Best Local Coding Models Ranked: Every VRAM Tier, Every Benchmark (2026)

January 28, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Qwen 2.5 Coder still dominates for autocomplete and FIM at every tier. Qwen 3.5 9B is the chat/reasoning coding model at 8GB, and the 27B (LiveCodeBench 80.7, SWE-bench 72.4) is a beast at 24GB with native tool calling. Qwen3-Coder-Next just hit #1 on SWE-rebench Pass@5 (64.6%) — beating every closed model including Claude Opus 4.6 and GPT-5.2. It's an 80B MoE with only 3B active, instruct-mode (not thinking), Apache 2.0, and fits on a 48GB Mac at Q4.

📚 **More on this topic:** VRAM Requirements · GPU Buying Guide · Ollama vs LM Studio · Qwen 3.5 Small Models: 9B Beats 30B

GitHub Copilot costs $10-19/month. ChatGPT Plus is $20. Claude Pro is $20. That's $120-240 per year for coding assistance that sends every line of your code to someone else's servers.

Local coding models flip that equation. You run them on your own hardware, your code never leaves your machine, and the total cost after setup is zero. The tradeoff used to be quality — local models couldn't compete with cloud. That changed in 2025. Open-source coding models now match GPT-4o on standard benchmarks, and they run on hardware most developers already own.

This guide covers which models to use at every VRAM tier, how they compare on real benchmarks, and exactly how to set them up in your editor.

Why Code Locally?

Four reasons developers are switching:

Your code stays private. Every prompt you send to Copilot or ChatGPT passes through corporate servers. If you're working on proprietary code, client projects, or anything sensitive, that's a risk. Local models process everything on your machine. Nothing leaves.

No recurring costs. $19/month for Copilot Business doesn't sound like much until you multiply it across a team or count years. Local models are free after the hardware investment — which you've probably already made if you have a GPU.

Works offline. Planes, coffee shops with bad WiFi, air-gapped environments, or just your ISP having a bad day. Local models don't care. No internet required.

No rate limits or surprise changes. No throttling during peak hours, no model swap without notice, no features removed from your tier. You control the model, the version, and the configuration.

What Makes a Good Coding Model

Not all LLMs are equal at code. The best coding models share four traits:

Code completion (FIM). Fill-in-the-middle support means the model can complete code given both the text before and after the cursor. This is what powers inline autocomplete in your editor. Not all models support FIM — coding-specific ones do.

Instruction following. "Refactor this function," "explain this regex," "write tests for this module." The model needs to follow natural-language instructions about code precisely, not just generate code from scratch.

Multi-language support. Most developers work across at least 2-3 languages. The model should handle Python, JavaScript/TypeScript, and your stack's other languages without falling apart.

Sufficient context window. Code context matters. A model that can only see 2K tokens is useless when your function depends on types defined 500 lines up. You want at least 8K, ideally 32K+.

The Benchmarks That Matter

Benchmark	What It Tests	Why It Matters
HumanEval	Generate correct Python functions from docstrings (164 tasks)	The standard for code generation quality
HumanEval+	Same tasks, 80x more test cases	Catches models that pass easy tests but fail edge cases
MBPP	974 programming problems	Broader than HumanEval, tests practical coding

Benchmark	What It Tests	Why It Matters
MultiPL-E	HumanEval translated to 18+ languages	Shows if the model only knows Python or actually handles JS, Rust, etc.
LiveCodeBench	600+ real coding contest problems	Tests harder, more realistic tasks

HumanEval pass@1 is the most commonly reported number. Higher is better. For reference: GPT-4o scores ~90%, GPT-3.5 scored ~48% when it launched.

Best Models by VRAM Tier

8GB VRAM (RTX 4060, 3070, 3060 Ti)

This is the most common GPU tier for developers. The good news: the best 7B coding model now outperforms much larger models from a year ago.

