
`LATTICEMIND_WEB_PORT`	Bind port (default `7865`).
`LATTICEMIND_WEB_OPEN_BROWSER`	`0` to never open a local browser.
`LATTICEMIND_WEB_BASE_PATH`	Explicit proxy prefix (e.g. `/proxy/7865`) for reverse proxies.
`LATTICEMIND_CLOUD_MODE`	`1` enables cloud-friendly defaults (no browser, proxy-aware base path).
`LATTICEMIND_UI_MODE`	Force an interface, e.g. `cli` for headless servers.

6.1.6 Solver commands, license & pseudopotentials

LatticeMind stores one launch command per solver. The Web UI persists these to `~/.latticemind/dashboard_settings.json` under `solver_commands`; environment variables override them.

Variable	Purpose
`LATTICEMIND_RESCU_CMD` (or `RESCU_CMD` / `RESCU_EXECUTABLE`)	RESCU launch command (default: `rescu` on `PATH`).
`LATTICEMIND_NANODCAL_CMD`	NanoDCAL launch command.
`LATTICEMIND_NANODCAL_DATA_PATH` (or `NANODCAL_DATA_PATH`)	Extracted NeutralAtomDatabase root. LatticeMind searches it recursively for the `*.nad` or `*.mat` files required by the requested elements.
`LATTICEMIND_VASP_CMD`	VASP launch command (e.g. `vasp_std`).
`LATTICEMIND_LICENSE_FILE`	Exact RESCU license file used at LatticeMind startup. If unset, LatticeMind expects exactly one `*.lic` or `*.license` file under `~/.nanoacademic/RESCU`. This startup check also applies when NanoDCAL is selected.
`LATTICEMIND_PSEUDO_ROOT` (or `RESCU_PSEUDO`)	Pseudopotential database folder (containing `nano_tm_lda/` and `nano_tm_pbe/`). The wheel ships a runtime bundle, so this is optional.
`VASP_PP_PATH`	VASP POTCAR library directory.
`LATTICEMIND_RESCU_SERIAL_ONLY_CALC`	Comma-separated calc types forced to run serially (default covers `dfpt-phonon-bs`, `dfpt-phonon-dos`, `dfpt-optic`, `dfpt-raman`).

The corresponding `dashboard_settings.json` shape:

```
{
  "solver": "rescu",
  "solver_commands": {
    "rescu": "/opt/esapp/rescu",
    "vasp": "vasp_std",
    "nanodcal": "/opt/nanodcal/bin/nanodcal"
  }
}
```

6.1.7 Parallel execution

Control local and remote parallel launches (also set from the Web UI Settings panel), and mapped to Slurm resources for remote jobs.

Variable	Purpose
`LATTICEMIND_PARALLEL_ENABLED`	`1` to enable parallel execution.
`LATTICEMIND_PARALLEL_LAUNCHER`	`none` | `mpirun` | `mpiexec` | `srun` | `custom` | `profile`.
`LATTICEMIND_PARALLEL_CUSTOM_LAUNCHER`	Custom launcher prefix, e.g. `jsrun -n 4`.
`LATTICEMIND_MPI_RANKS`	MPI rank count (also Slurm `--ntasks`).
`LATTICEMIND_THREADS_PER_RANK`	OpenMP/BLAS threads per rank (also Slurm `--cpus-per-task`).
`LATTICEMIND_VASP_KPAR` / `LATTICEMIND_VASP_NCORE` / `LATTICEMIND_VASP_NPAR`	Optional VASP INCAR parallelization overrides.

6.1.8 Remote / HPC execution

Remote targets are defined in `~/.latticemind/remote_profiles.toml` (see *Step 2 — Describe the target*). These variables select and describe the active target:

Variable	Purpose
`LATTICEMIND_EXECUTION_BACKEND`	`local` | `remote` | `slurm`.
`LATTICEMIND_REMOTE_PROFILE`	Default remote profile name.
`LATTICEMIND_REMOTE_PROFILE_<SOLVER>`	Per-solver remote profile override (e.g. `LATTICEMIND_REMOTE_PROFILE_NANODCAL`).
`LATTICEMIND_REMOTE_SCHEDULER`	`direct` | `slurm`.
`LATTICEMIND_RESCU_PLOT`	Set `0` to disable RESCU's built-in plotter (`rescu -p`) for this session; Python post-processing still generates every figure. Use on machines without a MATLAB Java runtime (JVM). `/rescu-plotting off` saves the same choice permanently with your settings.
`LATTICEMIND_RUNTIME_PROFILE`	Named runtime/deployment profile.
`LATTICEMIND_SYSTEM_DESCRIPTION` / `LATTICEMIND_ENVIRONMENT_NOTES`	Advisory deployment context used when planning run scripts and reports.
`LATTICEMIND_OPERATOR_PROMPT_FILE`	File of advisory operator instructions (deployment context only).

