# **Quick rules for using these prompts**

* Feed the netdoctor\_spec.json to Cursor first (or paste the JSON into the first prompt).
* After each step, test locally (commands provided). If something fails, re-run the Cursor prompt in the step with any failing logs and ask it to fix issues.
* Commit often using the suggested commit messages.
* Use the provided acceptance criteria — only move to the next step when they are satisfied.

## **Step 0 — Upload the JSON schematic to Cursor**

Paste the entire JSON schematic into Cursor or upload netdoctor\_spec.json and then send this prompt:

**Prompt 0 (upload confirmation + plan):**

I have uploaded `netdoctor\_spec.json`, a complete schematic for the NetDoctor project. Read it fully and confirm you understand all sections. Then output a one-paragraph summary of the project goals and a step-by-step implementation plan we will follow, broken into the phases: preparation, prototyping, MVP backend, MVP GUI, polish, advanced features, packaging, docs & release. Keep the plan concise (bullet list with steps).

**Acceptance criteria**

* Cursor summarizes correctly and returns the phase breakdown.

## **Step 1 — Create repo skeleton and tooling**

**Goal:** create initial repo structure, pyproject, requirements, pre-commit, and CI skeleton.

**Prompt 1:**

Using `netdoctor\_spec.json`, create a Git repository skeleton for the NetDoctor project. Create files and folders exactly as in the "recommended\_file\_structure" section. Add:

- `pyproject.toml` (Poetry format) with minimal dependencies: PySide6, psutil, pyqtgraph, dnspython, requests.

- `requirements.txt` mirroring dependencies.

- `README.md` with project name and short summary (from JSON).

- `.gitignore` for Python, venv, build artifacts.

- `.github/workflows/lint.yml` and `.github/workflows/test.yml` stub workflows that run black, ruff, pytest.

- `pre-commit` config to run black and ruff.

Provide the exact files and their contents. Show the git commands to initialize the repo, add files, and first commit with message "chore: initial project skeleton & toolchain".

**Local commands to run after Cursor outputs files:**

git init

# create virtualenv and install if desired

# review files, then:

git add .

git commit -m "chore: initial project skeleton & toolchain"

**Acceptance criteria**

* Files are present locally as shown and git status clean after commit.

## **Step 2 — Dev environment & run scaffolding**

**Goal:** runnable main.py that launches an empty PySide6 window.

**Prompt 2:**

Create a minimal runnable PySide6 application in `netdoctor/main.py` that launches a main window with title "NetDoctor" and the left navigation rail with placeholder buttons for Dashboard, System, Network Tools, Reports, Settings. Use PySide6 only. Add `if \_\_name\_\_ == "\_\_main\_\_"` block. Provide code, and a `launch` instruction to run it.

Also add a `launch.json` snippet for VS Code (optional) and a short README instruction on how to run the dev app.

Commit files with message "feat: add minimal PySide6 app shell with nav".

**Local test commands:**

python -m netdoctor.main

# or run via VS Code

**Acceptance criteria**

* App window opens without errors and shows nav placeholders.

## **Step 3 — Implement core worker model**

**Goal:** a robust QRunnable worker pattern with signals and cancellation.

**Prompt 3:**

Implement `netdoctor/workers/task\_worker.py` with:

- `WorkerSignals` (QObject): progress(int), row(dict), log(str), error(str), finished(object).

- `TaskWorker` class (QRunnable) that wraps any callable and supports a thread-safe cancellation flag (e.g., `self.\_cancel = threading.Event()` or similar).

- Provide usage example code snippet (in a comment) showing how to start the worker using `QThreadPool.globalInstance().start(TaskWorker(...))` and connect signals.

Add unit tests `tests/test\_worker.py` that exercise running a sample long-running function and cancellation.

Commit as "feat: worker model with cancellation & signals".

**Local tests:**

pytest -q

**Acceptance criteria**

* Tests pass for worker behavior (mock a short sleep function).

## **Step 4 — Implement systeminfo backend (psutil)**

**Goal:** backend module returning system metric snapshots.

**Prompt 4:**

Create `netdoctor/core/systeminfo.py`. Implement functions:

- `get\_system\_overview()` -> dict with cpu\_percent, per\_cpu\_percent, memory\_total, memory\_used, swap, disk\_partitions (list with total & used), uptime, load\_avg (where available), interfaces (list of dicts: name, ip, mac, mtu, rx\_bytes, tx\_bytes).

