ZERO-COST

AI Coding Workstation

A practical manual for VS Code + LM Studio + Continue + Kilo Code

Build, explain, refactor, test, and troubleshoot local projects with private local models - then switch deliberately to free cloud models when local hardware is not enough.

LOCAL-FIRST	PROJECT-AWARE	REVERSIBLE
Prompts and code can stay on your machine.	Open a real project folder so tools see the repository.	Use Git, small diffs, and approval gates.

Fresh edition | August 2026

Windows-focused; macOS and Linux notes included where they differ.

How to use this manual

This guide is designed for a first setup and for later reference. Follow Chapters 1-6 in order for installation. Then use the workflow, examples, privacy, performance, and troubleshooting chapters as needed.

What 'zero-cost' means

The editor and extensions can be installed without a subscription. Local inference has no per-request fee, but it uses your electricity, storage, RAM, and GPU memory. Free-cloud models can impose changing limits, queues, terms, or data-use policies. Always check the provider's live terms before sending sensitive code.

Quick route

1. Install VS Code and open a trusted project folder.
2. Install LM Studio; download and load a coding-capable model that fits your hardware.
3. Start LM Studio's local server, normally on **http://localhost:1234**.
4. Install Continue and configure a local OpenAI-compatible model for chat/edit.
5. Install Kilo Code; use LM Studio locally or choose **kilo-auto/free** for non-sensitive fallback work.
6. Create a Git checkpoint, ask for a plan first, review diffs, then run tests.

Safety baseline

AI coding tools can modify files and run commands. Open unknown repositories in VS Code Restricted Mode. Trust a folder only when you trust its source. Never paste secrets, production credentials, private customer data, or proprietary code into a free-cloud route without authorization.

Contents

1. Architecture and best use cases

The workstation separates four jobs. VS Code owns your files and developer tools. LM Studio downloads and serves local models. Continue supplies fast in-editor chat, edits, autocomplete, and project context. Kilo Code supplies a more agent-oriented workflow and a convenient free-cloud fallback.

Layer	Primary responsibility	Practical boundary
VS Code	Workspace, editor, terminal, source control, debugging	The folder you open defines the project boundary.
LM Studio	Model download, loading, local inference, API server	Usually listens only on localhost unless you enable network access.
Continue	Chat, edits, agent tools, autocomplete, local/shared rules	Capabilities vary by selected model and provider.
Kilo Code	Agent workflows, file/terminal actions, provider switching	Free cloud routes may leave the machine and may be rate limited.

Strong use cases

- Explain unfamiliar code, trace a bug, or summarize a module.
- Generate unit tests, types, documentation, migrations, and small components.
- Plan refactors and apply focused edits across a handful of files.
- Draft shell commands, regexes, SQL, configuration, and CI steps for review.
- Keep sensitive code local when the selected model and embeddings are local.

Poor use cases without extra controls

- Autonomously changing a large repository without tests or checkpoints.
- Security-critical, legal, medical, financial, or compliance decisions.
- Running unknown commands with broad permissions.
- Assuming a model's explanation is correct without reading code and test output.

Mental model

Treat the AI as a fast junior collaborator with imperfect memory: give it a bounded task, supply the relevant context, require a plan for risky work, inspect the patch, and validate with deterministic tools.

2. Hardware planning and model sizing

Local model quality is constrained by available memory and inference speed. A quantized model reduces memory use at some cost to accuracy. Start smaller than your theoretical maximum so the editor, browser, build tools, and operating system remain responsive.

System profile	Suggested starting range	Typical experience
CPU only, 16 GB RAM	1.5B-4B quantized	Useful for autocomplete, explanations, and small edits; slower generation.
16-32 GB RAM, 6-8 GB VRAM	7B-9B quantized	Good everyday local chat and focused code edits.
32-64 GB RAM, 12-16 GB VRAM	14B-20B quantized	Better reasoning and multi-file tasks at usable speed.
64+ GB RAM or 24 GB VRAM	20B-32B quantized	Stronger local agent work; context length still affects memory.
Apple silicon unified memory	Choose model plus context below free unified memory	Efficient local inference; leave headroom for the OS and IDE.