6.1.9 Documentation retrieval

The RESCU documentation is **built in and works offline** — no configuration required. Optional overrides:

Variable	Purpose
`RESCU_DOCS_DIR`	Read the RESCU docs from a different local folder.
`RESCU_RAG_BACKEND`	Set to `openai` to retrieve from an OpenAI vector store instead of the built-in docs.
`OPENAI_VECTOR_STORE_ID`	Your vector store ID (`vs_…`) when `RESCU_RAG_BACKEND=openai`.

6.1.10 Structure sourcing

Variable	Purpose
`MATERIALS_PROJECT_API_KEY`	Enables seeding structures from the Materials Project database.

6.1.11 A complete `.env` example

```
# --- AI provider ---
RESCU_LLM_PROVIDER=openai
OPENAI_API_KEY=your_key_here
OPENAI_MODEL_QUICK=gpt-5.4-mini
OPENAI_MODEL_THOROUGH=gpt-5.4

# --- Output & reports ---
LATTICEMIND_ARTIFACTS_DIR=/data/latticemind_outputs
RESCU_REPORT_FORMAT=html

# --- RESCU solver ---
LATTICEMIND_RESCU_CMD=/opt/esapp/rescu
LATTICEMIND_LICENSE_FILE=/opt/licenses/rescu.lic

# --- NanoDCAL solver (when selected) ---
LATTICEMIND_NANODCAL_CMD=/opt/nanodcal/bin/nanodcal
LATTICEMIND_NANODCAL_DATA_PATH=/opt/nanodcal/NeutralAtomDatabase

# --- Optional: Materials Project structure sourcing ---
MATERIALS_PROJECT_API_KEY=your_key_here
```

6.2 Command-line & interactive commands

This page is the complete reference for LatticeMind's command-line entry points, their flags, and every interactive (slash) command. For a friendlier, task-first introduction see *Interactive commands*.

6.2.1 Console commands

The wheel installs these commands on your `PATH`:

Command	Purpose
`latticemind`	Start the assistant. Prompts you to choose the Dashboard, Web UI, or plain CLI (unless `LATTICEMIND_UI_MODE` forces one).
`latticemind-web`	Launch the browser-based Web UI directly.

6.2.2 Shell discovery and one-shot commands

The `latticemind` command also provides non-interactive commands for shell users and general agents. Add `--json` or `--format json` anywhere in a command to receive the versioned JSON response documented in *Using LatticeMind from Codex, Claude, and shell agents*.

Command	Purpose
`latticemind --version`	Print the installed LatticeMind version without a license/provider check.
`latticemind capabilities --json`	Return solvers, shell commands, slash commands, exit codes, and execution safeguards.
`latticemind commands [filter] --json`	Return the full interactive command catalog or one category/search match.
`latticemind docs <question>`	Print one concise answer after inferring LatticeMind, RESCU, NanoDCAL, or all corpora from the question.
`latticemind docs --list`	List packaged LatticeMind manual source paths.
`latticemind docs --source <path>`	Read one exact packaged LatticeMind manual `.rst` page.
`latticemind protocol --json`	Return the versioned runtime protocol manifest.
`latticemind doctor --json`	Run read-only local and remote-configuration checks.
`latticemind project inspect [path] --json`	Read project workflow, calculation status, artifact status, and indexed files.
`latticemind project ask <question> --project <path> --json`	Answer from existing project files and matching LatticeMind manual sections without writing project metadata.
`latticemind run <prompt> --json`	Run one planning/validation turn; solver execution is disabled by default.
`latticemind run <prompt> --execute --approve-execution --json`	Run one turn with explicit permission for the validated solver launch.
`latticemind run <prompt> --execute --approve-execution --detach`	Release the approved calculations and return immediately. See *Releasing a long calculation*.

See *Using LatticeMind from Codex, Claude, and shell agents* for the recommended Codex/Claude command sequence, JSON fields, exit codes, project continuation, and license/provider setup.

6.2.3 Releasing a long calculation

A calculation does not need LatticeMind to stay running. Adding `--detach` releases the approved calculations to run independently — as a Slurm job, an `nohup`-ed SSH bundle, or a detached local supervisor — and returns immediately:

```
latticemind run "scf + bs + dos for Si: mp-149" \
    --execute --approve-execution --detach
```

You can then close the terminal, log out, or shut down the client machine if the work is running remotely. Pick the project up later with:

```
latticemind run --project <project-dir> "continue"
```

LatticeMind reattaches to the released work, collects whatever finished while it was away, and carries on from there. Inside the assistant, `/status` reports what is running and what has completed, and `/stale` reports steps whose inputs changed and therefore need recomputing.