Add `tests/test\_systeminfo.py` that verifies keys exist and types are correct (mock psutil where necessary).

Commit "feat: add systeminfo backend (psutil)".

**Local tests:**

pytest tests/test\_systeminfo.py -q

**Acceptance criteria**

* get\_system\_overview() returns sensible keys on your system; unit tests pass.

## **Step 5 — Implement ping backend with fallback**

**Goal:** robust ping function (raw ICMP if permitted, subprocess fallback).

**Prompt 5:**

From the JSON spec, implement `netdoctor/core/ping.py` with:

- `ping\_host(host, count=4, timeout=2, ipv6=False)` -> returns list of dicts per ping (seq, rtt\_ms, ttl, success, error)

- Attempt raw ICMP socket approach on platforms that allow it; if permission denied or not supported, fallback to invoking the system `ping` command and parse output for Linux/macOS/Windows.

- Implement `ping\_sweep(network\_cidr, concurrency=50)` that pings many addresses in parallel using ThreadPoolExecutor and yields results as they arrive.

Also add unit tests `tests/test\_ping.py` for ping\_host against `127.0.0.1` and for fallback parsing functions. Be conservative for CI (mark network tests with pytest marker `network` to skip by default).

Commit "feat: ping module with subprocess fallback + sweep".

**Local tests:**

pytest -q -k ping -m "not network" # run unit tests only

# or to run actual network tests, run with marker:

pytest -q -m network

**Acceptance criteria**

* ping\_host("127.0.0.1") returns success in unit test environment; fallback code path exists.

## **Step 6 — Implement portscanner backend (TCP connect)**

**Goal:** simple concurrent TCP connect scanner.

**Prompt 6:**

Implement `netdoctor/core/portscanner.py` with:

- `scan\_ports(host, ports, timeout=1.0, concurrency=100)` -> returns list of dict: {port, state: open/closed, banner (optional)}.

- Use `socket.create\_connection` with timeouts; concurrency via ThreadPoolExecutor.

- Add optional param `banner\_grab` which, if True, attempts a short recv() after connect to capture a banner.

- Add unit tests mocking sockets to verify open/closed code paths.

- Add detection function `detect\_nmap()` that returns the path/version if nmap is present; do not call nmap yet.

Commit "feat: tcp portscanner backend".

**Acceptance criteria**

* Tests pass and example scan against localhost shows expected behavior for common port (like closed ports).

## **Step 7 — Implement dns + whois backends**

**Goal:** dnspython-based DNS queries and WHOIS wrapper.

**Prompt 7:**

Create `netdoctor/core/dns.py` using dnspython to run queries for record types A, AAAA, MX, NS, TXT, CNAME. Provide `query\_records(domain)` returning dict of lists.

Create `netdoctor/core/whois.py` that uses `python-whois` if available, else fallback to subprocess `whois` where present. Provide safe parsing.

Add unit tests for DNS with mocking.

Commit "feat: dns & whois backends".

**Acceptance criteria**

* Unit tests for DNS parsing pass with mocked responses.

## **Step 8 — Build MVP GUI views (Ping + System + PortScan)**

**Goal:** wire GUI to backend via worker model, show results in tables and charts.

**Prompt 8:**

Using the existing PySide6 app shell, implement three views:

1) `netdoctor/gui/views/ping\_view.py`:

- Header input for host, count, start/stop.

- Use `TaskWorker` to run `ping.ping\_host` or `ping\_sweep`.

- Show results in `QTableView` (columns: seq, host, rtt\_ms, ttl, status).

- Show latency line chart using `pyqtgraph.PlotWidget`.

- Allow CSV export of the table.

2) `netdoctor/gui/views/system\_view.py`:

- On load, call `systeminfo.get\_system\_overview()` via worker and populate KPI cards and a per-interface table.

- Use sparklines (pyqtgraph) to display CPU and network usage; start a refresh timer (1s) for live updates with ability to stop.

3) `netdoctor/gui/views/portscan\_view.py`:

- Inputs for host and port ranges, thread count.

