Replace GitHub Copilot With Local LLMs in VS Code — Free, Private, No Subscription

March 1, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Install Continue (open source, Apache 2.0) in VS Code, run Ollama with Qwen 2.5 Coder 7B, and you get tab completion + chat for free on 8GB VRAM. On 24GB, Qwen 2.5 Coder 32B matches GPT-4o on coding benchmarks. You lose some speed versus Copilot, but your code never leaves your machine and there's no $19/month bill.

📚 **Related:** Best Models for Coding Locally · Local Alternatives to Claude Code · Ollama Troubleshooting Guide · VRAM Requirements

GitHub Copilot costs $10/month for individuals and $19/month for business users. Every keystroke, every prompt, every line of your code goes to Microsoft's servers. And if you've used Copilot during peak hours, you've hit the rate limits — that spinning cursor while you wait for a suggestion that may never come.

Local LLMs flip all of that. Your code stays on your machine. The cost after hardware is zero. No rate limits, no internet required, no subscription that quietly renews every month. The tradeoff used to be quality — local models couldn't keep up. That's no longer true. Qwen 2.5 Coder 32B scores 92.9% on HumanEval, matching GPT-4o. The 7B variant hits 88.4% and runs on an 8GB GPU.

This guide walks through the best VS Code extensions for local code completion, a step-by-step setup with Continue + Ollama, and which models to run at every VRAM tier.

Why Replace Copilot

Four reasons developers are making the switch:

Your code stays on your machine. Copilot sends your code context to GitHub's servers with every request. If you work on proprietary code, client projects, or anything under NDA, that's a liability. Local inference means nothing leaves your hardware. Period.

No recurring cost. $10/month is $120/year. $19/month business tier is $228/year. Multiply that across a team. Local models cost nothing to run once you own the GPU — and if you're a developer with a gaming card, you already do.

No rate limits. Copilot throttles heavy users. Microsoft has tightened limits repeatedly since 2025, and Business tier users report slower completions during peak hours. Your local model runs at the same speed whether it's 3am or 3pm.

Works offline. Planes, trains, coffee shops with spotty WiFi, air-gapped environments. Copilot goes silent without internet. Local models don't care.

The honest tradeoff: Copilot is faster. Cloud GPUs outperform consumer cards, and Copilot's completions arrive in 200-400ms. Local models on a 3090 take 500ms-2s depending on the model. For most developers, that's still fast enough. But if you need instant completions on every keystroke, cloud wins on raw latency.

The Extensions, Ranked

Not every VS Code extension that claims local model support actually delivers. Some have archived repos. Others quietly require paid accounts for the good features. Here's what actually works in March 2026.

1. Continue — Best Overall

Detail	Info
GitHub Stars	~31,600
License	Apache 2.0
Ollama Support	Native, first-class
Autocomplete	Yes (FIM-based tab completion)
Chat	Yes (sidebar + inline)
Agent Mode	Yes (tool calling, file edits)
Latest Version	v1.2.16 (Feb 2026)

Continue is the clear winner for local coding in VS Code. It has native Ollama integration, supports both tab autocomplete and chat with separate models for each, and it's fully open source under Apache 2.0.

What makes Continue stand out is the multi-model setup. You can run a small, fast model (Qwen 2.5 Coder 1.5B) for autocomplete and a larger model (Qwen 2.5 Coder 32B) for chat and

refactoring. Autocomplete stays snappy while complex questions get the big model's full attention.

Continue also supports context providers: `@codebase` to search your whole repo, `@file` to reference specific files, `@docs` to pull from documentation, `@terminal` for recent terminal output. These close much of the gap with Copilot's cloud-powered context.

The newer versions added agent mode with MCP (Model Context Protocol) support, letting the LLM edit files and run terminal commands autonomously. It's closer to Claude Code than to traditional autocomplete.

Who it's for: Anyone replacing Copilot. It's the most complete, best maintained option.

