A TERRABYTE FIELD GUIDE

Building a Roblox Game With AI

TERRABYTE SOLUTIONS

> **This is the first thing I ever typed about the game I went on to ship.** Typos and all:

```
I wnt to brainstorm for a new game called BlockterDivide. It is a fan copy
roblox game of Spectre Divide, the epic shooter wher you could control 2
bodies. It was shrouds game but it "flopped" - it was really fun :(
I want to plan all of the game systems and mechanics before going into ...
```

> That became **ShellShift**, a tactical shooter live on Roblox under **TerraByteStudios**. Every example in this guide comes from building it.
>
> **I didn't write the code.** I described what I wanted in English, pressed Play, and said what was wrong. This guide teaches that loop. The better you describe things, the bigger the thing you can make.

The Only Skill That Matters

No coding. No math. Just **describing**, which turns out to be a real skill with a ceiling you'll take months to hit.

Look at that first message again. Misspelled, rambling, and it names another game instead of explaining the mechanic. It worked anyway, because it said clearly what I wanted.

What you give it	What you get back
"make a shooter"	a generic shooter. Fine. Boring. Not yours
"a shooter where you control two bodies and swap between them"	something recognizably your idea
a page on the swap, the buy phase, the weapons, the feel	a game that keeps making sense as it grows
a **folder** of pages it reads every time	a game you're still building in three months

That's the arc of this guide. Start by saying it. Want more control, write it down. Repeating yourself, save it. Each step gives your AI more to work with, and each step buys you a bigger game.

None of this is gatekept behind programming. The people who build the best things this way aren't the best programmers. They're the ones who can see the game clearly enough to describe it.

You're not learning to code. You're learning to describe, and then to check the work.

BY THE END OF THIS GUIDE

You'll have:

- A Roblox game published and playable by other people.
- An AI agent that knows your project and picks up where it left off.
- A folder that explains your game to any AI you point at it.
- A way of working that doesn't fall apart when the game gets big.

And you'll know how to:

- Connect an AI agent to a real application and prove it worked.
- Turn a vague idea into something playable the same day.
- Tell when your AI is wrong, before it costs you a week.
- Make your own art, sound and models without learning to draw or model.

Part	What it covers
1 · The Boring Bit	what to install, what it costs, get the project folder, connect it to Studio
2 · Say It, Play It, Say What's Wrong	the loop you'll live in, and the console that shows you the truth
3 · Say More, Get More	the files that make your AI better every week instead of worse
4 · Look and Sound Like Yours	assets, images, sound, 3D models, animation, and the screen
5 · Make It Real	the walls you'll hit, publishing, real testers, and not getting cheated

Part 1 · The Boring Bit (once, then never again)

You need Roblox Studio, an AI agent that can use tools, and about ten minutes.

WHAT TO INSTALL

1. **Roblox Studio** — free from Roblox. Where the game lives.
2. **An AI coding agent.** Claude Code, opencode, or anything that can use tools.
3. **The Roblox Studio MCP server.** A small free add-on that lets your agent talk to Studio. Roblox publishes it. Follow its install page.
4. **Blender and the Blender MCP server.** Both free, and only if you want your own 3D models later. Skip for now.

Download all of that first. Nothing below works until it's on your machine.

WHAT "MCP" MEANS, AND THE ORDER TO DO THIS IN

MCP is just the standard way tools plug into an AI agent. Don't let the acronym put you off. Roblox has one, Blender has one, hundreds of apps have one. **Install one and you've learned to install all of them.**

Then three steps, and **the order matters:**

1. **Get your project folder.**
2. **Connect your agent to Studio.**
3. **Start prompting.**

Each step proves the one before it worked. Do them out of order and you'll be debugging two things at once without knowing which is broken.

WHAT THIS COSTS

Worth being upfront: this is the one part that isn't free. Your agent is doing real work on somebody's servers, and heavy days cost real money. Prices move, so check before you commit, but this is the shape of it as I write.