- Start button triggers `portscanner.scan\_ports` in worker.

- Results table with port, state, service (if known), and a "Banner" action to show raw banner output in modal.

For each view, implement proper enabling/disabling of Start/Stop and wire worker signals to update UI.

Add simple smoke tests or manual test instructions.

Commit "feat: add ping, system, and portscan GUI views wired to backend using worker model".

**Local tests:**

* Launch app and manually test Ping to 127.0.0.1, System view shows KPIs, Portscan against localhost.

**Acceptance criteria**

* Views function, worker updates UI without freezing, charts refresh.

## **Step 9 — Implement persistence/history & export**

**Goal:** save session data and support CSV/JSON export.

**Prompt 9:**

Create `netdoctor/storage/history.py` that:

- Provides `save\_session(session\_id, meta, results)` and `list\_sessions()` and `load\_session(session\_id)`.

- Use SQLite (via `sqlite3`) or JSON files (configurable); default to SQLite in user app data folder.

- Implement `netdoctor/core/report.py` with `export\_csv(session)` and `export\_json(session)`.

Add UI elements in Reports/History view to list saved sessions and export them.

Commit "feat: add history & export functionality".

**Acceptance criteria**

* After running a scan, user can save and later load/export session.

## **Step 10 — Add settings & dependency detection**

**Goal:** detect optional deps, implement settings UI.

**Prompt 10:**

Add `netdoctor/config.py` for global defaults and `netdoctor/gui/views/settings\_view.py` UI that allows:

- Configure default timeouts, concurrency.

- Paths for external tools (nmap).

- Theme toggle.

- Privacy/legal acknowledgement checkbox required before running scans.

Implement runtime detection utilities to detect presence of scapy, nmap, pysnmp. Show detected state in Settings view.

Commit "feat: settings UI and optional dependency detection".

**Acceptance criteria**

* Settings are persisted and optional deps detection displays correctly.

## **Step 11 — Add linting, tests CI, and pre-commit hooks**

**Goal:** ensure code quality and CI runs.

**Prompt 11:**

Add/expand `.github/workflows/lint.yml` to run black --check, ruff, mypy. Add `.github/workflows/test.yml` to run pytest with caching. Ensure pre-commit hooks run black and ruff.

Also add a `Makefile` with common dev tasks: `make venv`, `make install`, `make test`, `make run`.

Commit "chore: add CI workflows, pre-commit, and Makefile".

**Acceptance criteria**

* GitHub Actions pipeline templates present; running make test locally runs pytest.

## **Step 12 — Polish UI styling and animations**

**Goal:** apply theme, icons, and subtle animations.

**Prompt 12:**

Apply UI styling:

- Integrate `qdarktheme` (or a bundled QSS) with palette from `netdoctor\_spec.json`.

- Replace placeholder icons with SVG icons using `qtawesome` or embedded SVGs (add `netdoctor/resources/icons/\*.svg`).

- Add a slide-in animation for the left nav using `QPropertyAnimation`.

- Implement toast notifications (small sliding widgets) for success/failure messages.

Provide QSS snippets and update main window to apply the theme.

Commit "feat: theme, icons, and UI polish".

**Acceptance criteria**

* Theme loads, icons display, nav animates, toasts appear on completed scans.

## **Step 13 — Write tests for critical flows & add markers**

**Goal:** robust unit tests for non-network logic and CI configuration to skip network tests.

**Prompt 13:**

Add unit tests for core modules (systeminfo, ping fallback parsers, portscanner parse logic, DNS parsing). Mark real network tests with pytest marker `network`. Configure `pytest.ini` to skip `network` by default unless `-m network` provided. Ensure tests are referenced by CI.

Commit "test: add unit tests and network marker config".

**Acceptance criteria**

* pytest runs by default without network tests; unit tests pass.

## **Step 14 — Optional: Add scapy, pysnmp, nmap wrappers (post-MVP)**

**Goal:** optional advanced features per spec.

**Prompt 14:**

Implement optional modules:

- `netdoctor/core/arp\_scan.py` using scapy when available, else using OS ARP methods.

- `netdoctor/core/snmp.py` using pysnmp with a simple `poll\_oids` function.