2. Tabby — Best for Teams

Detail	Info
GitHub Stars	~33,000
License	Apache 2.0 (core) + proprietary EE
Ollama Support	Yes (HTTP connector)
Autocomplete	Yes (FIM, server-based)
Chat	Yes + Answer Engine
Self-Hosted	Yes (Rust server)
Latest Version	v0.32.0 (Jan 2026)

Tabby takes a different approach: you run a self-hosted server, and the VS Code extension connects to it. That server handles model inference, codebase indexing, and team management.

That architecture is why Tabby works well for teams. You set up one GPU server, and every developer gets completions from it. The Answer Engine feature indexes your internal docs and codebase for RAG-powered answers to project-specific questions.

The setup cost is higher than Continue. You're running a Rust-based server process, configuring model backends, managing infrastructure. For a solo developer, that's overkill. For a team of 5-10 developers sharing one beefy GPU server, it pays for itself on day one versus Copilot Business licenses.

Tabby supports Ollama as a backend, so you configure models the same way. One catch: the FIM prompt template needs to be explicitly set in the Tabby config for your chosen model.

Who it's for: Teams who want a self-hosted Copilot replacement with shared infrastructure. Solo devs should use Continue instead.

3. CodeGPT — Simplest Setup

Detail	Info
VS Code Installs	~2.29 million
License	Freemium (local features free)
Ollama Support	Yes, native
Autocomplete	Yes
Chat	Yes (slash commands)
Latest Version	v3.16.23

CodeGPT has the largest install base of any AI coding extension outside of Copilot itself. The appeal is simplicity — install the extension, select Ollama as your provider, pick a model, and it works.

The built-in slash commands (`/Fix`, `/Document`, `/Refactor`, `/Unit Testing`) are convenient. The downside is less configuration depth than Continue. You can't separately assign models for autocomplete versus chat, and the advanced context features (codebase-wide RAG, custom context providers) aren't as developed.

CodeGPT is backed by a commercial company, and some features push toward their cloud platform. The local Ollama path works fine, but it's clearly not their primary focus.

Who it's for: Developers who want the simplest possible setup and don't need advanced configuration.

4. Cody — Enterprise Only (Individual Plans Discontinued)

Sourcegraph discontinued Cody's Free and Pro plans in July 2025. New individual signups are no longer available. If you had Cody Free/Pro, you've been migrated to Sourcegraph's new tool, Amp.

When Cody did support Ollama, the local implementation was experimental and limited — autocomplete context was restricted to the current file only, while the cloud version sent context from all open editors. That gap was never fixed before the plan was discontinued.

Skip this unless you're on an enterprise Sourcegraph contract.

5. Twinny — Archived, Don't Use

Twinny was a popular lightweight option for local autocomplete in VS Code. The repository was archived in November 2025 and is no longer maintained. A fork called "twinny ex" exists on the marketplace, but it's a private continuation with no clear maintenance commitment.

If you see old guides recommending Twinny, they're outdated. Use Continue instead.

Quick Comparison

Extension	Best For	Ollama	Tab Complete	Chat	Agent	Status
Continue	Solo devs, all-around	Native	Yes (FIM)	Yes	Yes	Active
Tabby	Teams, self-hosted	Yes	Yes (FIM)	Yes	No	Active
CodeGPT	Simplest setup	Native	Yes	Yes	No	Active
Cody	Enterprise only	Was experimental	—	—	—	Individual plans dead
Twinny	Nobody (archived)	Was native	—	—	—	Archived Nov 2025

Setup Walkthrough: Continue + Ollama + Qwen 2.5 Coder 7B

This is the setup I recommend for most developers. It runs on 8GB VRAM, gives you both tab completion and chat, and takes about 10 minutes.

Prerequisites

- VS Code installed
- A GPU with at least 6GB VRAM (RTX 3060, 4060, or equivalent)
- ~5GB free disk space for the model

Step 1: Install Ollama

```
# Linux
curl -fsSL https://ollama.com/install.sh | sh

# macOS
brew install ollama

# Windows — download from https://ollama.com/download
```

Verify it's running:

```
ollama --version
```

If you hit issues, check our Ollama troubleshooting guide.