Your situation	What I'd use	Why
Nothing to spend	opencode pointed at a free cloud model, or one running on your own machine	Costs nothing. Works. Expect to ask for smaller pieces and check more often
Best value	**GPT Plus at $20/mo, driving the Codex CLI**	Where I'd start if money matters. Use the **CLI**, not the desktop app: it's far less restricted, and much less confusing than it first looks
Money isn't the constraint	**Claude Code on Anthropic's $200/mo plan**	The strongest overall. Anthropic wrote the MCP standard, so setup is smoothest: ask Claude to set up an MCP server and it does it, and it tells you plainly which parts you have to do yourself

Results vary, and sometimes wildly. What matters isn't the brand. It's whether your agent can hold a plan together over many steps and use tools without losing the thread. Stronger models do that well. Smaller or local ones wander off, forget what they were doing, or report finishing something they never started.

Don't let the setup be the reason you never start. You can move to a better model later, and the folder of writing you've built goes with you.

STEP 1 — GET YOUR PROJECT FOLDER

Your game needs somewhere for everything to live before you build anything. A few folders, each with a note saying what belongs in it, and one file at the top that maps them all. Your agent reads that map every session, so it always knows where to put things and where to look.

Don't build it. Download it.

```
github.com/TerraByte-Dev/ai-roblox-starter
```

Getting it takes a minute, and you don't need a GitHub account or git installed:

1. Open that link and click the green **Code** button, then **Download ZIP**.
2. Unzip it wherever you keep your stuff.
3. **Rename the folder** from `ai-roblox-starter` to whatever your game is called.
4. Start your agent from inside that folder.

That's your whole project set up. *(If you already use git, clone it or hit Use this template instead.)*

It's an empty project ready to point an agent at, and it carries more than a folder skeleton:

- `AGENTS.md` — the routing table, the standing rules, and the whole feedback loop written out as instructions your agent follows without being told.
- **A** `CONTEXT.md` **in every folder** saying what belongs there and what doesn't.
- `FEEDBACK.md` **and** `TIMELINE.md` already wired together, with the round protocol spelled out. Part 3 explains why these two matter more than anything else in the project.
- `PATCHNOTES.md`, `playtests.md`, `assets.md`, `audio.md` and a home for your own saved prompts, all with headers ready to fill in.
- `START-HERE.md` so a fresh agent, or a friend, can pick the project up cold.

It's better than what you'd get by asking. I've put real work into these files over a whole build, and the rules in them came from things going wrong. **If it saves you an afternoon, star it** so other people find it.

FOLDERS ARE ROOMS. AGENTS.MD IS THE FLOOR PLAN BY THE DOOR.

Each folder has a note saying what it's for. One file at the top lists the rooms, so nothing gets guessed at.

The pattern has a name, the **Interpretable Context Methodology**, but it's simpler than the name suggests. It's a labeled cupboard, and the labels are for a reader who forgets everything overnight.

Check the map works before you trust it. Ask your agent *"where would you file 'the player's save data got wiped'?"* If it answers straight away, you're set. If it hesitates, fix the routing table in `AGENTS.md`, not the folders.

WHY IT MATTERS MORE THAN IT LOOKS

Your agent will generate a lot of writing. Design notes, decisions, explanations of bugs it fixed, asset lists, reasons it did something a particular way.

All of that is worth keeping. None of it survives in a chat. With nowhere to put it, it gets summarized away and you ask the same questions again next week.

With somewhere to put it, it stacks. Every session adds to a project that gets easier to work on instead of harder. That's the opposite of how most solo game projects go.

You're not doing paperwork. You're building the thing that makes your AI good at *your* game.

THE ONE FILE YOUR AGENT ALWAYS READS

Every agent harness looks for one particular file in your project folder and reads it automatically at startup, before you type a word. Whatever is in that file works like standing instructions: your agent has already read it, every single time you talk to it. That's what makes it the most valuable page in the project, and it's the whole reason writing things down beats saying them.

The catch is the filename, and it changes by harness. Claude Code looks for `CLAUDE.md`. Most others, Codex included, look for `AGENTS.md`.

So do this once and stop caring which one you use. Put the real content in `AGENTS.md`, and make `CLAUDE.md` a single line:

```
Read AGENTS.md and follow it.
```

Now either harness picks it up. Switch tools next month and your project comes with you. Which of the two holds the real content doesn't matter, as long as the other one points at it. The starter ships both, already wired together, so this is done for you.