- `netdoctor/core/nmap\_wrapper.py` to call nmap and parse XML output using `python-nmap` or manual XML parsing.

These should be optional and gracefully disabled when dependencies are missing.

Commit "feat: optional advanced network modules (scapy, pysnmp, nmap wrapper)".

**Acceptance criteria**

* If deps are installed they work; otherwise UI warns and hides options.

## **Step 15 — Packaging with PyInstaller & release artifacts**

**Goal:** produce per-OS build script and GitHub Actions build step.

**Prompt 15:**

Add a `build` folder containing `pyinstaller.spec` and a `scripts/package.sh` that builds a single-file executable (Linux) and instructions for Windows/macOS. Add a GitHub Actions `build.yml` workflow that creates build artifacts using appropriate runners and uploads them as release assets.

Provide guidance for signing executables (notes) and include a sample release draft template.

Commit "chore: add packaging scripts and release workflow".

**Local test:**

# Example for Linux

pyinstaller --onefile netdoctor/main.py --name netdoctor

**Acceptance criteria**

* A working Linux executable is built locally; workflows present for CI builds.

## **Step 16 — Documentation, README polish, demo GIF**

**Goal:** publish documentation and a short demo.

**Prompt 16:**

Create `docs/` with an installation guide, user guide (how to run a ping scan, how to export), and developer guide (how to add a new view). Add screenshots of the app (or placeholder images) and create a short demo GIF instructions (how to record with asciinema/peek). Update README with badges and a link to docs.

Commit "docs: add user & developer docs, README polish".

**Acceptance criteria**

* README and docs available; instructions to run app are clear.

## **Step 17 — Iterate UI/UX and collect feedback**

**Goal:** polish UX, performance, edge-case handling.

**Prompt 17:**

Run a UX pass: review all module UIs, ensure errors are handled gracefully, add help tooltips, keyboard shortcuts (Ctrl+R run, Ctrl+E export), and optimize ThreadPool concurrency to prevent UI lock.

List any performance bottlenecks and propose fixes. Make changes to address them.

Commit "chore: UX pass & performance improvements".

**Acceptance criteria**

* App is responsive under moderate load and keyboard shortcuts work.

## **Step 18 — Final checks & release**

**Goal:** finalize versioning, changelog, and create GitHub release.

**Prompt 18:**

Prepare release v0.1.0:

- Update `pyproject.toml` version and create `CHANGELOG.md` with highlights.

- Create a GitHub release draft with release notes and attach build artifacts from CI (or local builds).

- Provide a short "How to try" in README including the explicit warning about legal scanning and the privacy settings.

Commit "chore: release v0.1.0" and produce release notes text for GitHub.

**Acceptance criteria**

* Release draft exists and artifacts attached.

## **Step 19 — Handover JSON & final schematic**

**Goal:** produce final JSON schematic (updated with file hashes/versions) for handing to other AIs.

**Prompt 19:**

Generate a final `netdoctor\_spec\_final.json` that:

- Reflects the current repo (files present, implemented modules, versions in pyproject).

- Lists optional dependencies vs installed ones (detection).

- Includes final color palette, font choices, and exact QSS snippets used.

- Includes the GitHub Actions pipeline names and README link.

Attach this JSON for future automation/handoff.

**Acceptance criteria**

* netdoctor\_spec\_final.json generated and matches repo state.

## **Extra prompts you can use for troubleshooting or refining**

**Fix failing tests:** Paste failing pytest output and ask:  
  
 Tests failing with this output: <paste>. Please diagnose root cause, modify code/tests to fix, and explain the fix. Provide diffs for files to update.

**Refactor a module to async:** Convert ping and portscanner to asyncio (qasync integration) and show changes to worker model and GUI integration.

**Design polish:** Produce 3 UI mock variations (dark, light, compact) as SVG layout images for the main screen. Use the color palette and layout described in netdoctor\_spec.json.

## **Commit message conventions I recommend**

* chore: config, docs, formatting
* feat: new feature or module
* fix: bugfix
* test: tests added or updated
* refactor: internal code changes

## **Final notes & acceptance testing checklist**

Before calling this “MVP complete,” verify:

* The app runs on your target OS and opens cleanly.
* Ping, PortScan, DNS, System Overview work and do not freeze UI.
* Worker cancellation stops ongoing tasks.
* Export functions (CSV/JSON) produce correct files.
* History save/load works.
* Tests pass (non-network) and CI is active (lint + tests).
* Packaging for at least one OS builds an executable.
* README & docs explain usage and legal warnings.