Step 2: Pull Qwen 2.5 Coder 7B

```
ollama pull qwen2.5-coder:7b-instruct-q4_K_M
```

This downloads ~4.5GB. The Q4_K_M quantization gives you the best quality-per-VRAM ratio. For autocomplete specifically, also grab the smaller model:

```
ollama pull qwen2.5-coder:1.5b-instruct-q4_K_M
```

The 1.5B model is ~1GB and responds faster for tab completions. Use the 7B for chat.

Step 3: Install Continue in VS Code

1. Open VS Code
2. Go to Extensions (Ctrl+Shift+X)
3. Search "Continue"
4. Install "Continue - Codestral, Claude, and more" by Continue
5. The Continue sidebar panel appears on the left

Step 4: Configure Continue for Ollama

Continue uses a `config.yaml` file. Open it via the Continue sidebar → gear icon, or find it at `~/.continue/config.yaml`.

Replace the contents with:

```
models:
  - name: Qwen 2.5 Coder 7B
    provider: ollama
    model: qwen2.5-coder:7b-instruct-q4_K_M
    roles:
      - chat
      - edit

tabAutocompleteModel:
  provider: ollama
  model: qwen2.5-coder:1.5b-instruct-q4_K_M

contextProviders:
  - name: codebase
  - name: file
  - name: terminal
```

This sets up:

- **Chat and edit:** Qwen 2.5 Coder 7B (better quality for conversations and refactoring)
- **Tab autocomplete:** Qwen 2.5 Coder 1.5B (faster response for inline completions)
- **Context providers:** Codebase search, file references, and terminal output

Step 5: Test It

Open any code file. Start typing a function and pause — you should see ghost text autocomplete suggestions. Press Tab to accept.

Open the Continue sidebar (Ctrl+L) and ask: “Explain this file” or “Write a test for this function.” The 7B model handles these well.

What to Expect

- **Tab completions:** 100-300ms on a modern GPU. Nearly instant for the 1.5B model.

- **Chat responses:** 2-5 seconds for the first tokens from the 7B model, then streaming at 30-40 tok/s on a 3060 12GB.
- **Quality:** The 7B model handles single-function completions, docstring generation, and simple refactors well. It struggles with complex multi-file reasoning. That's where you want a bigger model or cloud fallback.

Advanced Setup: Qwen 2.5 Coder 32B on 24GB VRAM

If you have an RTX 3090 or 4090, the 32B model is worth the VRAM. It scores 92.9% on HumanEval, matching GPT-4o, and handles complex refactoring and multi-step reasoning that the 7B can't touch.

```
ollama pull qwen2.5-coder:32b-instruct-q4_K_M
```

This pulls ~20GB. Update your `config.yaml`:

```
models:
  - name: Qwen 2.5 Coder 32B
    provider: ollama
    model: qwen2.5-coder:32b-instruct-q4_K_M
    roles:
      - chat
      - edit

tabAutocompleteModel:
  provider: ollama
  model: qwen2.5-coder:7b-instruct-q4_K_M

contextProviders:
  - name: codebase
  - name: file
  - name: terminal
  - name: git
```

Now you're running:

- **Chat/edit:** 32B (near-Copilot quality for complex tasks)
- **Tab autocomplete:** 7B (fast enough for inline, better quality than 1.5B)

On an RTX 3090, expect the 32B model to generate at 15-18 tok/s. Slower than Copilot, but the quality per token is higher than anything in the 7-14B range. For the type of work where you actually need the chat — explaining unfamiliar code, designing a function's interface, catching subtle bugs — the wait is worth it.

What Works Well Locally

Tab completion and autocomplete. This is where local models shine. FIM (fill-in-the-middle) trained models like Qwen 2.5 Coder are purpose-built for this task. With the 7B model, completions are fast and accurate for single-line and short multi-line suggestions. It's the closest experience to Copilot.