One rule that follows from this, and it catches everybody. Your agent reads that file *once*, at startup. So **any time you or your agent changes `AGENTS.md`, restart the agent.** In a terminal that's **Ctrl+C**, then launch it again. Skip it and you'll spend an hour wondering why it's ignoring a rule you can see on your own screen.

Set up the rooms before you move the furniture in.

STEP 2 — CONNECT YOUR AGENT TO STUDIO

Open your game in Studio and **leave it open**. Then, with your agent running from your project folder, paste this:

```
Call get_place_info and tell me the name of the place I have open.
```

It should say your game's name. That's the whole test, and it's worth doing before anything else.

WHY ONE READ, AND NOT THE TOOL LIST

The Roblox Studio MCP (I'll call it **the bridge**) is two pieces: a program on your computer, and a plugin inside Studio. Both have to be running.

Here's the catch. **Your AI listing the Studio tools only proves half of it works.** The Studio half can be dead and everything still looks fine from your side.

So don't trust the list. Make it read something real, and check the answer is *your* game.

IF THE ANSWER IS WRONG, OR NOTHING COMES BACK

- Check the plugin is **enabled** in Studio's **Plugins** tab.
- Make sure your game is actually open in Studio.
- **Fully restart your agent**, not just the conversation.

That fixes it almost every time. And don't start debugging your game until this passes: **a dead bridge looks exactly like broken code.**

STEP 3 — YOUR FIRST FIVE MINUTES

Don't plan. Don't design. Ask for the smallest playable version of your idea and watch it exist.

```
I want to make a tactical shooter in Roblox, like Spectre Divide — the one
where you control two bodies and swap between them.

Build me the smallest version I can actually play right now:
a flat map, one gun that shoots, and a key that swaps me between two bodies.

Nothing else yet — no rounds, no buying, no menus, no UI.
When you're done, tell me what I should look for when I press Play.
```

Then **press Play.** Something will be wrong. That's the loop starting.

Swap in your own idea and send it now. It doesn't have to be a shooter and it doesn't have to be sensible. A haunted pizza shop. A tycoon on the back of a giant turtle. A parkour course made of breakfast. The machine has no opinion about your idea, only about how clearly you described it.

Ask for the smallest playable version first. Always. One broken minute of play teaches more than an hour of planning.

Part 2 · Say It, Play It, Say What's Wrong

This is the loop you'll live in. It's a conversation, and the whole cycle is simple:

```
1. You describe what you want.        Your agent builds it.
2. Publish the game.                  File → Publish to Roblox.
3. Play it.                           Live, with a friend or on your own.
4. Write down what's wrong.           While you play, not after.
5. Hand the list back.                Go to 1.
```

That's the job. It feels clumsy for about a day, then it gets fast.

TEST ON THE LIVE GAME, NOT JUST IN STUDIO

Two ways to try something. Press **Play** inside Studio for a quick look, or **publish** and play the real thing on Roblox like anyone else.

Use Studio's Play for fast checks. Do your real testing on the published game. I did nearly all of mine there. Studio's Play is a simulation, some things don't run in it at all, and client changes need a publish before anyone else sees them.

Publish it as a Beta and stop worrying. Nobody expects a Beta to be finished. That one decision removes the fear of shipping something rough, which is what stops most people shipping at all.

ASK WHAT THINGS ARE CALLED

You don't need to learn Roblox before you start. When your agent mentions something you don't recognize, **ask**. One message answers a question you'd have had a hundred more times:

"What's the difference between a ScreenGui, a SurfaceGui and a BillboardGui, and when would I use each?"

- **ScreenGui** — flat on the player's screen. Ammo counter, menus.
- **SurfaceGui** — stuck to the face of a part, out in the world. A scoreboard on a wall.
- **BillboardGui** — floats in the world, always turns to face you. Name tags, an icon over a pickup.

Now you can ask for the right thing instead of describing it and hoping.

The learning curve here is knowing what's possible in Studio. That's all it is. So play other people's games and pay attention. **If you've seen someone do it, assume you can get it done.** Then go ask your agent how.

YOU PRESS PLAY. IT CAN'T.