If you want, I can now:

* Produce the exact first Cursor prompt file for Step 1 containing netdoctor\_spec.json and the initial repo skeleton instructions; or
* Generate the starter code for main.py, task\_worker.py, and systeminfo.py right here so you can paste to Cursor or create files locally.

Which of those two do you want next?

ChatGPT can make mistakes. Check important info. See Co

Troubleshooting

# **How to use these prompts (short guide)**

1. Gather context: stack trace, failing test output, relevant code file(s).
2. Paste the exact prompt and include the attachments in Cursor.
3. Run suggested fixes locally. If still broken, re-run with the **“If failing, do this next”** follow-up prompt (provided for each category).
4. Commit changes with a clear message and move on.

## **1) Generic "Fix failing tests" prompt**

Use when pytest or unit tests fail.

**Prompt:**

I ran pytest and got failing tests. Here is the full failing output and the files it references:

<PASTE full pytest output here>

Also attach the project files referenced in the stack traces (paths & content). Please:

1. Diagnose the root causes (list each failing test and what caused it).

2. Propose code changes and show exact diffs/patches for the files to modify.

3. Update or add unit tests if needed (explain why).

4. Explain why your fix is safe and any edge-cases to watch.

Return the exact git commands to apply the patch and run tests again.

Acceptance criteria: after applying the changes and running `pytest -q`, all previously failing tests pass.

**Attach:** full pytest output, relevant source files, tests.

**If still failing (follow-up):**

The fixes didn't pass. Here is the new pytest output: <PASTE>. Please iterate: identify remaining failures, provide corrective diffs, and explain intermediate debugging checks I should run.

## **2) Runtime exception / stack trace debugging**

Use when the app crashes with an exception.

**Prompt:**

The application threw this exception with stack trace:

<PASTE full stack trace and any console logs>

Attach the source file(s) referenced. Please:

1. Explain the root cause in 2–3 sentences.

2. Provide an exact code change (diff) to fix the error.

3. Add defensive checks / unit tests to prevent regression.

4. Suggest logging improvements or guards for future similar errors.

Acceptance criteria: after applying patch, running the scenario that caused the exception no longer crashes and the app logs a handled/meaningful error.

**Attach:** stack trace, relevant source files, minimal reproduction steps.

## **3) UI / PySide6 layout bug or frozen UI**

Use when the UI freezes, layout is broken, or widgets don't appear.

**Prompt:**

UI issue encountered:

- Describe: <one-liner: freeze/layout/incorrect rendering>

- Steps to reproduce: <list>

- Attach screenshot(s) and any console/Qt warnings.

Attach relevant UI files (.py/.ui) and worker code (QRunnable or threads). Please:

1. Diagnose why the UI is freezing or layout failing.

2. Provide exact code diffs to fix (e.g., move blocking I/O to worker, fix layout parents).

3. If freeze: explain where blocking call is and convert to QRunnable/ThreadPool or qasync coroutine.

4. Provide a minimal test snippet to verify fix.

Acceptance criteria: UI remains responsive during operation; layout renders correctly on typical screen sizes.

**If freeze persists (follow-up):**

After applying the change, UI still freezes. Here is `ps`/profiling or logs showing CPU usage and current stack traces. Please provide advanced debugging steps (how to attach a thread dump / Qt warnings) and a refined patch.

## **4) Concurrency / Worker cancellation issues**

Use when workers don't stop or continue to emit after cancel.

**Prompt:**

Worker cancellation issue:

- Describe behavior: worker doesn't stop on cancel / continues updating UI

- Attach worker implementation file and any signals/slots code.

Please:

1. Identify why cancellation flag isn't working or isn't checked (show exact line(s)).

2. Provide a robust worker pattern patch that uses a thread-safe Event or atomic flag, ensures sockets/timeouts are closed, and emits a 'cancelled' finished state.