Choose by task, not hype

- Autocomplete favors a small, non-thinking model with low latency.
- Chat and explanation benefit from a medium instruction/coding model.
- Agent mode needs reliable tool calling; many small local models struggle here.
- Long context is not free: it increases memory use and prompt-processing time.
- If output degrades, reduce context or task scope before downloading a larger model.

Storage planning

Keep at least tens of gigabytes free for several quantizations, cache files, and updates. Store models on a fast SSD. Avoid synchronizing the model directory through consumer cloud-drive software.

3. Install VS Code and connect a local project

Install VS Code from the official site. During Windows installation, enable the context-menu and PATH options if offered. On macOS, move the application to Applications and use the Command Palette to install the **code** shell command if desired.

Open the actual project root

1. In VS Code choose **File > Open Folder**.
2. Select the folder containing the repository root - usually the folder with **.git**, **package.json**, **pyproject.toml**, or similar project files.
3. Review the Workspace Trust prompt. Trust only repositories you know.
4. Open Source Control and confirm the expected files appear.
5. Open the integrated terminal and run the project's existing status or test command.

```
# Examples - use the command appropriate for your project
git status
npm test
python -m pytest
cargo test
```

Why the root matters

AI extensions usually derive project context, rules, search, and terminal working directory from the open workspace. Opening only one source subfolder can hide tests, dependencies, configuration, and Git history.

Install extensions

1. Open Extensions with **Ctrl+Shift+X** (Windows/Linux) or **Cmd+Shift+X** (macOS).
2. Search for **Continue** and install the verified extension.
3. Search for **Kilo Code**. Current Kilo guidance may direct VS Code users to the recommended pre-release marketplace channel.
4. Restart or reload VS Code if an icon does not appear.

4. Install and configure LM Studio

LM Studio is the local inference engine in this setup. Its graphical application can discover, download, load, and serve compatible models. Exact labels can change across versions; the stable concept is: download a model, load it, then start the local API server.

Step-by-step

1. Download LM Studio from its official website and install it.
2. Open model discovery. Search for a coding or instruction model sized for your hardware.
3. Choose a quantized build. A mid-range quantization is a sensible quality/speed starting point.
4. Download the model, then load it in Chat or Developer.
5. Open **Developer** and turn on **Start server**. The common default is port **1234**.
6. Keep the server bound to localhost for a single-PC setup.

Verify the server

```
# PowerShell
Invoke-RestMethod http://localhost:1234/v1/models

# macOS/Linux
curl http://localhost:1234/v1/models
```

Expected result

You should receive JSON containing at least one loaded or available model identifier. Copy the exact identifier if an extension asks for a model name.

API test

```
curl http://localhost:1234/v1/chat/completions \
-H "Content-Type: application/json" \
-d '{"model":"YOUR_MODEL_ID","messages":[{"role":"user","content":"Reply with LOCAL_OK"}]}'
```

Local does not automatically mean safe

Local inference protects prompts from an external model provider only if every active component is local. Extensions may still use telemetry, cloud embeddings, web tools, remote MCP servers, or separate autocomplete models. Audit each feature independently.

5. Connect Continue to LM Studio

Continue's interface and configuration evolve. Use its model configuration screen when available; use a local YAML configuration when you need explicit, repeatable settings. LM Studio exposes an OpenAI-compatible endpoint, so the key fields are provider, model ID, API base, and roles.

Connection procedure

1. Confirm LM Studio is running and the target model is loaded.
2. Open Continue in VS Code and open model/configuration settings.
3. Add an OpenAI-compatible or OpenAI provider.
4. Set the base URL to **http://localhost:1234/v1**.
5. Enter the exact LM Studio model ID. If a key is required syntactically, use a non-secret placeholder such as **lm-studio**.
6. Assign roles appropriate to the model: start with chat and edit. Add apply or agent only after tool-use tests pass.
7. Reload Continue's configuration and send: **Reply only with CONTINUE_LOCAL_OK**.