Your AI can check that code is *shaped* right. It can't tell whether the game is fun, whether the gun feels weak, or whether the swap is too slow. Across the whole ShellShift build the AI started a playtest sixteen times, against thousands of other actions.

That's your job. It builds, you judge. You're the only one who knows what the game is supposed to feel like.

So never accept *"it works."* Ask what you should **see**. If it can't tell you what to look for, it doesn't know either.

IF AN EDIT SEEMS TO DO NOTHING

Stop the playtest and start it again before assuming it's broken. A running game holds on to the version it already loaded. This wastes a lot of people's afternoons.

PLAIN ENGLISH REALLY IS ENOUGH

Real messages from the ShellShift build. All of them worked:

- *"Slugger should only one shot close range. Close range id define as 58 studs in this instance, it also should have a mag size of 3, and 9 reserve."*
- *"too big, dont let it run off past the edge of the player screen"*
- *"way too big and blocks the skins ui"*

None are well spelled or nicely phrased. They work because each names **a specific thing** and **what it should do instead**.

Compare *"the gunplay feels off."* That buys you a guess. **Name the thing, name the target.**

PASTE ERRORS, DON'T DESCRIBE THEM

If Studio shows a red error, paste the whole thing. Don't summarize. The error says exactly which line broke, and retyping it as *"something went wrong with the gun"* throws that away.

THE ONE LIE YOU'LL BE TOLD

When your AI runs something in Studio, the reply says whether the *message* arrived. It says nothing about whether your *code* was happy. An error that gets caught and handed back politely comes home labeled **"executed successfully."**

Every real failure in the ShellShift build wore that label. So when your agent says something worked, ask:

```
Don't tell me it ran. Tell me what it actually returned, and tell me what
I should SEE on screen when I press Play.
```

The AI marks its own homework and always gives itself an A. You press Play.

OPEN THE CONSOLE

Press **F9** in a running game and you get the developer console: everything your code printed, every error and warning, plus live stats on performance and network. It works in a Studio playtest and in the published game.

That turns your agent into something that can leave you notes inside the game while you play.

```
I can't work out why <thing> isn't working. Add print statements showing
<the values you care about> at each step, so I can watch them in the console
while I play.

Tell me exactly what I should see printed if it's working properly.
```

Play, press F9, read, paste back what you saw. This is how you stop describing symptoms and start reporting values. *"The swap feels broken"* buys you a guess. *"It prints the request but never the confirm"* tells your agent exactly which half is dead.

Have it take the prints back out when you're done, or the console fills with noise and stops being useful.

Part 3 · Say More, Get More

Here's the biggest jump in what you can build.

A sentence buys a sentence's worth of game. A page buys a page's worth. Everything your AI knows about your game, it knows because you told it. Anything you told it in a chat message is gone the moment the conversation gets long enough to be summarized.

So stop typing things and start **writing them down**. Same English, same describing. It just lasts.

WHY A FILE BEATS A MESSAGE

Every AI has a memory limit. When a long conversation fills it, the whole thing gets squashed into a summary and details fall out. In the ShellShift build that happened roughly every forty messages.

Anything in a file survives, because your agent re-reads it. Anything you only said out loud does not. You'll notice, because you'll be explaining your economy for the third time on a Thursday.

This is the difference between an assistant who forgets you every morning and one who's been on the project since day one.

KEEP THE MAP SHORT AND CURRENT

You made `AGENTS.md` when you set up the floor plan. Your agent reads it automatically every session, which makes it the most valuable page in your project and the easiest to wreck.

Put in only what you never want to say twice. **Keep it under a page.** A rule buried on page four is a wish. If you can't read the file aloud in a minute, cut something.

Add standing rules as you find them. You'll know which: they're whatever you just said for the second time.

```
Add these to the Rules section of AGENTS.md, and follow them from now on:

- Ask me before adding a whole new system. Small changes, just do them.
- The server decides who took damage and what they own. Never the client.
- One system, one script. Don't pile new features into an existing file.
- If you're not sure what I meant, ask instead of guessing.
- When we learn something worth keeping, write it into the right folder,
  and if it doesn't have a home yet, tell me and update the routing table.
```

DESCRIBE THE WHOLE GAME ONCE

`AGENTS.md` holds the always-true things. The **design doc** is where you get to dream, and it's where most people leave the most on the table.