Model	HumanEval	FIM	Context	VRAM (Q4)	Best For
Qwen 2.5 Coder 7B	88.4%	Yes	128K	~5 GB	Best for autocomplete/FIM
Qwen 3.5 9B ⭐	—†	No	262K	6.6 GB	Best for chat coding, multimodal
DeepSeek Coder V2 Lite	81.1%	Yes	128K	~5 GB	Reasoning-heavy tasks
DeepSeek Coder 6.7B	~65%	Yes	16K	~4.5 GB	Lightweight, fast
CodeLlama 7B	~30%	Yes	16K	~4.5 GB	Legacy. Skip for new setups.

†Qwen 3.5 9B isn't a coding-specific model so HumanEval isn't reported. Its LiveCodeBench v6 score is 65.6, and it beats Qwen3-30B on GPQA Diamond (81.7 vs 77.2) and IFEval (91.5 vs 88.9).

For autocomplete: Qwen 2.5 Coder 7B. Still the FIM king. 88.4% HumanEval at 7B parameters, beats CodeStral-22B and DeepSeek Coder 33B V1. FIM support, 128K context, 92+ languages. This is what powers your tab-complete.

```
ollama pull qwen2.5-coder:7b
```

For chat-based coding: Qwen 3.5 9B. This dropped March 2, 2026 and it's a different animal. Not a coding-specific model, but a general-purpose model with native multimodal and 262K

context that happens to be excellent at code. What makes it worth running alongside your autocomplete model:

- **Reads images natively.** Paste a code screenshot, whiteboard photo, or error dialog and it understands them. No separate vision model. Same weights handle text and images.
- **262K context.** Enough to fit large files or multiple files in one prompt. The Qwen 2.5 Coder 7B tops out at 128K and eats more VRAM doing it.
- **Thinking mode.** Enable it for harder problems: `ollama run qwen3.5:9b` with `/set parameter num_predict 81920` and the model generates `<think>` blocks before answering. This substantially improves results on complex debugging and refactoring tasks.
- **6.6GB on Ollama at Q4_K_M.** Fits 8GB VRAM with room to spare.

```
ollama run qwen3.5:9b
```

The play at 8GB: run Qwen 2.5 Coder 7B for autocomplete, swap to Qwen 3.5 9B for chat, debugging, and code review. They don't need to run simultaneously.

DeepSeek Coder V2 Lite is still solid for reasoning-heavy tasks. 16B MoE with only 2.4B active parameters, so it's fast and memory-efficient.

CodeLlama 7B is showing its age. At ~30% HumanEval, it's been lapped by models half its size from newer families. Only use it if you have a specific reason (e.g., Meta's Llama license requirements).

Laptops and Integrated Graphics: Qwen 3.5 4B

If you're on a laptop with no discrete GPU, or you need a coding assistant on integrated graphics, the Qwen 3.5 4B is the new minimum viable coding model. It dropped alongside the 9B on March 2, 2026.

At 3.4GB on Ollama, it runs on basically anything with 4GB of RAM to spare. LiveCodeBench v6 score of 55.8, GPQA Diamond of 76.2, and it's natively multimodal with the same 262K context as its bigger sibling. It won't replace a proper coding model for complex tasks, but for quick code explanations, simple refactoring, and reading error screenshots on a thin laptop, nothing else at this size comes close.

```
ollama run qwen3.5:4b
```

This replaces anything sub-4B you might have been running for lightweight coding. The full breakdown is here.

12-16GB VRAM (RTX 3060 12GB, 5060 Ti 16GB, 4060 Ti 16GB)

The mid-range tier opens up significantly better models. If you've got 16GB of VRAM, you can run 14B models at high quality or squeeze in a quantized 33B.

Model	HumanEval	FIM	Context	VRAM (Q4)	Best For
Qwen 2.5 Coder 14B	~89%	Yes	128K	~9 GB	Best at this tier
DeepSeek Coder 33B (Q3)	70%	Yes	16K	~16 GB	24GB model squeezed down
CodeLlama 13B	~36%	Yes	16K	~8.5 GB	Legacy. Outclassed.