3. Add test-case or reproducible script to demonstrate cancel behavior.

Acceptance criteria: calling Cancel results in worker stopping within a reasonable time (<2s) and no further UI updates from that worker.

## **5) Packaging / PyInstaller build failure**

Use when PyInstaller build errors out or runtime executable missing resources.

**Prompt:**

PyInstaller build failed. Paste the full build log and the pyinstaller .spec content:

<PASTE logs and spec file>

Attach the project tree (especially resource files, .ui, icons). Please:

1. Diagnose missing imports/data files and explain why PyInstaller missed them.

2. Provide updated `.spec` and exact `pyinstaller` command to produce a working single-file build.

3. If runtime fails due to missing plugins (Qt libs), add hooks or runtime path changes and give test steps to validate on target OS.

Acceptance criteria: produced executable runs on the target OS and loads UI and resources without missing-file errors.

**If cross-platform packaging:**

Also include CI steps or Docker-based approach for reproducible builds and note any platform-specific caveats (macOS signing, Windows UAC).

## **6) CI pipeline failing (GitHub Actions)**

Use when workflows fail.

**Prompt:**

CI workflow failing. Paste the failing run log from GitHub Actions and the workflow YAML file:

<PASTE logs and .github/workflows/...yml>

Please:

1. Explain the failing step(s).

2. Provide corrected workflow YAML or commands with minimal changes.

3. Suggest caching, matrix runners, or environment setup to speed and stabilize CI.

4. Provide a local reproduce command using act (optional) or docker.

Acceptance criteria: After updating workflow, subsequent commit triggers a passing run for lint and tests.

## **7) Dependency / import errors (version mismatches)**

Use when imports fail or behavior differs between dev machines.

**Prompt:**

Import / dependency issue:

- Paste the full ImportError/Traceback.

- Provide `pyproject.toml` or `requirements.txt` and `pip freeze` of the environment.

Please:

1. Identify the version incompatibility or missing dependency.

2. Recommend exact version pins or conditional imports and show diffs to requirements or code.

3. Suggest environment setup commands and a simple verification command.

Acceptance criteria: After adjusting deps and reinstalling, import errors are resolved in a fresh virtualenv.

## **8) Performance profiling / UI lag**

Use when tasks are slow or UI lags under load.

**Prompt:**

Performance issue: describe symptom (slow scan, UI lag). Provide sample dataset or test steps and attach CPU profile or wall-clock timings if available.

Please:

1. Suggest profiling steps (e.g., cProfile, py-spy, perf) and exact commands to run that capture the issue.

2. Analyze the output you provide and highlight hotspots (functions responsible).

3. Provide code-level optimizations (diffs), concurrency improvements, or architecture changes (queue batching, rate limiting).

4. If applicable, propose caching strategies, database indexing, or streaming updates to UI rather than full re-renders.

Acceptance criteria: after applying changes, the scenario runs at least X% faster (specify target) or UI remains responsive under the same load.

## **9) Cross-platform bugs (Windows vs Linux vs macOS)**

Use when behavior differs between OSes (permissions, path, raw sockets).

**Prompt:**

Cross-platform discrepancy:

- Describe difference and steps to reproduce on both OSes.

- Provide platform-specific logs, `uname`/OS version, Python version, and relevant code that branches on platform.

Please:

1. Explain why behavior differs and propose cross-platform-safe alternatives (use subprocess fallback, permissions check, or platform-specific branches).

2. Provide code diffs to implement robust fallbacks with clear docstrings.

3. Add platform-specific tests or CI matrix instructions.

Acceptance criteria: behavior consistent across target OSes or documented and handled with clear fallbacks.

## **10) Security / sensitive data concerns**

Use when exporting reports or collecting potentially sensitive data.

**Prompt:**

Security/privacy audit request:

- Describe the data collected and exported and show the export code path.

- Indicate any fields that may be sensitive (IPs, usernames, logs).

Please:

1. Point out any potential leakage or security problem.

2. Provide code diffs to redact or hash sensitive fields (configurable opt-in/opt-out).

3. Suggest secure storage best practices (encryption at rest for DB, or user opt-in).

4. Provide recommended README wording and in-app warning text.

Acceptance criteria: exported reports have a configurable redact option and README documents privacy risks.