Released work is tracked through an **execution lease** recorded inside the project. `/lease status` shows the lease and whether it still blocks a new launch; if a launch ever reports that it is waiting on a lease you know is finished, `/lease clear` removes it and the next run proceeds normally. Finished foreground runs leave no lease behind, and a stale lease left by an interrupted session heals automatically on the project's next run.

This is the recommended way to run anything long — geometry relaxations, phonon dispersions, or large supercells — and the only practical way to use a batch scheduler, where the job may not start until long after you submit it.

Solver documentation searches return excerpts only. `--source` cannot read an exact RESCU or NanoDCAL page. The wheel contains the sanitized RESCU text corpus and the hand-reviewed NanoDCAL keyword/calculation-name schema; it does not contain the raw mirrors from `solver_docs/rst_manual` or `solver_docs/nanodcal`.

`docs` uses `--corpus auto` by default. Use `--corpus <name>` to override the inferred corpus, `--json` for a compact machine-readable response, and `--verbose` for full fields on the returned excerpts, including scores, distribution names, and corpus counts. `--limit N` changes the maximum number of returned excerpts. These commands also bypass the interactive banner and interface chooser when an older installation still maps `latticemind` to `runner.py`.

6.2.4 `latticemind-web` options

Flag	Meaning
`--host <addr>`	Bind host. Default: `LATTICEMIND_WEB_HOST` or `127.0.0.1`; cloud mode keeps `127.0.0.1`; notebook platforms proxy to it.
`--port <n>`	Bind port. Default: `LATTICEMIND_WEB_PORT` or `7865`.
`--no-open`	Do not attempt to open a local browser (use on servers/cloud).
`--open-browser`	Force opening a local browser.

Example:

```
latticemind-web --port 7865 --no-open
```

6.2.5 Remote utility subcommands

Before opening the interactive app, you can manage remote SSH sessions directly from the shell. These are handled by `latticemind` at startup:

```
latticemind remote setup <profile>    # install SSH session-reuse settings
latticemind remote warm <profile>     # open one approved SSH session (2FA once)
latticemind remote status <profile>   # check recorded remote job status
latticemind remote close <profile>    # close the saved SSH session early
```

See *Remote & HPC execution* for the full remote/HPC workflow.

6.2.6 Interactive command palette

Inside any interface, type a command beginning with `/`. Type `/commands` (alias `/help`) to browse them in-app. The full palette:

Help

Command	Description	Example
`/commands`	Show the command palette (alias `/help`).	`/commands`
`/help <topic>`	Explain a command or topic.	`/help execution`
`/why`	Explain the current pause, blocker, or suggested action (alias `?`).	`/why`
`/tips`	Show context-aware next actions.	`/tips`
`/examples`	Show short example prompts.	`/examples`

Project

Command	Description	Example
`/new`	Start a fresh managed project.	`/new`
`/projects [limit]`	List recent indexed projects.	`/projects 8`
`/resume <project>`	Resume a project by id, prefix, or path.	`/resume last`
`/fork <project>`	Fork a project before experimenting.	`/fork last`
`/attach .`	Adopt the current folder as the active project.	`/attach .`
`/attach-folder <path>`	Adopt an existing folder without copying it.	`/attach-folder .`
`/attach <file...>`	Attach documents/files to the active or next project.	`/attach notes.pdf`
`/workspace`	Inspect the current folder before choosing new/resume/attach.	`/workspace`
`/title <name>`	Set a readable display title (alias `/rename`).	`/title Silicon DFPT`
`/whereami`	Show active project, mode, backend, settings (alias `/project`).	`/whereami`
`/status`	Show workflow and calculation status.	`/status`

Files

Command	Description	Example
`/tree`	Show key project folders and artifacts.	`/tree`
`/inputs`	List known solver input files.	`/inputs`
`/outputs`	List known results, plots, logs, and reports.	`/outputs`

Workflow & validation

Command	Description	Example
`/failed`	List failed calculation instances and suggested reruns.	`/failed`
`/stale`	List stale calculation instances.	`/stale`
`/continue <target>`	Continue a failed/stale workflow target.	`/continue dfpt_phonon_bs`
`/rerun <target>`	Force a target to rerun.	`/rerun band_structure`
`/validator <on|off>`	Enable or skip validation.	`/validator on`
`/readiness`	Show launch probability and gate findings.	`/readiness`
`/contract`	Show workflow-contract summary.	`/contract`
`/warnings`	Show validation/readiness warnings.	`/warnings`

