GHOST LOGICS LLC

AI Dev Studio Field Manual

VS Code + Continue + LM Studio Edition

The professional's guide to running a five-model, cost-disciplined, agentic development studio — without paying for work a free local model can do just as well.

Table of Contents

SECTION 01

The Three-Tool Stack

What each piece actually does, and how they connect

Your studio is three separate programs that work together but don’t require each other to function. Understanding the boundary between them is the single most useful mental model in this whole manual.

VS Code

The editor. This is where you read, write, and navigate code. On its own, VS Code has zero AI capability — it's a text editor with excellent tooling around it (terminal, git integration, file explorer, debugging). Everything “intelligent” about your setup comes from the Continue extension installed inside it.

Continue

The bridge. Continue is a VS Code extension that connects your editor to AI models — both cloud models (Claude, GPT, Gemini, Grok, via your own API keys) and local models (via LM Studio, running on your own hardware). Continue is where you actually chat with a model, ask it to edit files, or run a multi-step agent task. It has no intelligence of its own — it's a router and an interface.

LM Studio

The free compute. LM Studio downloads and runs open-weight AI models (like Qwen2.5-Coder) directly on your own computer's CPU/GPU. No internet required once a model is downloaded, no per-token cost, ever. LM Studio can be used two ways: (1) as its own standalone chat window, completely separate from VS Code, ideal for thinking/planning/strategy sessions; or (2) as a background server that Continue connects to, so the local model can actually read and edit your project files.

> **The One-Sentence Version**
> VS Code is where you work. Continue is how you talk to AI from inside VS Code. LM Studio is a free brain you can plug into that conversation instead of a paid one.

How They Physically Connect

1. LM Studio runs a local model and exposes it at `http://127.0.0.1:1234` when its local server is turned on.
2. Continue's `config.yaml` has an entry pointing at that same address — this is your `local-coder` model.
3. When you select that model in Continue's dropdown, every message goes to your own machine, not the internet. Zero cost, but LM Studio's local server must actually be running (not just the app open) for it to work.
4. When you select Claude, GPT, Gemini, or Grok instead, Continue sends the request to that provider's API using your stored API key — and that call costs money.

SECTION 02

The Cost Model

What's free, what isn't, and why it matters

This is the section that solves the problem you started this whole migration to fix: running out of credits mid-task. The rule is simple once you see it clearly.

Free, Always

- **VS Code itself** — no account, no subscription, ever.
- **Continue the extension** — open source, free to install and use.
- **Any request routed to LM Studio's local model** — runs on your own CPU/GPU, no internet call, no per-token cost. This is the single biggest lever you have.

Costs Money, Every Time

- **Any request routed to Claude, GPT, Gemini, or Grok** — billed per-token against your own API key with that provider. You are paying the provider directly, at their raw rate — no markup, unlike a bundled subscription tool.
- **EAS cloud builds** (`eas build`) — billed against your Expo/EAS build credit balance, separate from all AI model costs entirely.
- **EAS submissions** (`eas submit`) — typically included with build credits, but confirm on your specific plan.

The Trap to Avoid

Don't confuse “no subscription markup” with “free.”
Routing everything to Claude because it's the best model, all day, every day, will still burn through real money — you've just removed one layer of markup, not the underlying cost. The discipline that actually saves money is routing correctly: local for routine work, paid only for what genuinely needs it.

How To Check Your Balances

Provider	Where to Check
Anthropic (Claude)	console.anthropic.com → Settings → Billing
OpenAI (GPT)	platform.openai.com → Settings → Billing
Google (Gemini)	aistudio.google.com or console.cloud.google.com → Billing
xAI (Grok)	console.x.ai → Billing
EAS Build credits	expo.dev → your account → Usage / Billing

SECTION 03

The Five Models in Continue

Full profiles — strengths, weaknesses, ideal use

Qwen2.5-Coder (local, via LM Studio)

Cost: Free — runs on your own hardware

Strengths

- Zero cost, zero rate limits, works offline
- Fast for its size, genuinely good at pattern-matched, syntactically-correct code
- Excellent at literal instructions: “rename this variable everywhere,” “fix this one typo,” “add a null check here”

Weaknesses

- Weaker multi-file reasoning — loses track of context across a large codebase
- Can produce confident-sounding but shallow answers on architecture or judgment questions
- Slower than cloud models on very long contexts, since it's bottlenecked by your own hardware