The winner: Qwen 2.5 Coder 14B. It surpasses CodeStral-22B and DeepSeek Coder 33B on benchmarks despite being smaller. At Q4 quantization it needs only ~9GB, leaving room for long context on a 16GB card. State-of-the-art on over 10 code evaluation benchmarks.

```
ollama pull qwen2.5-coder:14b
```

DeepSeek Coder 33B at Q3 quantization technically fits in 16GB, but it's a tight squeeze with degraded quality. If you have exactly 16GB, the Qwen 14B at higher quantization (Q5 or Q6) will outperform it in practice.

24GB VRAM (RTX 3090, 4090)

This is where local coding gets seriously competitive with cloud models. A used RTX 3090 at $700-850 gives you access to models that match GPT-4o.

Model	HumanEval	FIM	Context	VRAM (Q4)	Best For
Qwen 2.5 Coder 32B	92.7%	Yes	128K	~20 GB	Best FIM/autocomplete at 24GB
Qwen 3.5 27B	—	No	262K	~16 GB	Dense, ties GPT-5 mini on SWE-bench
Qwen 3.5 35B-A3B	—	No	262K	~20 GB	MoE, fast agentic coding
		Yes	16K	~20 GB	Older but solid

Model	HumanEval	FIM	Context	VRAM (Q4)	Best For
DeepSeek Coder 33B	70% (78%*)				
CodeLlama 34B	53.7%	Yes	16K	~20 GB	Legacy. Outclassed.

*78% with CodeFuse fine-tuning

For autocomplete: Qwen 2.5 Coder 32B. Still the FIM king at 24GB. 92.7% HumanEval, 73.7 on Aider (code repair), 128K context. At Q4_K_M (~20GB), it fits on a single 24GB card.

```
ollama pull qwen2.5-coder:32b
```

For chat coding and reasoning: Qwen 3.5 27B. A dense 27B model that ties GPT-5 mini on SWE-bench Verified (72.4%) and fits at ~16GB Q4, leaving 8GB for context on a 24GB card. No FIM support, but the 262K context window and native multimodal make it a strong pair with the Coder 32B. Use Coder for tab-complete, 27B for "refactor this module" conversations.

The March benchmarks filled in the picture: **LiveCodeBench v6 of 80.7** and **Terminal-Bench 2 of 41.6** (GPT-5 mini scores 31.9 on the same test). It also has native tool calling via the `qwen3_coder` parser — VS Code extensions like Continue can now dispatch tool calls directly to the 27B for file edits, shell commands, and web lookups without external wrappers.

```
ollama run qwen3.5:27b
```

For fast agentic coding: Qwen 3.5 35B-A3B. This MoE model holds 35B total parameters but only fires 3B per token. On an RTX 3090, one user measured 112 tok/s, and another scaffolded 10 files and 3,483 lines from a single spec. It beats the previous-gen Qwen3-235B-A22B on GPQA Diamond (84.2 vs 81.1) with 1/7th the parameters. The catch: it needs ~20GB to load despite only 3B being active, so you won't fit it alongside another model on 24GB.

```
ollama run qwen3.5:35b-a3b
```

The 24GB tier now has three distinct coding strategies: Coder 32B for FIM, 27B dense for chat reasoning, 35B-A3B MoE for fast agentic work. If you have a 3090 or 4090, pick two and swap between them.

48GB+ / Mac: Qwen3-Coder-Next (Agentic Coding)

Released February 3, 2026, Qwen3-Coder-Next is a different beast. It's an 80B MoE that activates only 3B parameters per token, built specifically for agentic coding workflows — the kind where the model reads your repo, plans changes across multiple files, and executes them. It's **instruct-mode, not thinking** — no `<think>` blocks, just direct tool-use and code generation, which keeps latency low for agentic loops.