Execution

Command	Description	Example
`/execute <on|off>`	Enable or disable solver execution.	`/execute on`
`/autopilot <on|off>`	Auto-approve execution gates when ready.	`/autopilot off`
`/rescu <command>`	Set the RESCU executable or launch command.	`/rescu /opt/esapp/rescu`
`/compute <local|remote|slurm>`	Choose local, SSH, or Slurm execution.	`/compute slurm`
`/launch-preview`	Preview what would run before launch (alias `/dry-run`).	`/launch-preview`
`/cost`	Estimate resource risk and staged-run recommendations.	`/cost`
`/rescu-plotting [on|off]`	Enable or disable RESCU's built-in plotter (`rescu -p`); the choice is saved with your settings. Disable it on machines without a MATLAB Java runtime (JVM) — Python post-processing still generates every figure.	`/rescu-plotting off`
`/lease status`	Show the recorded execution lease and whether it blocks the next launch (alias `/lease`).	`/lease status`
`/lease clear`	Clear a settled or stale execution lease so the next run launches normally. Refuses to clear a lease whose wave is still running unless given `force`.	`/lease clear`

Remote / HPC

Command	Description	Example
`/remote wizard`	Create or update a remote profile with guided prompts.	`/remote wizard`
`/remote profiles`	List remote profiles.	`/remote profiles`
`/remote profile <name>`	Select a remote profile.	`/remote profile cluster`
`/remote setup [profile]`	Install SSH session-reuse settings.	`/remote setup cluster`
`/remote warm [profile]`	Open a reusable SSH session for 2FA.	`/remote warm cluster`
`/remote test [profile]`	Run SSH/transfer/scheduler preflight.	`/remote test cluster`
`/remote doctor`	Explain remote readiness and next checks.	`/remote doctor`
`/remote status`	Check recorded remote job status.	`/remote status`
`/remote fetch`	Fetch recorded remote job outputs.	`/remote fetch`
`/remote logs [lines]`	Tail latest remote execution logs.	`/remote logs 80`
`/remote workdir`	Show remote work directory and local bundle paths.	`/remote workdir`
`/remote cancel`	Cancel the recorded remote job.	`/remote cancel`

Post-processing

Command	Description	Example
`/postprocess`	Run or preview post-processing scripts.	`/postprocess`
`/plots`	List generated plots.	`/plots`
`/analysis`	Show the analysis summary.	`/analysis`

Runtime, cloud & session

Command	Description	Example
`/provider [name]`	Show the active AI model provider and models, or switch to another (alias `/llm`).	`/provider`
`/model [id|off]`	Show the active model, pin an exact model id, or clear the pin.	`/model`
`/doctor`	Read-only environment diagnostics (CLI, web, RESCU, remote).	`/doctor`
`/runtime`	Inspect or persist runtime/deployment context.	`/runtime`
`/language <auto|en|fr|zh>`	Choose the final report language (alias `/report-language`).	`/language auto`
`/qbraid`	Show the qBraid Web UI and proxy quickstart.	`/qbraid`
`/cloud status`	Show cloud/headless web settings.	`/cloud status`
`/web`	Launch the Web UI.	`/web`
`/exit`	Exit LatticeMind.	`/exit`

6.2.7 Environment variables

Every setting also has an environment variable. See *Settings & configuration reference* for the complete list.

6.3 Remote & HPC execution

LatticeMind builds inputs locally and can offload the actual calculation to a remote workstation or a Slurm HPC cluster. It sends a **self-contained job bundle** — inputs, scripts, pseudopotentials (and, for NanoDCAL, the basis files), a manifest, and a generated job script — runs it over SSH, and fetches the verified results back.

> **Important**
>
> LatticeMind **never sends your API key** and **never runs the AI agent remotely**. Only the solver job is executed on the remote host. It also never stores your password, private key, or 2FA code.

6.3.1 Overview

6.3.2 Step 1 — Confirm plain SSH works

Before involving LatticeMind, make sure ordinary SSH already connects to your machine or cluster:

```
ssh my-profile true
```

LatticeMind drives `ssh`, `scp`/`rsync`, and Slurm commands through your normal SSH configuration. Fix any SSH issues here first.

6.3.3 Step 2 — Describe the target

Define one or more profiles in `~/.latticemind/remote_profiles.toml`.