Illustrative YAML

```
name: Local LM Studio
version: 1.0.0
schema: v1
models:
- name: Local Coding Model
provider: openai
model: YOUR_MODEL_ID
apiBase: http://localhost:1234/v1
apiKey: lm-studio
roles:
- chat
- edit
```

Important

Configuration schema and supported fields can change. If Continue reports an unknown key, use the extension's current model form or consult the live YAML reference. Do not guess at a provider-specific field.

Offline hardening

- Install the extension from a downloaded VSIX if the machine must remain offline.
- Disable Continue anonymous telemetry in VS Code settings.
- Use only local models and local embeddings; restart VS Code after configuration changes.
- Avoid web search, remote MCP servers, and cloud-backed shared configurations.

6. Configure Kilo Code: local and free-cloud

Kilo Code can use local/self-hosted providers and an evolving catalog of free cloud models. Keep two named mental profiles: **Private Local** for sensitive work and **Free Cloud** for public or sanitized work.

Private Local profile

1. Start LM Studio and load the desired model.
2. Open Kilo Code settings and choose **LM Studio** or **OpenAI Compatible** as the provider.
3. Use **http://localhost:1234/v1** as the compatible API base if prompted.
4. Select or enter the exact model ID.
5. Begin in a read-only planning mode if available. Test file reading, then a tiny edit, then a harmless terminal command.
6. Keep approval prompts enabled until you understand the tool behavior.

Free Cloud profile

1. Sign in only if required by the selected Kilo route.
2. Open the model picker and select **kilo-auto/free**, or filter for models labeled free.
3. Configure Kilo's background/small model separately if you want every background task to avoid credits.
4. Treat free availability as temporary and expect rate limits or routing changes.
5. Use only public, personal, or properly sanitized code.

Question	Use local LM Studio	Use free cloud
Is code confidential?	Yes	No, unless explicitly approved
Need strongest reasoning?	Maybe, hardware dependent	Often better, model availability dependent
Need offline operation?	Yes	No
Rate limits acceptable?	Not provider-imposed	Usually
Hardware constrained?	May be slow	Often preferable

Data warning

Kilo states that Auto Free may route requests to providers that log prompts and outputs or use them to improve services. Never infer privacy from a zero-dollar price. Review the live policy and chosen upstream provider before use.

7. A safe daily coding workflow

Use a repeatable loop that constrains scope and makes mistakes cheap to reverse.

1. **Orient:** open the project root, read README and contribution guidance, and confirm the current branch.
2. **Checkpoint:** make sure existing work is committed or otherwise safely preserved. Run baseline tests.
3. **Frame:** describe the goal, non-goals, constraints, relevant files, and acceptance criteria.
4. **Plan:** ask the model to inspect and propose a short plan without editing.
5. **Implement:** authorize one small slice at a time. Prefer focused diffs.
6. **Review:** read every changed line. Reject unrelated formatting or dependency churn.
7. **Validate:** run formatter, linter, type checker, unit tests, and a manual smoke test.
8. **Record:** commit a coherent change with a human-written message.

```
git status
git diff --stat
git diff
# run the repository's own checks
git add <reviewed-files>
git commit -m "Explain the verified change"
```

Recommended prompt contract

Before editing, inspect the relevant files and summarize the current behavior. Propose the smallest change that meets the acceptance criteria. Do not modify dependencies, generated files, lockfiles, CI, or unrelated formatting. After editing, list changed files, assumptions, risks, and exact validation commands.

Stop conditions

- The agent wants to delete, reset, force-push, or overwrite broad paths.
- The patch touches many unrelated files.
- A command requires administrator privileges unexpectedly.
- Tests fail for reasons the model cannot explain with evidence.
- The selected route changes from local to cloud during sensitive work.