Metric	Score
SWE-bench Verified	70.6%
SWE-bench Pro	44.3%
SWE-rebench Pass@5	**64.6% (#1 overall)**
Context	256K (extendable to 1M with YaRN)
VRAM (Q4)	~35-40 GB
License	Apache 2.0

That SWE-rebench Pass@5 of 64.6% is the headline number: it's **#1 on the entire SWE-rebench leaderboard**, beating every closed model including Claude Opus 4.6 (58.3%), GPT-5.2-medium (60.4%), and Gemini 3 Pro (58.3%). SWE-bench Verified at 70.6% puts it ahead of DeepSeek V3.2 (70.2%) on real-world GitHub issue resolution. The 3B active parameter count means throughput is high for its capability level — 20-40 tok/s on consumer hardware, 10-15 tok/s on a Mac Mini M4 Pro 64GB.

```
ollama pull qwen3-coder-next
```

The catch: VRAM. At Q4, it needs ~35-40GB. A single RTX 3090/4090 can't hold it without CPU offload (which drops you to ~12 tok/s). This model shines on:

- **Mac with 48GB+ unified memory** — runs natively, no offloading
- **Dual 24GB GPUs** — if you've got two 3090s
- **CPU offload on 24GB GPU + 32GB RAM** — usable but slower

Unsloth requantized GGUFs (March update): Unsloth's initial Qwen3-Coder-Next GGUFs had a bug that applied MXFP4 quantization to attention tensors instead of just routed expert weights, causing measurable quality degradation. This was fixed on February 27 — all quants were regenerated. If you downloaded before that date, re-download. Use `Q4_K_M` or `MXFP4_MOE` variants for the best quality-to-size ratio.

This isn't a replacement for Qwen 2.5 Coder 32B on a single 24GB card. It's what you graduate to when you want agentic workflows — tools like Aider or Claude Code pointed at a local model that can reason across an entire codebase. If you have the memory, it's the strongest open-source coding agent model available right now.

The Master Comparison

Every model side by side:

Model	Params	HumanEval	SWE-bench	VRAM (Q4)	FIM	Context	License
Qwen3-Coder-Next ⭐	80B (3B active)	—	70.6% (rebench P@5: 64.6%)	~38 GB	No	256K	Apache 2.0
Qwen 3.5 35B-A3B	35B (3B active)	—	69.2%	~20 GB	No	262K	Apache 2.0
Qwen 3.5 27B	27B	—	72.4%	~16 GB	No	262K	Apache 2.0
Qwen 2.5 Coder 32B	32B	92.7%	—	~20 GB	Yes	128K	Apache 2.0
Qwen 2.5 Coder 14B	14B	~89%	—	~9 GB	Yes	128K	Apache 2.0
Qwen 2.5 Coder 7B	7B	88.4%	—	~5 GB	Yes	128K	Apache 2.0
Qwen 3.5 9B	9B	—†	—	6.6 GB	No	262K	Apache 2.0
Qwen 3.5 4B	4B	—†	—	3.4 GB	No	262K	Apache 2.0
DS Coder V2 Lite	16B (2.4B active)	81.1%	—	~5 GB	Yes	128K	MIT

Model	Params	HumanEval	SWE-bench	VRAM (Q4)	FIM	Context	License
DS Coder 33B (V1)	33B	70%	—	~20 GB	Yes	16K	Permissive
CodeLlama 34B	34B	53.7%	—	~20 GB	Yes	16K	Llama
CodeLlama 7B	7B	~30%	—	~4.5 GB	Yes	16K	Llama

†Qwen 3.5 models are general-purpose multimodal, not coding-specific. LiveCodeBench v6: 65.6 (9B) / 55.8 (4B). They excel at reasoning, instruction-following, and vision tasks — see full benchmarks.

Four tiers of coding model now, each best at different things. Qwen 2.5 Coder owns FIM autocomplete — nothing touches it for tab-complete at any size. Qwen 3.5 9B/4B handle chat-based coding with native vision and 262K context. At 24GB, the Qwen 3.5 27B (72.4% SWE-bench, dense) and 35B-A3B (112 tok/s on a 3090, MoE) give you two strong options for chat coding and agentic workflows. And Qwen3-Coder-Next is the 48GB+ agentic option — it resolves real GitHub issues at 70.6% on SWE-bench, frontier-model territory for hardware you own.