Direct SSH workstation:

```
[profiles.office]
host = "my-office-computer"
scheduler = "direct"
remote_root = "/home/myuser/latticemind_runs"
```

(continues on next page)

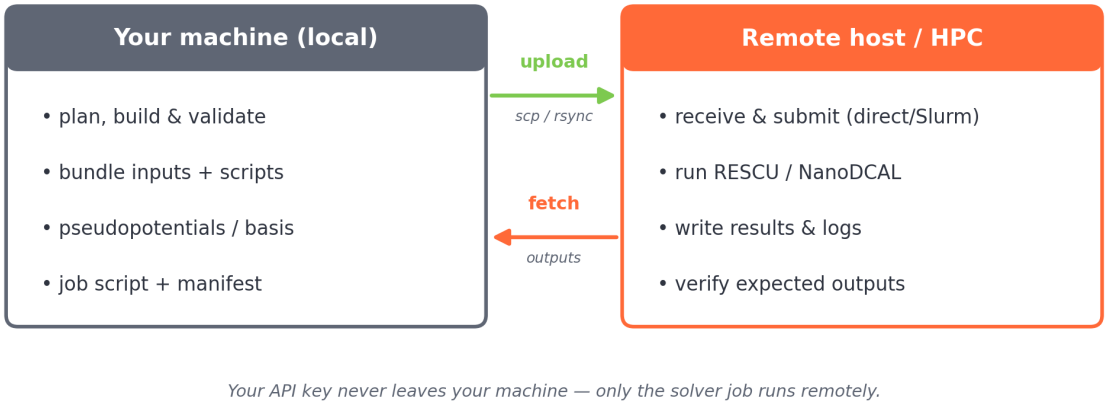

Fig. 6.3.1: Remote execution: LatticeMind bundles the validated job locally, uploads it, submits it (directly or via Slurm), then fetches the verified outputs back into the project.

(continued from previous page)

```
transfer = "scp"
rescu_command = "rescu"
shell_preamble = ""
```

Slurm cluster:

```
[profiles.cluster]
host = "my-cluster-login"
scheduler = "slurm"
remote_root = "/scratch/myuser/latticemind_runs"
transfer = "rsync"
rescu_command = "rescu"
shell_preamble = "module load rescu"
account = "my-account"
partition = "short"
time = "02:00:00"
nodes = 1
ntasks = 4
cpus_per_task = 1
mem = "8G"
```

Table 6.3.1: Remote profile fields

Field	Meaning
`host`	SSH host (as in your `~/.ssh/config`) to connect to.
`scheduler`	`direct` for a plain SSH workstation, `slurm` for an HPC cluster.
`remote_root`	Base directory on the remote host where job bundles are staged.
`transfer`	`scp` or `rsync` for uploading and fetching.
`rescu_command`	The RESCU launch command on the remote host. For NanoDCAL use `nanodcal_command` instead.
`shell_preamble`	Shell lines run before the solver, e.g. `module load rescu`.
`account`, `partition`, `time`, `nodes`, `ntasks`, `cpus_per_task`, `mem`	Slurm resource request for `scheduler = "slurm"`.

6.3.4 Step 3 — Select and test the profile

From inside LatticeMind:

```
/remote profiles            # list profiles
/remote profile office      # select one
/remote setup office        # add reusable-SSH settings (helps with 2FA)
/remote warm office         # open one approved SSH session (approve 2FA once)
/remote test office         # check SSH, solver, and scheduler
/compute remote             # or: /compute slurm
/execute on
```

A convenient shortcut runs the whole selection interactively:

```
/remote wizard
```

> **Tip**
>
> If your machine uses 2FA, `/remote warm` lets you approve it **once**; LatticeMind reuses that single approved SSH session for upload, submit, status, logs, and fetch.

You can also run remote commands from the shell before opening the app:

```
latticemind remote setup office
latticemind remote warm office
latticemind remote status office
latticemind remote close office   # close the saved SSH session early
```

6.3.5 Step 4 — Launch and track jobs

Before any remote run, LatticeMind shows the exact target, remote directory, upload method, job script, inputs, and Slurm resources for you to review. After submitting, track the job for the active project:

```
/remote status
/remote logs 80
/remote fetch
/remote fetch latest
```

(continues on next page)

(continued from previous page)

```
/remote workdir
/remote cancel
/remote doctor      # remote/HPC readiness checks
```

6.3.6 Serial-only guardrails

Some RESCU post-analysis stages must run serially. LatticeMind parses each input before launch and, for guarded stages such as `dfpt-phonon-bs`, `dfpt-phonon-dos`, `dfpt-optic`, and `dfpt-raman`, removes MPI launchers and unsafe parallel flags. For Slurm bundles containing only guarded serial stages, it also submits with one node, one task, and one CPU per task. Override the guarded list with `LATTICEMIND_RESCU_SERIAL_ONLY_CALC_TYPES` only if your RESCU build supports those stages in parallel.