Be greedy here. Describe the game you actually want, not the version you think is realistic. Asking for everything and cutting back costs you nothing.

```
I want to properly design this before we build more. Ask me questions
first — as many as you need — then write it up as 01-design/GAME.md.

Here's the start: a 3v3 tactical shooter where each player controls two
bodies and swaps between them. Plant and defuse. You buy weapons at the
start of a round with money you earned last round.

Before you write anything, go and RESEARCH how real games in this genre
actually handle this, and keep it grounded — no inventing. What rig do
Roblox FPS games use, and what are the tradeoffs?

Cover: the round loop, what you can buy and roughly in what order, what
makes it fun in minute one AND on day three, and how a match ends.

Then be honest — which of these is going to be the most annoying to
build, and what should we cut?
```

MAKE IT INTERVIEW YOU

That *"ask me questions first"* line does a lot of work. Left alone, an AI fills your gaps with the most average answer available, and average is what makes a game forgettable.

Made to ask, it surfaces things you hadn't decided. In ShellShift that's where the good parts came from: a request-a-buy feature so teammates can buy your loadout when you're broke, a hidden rank running under casual play from day one, and the name **ShellShift** itself, which started as the word for the body you *aren't* controlling.

Those answers are your game. Nobody else's shooter has them.

MAKE IT RESEARCH, THEN MAKE IT ARGUE

Research first. *"Go find out how real games solve this, and keep it grounded"* is a completely different instruction from *"design this."* One gets you an informed answer. The other gets you a plausible one.

Then make it attack the design. One line, worth more than anything else here:

"Now critique this. What's boring? What contradicts something else? What can't realistically be built by one person? Give me a numbered list of what to cut."

An AI asked to please you says your idea is great. An AI asked to attack it finds the hole in your economy before you build it.

PROVE THE RISKY BIT FIRST

Whatever your game leans on hardest, build a throwaway test for that one thing before you build anything that looks like a game. No menus, no art, no content. Just the risky part, and a number.

ShellShift's whole idea was the two-body swap, so the first real session built nothing else. It had to answer a question, not a feeling: does the swap still feel instant when the connection is bad?

The answer came back as **swap confirmed in 66 milliseconds plus your ping, holding steady as ping gets worse.** That's a go. Had it come back slow or jittery, this was a different game, and finding that out on day one costs a night instead of a month.

```
Before we build anything real, I want a test that answers a number.

Build a throwaway place whose ONLY job is to prove <your riskiest thing>.
Nothing else — no menus, no art, no scoring.

Then let me try it myself under <the bad conditions you're worried about>.

Print the measurement at the end. I want a NUMBER, not "it works".
Write what you built and what it showed into 02-build/TIMELINE.md.
```

A test that answers a number beats ten that look like the game.

BATCH WHAT YOU NOTICE

This habit saves the most time, and ShellShift got it most wrong.

You press Play, spot something small, type *“make the ammo counter smaller”*, and wait. Then you press Play again, spot the next thing, wait again. Most messages in the real build were tiny one-liners like that. Hundreds of them. Each cost a full round trip while the AI worked out where everything lived.

Play with a file open instead. Write as you go. Don’t stop to send.

```
# FEEDBACK — round 3

## Guns
- Slugger one-shots way past close range. Should fall off hard after ~58 studs.
- No reload sound on the Ranger.
- Muzzle flash is way too big, blocks half the screen.

## Buy phase
- I can still shoot during the buy phase. Shouldn't be able to.
- Prices are unreadable on a phone.

## Swap
- Swapping resets my camera to face north. Keep my look direction.
```

Then send **one** message:

```
Read 02-build/FEEDBACK.md and work it top to bottom.

For each item: make the change and tick it off in the file with one line
saying what you did.
If something's unclear or you disagree, leave it unticked and write your
question underneath instead of guessing.
Don't touch anything that isn't in the file.

Once I've confirmed it plays: append a dated summary of what changed to
02-build/TIMELINE.md, then wipe FEEDBACK.md back to an empty heading
ready for the next round.

Then tell me what to look at when I press Play.
```

TWO FILES, TWO DIFFERENT JOBS

`FEEDBACK.md` **is a whiteboard.** One round of notes, and only one. You fill it while you play, your agent works it, you confirm it plays, then it gets **wiped clean** for the next round. It’s never a history and it should never grow.