## **11) UX / styling polish issues**

Use when visuals are off or theme inconsistent.

**Prompt:**

UI/Theme inconsistency:

- Attach screenshots and describe what looks wrong (color contrast, font size, spacing, component alignment).

- Attach current QSS or theme code.

Please:

1. Provide a focused visual patch: updated QSS (diff) and icon suggestion swaps.

2. Recommend micro-interaction tweaks (duration for animations, easing).

3. Provide before/after screenshot mockups or exact CSS-like QSS changes.

Acceptance criteria: updated QSS applied and visuals look consistent per provided palette; toolbar/nav spacing fixed.

## **12) Refactor / architecture change request**

Use when you want to convert workers to asyncio/qasync, or split monolith into modules.

**Prompt:**

Refactor request:

- Explain current architecture and the desired target (e.g., convert QRunnable to asyncio + qasync integration).

- Provide files that need conversion and any constraints (must remain testable).

Please:

1. Provide a step-by-step refactor plan with small commits that preserve functionality at each step.

2. Provide diffs for the first commit that introduces asyncio entry points and tests, and subsequent commits for full conversion.

3. Suggest migration tests and CI changes.

Acceptance criteria: After incremental commits, tests pass and both sync and async modes work (or sync mode is fully replaced intentionally).

## **13) Iterative "Explain and patch" template (short, general)**

A rapid-fire prompt for many quick issues.

**Prompt:**

Explain-and-patch:

- Here are the logs/errors: <PASTE>

- Files: <list & attach relevant files>

Please:

1. Explain in plain English what is broken.

2. Provide the single smallest code patch (diff) that most likely fixes it.

3. Provide steps to verify fix locally.

Acceptance criteria: after applying the patch and running the reproducer, the error ceases.

## **14) When all else fails — create a minimal reproducible example (MRE)**

Use when the bug is complex and you need to isolate it.

**Prompt:**

Help me build a minimal reproducible example for this bug. The original project is large and failing with: <PASTE error>. Please:

1. Suggest the minimal subset of files and code that reproduce the error.

2. Provide a zip/tarball structure or explicit files for the MRE.

3. Provide instructions to run only the MRE and reproduce the bug.

Acceptance criteria: MRE reproduces the bug on a fresh environment and is small enough (~<200 LOC).

## **15) Conventions & best practices (meta-prompt)**

Ask for a checklist for future-proofing projects.

**Prompt:**

Provide a concise checklist (20 items max) of conventions and best practices to follow for this Python PySide6 project to make future debugging and CI easier. Include items for testing, logging, error handling, packaging, dependency management, code style, and release notes conventions.

**Expected output:** actionable checklist you can paste into repo CONTRIBUTING.md.

# **Small tips on how to paste logs and diffs into Cursor**

* Keep logs intact and full — often the last 40–80 lines are most useful.
* For large files, paste only the failing file(s) and the surrounding 30–50 LOC.
* When attaching diffs, show unified diff format (git diff --staged style) so the assistant can suggest patches in the same format.

# **Final note — iterative loop pattern**

1. Run failing case → capture logs/tests.
2. Paste logs + use one of the prompts above.
3. Apply patch and run locally.
4. If still failing, paste new logs and use the “If still failing” follow-up.
5. When passes, commit with a clear message and move on.

# High-level strategy & scope

* **Ship a tiny, working MVP first.** Pick 3 features (Ping, System Overview, TCP Port Scan) and make them rock-solid before adding bells. Recruiters and hiring managers care more about a polished, usable slice than an unfinished monster.
* **Iterate in short cycles.** Implement a small feature → test locally → commit → push → run CI. Frequent commits and releases prevent scope creep.

# Project setup & tooling

* **Use poetry** for dependency & virtualenv management (or pip+venv if you prefer). Lock versions for reproducible builds.
* **Enable pre-commit hooks** (black, ruff/flake8, isort) to keep code consistent.
* **Use type hints + mypy** gradually — they catch many bugs early. Add them to core modules first.
* **Add linters to CI** early so style doesn’t become a problem later.