6.3.7 Runtime profiles (deployment context)

Remote *profiles* describe how jobs launch. **Runtime profiles** describe the broader shell/cloud environment and advisory operator instructions LatticeMind should consider when planning run scripts and reports. Set them interactively:

```
/runtime
/runtime cloud on
/runtime system Linux container with RESCU 3.x under /opt/esapp
/runtime note Slurm jobs must use /scratch/myuser and module load rescu
/runtime prompt Use the shared scratch filesystem for generated job files.
/runtime file /home/myuser/latticemind_operator_prompt.txt
/runtime save
```

Or via environment variables in cloud deployments:

```
export LATTICEMIND_CLOUD_MODE=1
export LATTICEMIND_RUNTIME_PROFILE=cluster
export LATTICEMIND_SYSTEM_DESCRIPTION="Linux container with RESCU under /opt/esapp"
export LATTICEMIND_ENVIRONMENT_NOTES="Use /scratch/myuser; module load rescu before␣
↪jobs"
export LATTICEMIND_OPERATOR_PROMPT_FILE=/home/myuser/latticemind_operator_prompt.txt
export LATTICEMIND_EXECUTION_BACKEND=slurm
export LATTICEMIND_REMOTE_PROFILE=cluster
export LATTICEMIND_REMOTE_SCHEDULER=slurm
```

Or persist them in `~/.latticemind/config.json` (or a file named by `LATTICEMIND_CONFIG_FILE`):

```
{
  "runtime_profile": {
    "runtime_profile_name": "cluster",
    "cloud_mode": true,
    "system_description": "Linux container with RESCU installed under /opt/esapp",
    "environment_notes": "Use /scratch/myuser; module load rescu before jobs",
    "operator_instructions": "Use the shared scratch filesystem for generated job␣
↪files.",
    "execution_backend": "slurm",
    "remote_profile": "cluster",
    "remote_scheduler": "slurm",
    "rescu_command": "/opt/esapp/rescu"
  }
}
```

Note

Operator instructions are **advisory deployment context**. They do not enable or disable solver execution. Use `/execute on` or `/execute off` for interactive work; agent commands require both `--execute` and `--approve-execution` before a solver may launch.

See also

A full worked example — recovering an interrupted silicon DFPT phonon run on a remote cluster — is in *Running on an HPC cluster*.

6.4 Solvers & calculation types

LatticeMind drives three solver backends. A single project is bound to **one** solver — LatticeMind does not mix solver vocabularies within a project. The solver is inferred from your prompt (for example *"Using NanoDCAL, …"*) or set explicitly.

Solver	Status	Summary
RESCU	**Primary, stable**	Real-space / atomic-orbital Kohn–Sham DFT. The default and most thoroughly tested backend.
NanoDCAL	**Experimental**	LCAO quantum-transport solver. Two-probe devices, effective mass, complex bands, phonons, and ground-state properties. See *NanoDCAL support (experimental)*.
VASP	**Early, experimental**	Plane-wave DFT for multi-stage workflows.

6.4.1 RESCU calculation types

LatticeMind supports the following RESCU calculations, and chains them automatically (for example SCF → DOS, or SCF → band structure):

Calculation	What you get
Self-consistent field (SCF)	Ground-state electron density and total energy; the foundation for most other steps.
Band structure	Electronic bands along a k-point path, with automatic high-symmetry path selection and band-gap extraction.
Density of states (DOS)	Total DOS, and projected/local DOS (PDOS/LDOS) via atomic orbitals.
Structural relaxation	Relaxed atomic positions (and cell), via steepest-descent or non-linear conjugate-gradient. The report gives the relaxed lattice parameters, the converged energy, and the force norm across iterations, so you can see whether the geometry actually settled.
DFT+U	Hubbard-corrected DFT for correlated systems (e.g. transition-metal oxides), with a curated U table.
DFPT phonons	Phonon band structures and DOS, and dielectric/IR/Raman response via density functional perturbation theory.
Magnetism	Spin-polarized (collinear) and non-collinear calculations, including spin-orbit coupling. A spin-polarized run reports the converged magnetic moment, total and per atom.
Alloys & defects	Deterministic alloy supercells and point-defect structures, including defect formation-energy workflows.
Surfaces & work functions	Symmetric slabs built from the parent crystal, with the surface cell, layer count and vacuum you ask for. The work function is read from the plane-averaged electrostatic potential and reported alongside the measured value for that surface. Ask for several slab thicknesses in one request to see how the work function converges with thickness.

Note

You never write RESCU keywords yourself. LatticeMind selects the calculation type, parameters, and hand-offs, and renders the input decks. The generated `inputs/*.input` files are standard RESCU decks you can inspect, keep, and re-run.

6.4.2 How the solver is chosen

- **From your prompt.** *"Using NanoDCAL, run a transmission calculation…"* binds the project to NanoDCAL; an unqualified DFT request uses RESCU by default.
- **Explicitly.** Set the active solver for a project through the interface, or configure the default (see *Settings & configuration reference*).