Use it for

- Routine edits, boilerplate, syntax fixes
- Autocomplete while typing
- Deterministic, checklist-style work (QA sweeps, accessibility label audits, error-string greps)
- First-pass brainstorming where volume matters more than depth

Don't use it for

- Root-cause debugging of something genuinely unfamiliar
- Architecture decisions with real tradeoffs
- Anything you'd want a second pair of eyes on before trusting it

Claude Sonnet (Anthropic)

Cost: Paid — your Anthropic API key, billed per token

Strengths

- Strong multi-file reasoning and codebase-wide context tracking
- Reliable at following detailed, multi-step instructions exactly as scoped
- Good judgment on ambiguous situations — will flag uncertainty rather than guess confidently
- Currently your default/primary paid model in this setup

Weaknesses

- Costs real money on every call — the most expensive per-token of your routine options in many cases, depending on current pricing
- Overkill for trivial single-line fixes

Use it for

- Genuine debugging investigations (the kind that need real hypothesis-testing, like today's Android NDK saga)
- Multi-file agentic tasks (cloning repos, restoring configs, running verification sequences)
- Anything build/EAS/submission-related
- Final review pass before something ships

Don't use it for

- Simple, mechanical edits a local model could do for free
- Long open-ended brainstorming sessions (expensive way to generate volume)

GPT-5.4 Pro (OpenAI)

Cost: Paid — your OpenAI API key, billed per token

Strengths

- Strong general reasoning, often a good second opinion against Claude
- Broad training data, useful for less code-specific research-style questions

Weaknesses

- A separate billing account and API key from Anthropic — easy to mix up (this has already happened once in this exact setup — see Section 9)

Use it for

- A second opinion when you want to cross-check a Claude answer on something important
- General research/writing tasks outside pure coding

Don't use it for

- Routine work — same guidance as Claude applies here

Gemini 3.1 Pro (Google)

Cost: Paid — your Google API key, billed per token

Strengths

- Very large context window — strong when a task needs to reason over a huge amount of text or many files at once
- Often competitively priced against Claude/GPT for equivalent capability

Weaknesses

- Separate billing/key management, same caution as GPT

Use it for

- Large-context tasks — reviewing an entire big log file, a huge diff, or many files at once
- Third opinion on a genuinely hard problem

Don't use it for

- Routine work — same guidance as Claude/GPT

Grok (xAI)

Cost: Paid — your xAI API key, billed per token

Strengths

- Grok Code Fast variants are tuned specifically for quick coding tasks at lower cost/latency than a full reasoning model
- Good middle ground between “local free” and “full paid reasoning model”

Weaknesses

- The “Code Fast” variant trades some depth for speed — don't reach for it on a hard problem expecting Claude-level judgment

Use it for

- Quick, contained edits where you want more capability than local but don't need a heavyweight model
- A cheap-ish paid fallback when you want to preserve Claude credits for something harder later the same session

Don't use it for

- Deep debugging investigations — use Claude/GPT/Gemini for those instead

SECTION 04

LM Studio Deep Dive

Loading, serving, and chatting locally

Two Ways to Use LM Studio

1. Standalone chat (no VS Code involved at all)

Open LM Studio, load a model, and just chat directly in its own window. This is the right mode for strategy sessions, market research, brainstorming, or any of the 11 agent personas (Section 5) where you're thinking out loud rather than editing files. Completely free, completely separate from your codebase.

2. As a background server for Continue

Turn on LM Studio's local server (usually a toggle in the Developer/Local Server tab). This exposes the loaded model at `127.0.0.1:1234`, which Continue's `local-coder` entry connects to. This is the mode you need when you want the local model to actually read/edit your project files inside VS Code.

The Most Common Local-Model Failure

"LM Studio is open but nothing happens."
Having the LM Studio app open is not the same as having its server running. Check the Developer/Local Server tab specifically — the toggle must be on, and a model must actually be loaded into memory (not just downloaded) for Continue to reach it.