# Architecture & code hygiene

* **Keep UI <-> logic separated.** GUI calls should only call worker interfaces that expose plain Python functions (pure logic). This makes testing and reuse trivial.
* **Make core modules testable.** Avoid direct Qt dependencies in core/\* modules so you can unit-test them easily.
* **Design for optional dependencies.** Make scapy/pysnmp/nmap optional and detect availability at runtime — don’t import them at top-level.

# Concurrency & responsiveness

* **Never block the Qt main thread.** Always use QRunnable/QThreadPool or qasync+asyncio. Test long-running operations under load.
* **Implement proper cancellation.** Workers should poll a thread-safe Event or flag and clean up sockets/timers on cancel.
* **Use small timeouts and rate-limits by default** for scans. Safe defaults avoid accidental DoS-like behavior.

# Testing strategy

* **Split tests by type:** unit, integration (network), UI. Mark network tests with pytest.mark.network and skip them by default on CI.
* **Mock network IO** for most unit tests. Use tools like unittest.mock or local fake sockets.
* **Add smoke tests in CI** that run the app shell or minimal GUI checks (e.g., using pytest-qt).

# Cross-platform & platform-specifics

* **Windows:** ICMP raw sockets often require admin; Npcap/WinPcap needed for scapy. Document how to run as admin and how to install Npcap.
* **macOS:** Code signing / notarization is painful if you plan to distribute widely later. Consider delaying macOS binaries until you’re ready to handle signing.
* **Linux:** libpcap is usually available; Docker can reproduce environments.
* **Document these platform instructions** clearly in README.

# Security, privacy & legal

* **Explicit permission warning** before scans. Add a mandatory checkbox that the user acknowledges permission to scan networks.
* **Do not collect telemetry** unless user opt-in explicitly. If you ever add telemetry, store only telemetry-safe, anonymized metrics and be transparent.
* **Redaction & export controls:** let users redact IPs and usernames in exported reports.
* **Secure secrets:** if users supply API keys (WHOIS/registrar), store them in OS keyring (or encrypted config), not in plain files.

# Packaging & distribution

* **Expect packaging friction.** PyInstaller often needs extra hooks for Qt plugins and data files (.ui, icons). Test packaging early.
* **Use CI matrix** for builds (Linux/Windows/macOS runners). For cross-platform reproducible builds, consider ephemeral runners or GitHub-hosted runners (no cross-compiling).
* **Sign your binaries** (Windows signtool, macOS codesign) before public releases if you want users to trust them.

# UX & polish

* **Focus on discoverability.** Tooltips, example targets (localhost), and sensible defaults reduce friction.
* **Make errors actionable.** Don’t just show a stack trace — show a human-friendly message and a suggested remedy.
* **Provide sample workflows** in the README (e.g., “How to triage slow network with NetDoctor”).

# Documentation & portfolio

* **Capture demo GIFs early.** A short 10–20s demo GIF is gold on GitHub README and LinkedIn.
* **Write a short “How I built NetDoctor”** walkthrough for your portfolio — employers love the story and decisions.
* **Include screenshots for each major feature.**

# Dev workflow & collaboration

* **Use feature branches + PRs**, even if you’re solo. It’s a habit reviewers expect.
* **Write clear commit messages** (feat:, fix:, chore:, test:) — you already have a good convention in the roadmap.
* **Open an ISSUE template** for bugs/feature requests to track scope.

# Observability & debugging

* **Structured logging** (use logging with clear levels). Add a debug mode that writes detailed logs to file (rotating logs).
* **Add a diagnostic export** (bundle logs + systeminfo + config) to help troubleshooting.
* **Include health endpoints** in the CLI or a small local web server mode if you want remote checks later.

# Performance & scaling

* **Start conservative** with concurrency/timeouts; profile before making aggressive speedups.
* **Batch UI updates** instead of updating row-by-row for very large results (use model.beginResetModel / endResetModel patterns).
* **Limit result set sizes** or paginate in GUI for huge scans.

# Legal / ethical checklist before sharing or demoing scans

* Only demo scans on your own network or public test networks.
* Add an in-app **demo mode** that uses prerecorded sample data — great for demos without network risk.

# Quick wins to impress employers

* Polished README, demo GIF, one-click installer or AppImage, clear LICENSE, well-structured repo.
* Include "How to run in 3 steps" and "What I learned building this" in README.
* Add automated tests and CI badge to README.