6.4.3 Choosing between solvers

If you want to…	Use
Electronic structure, phonons, magnetism, defects of a bulk crystal	RESCU
Quantum transport through a two-probe device (transmission, I–V, conductance), effective mass, complex bands	NanoDCAL
A plane-wave cross-check or an existing VASP-based multi-stage workflow	VASP

See also

- RESCU tutorials: *Basic tutorials*.
- NanoDCAL details: *NanoDCAL support (experimental)*.

6.5 NanoDCAL support (experimental)

Note

NanoDCAL support is **experimental and in active development**. Every calculation type listed here has been run end-to-end on real hardware during internal testing — but the surface is smaller and evolving faster than the RESCU backend. *Last exercised against NanoDCAL 2022A (remote) and 3.1.1 (local), LatticeMind ≥ 0.1.9.*

LatticeMind turns a plain-language request into a complete, launch-ready NanoDCAL workflow: it selects the calculation steps, builds the crystal or two-probe device structure, renders the input decks with a curated parameter vocabulary, stages the LCAO basis files with the project so it runs anywhere, executes the decks locally or on a remote server, and verifies the outputs before reporting results.

6.5.1 Configure NanoDCAL before the first run

Configure a valid RESCU license for LatticeMind startup; the same license supports RESCU execution. To run NanoDCAL calculations, also configure:

1. A NanoDCAL executable, selected by `LATTICEMIND_NANODCAL_CMD` or the Web UI's **Local NanoDCAL command** field.
2. A separate NanoDCAL license on the computer that runs the solver. Test a small NanoDCAL example directly before launching it through LatticeMind.
3. The extracted NeutralAtomDatabase from the Nanoacademic Portal, selected by `LATTICEMIND_NANODCAL_DATA_PATH` or the Web UI's **NanoDCAL neutral atom data** field.

LatticeMind searches the configured database recursively, verifies that every requested element has a `*.nad` or `*.mat` file, and stages the selected files under the project's `nanodcal_basis/` directory. When sharing a project, please keep that directory within the scope of your Nanoacademic license. See *Licenses and solver executables* for complete local and remote setup.

6.5.2 Supported and tested calculation types

The prompts below have each produced a complete, verified run. Substitute your own material freely.

Ground state and energetics

Calculation	Example prompt	Key outputs
SCF	"Using NanoDCAL, run an SCF calculation for silicon."	`NanodcalObject.mat`, total energy
Total energy	"Using NanoDCAL, calculate total energy for silicon."	`TotalEnergy.mat`
Charge analysis	"Using NanoDCAL, calculate charge analysis for silicon."	`Charge.mat`, per-atom charges
Effective potential	"Using NanoDCAL, calculate effective potential for silicon."	`EffectivePotential.mat`
Electron density	"Using NanoDCAL, extract electron density for silicon."	`TotalElectronDensity.mat`, integrated electron count
Starting fields	"Using NanoDCAL, create initialization, non-SCF rigid atomic field, and Harris-field decks for diamond germanium."	initialization, non-SCF, and Harris-field objects
Combined ground-state properties	"Using NanoDCAL, calculate total energy, effective potential, electron density, and charge analysis for rocksalt MgO."	all four property files after one SCF calculation

Electronic structure

Calculation	Example prompt	Key outputs
Band structure	"Using NanoDCAL, run an SCF and band-structure calculation for silicon."	`BandStructure.mat`, band gap, plot
Full band structure	"Using NanoDCAL, run SCF and calculate full band structure for aluminum."	`FullBandStructure.mat`
Complex band structure	"Using NanoDCAL, calculate the complex band structure for silicon."	`ComplexBandStructure.mat`
Density of states	"Using NanoDCAL, calculate density of states for silicon."	`DensityOfStates.mat`, plot
Joint DOS	"Using NanoDCAL, calculate the joint DOS for silicon."	`JointDensityOfStates.mat`
Effective mass	"Using NanoDCAL, calculate the effective mass for GaAs."	`EffectiveMass.mat`
Eigen-states and wavefunctions	"Using NanoDCAL, run SCF and calculate eigen-states, real-space wavefunction, and effective mass for silicon."	`EigenStates.mat`, `RealSpaceWavefunction.mat`
Post-SCF and momentum	"Using NanoDCAL, run SCF, post-SCF spin-orbit coupling, and momentum analysis for silicon."	`CalculatedResults.mat`, `Momentum.mat`
Spin polarization	"Using NanoDCAL, calculate spin polarization for ferromagnetic bcc iron."	`SpinPolarization.mat`

Structure and mechanics

Calculation	Example prompt	Key outputs
Relaxation	"Using NanoDCAL, relax rocksalt MgO and calculate elastic modulus."	relaxed structure
Forces and stress	"Using NanoDCAL, calculate the force and stress for silicon."	`Force.mat`, `Stress.mat`
Elastic modulus	"Using NanoDCAL, relax diamond germanium and calculate force, stress, and elastic modulus."	`ElasticModulus.mat`