→ Check what fits your hardware with our Planning Tool.

Best Model by Language

All models above are multi-language, but some have particular strengths.

Language	Best Local Model	Notes
Python	Qwen 2.5 Coder (any size)	Best benchmarked language across all models
JavaScript/ TypeScript	Qwen 2.5 Coder 14B+	Strong JS/TS support; 7B handles it well too
Rust	Qwen 2.5 Coder 32B	Smaller models struggle with Rust's borrow checker; 32B handles it
Go	Qwen 2.5 Coder 14B+	Clean Go output from 14B up
C/C++	DeepSeek Coder 33B	Slightly better at low-level memory management patterns

Language	Best Local Model	Notes
Java	Qwen 2.5 Coder 14B+	Good boilerplate generation, understands frameworks
SQL	Qwen 2.5 Coder (any size)	82% on Spider benchmark — well ahead of competitors

The honest caveat: For Python and JavaScript, the 7B Qwen Coder is genuinely excellent. For Rust, C++, and other complex compiled languages, bigger models produce noticeably better results. If Rust is your primary language and you only have 8GB, expect some friction — the model will get syntax right but occasionally misunderstand lifetime annotations or trait bounds.

How to Set Up Local Coding in Your Editor

Option 1: VS Code + Ollama + Continue (Recommended)

This is the free, open-source Copilot replacement. Continue is a VS Code extension that connects to Ollama for both chat and autocomplete.

Step 1: Install Ollama

If you haven't already, follow our Ollama setup guide. One command on any OS.

Step 2: Pull your coding model

```
# Pick your tier:
ollama pull qwen2.5-coder:7b     # 8GB VRAM
ollama pull qwen2.5-coder:14b    # 16GB VRAM
ollama pull qwen2.5-coder:32b    # 24GB VRAM
```

Step 3: Install Continue extension

Open VS Code → Extensions (Ctrl+Shift+X) → Search "Continue" → Install.

Step 4: Configure Continue

Create or edit `~/.continue/config.yaml`:

```
name: Local Coding
version: 0.0.1
schema: v1
models:
  - uses: ollama/qwen2.5-coder-7b
```

For autocomplete (tab completion), Continue uses a separate, smaller model by default. You can also point it at your main coding model.

Step 5: Start coding

- **Chat:** Click the Continue icon in the sidebar, ask questions about your code
- **Autocomplete:** Start typing and suggestions appear inline
- **Edit:** Select code, press Ctrl+I, describe the change you want

Everything runs locally. No API keys, no accounts, no internet.

Option 2: LM Studio as Backend

If you prefer LM Studio's visual interface, you can use it as a backend for Continue too. Start LM Studio's local server, then point Continue at `http://localhost:1234/v1`.

This is useful if you like browsing and switching between models visually.

Option 3: Tabby (Self-Hosted Copilot)

Tabby is a self-hosted AI coding assistant with its own VS Code extension, JetBrains plugin, and Vim support. It's more opinionated than Continue — closer to a full Copilot replacement with built-in code indexing.

```
docker run -d --gpus all -p 8080:8080 \
  -v $HOME/.tabby:/data \
  registry.tabbyml.com/tabbyml/tabby serve \
  --model Qwen2.5-Coder-7B \
  --device cuda
```

Tabby works well for teams who want a shared local coding server. For solo developers, Continue + Ollama is simpler.

Option 4: Aider (Terminal)

Aider is a terminal-based coding assistant that edits files directly. Point it at your local Ollama instance:

```
pip install aider-chat
aider --model ollama/qwen2.5-coder:7b
```

It understands your git repo, makes edits across files, and creates commits. Best for developers who live in the terminal.