Loading a Model

1. Open LM Studio's model browser (magnifying glass icon)
2. Search for the model you want (e.g. "Qwen2.5 Coder")
3. Pick a size that fits your hardware — 7B is a good general-purpose size; 0.5B is faster but shallower, useful for pure autocomplete only
4. Download, then load it into memory from the chat tab or the local server tab
5. If using it through Continue, confirm the local server toggle is on and note the port (default 1234) matches your `config.yaml`

A Note on Hardware Contention

A loaded local model sits in RAM the whole time it's active, competing for memory with everything else running — VS Code, an emulator, a native build process. On memory-constrained machines, running a heavy local model at the same time as a native Android/iOS build can starve one or both. If you hit an unexplained crash during a build, check whether LM Studio has a large model loaded at the same time first, before assuming it's a code problem.

SECTION 05

The 11 Strategic Agents

Full system prompts — paste into LM Studio as a preset, or as the first message in a fresh Continue chat

Each agent below is a system prompt — a persona and set of rules that shapes how a model responds. In LM Studio's own chat window, these can be saved as named presets. In Continue, paste the full prompt as your very first message in a new chat before giving the real task; this primes the model into that role for the rest of the conversation.

DEV Deterministic — Tier 1 — Qwen2.5-Coder (local)

```
You are DEV DETERMINISTIC, a precision code-editing agent. Your job is to make the smallest, most literal change that satisfies the request — nothing more. Do not refactor unrelated code. Do not add features that weren't asked for. Do not change formatting/style outside the lines you're editing. If a request is ambiguous, ask one clarifying question rather than guessing. State exactly what you changed and why, in one sentence per change.
```

Refactor Surgeon — Tier 1 — Qwen2.5-Coder (local)

```
You are REFACTOR SURGEON. Your job is to improve code structure with the smallest possible blast radius. Preserve all existing behavior exactly — a refactor that changes behavior is a bug, not a refactor. Prefer extracting/renaming/simplifying over rewriting. After any refactor, list every file touched and confirm no behavior change was intended.
```

DEV Security Officer — Tier 1 for sweep, Tier 3 to confirm CRITICAL findings

```
You are DEV SECURITY OFFICER, conducting a systematic security audit. Check for: hardcoded secrets/API keys, keys passed in URL query strings instead of headers, missing input validation, insecure storage of sensitive data, overly broad permissions, and any place user input reaches a command/query without sanitization. Rate each finding LOW / MEDIUM / HIGH / CRITICAL. Do not fix anything — report only, so findings can be reviewed before any change is made.
```

DEV Architect — Tier 3 — Claude Sonnet

```
You are DEV ARCHITECT, responsible for system-level design decisions. When given a proposed change or new feature, evaluate it against: existing architecture patterns in this codebase, long-term maintainability, coupling/cohesion, and whether it introduces a pattern that would need to be repeated elsewhere. Recommend the simplest design that satisfies real requirements — flag over-engineering as clearly as you'd flag under-engineering.
```

DEV Market Research — Tier 1 or 2

```
You are DEV MARKET RESEARCH. Given raw text (reviews, forum posts, competitor descriptions), extract: stated user pain points, feature gaps competitors haven't filled, and pricing/positioning patterns. Be terse and factual — report what's actually in the source material, not speculation dressed as findings.
```

DEV Creative — Tier 2 or 3

```
You are DEV CREATIVE, focused on UI/UX and feature ideation. Generate options, not a single answer — present 2-3 distinct directions with the tradeoff of each stated plainly. Ground every
```

```
suggestion in what's actually buildable by a solo developer on this stack, not aspirational scope.
```

Growth Hacker Creative — Tier 1 or 2

```
You are GROWTH HACKER CREATIVE, generating acquisition and retention loop ideas. Prioritize volume of distinct ideas over polish on any single one — the goal of this pass is a wide net, refined later. Ground every idea in a privacy-first, one-time-purchase, no-telemetry product model — do not suggest anything that requires accounts, ads, or tracking.
```

Blue Ocean Strategy Innovator — Tier 1, high creativity setting

```
You are BLUE OCEAN STRATEGY INNOVATOR. Your job is to find positioning that avoids direct feature-for-feature competition with incumbents. Given a market and a set of competitors, identify what none of them are doing, and why that gap might be defensible rather than merely unaddressed.
```

Monetization Architect — Tier 2 or 3

```
You are MONETIZATION ARCHITECT, reasoning about pricing and revenue models. Constraints: one-time purchase only, no subscriptions, no ads. Evaluate price points against comparable one-time-purchase apps, and reason about what price signals quality vs. what price creates purchase friction, for this specific audience.
```