Phonons (finite-displacement Hessian)

Calculation	Example prompt	Key outputs
Phonon band structure	"Using NanoDCAL, calculate the Hessian and phonon band structure for silicon."	`Hessian.mat`, `PhononBandStructure.mat`
Phonon DOS	"Using NanoDCAL, calculate the Hessian and phonon density of states for silicon."	`PhononDensityOfStates.mat`
Phonon full band structure	"Using NanoDCAL, calculate the Hessian, phonon band structure, and phonon full band structure for rocksalt MgO."	`PhononFullBandStructure.mat`

Quantum transport (two-probe devices)

LatticeMind builds the full lead-SCF → device-SCF → property chain automatically from a one-line request.

Calculation	Example prompt	Key outputs
Transmission	"Using NanoDCAL, run a transmission calculation along z for a homogeneous aluminum two-probe device."	`Transmission.mat`, T(E)
Conductance	"Using NanoDCAL, run a conductance calculation along z for a homogeneous aluminum two-probe device."	`ConductanceAndCurrent.mat`
I–V curve	"Using NanoDCAL, run an I-V curve calculation along z for a homogeneous aluminum two-probe device."	`CurrentVoltageCurves.mat`
Scattering states	"Using NanoDCAL, run a scattering-states calculation along z for a homogeneous aluminum two-probe device."	`ScatteringStates.mat`
Transmission channels	"Using NanoDCAL, run SCF and transmission-channel analysis for ferromagnetic bcc iron."	`TransmissionChannel.mat`

6.5.3 Publish-ready figures and numeric summaries

After NanoDCAL writes a `.mat` result, LatticeMind reads the physical axes and units from that file and writes these project-local files:

- `results/analysis_summary.json` contains scalar and array-shape summaries, including the band gap, integrated electron count, transmission at the Fermi level, force maximum, stress maximum, and transport channel count when those quantities are present.
- `figures/<name>.png` is the 360 dpi report image embedded in `report.html`.
- `figures/<name>.svg` and `figures/<name>.pdf` are editable or vector exports for a manuscript.
- `results/figure_manifest.json` names the source `.mat` file and records every PNG, SVG, and PDF path, plus PNG dimensions and DPI.

The figure type follows the solver data:

- Band structures use Fermi-referenced energy axes. Effective-mass figures use electron/hole dispersion fits and directional components in units of m_e.
- `EigenStates.mat` contains only the bands and k-points requested by `calculation.eigenStates.numberOfBands` and `kSpaceGridNumber`. For the silicon eigenstate prompt, `[1,1]` at a `[1,1,1]` grid selects one state below and one state above E_F at Γ. Their energy separation is a selected direct Γ-point separation, not the fundamental indirect band gap; request a band-structure calculation to determine the latter.
- `EffectivePotential.mat` and `TotalElectronDensity.mat` produce three orthogonal slices plus plane-averaged profiles.

- `TransmissionChannel.mat` produces transverse-k maps split by spin.
- Force, stress, and elastic-modulus figures use eV/Angstrom and GPa.
- `NanodcalObject.mat`, `TotalEnergy.mat`, and `CalculatedResults.mat` contribute descriptors but do not create a figure from saved-object or scalar-only arrays.

If one file contains an unsupported array shape, LatticeMind records the exact filename and parser error in `results/analysis_summary.json.warnings` and continues processing the other files. A follow-up request can regenerate the figures without rerunning NanoDCAL:

```
Reprocess the existing NanoDCAL outputs and regenerate all report figures
without rerunning the solver.
```

6.5.4 Execution options

- **Local** — runs against a local NanoDCAL installation.
- **Remote** — a profile in `~/.latticemind/remote_profiles.toml` with the exact fields `host`, `nanodcal_command`, and `shell_preamble`. LatticeMind stages a self-contained bundle (decks, basis files, job script), runs it over SSH, and fetches the verified outputs back into the project. Multiple calculations can run in parallel on a multi-core server. See *Remote & HPC execution*.
- **Portable basis** — basis files travel with the project (`nanodcal_basis/`), so a project generated on one machine runs unchanged on another.

See also

For a worked NanoDCAL tutorial, see *NanoDCAL: electronic structure and transport (experimental)* in the advanced tutorials.

CHAPTER

SEVEN

CONTACT US

For general inquiries about LatticeMind or other Nanoacademic Technologies Inc. products, please use the Contact Us Form[5].

For technical support—including bug reports, feature requests, and other technical questions—please use the Nanoacademic Customer Support Center[6].

[5] https://nanoacademic.com/contact-us/
[6] https://nanoacademic.atlassian.net/servicedesk/customer/portal/1