`TIMELINE.md` **is the log.** Append-only, newest at the top, nothing ever deleted. When a round finishes, a dated summary lands here and stays.

Keeping them apart stops both from rotting. A whiteboard nobody erases becomes an unreadable pile you stop opening. A log people keep editing stops being something you can trust.

WHY THIS BEATS ONE-AT-A-TIME

It works out where everything lives once instead of forty times. Related fixes get made together and stay coherent. You stay in the game instead of bouncing in and out. When the round closes you get a real record in `TIMELINE.md` without having written one.

The *“leave it unticked and ask”* line is the quiet hero. It turns a silent wrong guess into a question you can answer, which is the difference between catching a misunderstanding now and catching it three rounds later.

Some things still deserve their own message: anything actually broken, and any change of direction. Feedback files improve what exists. They don’t change your mind about what you’re making.

STOP RETYPING YOURSELF

By week two you’ll notice you keep typing the same kinds of things. That’s a signal. **Anything you’ve explained twice should live in a file.**

LET THE ROOMS GROW

The floor plan isn't fixed. When you notice writing with no obvious home, like weapon balance notes or everything about one map, that's a new room:

"Make a folder for this, give it a CONTEXT.md, and add it to the routing table in AGENTS.md."

The routing table is what everyone forgets to update. A folder your agent doesn't know about is one it won't use, and you end up with an orderly cupboard nobody opens.

SAVE YOUR GOOD PROMPTS

When a prompt works well, keep it. A file of your own best prompts beats any list somebody else wrote, because yours are shaped around your game.

"Whenever a prompt of mine works especially well, save it to `02-build/prompts.md` with a one-line note on what it's for."

Six weeks in, that file is what makes you fast. It's the difference between starting from scratch every session and starting from everything you already worked out.

Tell it to remember without being asked. One line in `AGENTS.md` changes how much you have to hold in your head:

```
When we learn something worth keeping — a bug we fixed, an asset ID, a limit
we hit, a decision I made — write it down in the right file straight away,
in the same reply. Don't ask me first, and don't wait until I remember to ask.
```

Spend that effort on **things you corrected**. Plain facts can be looked up again. A correction is the one thing your AI won't rediscover on its own.

Everything you write down, you never have to say again.

Part 4 · Make It Look and Sound Like Yours

Same move as everything else. **Describe the thing, get the thing.**

THE ONE RULE FOR EVERYTHING YOU ADD

Models, pictures and sounds all work the same way. You upload the file, Roblox stores it and hands back **a number**, and your code only uses that number. **The number is the asset.** Lose the list and your files are unreachable even though they still exist.

```
Whenever I give you an asset ID, add a row to 03-assets/assets.md with:
name | id | what it is | where it's used | today's date

Always use the name in code, never the bare number.
If you need an asset that isn't in that list, stop and ask me — never invent an ID.
```

Two things that will confuse you. A new upload comes back looking *failed* until Roblox's moderation clears it, so upload the day before you need things. And Roblox scans asset *names* for rude words hidden inside longer words. One innocent ShellShift camo got blocked over three letters in the middle of a word. Give files boring names, keep the pretty ones in your code.

PICTURES AND TEXTURES

Whatever image generator you like, the loop is: **describe it → generate → name it → upload it → paste the number back.**

DESCRIBE IT LIKE YOU WOULD TO A PERSON

"A camo texture" gets you a stock camo. What you want is closer to *"a tiling digital camo in dark navy and gray, high contrast, no logos or text, readable wrapped around a gun barrel on a phone screen."*

Say what it's **for**, what it must **not** have, and where it will be **seen**. Those three do most of the work.

Keep the descriptions in a file in `03-assets/`, one line each. ShellShift ended up with over a hundred camos. They look like a set because the briefs lived in a file instead of a conversation.

SAY WHAT TO TAKE FROM A REFERENCE

If you paste a picture as an example, say what you want from it. A picture alone is ambiguous. The shape? The colors? The mood?