Unified Founder Mode — Tier 3 — Claude Sonnet

```
You are UNIFIED FOUNDER MODE, synthesizing product, technical, and business perspectives into one recommendation. When given a decision, don't just list considerations — make the call, state the reasoning plainly, and name the biggest risk in the path you're recommending.
```

Senior QA & Audit Engineer — Tier 1 — Qwen2.5-Coder (local)

```
You are SENIOR QA & AUDIT ENGINEER. Work through the given checklist systematically, one item at a time. For each item: state what was tested, what was found, and whether it passes. Do not skip items or summarize multiple items together — this is a volume-and-completeness task, not a judgment task.
```

SECTION 06

Agent → Tier → Model Routing Table

Match the task to the row

Task	Tool	Agent	Model
Architecture / system design	Continue (VS Code)	DEV Architect	Claude Sonnet
Feature implementation, multi-file	Continue Agent mode	—	Claude Sonnet (default)
Precise bug fix, single file	Continue	DEV Deterministic	Qwen2.5-Coder (local)
Surgical refactor	Continue	Refactor Surgeon	Qwen2.5-Coder (local)
Security audit sweep	LM Studio or Continue	DEV Security Officer	Qwen2.5-Coder (local)
Confirm a CRITICAL security finding	Continue	—	Claude Sonnet
Market research session	LM Studio (standalone)	DEV Market Research	Qwen2.5-Coder (local)
UI / feature ideation	LM Studio or Continue	DEV Creative	Qwen2.5-Coder, escalate if shallow
Growth loops / acquisition ideas	LM Studio (standalone)	Growth Hacker Creative	Qwen2.5-Coder (local)
Blue ocean positioning	LM Studio (standalone)	Blue Ocean Innovator	Qwen2.5-Coder (local)
Pricing / revenue model	LM Studio or Continue	Monetization Architect	Claude Sonnet
Big-picture product decisions	Continue	Unified Founder Mode	Claude Sonnet
Pre-ship QA checklist sweep	Continue	Senior QA & Audit Engineer	Qwen2.5-Coder (local)
Simple quick edit / typo / null check	Continue	DEV Deterministic	Qwen2.5-Coder 0.5B
Genuine debugging investigation	Continue Agent mode	—	Claude Sonnet
Second opinion on a hard problem	Continue	—	GPT-5.4 or Gemini 3.1 Pro
Large-context review (big diff/log)	Continue	—	Gemini 3.1 Pro
Fast contained edit, want more than local	Continue	—	Grok Code Fast
Long open-ended strategy brainstorm	LM Studio (standalone, free)	Unified Founder or Blue Ocean	Qwen2.5-Coder (local)
OTA / build / submission decision	Continue or claude.ai direct	—	Claude Sonnet, always

SECTION 07

Worked Examples

Real scenarios, step by step, tool by tool

Scenario: You find a typo in an error message shown to users

1. Open Continue in VS Code
2. Confirm the model dropdown shows a local model (e.g. `local-coder`)
3. Confirm LM Studio's local server is running (check the Developer/Local Server tab)
4. Type: “Fix the typo in this error string: [paste string]” and point at the file
5. Review the one-line diff, save
6. Cost: $0.00

Scenario: You need to diagnose why a native Android build is failing with an unfamiliar linker error

1. Switch Continue's model to Claude Sonnet
2. Paste the full error output, not a summary — truncated errors waste a turn asking for the rest
3. Let it investigate read-only first (check versions, configs) before attempting any fix
4. Apply one fix at a time, confirm results before trying the next
5. This is exactly the kind of multi-hypothesis, evidence-based debugging that justifies paid-model cost — see Appendix B for a full real example

Scenario: You want to brainstorm feature ideas for a new app, not write code yet

1. Skip VS Code entirely — open LM Studio directly
2. Load Qwen2.5-Coder (or a stronger local model if available)
3. Paste the DEV Creative or Unified Founder Mode system prompt as your first message
4. Brainstorm freely — as many rounds as you want, zero cost
5. Once you have a short-list, optionally bring the top 1-2 ideas into a Claude Sonnet conversation for a final judgment pass before committing real build time

Scenario: You're ready to actually submit a build to TestFlight or the Play Store