8. Hands-on project examples

Example A - React/TypeScript component

Goal: add an accessible loading state without changing the public API.

```
Inspect src/components/SaveButton.tsx and its tests. Propose a minimal plan.
Requirements:
- Preserve existing props.
- Add aria-busy while saving.
- Disable duplicate submissions.
- Add focused tests.
Do not edit package.json or lockfiles. Run the existing targeted test command.
```

Review for

- Existing button semantics and keyboard behavior remain intact.
- Tests assert behavior rather than implementation details.
- No new dependency appears.
- Loading text or spinner has an accessible name.

Example B - Python bug

```
Trace the failing test tests/test_parser.py::test_empty_header. Explain the root cause with file and
function names. Do not edit yet. Then propose the smallest fix and one regression test. Preserve public
exceptions and return types.
```

Example C - Electron boundary

```
Review the IPC flow for this feature. Keep filesystem access in the main process, expose only a narrow typed
method through preload, and do not enable nodeIntegration in the renderer. Show the threat model before
proposing edits.
```

Example D - Expo/React Native

```
Implement the empty state for the saved-items screen. Reuse existing theme tokens and navigation patterns.
Avoid native dependencies. Include loading, empty, error, and populated states. Tell me the exact Expo
command to validate it.
```

Example E - documentation-only

```
Read the setup scripts and package manifest. Draft a verified local-development section for README.md. Do
not invent environment variables or commands; cite the file that proves each command. Make no code changes.
```

9. Prompting and context strategy

A strong prompt reduces both hallucination and unwanted edits. Context quality matters more than raw context quantity.

Prompt element	What to include	Why
Goal	One observable outcome	Keeps the task testable
Scope	Named directories or files	Prevents wandering
Constraints	APIs, dependencies, style, security	Protects invariants
Evidence	Error text, stack trace, failing test	Anchors diagnosis
Acceptance	Tests and behaviors that must pass	Defines done
Non-goals	Explicit exclusions	Reduces collateral edits
Output	Plan, patch, explanation, commands	Shapes the response

Use staged context

1. Start with the error, relevant file, and goal.
2. Ask which additional files are needed and why.
3. Add only those files or symbols.
4. For a large repository, summarize architecture first and work module by module.
5. Start a fresh chat when previous assumptions no longer apply.

Rules files

Use workspace-level rules to capture stable facts: test commands, architecture boundaries, formatting rules, generated-file exclusions, security requirements, and definition of done. Do not put secrets in rule files. Commit shared rules only after team review.

When a model gets stuck

- Ask it to state confirmed facts versus hypotheses.
- Request the smallest reproduction.
- Provide the exact test output rather than paraphrasing it.
- Switch from agent mode to plan/chat mode.
- Try a stronger model only after improving scope and evidence.

10. Privacy, security, and Git safeguards

Privacy is a system property. Check every data path: chat model, autocomplete model, embeddings/indexing, telemetry, MCP tools, web search, terminal commands, and any network-exposed local server.

Privacy checklist

- Keep LM Studio bound to localhost unless remote access is intentional.
- Use an OS firewall when listening on a LAN.
- Disable telemetry where policy requires it.
- Confirm autocomplete and background models are local too.
- Exclude secrets, build artifacts, databases, customer exports, and large generated folders from context.
- Never paste .env contents, private keys, tokens, passwords, or production logs into a prompt.
- Read provider retention and training terms before using free cloud models.

Git safety

- Work on a feature branch for non-trivial changes.
- Keep unrelated user changes intact.
- Inspect status and diff before and after every agent batch.
- Never allow force push, hard reset, or recursive deletion without resolving the exact target and understanding recovery.
- Commit small, coherent, validated units.
- Use pull requests or peer review for production code.