"Use this for the color palette only, ignore the shape" is a different instruction from *"match this silhouette."* The one time a reference went in without instructions, the result came back rejected for not resembling it.

Tag the jobs you'll do yourself. When your agent writes a to-do list, have it mark which items need art or sound. Then you can knock out a batch in one sitting.

SOUND AND MUSIC

Sound is the fastest way to make a game feel finished, and beginners skip it. Two jobs, and different tools are good at each.

WHAT I'D USE

Music: Suno. It's what I used for ShellShift, and it's strong at full tracks. Menu loops, round-win stings, anything with structure.

Sound effects: ElevenLabs. Better than Suno for short one-shot sounds. Gunshots, reloads, button clicks, footsteps.

Describe sounds the way you'd describe them to a friend. *"A sharp punchy shotgun blast, quite dry, no long tail."* That works.

KEEP THEM GROUPED

Write the sound briefs into one file, grouped by what they're for, so a whole family comes out sounding related instead of one at a time. The prompt for that is below.

Same rule as textures: the brief lives in a file, not in a conversation. That's what keeps forty sounds sounding like they belong to one game.

```
Make 03-assets/audio.md. Group by what the sound is for: weapons, UI,
footsteps, round events, music. One line per sound saying what it should
sound like and how long it should be.

Match every name to its row in 03-assets/assets.md.
```

The trap that gets everyone: you test alone, hear everything, publish, then play with a friend and half the sounds are missing. Sounds must sit somewhere both players' games can reach, and the server has to decide when they play. **Always test audio with a second person on the published game.**

MODELS AND ANIMATION

Skip this until you want it. Free models from the Roblox library are a respectable way to ship your first game.

Be warned: this is the hit-or-miss part. Everything else in this guide worked reliably for me. Getting good 3D models out of an AI took the most attempts and produced the most rework. Budget more patience here than anywhere else.

BLENDER, THROUGH ITS MCP

Blender is free, and its MCP server lets your agent build models by writing code, so you never learn to model. Same pattern as Studio: install, connect, check it's alive, work in small steps.

One trap to know first. There's a tool that screenshots the Blender viewport so your AI can "see" the model. **It often returns a black image and reports success.** Have it render to a file and open that instead.

And have it **print numbers**. How many pieces, how big, does the bottom sit on the ground. A picture flattens things. Two parts that look joined can have a gap you'd only see from the side. Every ShellShift weapon got re-modeled after parts were found floating on renders that had already passed.

Write yourself a style skill. If you keep getting models that don't match, put your art style in a file your agent reads every time: proportions, chunkiness, how much detail, reference images. A consistent look is worth more than any single good model.

DECIDE THE COLORING BEFORE YOU MODEL

The expensive mistake, and it bites weeks after you cause it.

In Roblox a texture covers a whole mesh piece. A gun built as one merged lump takes exactly one camo, across scope, grip and barrel together.

Want parts colored separately, for camos or team tints? **It has to be built as separate named pieces from the start.** All seven ShellShift weapons were re-modeled mid-project over this, after eight had already shipped as single meshes.

One piece, one color. Split by material, not by name. Freeze names before export, because renaming afterwards breaks the model, the camo list and your asset list at once, silently.

ANIMATION: START WITH VIDEO

I never got animation where I wanted it. If you're starting out, **use Studio's built-in video-to-animation capture.** You record yourself doing the motion and it turns that into a Roblox animation. It's low effort and it's good enough to get a game feeling alive, which beats a perfect idle you never finish.

Getting models into Roblox: export as `.glb`, import through Studio's Asset Manager. One warning that wastes two attempts: **Roblox ignores plain colors set on materials in Blender.** They arrive flat gray. Either bake colors into a real texture, or import plain and have your agent color the pieces by name in code.

THE SCREEN (WHERE YOUR AI IS BLINDEST)

Your agent can't see your screen. It writes menus from imagination and gets them wrong in ways only eyes catch. Every menu in ShellShift shipped without the AI ever seeing it.