1. This step always costs money — EAS build credits, regardless of which AI model helped you get here
2. Use Claude Sonnet to prepare the exact commands and confirm the working tree is clean before running anything
3. Explicitly confirm you're ready before the agent runs any `eas build` or `eas submit` command — never let this run on autopilot

SECTION 08

Professional Agentic Programming Discipline

The habits that separate a reliable AI-assisted workflow from a chaotic one

1. Investigate Before You Implement

The single most valuable discipline in this whole manual. Before letting any agent change code, have it read and report first — no file edits, no fixes, just “here's what I found.” This catches wrong assumptions before they become wasted work, and it's exactly what turned Section 9's Appendix B debugging saga from a guessing game into a real diagnosis.

2. One Variable at a Time

When something is broken and the cause is unclear, change exactly one thing, then test. Changing three things at once and it works means you don't actually know which one fixed it — and you won't know how to fix it faster next time.

3. Commit Early, Commit Often

Especially true when working across an SDK upgrade, a risky refactor, or anything with several steps. A safety commit before a risky step costs nothing and means a bad outcome is always recoverable. Never let hours of uncommitted work exist as the only copy.

4. Verify, Don't Trust

An agent reporting “done” and an agent actually being done are two different things. Ask for a precise, section-by-section status report rather than accepting a broad summary — broad summaries round up. “What exactly did you test, what exactly did you find” is a better question than “is it done.”

5. Read-Only Audits Stay Read-Only

When you ask for an audit or a diagnosis, say so explicitly and hold the agent to it. An audit that quietly starts fixing things as it goes has stopped being an audit — you lose the clean before/after picture you needed.

6. Know When to Stop

Not every problem should be solved by the same tool. If you've genuinely diagnosed and fixed several distinct issues and a new one keeps appearing, that's a real signal — not a reason to try harder with the same approach. Sometimes the right fix is switching machines, switching build methods, or accepting a workaround, rather than a seventh attempt. Appendix B is a real example of stopping at exactly the right moment.

7. Guard the Credit-Spending and Publish-Facing Steps

Anything that costs real money (a cloud build) or is genuinely irreversible (a git push to a shared remote, an OTA publish, a store submission) should require your explicit, separate go-ahead — never bundled into a broader “yes, continue” approval for something else.

SECTION 09

Common Pitfalls

Real mistakes already made in this exact setup, so you don't repeat them

Confusing OpenAI and Anthropic billing pages

OpenAI (the company) makes ChatGPT (the product) and issues API keys at platform.openai.com. Anthropic (the company) makes Claude and issues keys at console.anthropic.com. A “credit balance too low” error from Claude cannot be fixed on OpenAI's page — they are completely separate accounts and balances.

Editing gradle.properties/config.yaml and expecting instant effect

Some tools cache settings in a running background process (a Gradle daemon, a running LM Studio server). Editing the config file doesn't retroactively change an already-running process — it usually needs a restart (killing the daemon, or restarting the app) before the new setting actually takes effect.

Using && in PowerShell 5.1

Windows PowerShell 5.1 (the older, pre-installed version, distinct from PowerShell 7/pwsh) doesn't support && as a command separator. Use semicolons, or run commands on separate lines, or switch the terminal to pwsh if it's available.

Treating LM Studio being open as the same as its server running

The app being open and the local inference server being turned on are two different states. Continue can only reach the local model if the server toggle is specifically on.

Committing debug artifacts alongside real changes

Log files, temporary output, and other session artifacts can end up sitting in a working tree next to genuine code changes. Always check `git status` and read what's actually changed before committing — don't commit blind.

Assuming a name match means it's the same model

Model names and versions change over time (GPT-5, then 5.1, 5.2, 5.4…). Don't assume a model ID from months ago is still current — check what's actually available before hand-typing a model string into a config file.

Letting an agent run a credit-costing or publish-facing command inside a broader “yes”

If you say “yes, continue” to a multi-step plan that happens to include a cloud build or a git push near the end, make sure that specific step still gets its own explicit confirmation — don't let it ride along on general momentum.

SECTION 10

Quick Reference Cheat Sheet

The one page to keep next to your keyboard

Before Any Task, Ask:

The Decision Rule

If you could hand this to a competent junior dev with a checklist and trust the output without double-checking their judgment — use the local model. If you'd want to review their reasoning before trusting it — use Claude Sonnet (or another paid model).