Docstring and comment generation. Ask the model to document a function and it does a solid job. The code is right there in context, so the model doesn't need deep project understanding — just the function signature and body.

Explain this code. Paste unfamiliar code into the chat, ask what it does. Local models handle this well because the full context is in the prompt. No codebase-wide reasoning required.

Simple refactoring. Rename variables, extract functions, convert a loop to a list comprehension, add type hints to a function. Single-file, well-scoped changes are reliable with 7B+ models.

Boilerplate generation. Test scaffolding, CRUD endpoints, configuration files, Docker setup. Pattern-heavy code where the model's training data directly applies.

What Still Struggles Locally

Large codebase context. Copilot's cloud infrastructure can process context from across your workspace. Local models are limited by their context window and your GPU's memory. Continue's `@codebase` provider helps by doing RAG over your repo, but it's not the same as having the full project in context. Complex questions that require understanding how 5 files interact still go wrong.

Multi-file edits. Ask a local 7B model to refactor an interface and update all its implementations across multiple files, and you'll get inconsistent results. The 32B model is better, but this remains the biggest gap between local and cloud. Multi-file agentic workflows really want frontier-class reasoning.

Very long completions. Generating a full 200-line class from a description is unreliable with smaller models. They lose coherence past 50-80 lines. The 32B model handles this better, but cloud models still have the edge.

Uncommon languages and frameworks. Qwen 2.5 Coder is strong in Python, JavaScript/TypeScript, Java, C++, Go, and Rust. For niche languages (Elixir, Haskell, COBOL) or very new frameworks, the training data thins out and quality drops.

Speed on complex prompts. A 32B model generating a detailed code explanation takes 10-15 seconds to produce a full response. Copilot returns similar quality answers in 2-3 seconds. For rapid-fire Q&A, the latency adds up.

Model Recommendations by VRAM

VRAM	Autocomplete Model	Chat Model	What You Get
6-8GB	Qwen 2.5 Coder 1.5B (Q4)	Qwen 2.5 Coder 7B (Q4)	Good tab completion, basic chat. Run one model at a time.
12GB	Qwen 2.5 Coder 7B (Q4)	Qwen 2.5 Coder 14B (Q4)	Strong tab completion, solid chat for single-file tasks.
16GB	Qwen 2.5 Coder 7B (Q4)	Qwen 2.5 Coder 14B (Q4)	Same models, more headroom for longer context windows.
24GB	Qwen 2.5 Coder 7B (Q4)	Qwen 2.5 Coder 32B (Q4)	Near-Copilot quality. Best single-GPU local coding setup.

Ollama Pull Commands by Tier

8GB VRAM:

```
ollama pull qwen2.5-coder:1.5b-instruct-q4_K_M
ollama pull qwen2.5-coder:7b-instruct-q4_K_M
```

12-16GB VRAM:

```
ollama pull qwen2.5-coder:7b-instruct-q4_K_M
ollama pull qwen2.5-coder:14b-instruct-q4_K_M
```

24GB VRAM:

```
ollama pull qwen2.5-coder:7b-instruct-q4_K_M
ollama pull qwen2.5-coder:32b-instruct-q4_K_M
```

A note on running two models: Ollama loads models into VRAM on demand and unloads them after a timeout (default 5 minutes). If your autocomplete and chat models don't fit in VRAM simultaneously, Ollama swaps them — which adds a few seconds of delay when switching between tab completion and chat. On 24GB, the 7B + 32B pairing is tight but workable. On 8GB, you're running one model at a time.

Tab Completion vs Chat: Different Models for Different Roles

Most guides treat these as interchangeable. They're not.

Tab autocomplete needs a FIM-trained model. Fill-in-the-middle means the model sees code before and after your cursor, then predicts what goes in the gap. This is what makes tab completions feel magical — the model knows you're inside a function that returns a specific type, and completes accordingly. Regular chat models don't support FIM. Use a coding-specific model (Qwen 2.5 Coder, StarCoder2, DeepSeek Coder) for autocomplete.