Practical Tips

FIM vs. Chat: Know the Difference

FIM (Fill-in-the-Middle) powers inline autocomplete — the cursor is in the middle of your code and the model predicts what goes there. This is what makes "tab complete" work. Qwen 2.5 Coder and DeepSeek Coder both support FIM.

Chat mode is for conversations: "explain this function," "refactor this class," "write tests." It's a different inference mode. Most editors let you use both simultaneously — FIM for autocomplete, chat for dialogue.

Keep a Small Model for Autocomplete

Autocomplete needs to be fast — under 200ms ideally. On 8GB, your main 7B coding model handles both chat and FIM fine. On 16-24GB, consider running a smaller model (Qwen 2.5 Coder 1.5B or 3B) for autocomplete and your bigger model for chat. This keeps tab-complete snappy while giving you full power for longer tasks.

Context Window vs. VRAM

Bigger context windows eat more VRAM. If you're working on a large codebase and want the model to "see" more files, you'll burn through your VRAM headroom fast. On 8GB, stick to 4-8K context for coding. On 16GB, you can push to 16K. On 24GB, 32K is comfortable.

For navigating large codebases, quantization at Q4_K_S instead of Q4_K_M saves a few hundred MB that can go toward context.

Tool Calling Fixes (March 2026)

If you're using Qwen models for agentic coding (tool calls, function calling), two recent updates matter:

llama.cpp merged fixes for Qwen's XML-tagged tool call format. Earlier builds would choke on the non-standard XML tags Qwen3-Coder uses for function names and parameters, or hit a `key_gdiff` calculation bug that caused looping output. The Qwen 3.5 chat template also had a tool-calling bug that's now patched. If you built llama.cpp before early March 2026, rebuild from source.

exllamav3 added full Qwen 3.5 support — both the dense models (27B) and MoE variants (35B-A3B), including multimodal. If you're running ExLlama for faster inference, Qwen 3.5 models now work out of the box.

Close Your Browser

Same advice as for any 8GB VRAM workload: Chrome's hardware acceleration eats GPU memory. Close it or disable GPU acceleration when running local models. On 24GB this matters less, but on 8-16GB it can be the difference between smooth inference and OOM errors.

The Bottom Line

Local coding models now cover three distinct jobs:

1. **Autocomplete/FIM:** Qwen 2.5 Coder (7B/14B/32B) — still the best at every tier, nothing has displaced it.
2. **Chat coding:** Qwen 3.5 9B — native vision, 262K context, thinking mode. The full family runs from 0.8B to 397B, all Apache 2.0.
3. **Agentic coding:** Qwen3-Coder-Next — #1 on SWE-rebench Pass@5 (64.6%), 70.6% SWE-bench Verified, needs 48GB+ but only activates 3B params so it's fast. Instruct-mode, not thinking.

The 2-model setup for 8GB VRAM:

```
# 1. Install Ollama (if you haven't)
curl -fsSL https://ollama.com/install.sh | sh

# 2. Autocomplete model (FIM, tab-complete)
ollama pull qwen2.5-coder:7b
```

```
# 3. Chat/reasoning model (multimodal, 262K context)
ollama pull qwen3.5:9b

# 4. Install Continue extension in VS Code, configure, and code
```

No subscriptions. No data leaving your machine. No rate limits. Just you, your code, and two models that cover different parts of the workflow.

Related Guides

- Qwen 3.5 Small Models: 9B Beats Last-Gen 30B
- Qwen 3.5 Complete Local Guide
- How Much VRAM Do You Need for Local LLMs?
- GPU Buying Guide for Local AI
- Ollama vs LM Studio: Which Should You Use?

Sources: Qwen2.5-Coder Technical Report, Qwen3-Coder-Next Model Card, Qwen3.5-27B Model Card, Qwen3.5-9B Model Card, Qwen 3.5 on Ollama, SWE-rebench Leaderboard, Unsloth Qwen3-Coder-Next GGUF, ExLlamaV3 GitHub, DeepSeek Coder GitHub, Continue.dev Ollama Guide, EvalPlus Leaderboard, Tabby

---