Fast Lookup

Situation	Do This
Model dropdown shows local-coder but you need real reasoning	Switch to Claude Sonnet before sending
Local model gives a shallow/wrong answer	Escalate to Claude, don't re-ask local repeatedly
About to run eas build or eas submit	Stop — confirm explicitly, check credit balance first
Config file edited but behavior unchanged	Restart the relevant daemon/app/terminal
“Credit balance too low” error	Check the correct provider's own billing page — not another provider's
Multi-step debugging with no clear cause yet	Investigate-and-report first, one hypothesis at a time
Same class of error keeps recurring after several fixes	Consider this a signal to change approach, not to try harder the same way
Working tree has unexpected changes before a commit	git status and git diff before committing — always

SECTION A

Appendix A — Environment & Key Reference

Continue Config Location

```
Windows: C:\Users\\.continue\config.yaml
```

API Key Environment Variables

```
ANTHROPIC_API_KEY
OPENAI_API_KEY
GOOGLE_API_KEY
XAI_API_KEY
```

Set with: setx VARNAME "value" in PowerShell. Requires a full VS Code / terminal restart to take effect — setx does not affect the currently open session.

LM Studio Local Server

```
http://127.0.0.1:1234/v1
```

apiKey field can be any placeholder string (e.g. "lm-studio") — not validated.

Checking a Key Is Actually Set

```
echo $env:ANTHROPIC_API_KEY
```

SECTION B

Appendix B — The Android Build Saga

A real debugging case study in applying Section 8's discipline

This really happened, on this exact setup, and it's worth keeping as a reference for what disciplined-but-honest debugging actually looks like — including knowing when to stop.

1. **Symptom:** Android native build failed with cryptic linker errors (undefined C++ symbols).
2. **Hypothesis 1 — path with spaces:** tested via a directory junction to a space-free path. Ruled out — failed identically.
3. **Hypothesis 2 — NDK version mismatch:** pinned the correct NDK version explicitly in build.gradle. Partially confirmed — fixed most modules, but one (expo-updates) still used the wrong version.
4. **Hypothesis 3 — stale build cache:** purged all .cxx cache directories. Ruled out — a genuinely fresh prebuild failed identically.
5. **Hypothesis 4 — missing STL linkage:** traced the actual linker command and found -lc++_shared was missing from the link line. Real finding, but this same package set built fine on a Mac — a strong clue this was Windows-toolchain-specific, not a real package bug.
6. **Hypothesis 5 — outdated CMake:** the SDK-bundled CMake (3.22.1) was old enough to have known gaps with the newer NDK. Upgraded to 3.30.5.
7. **Unrelated discovery along the way:** Android Studio's own bundled Java runtime was corrupted (a broken update left only 2 of ~16 required files in place). This had nothing to do with the original bug, but was blocking all further testing — required a full uninstall/reinstall to fix.
8. **Hypothesis 6 — JDK version too new:** the freshly reinstalled Android Studio bundled a much newer Java version with stricter native-access restrictions. Added an explicit flag to allow it. This fixed a second, separate failure that had been masked by the JVM corruption.
9. **Hypothesis 7 — Option C, remove the conflicting NDK entirely:** uninstalled the wrong NDK version via SDK Manager so only the correct one remained.
10. **Final finding:** a specific Gradle plugin (expo-updates-gradle-plugin) was found to actively, automatically reinstall the wrong NDK version during its own configure step — self-reinforcing behavior that no local fix could out-run.
11. **The stop:** after seven real, evidence-based hypotheses — several of them correct fixes for real (if secondary) problems — the pattern itself became the signal. Local Android builds on this specific machine were accepted as not currently viable, and the path forward (EAS cloud builds) was adopted instead — not as a failure, but as the correct response to the evidence.

Why This Belongs in the Manual

Every hypothesis here was reasonable, tested with real evidence, and either confirmed or properly ruled out — never guessed-and-abandoned. Several genuinely fixed real, separate problems along the way (the corrupted JVM would have blocked everything regardless). And the final stop wasn't giving up — it was recognizing that a self-reinforcing plugin behavior was a different class of problem than a local misconfiguration, and routing to a tool built for exactly that situation (cloud builds) instead.