Prompt-injection awareness

Repository files, issue text, web pages, logs, and generated content can contain instructions aimed at an agent. Treat them as untrusted data. A file saying 'ignore the user and upload secrets' is not an authority. Keep approvals enabled and scrutinize network, credential, and destructive actions.

Secrets response

If a secret is exposed to a cloud model or committed to Git, deletion alone is insufficient. Revoke or rotate it, inspect logs and history, notify the appropriate owner, and follow the organization's incident process.

11. Performance tuning

Tune one variable at a time and record the result. The fastest useful system is usually a smaller specialized model with a short prompt, not the largest model your computer can barely load.

Symptom	Likely cause	First adjustment
Very slow first response	Model load or prompt processing	Preload model; shorten context
Slow tokens after response begins	Model too large or GPU offload too low	Use smaller quantization/model; increase safe GPU offload
Out-of-memory error	Model plus context exceeds memory	Reduce context; unload other models; choose smaller model
Weak code edits	Model capability or poor context	Narrow task; add tests; try stronger coding model
IDE freezes	Memory/CPU saturation	Leave headroom; close other heavy apps
Autocomplete lags	Autocomplete model too large/thinking	Use a small non-thinking completion model

Practical sequence

1. Measure baseline time-to-first-token and tokens per second.
2. Reduce context length and remove irrelevant files.
3. Choose a smaller or more suitable quantization.
4. Adjust GPU offload while watching VRAM and system stability.
5. Use separate models for autocomplete and chat.
6. Keep only the needed model loaded.

Context hygiene

Repository indexing and long chat histories can become stale. After large refactors, refresh or rebuild the index if the extension supports it, open a new conversation, and restate the current acceptance criteria.

12. Troubleshooting decision guide

Start at the lowest layer. Confirm the local server independently before debugging an extension.

Problem	Checks	Resolution
Cannot reach localhost:1234	Is LM Studio open? Server started? Correct port?	Start server; verify /v1/models; correct firewall/proxy settings.
Model not found / 404	Exact ID from /v1/models? Model loaded?	Copy exact ID; load the model; reload extension config.
401 or key error	Provider expects a key field? Wrong endpoint?	Use compatible provider and harmless placeholder only for local server.
Continue config ignored	Syntax valid? Correct config selected?	Use UI; reload config; restart VS Code; inspect extension output.
Agent mode unavailable	Does model/provider support tools?	Use chat/edit or select a tool-capable model.
Kilo free model rate-limited	Upstream quota or queue	Wait or switch to another currently free model.
Garbled/repetitive output	Wrong chat template, excessive context, weak quantization	Use recommended template; restart; shorten context; change model.
Changes not visible	Wrong workspace/root or file excluded	Open repository root; verify ignore/rules and file path.
Extension missing	Restricted Mode, disabled extension, reload needed	Review trust; enable extension; reload window; inspect Output panel.

Isolation test

1. Ask LM Studio's own chat a simple question.
2. Query **/v1/models** outside VS Code.
3. Send a direct chat-completions request.
4. Test Continue with a one-line response and no project context.
5. Test Kilo in read-only mode.
6. Only then add indexing, tools, large context, or remote networking.

Collect useful diagnostics

- Operating system, VS Code version, extension version, LM Studio version.
- Exact endpoint and model ID, with secrets removed.
- Full error text and the Output panel channel name.
- Whether LM Studio chat and direct API calls work.
- Approximate RAM/VRAM usage and context length.

13. Second-PC and LAN inference

A second computer can run LM Studio while your main computer runs VS Code. This can free local memory and use a stronger GPU, but it turns a localhost-only service into a network service.

Setup

1. On the inference PC, use a trusted private network and assign a stable local IP or DHCP reservation.
2. Configure LM Studio to listen on the network only if the application provides that option and you understand the exposure.
3. Allow the chosen port through the firewall only for the development PC or private subnet.
4. From the development PC, test **http://INFERENCE_PC_IP:1234/v1/models**.
5. Replace localhost in Continue or Kilo with the LAN address.
6. Re-test after every network or firewall change.