Chat needs an instruction-following model. When you ask "refactor this function" or "explain this error," you want a model trained to follow natural language instructions. The instruct variants of Qwen 2.5 Coder handle both FIM and instruction following, which is why they're the default recommendation.

Speed matters more for autocomplete. You tolerate a 5-second wait for a chat response. You don't tolerate a 5-second wait for every tab completion. It breaks your flow. Use a smaller, faster model for autocomplete (1.5B-7B) and a larger, smarter model for chat (14B-32B).

Continue's config makes this separation explicit with `tabAutocompleteModel` and `models`. Once you try a dual-model setup, going back to one model for everything feels broken.

Performance: Local vs Copilot

Here's the honest comparison. Local wins on privacy and cost. Copilot wins on speed and multi-file context.

Metric	Copilot (Cloud)	Local 7B (8GB GPU)	Local 32B (24GB GPU)
Tab completion latency	200-400ms	300-600ms	500ms-1.5s
Chat first-token latency	1-2s	2-4s	4-8s
Chat throughput	50-80 tok/s	30-40 tok/s	15-18 tok/s
Multi-file context	Full workspace	RAG-based (partial)	RAG-based (partial)
HumanEval score	~90% (GPT-4o)	88.4% (Qwen 7B)	92.9% (Qwen 32B)
Monthly cost	$10-19	$0	$0
Code privacy	Sent to Microsoft	On your machine	On your machine
Works offline	No	Yes	Yes
Rate limits	Yes (tightening)	No	No

The 32B local setup is competitive with Copilot on quality and beats it on every other dimension except speed. The 7B setup is "good enough" for the majority of daily coding tasks — autocomplete, docstrings, explanations, simple edits — at a fraction of the latency cost versus the 32B.

Most developers who make this switch keep both tools for a while, then gradually drop Copilot as they get used to the slightly slower cadence.

Troubleshooting

Autocomplete not showing up? Make sure Ollama is running (`ollama serve` or check the system tray). Verify the model name in `config.yaml` matches exactly what you pulled. Open the Continue output panel (View → Output → Continue) for error messages.

Completions are slow? Check if Ollama is offloading to CPU. Run `ollama ps` to see the current model and memory usage. If the model doesn't fully fit in VRAM, inference slows way down. Use a smaller model or a more aggressive quantization (Q3_K_M instead of Q4_K_M).

Chat works but autocomplete doesn't? Your chat model likely doesn't support FIM. Make sure `tabAutocompleteModel` is set to a coding-specific model (Qwen 2.5 Coder, not a general chat model like Llama).

VRAM out of memory? Close other GPU-heavy applications. Check our VRAM requirements guide to verify your model fits. Consider using a smaller quantization or model size.

Models swapping constantly? If your autocomplete and chat models together exceed your VRAM, Ollama will swap between them. Set `OLLAMA_KEEP_ALIVE=10m` as an environment variable to keep the active model loaded longer, or use smaller models that both fit in VRAM.

The Bottom Line

Continue + Ollama + Qwen 2.5 Coder is the best free Copilot replacement in March 2026. On 8GB VRAM, you get solid tab completion and basic chat. On 24GB, you get near-Copilot quality across the board.

You'll trade some latency for complete privacy and zero recurring cost. For most developers, especially those on Linux, working on proprietary code, or just tired of another subscription, that trade is easy to make.

Start with the 7B setup, use it for a week, and decide whether the 32B upgrade is worth the disk space. You'll know within a day whether local coding fits your workflow.

Next steps:

- Best coding models in detail — deeper benchmarks and model comparisons
- VRAM requirements by model — what fits on your GPU
- Ollama troubleshooting — fix common setup issues
- Local alternatives to Claude Code — for agentic, terminal-based AI coding

Get notified when we publish new guides.

Subscribe — free, no spam