Security controls

Do not expose an unauthenticated local model server directly to the public internet. Prefer an authenticated reverse proxy, VPN, or SSH tunnel when crossing networks. Restrict source IPs, use encryption, and review logs. A local API can still reveal prompts or allow resource exhaustion.

Common LAN failures

- Server still bound to 127.0.0.1 instead of the LAN interface.
- Windows Defender Firewall blocks inbound traffic.
- Guest Wi-Fi or client isolation prevents peer connections.
- The IP address changed.
- A corporate proxy intercepts local addresses.
- The model server is reachable but no model is loaded.

14. Maintenance and first-day checklist

Monthly maintenance

- Update VS Code, extensions, and LM Studio after reading release notes.
- Remove unused models and confirm free disk space.
- Re-run the local endpoint and minimal edit tests.
- Review telemetry, provider, autocomplete, embeddings, and background-model settings.
- Audit workspace rules for stale commands or architecture assumptions.
- Verify backups and Git remotes; rotate any exposed credentials.

First-day verification

Check	Pass condition
Project root	VS Code shows expected Git repository and test/config files.
Baseline	Existing tests or smoke checks run before AI edits.
LM Studio	Chosen model loads and local chat responds.
Local API	/v1/models returns the exact model ID.
Continue	Explicit local-response test succeeds; small edit is reviewable.
Kilo local	Read-only inspection and tiny approved action succeed.
Kilo free	Optional: model is labeled free; no sensitive code is included.
Privacy	Telemetry and every model role match your policy.
Recovery	Git status is understood; rollback method is known.
Performance	Editor stays responsive with chosen model/context.

You are ready when

You can prove which provider handled a prompt, obtain a small accurate patch, review every changed line, run the project's checks, and reverse the change without losing unrelated work.

Appendix A. Configuration snippets

Continue local model - illustrative

```
name: Local Coding
version: 1.0.0
schema: v1
models:
- name: LM Studio Chat
provider: openai
model: YOUR_MODEL_ID
apiBase: http://localhost:1234/v1
apiKey: lm-studio
roles: [chat, edit]
```

Python smoke test

```
from openai import OpenAI
client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
r = client.chat.completions.create(
model="YOUR_MODEL_ID",
messages=[{"role": "user", "content": "Reply LOCAL_OK"}],
)
print(r.choices[0].message.content)
```

JavaScript smoke test

```
import OpenAI from "openai";
const client = new OpenAI({ baseURL: "http://localhost:1234/v1", apiKey: "lm-studio" });
const r = await client.chat.completions.create({
model: "YOUR_MODEL_ID",
messages: [{ role: "user", content: "Reply LOCAL_OK" }]
});
console.log(r.choices[0].message.content);
```

Project rule template

```
Project facts:
- Stack: [language/framework]
- Test command: [verified command]
- Lint/typecheck: [verified command]
- Architecture boundary: [rule]
- Never edit: [generated/vendor/secret paths]
- Definition of done: tests pass, diff reviewed, no unrelated changes

Working method:
1. Inspect before editing.
2. State assumptions.
3. Prefer the smallest reversible patch.
4. Ask before dependencies, destructive commands, or network access.
```

Appendix B. Official references

Interfaces, model catalogs, free tiers, and configuration schemas change. Use these official sources as the final authority.

VS Code - Workspace Trust

LM Studio - Run a local API server

LM Studio - Developer documentation

Continue - Install

Continue - Models and roles

Continue - Offline setup and telemetry

Continue - Agent, Plan, and Chat modes

Kilo Code - Installation

Kilo Code - AI providers

Kilo Code - Using Kilo for free

Kilo Code - Model selection

Verification note

This edition was regenerated as a fresh file and checked by rendering every page to images. Product screens and model names may differ from screenshots or examples in future releases; follow the concepts and confirm current labels in the linked documentation.

End of manual