That makes the interface the best use of your feedback file. Visual polish arrives as twenty small observations, which is what batching is for. Play with the file open.

```
Add a UI section to AGENTS.md and follow it from now on:
- Size and position everything in SCALE, not pixels. Pixels only look right
  on the exact screen they were made on.
- Assume a phone, held in two hands, with thumbs covering the bottom corners.
- Nothing may cover the top-right corner — that's Roblox's own menu button.
- Stick things to a screen edge on purpose, never by accident.
- Every screen gets its own script from line one.
```

Describe it like you're describing it to a friend. That's the technique.

Part 5 · Make It Real

THREE WALLS IF THE GAME GETS BIG

Roblox has hard limits, and one decision avoids all of them: **one system, one script.** In ShellShift the only two scripts that ever hit a wall were the two written before that rule existed. The forty-two written afterwards never did.

The wall	What it means	The fix
200 locals	one *function* holding too many named things. A 2,700-line file is fine	move the settings at the top into their own file
255 registers	one function juggling too much at once	**split the function.** Moving settings out won't help
200,000 characters	a single script got too long, and writes quietly stop working	split it. Past **205,801** characters it became impossible to edit at all

```
Rule for this session: one system, one script. Anything new gets its own
script from line one — don't add it to an existing file.

Before you write anything, tell me which file it belongs in and why.
If the answer is "the big one", propose splitting it first.
```

SAVING, PUBLISHING, NOT LOSING WORK

SAVE AND PUBLISH ARE DIFFERENT

Save (Ctrl+S) keeps your changes. Nobody else sees them. **Publish** (File → Publish to Roblox) puts them in the live game.

"I swear I fixed that" is usually a change that got saved and never published.

Reopen your game with **File → Open from Roblox**, not from a local file, or you'll edit a stale copy and wonder why nothing sticks.

Publish a version you can roll back to before each big change. Then every step is one undo from a working game.

WHEN YOUR GAME BECOMES TWO PLACES

Most real Roblox games are a lobby plus a match place. ShellShift is exactly that, and three things bite.

The first place you make is permanently the entry point. Decide early, it can't be changed later.

Only one place is open at a time and the bridge only sees the open one, so tell your agent which you're in.

If both places save player data, **merge on save**. Never overwrite, or one place wipes what the other knew.

PUT IT IN FRONT OF A REAL PERSON

Publish early and get **one person who isn't you** to play it. Hand it over and watch, don't narrate. You're the worst possible tester of your own game because you know where everything is.

Watch what they do before you listen to what they say. Where they get stuck is worth more than their opinion afterwards.

Test the end of your game early. Your best players get there faster than you think. ShellShift's testers hit the endgame in about a day and a half, and the whole progression had to be rebuilt and then redesigned three more times. Put your endgame in front of somebody before you tune the middle.

```
After each playtest, add to 04-ship/playtests.md: the date, what we tested,
what broke, and what felt wrong. One short entry, without me asking.
```

Your patch notes write themselves. You're already keeping `TIMELINE.md`. When you publish an update, turn the last stretch of it into something players can read:

```
Read 02-build/TIMELINE.md since the last release. Write player-facing patch
notes from it into 04-ship/PATCHNOTES.md, newest at the top.

Plain language, no internal file names, grouped by what a player will notice.
```

`TIMELINE.md` is for you and your agent. `PATCHNOTES.md` is for players, and it's what you paste into your game's update page or your Discord. Keeping a log stops feeling like admin the first time it hands you a week of patch notes for free.

THE TEN MINUTES THAT PROTECT YOU

Do this before strangers play. AI-written code copies the shape of working code faithfully, **including the shape that forgot to check anything.** It'll be perfect for honest players and fall apart for the first person who pokes at it.

The best catch a review ever made in ShellShift was exactly this: a round-state flag stayed on through warmup, letting players shoot during the buy phase. Nothing errored. It just quietly wasn't the game.

```
Audit this game as if you were trying to cheat at it:

- Every message my device can send the server: does the SERVER check it's
  allowed? List each one and what it checks.
- Anything that matters — damage, money, unlocks — is the SERVER the one
  deciding it? Show me where.
- What could someone send directly to give themselves something free?

Give me the findings as a list I can fix one at a time.
```

Never trust the player's device. It can be made to say anything. The server decides what's real.

Roblox explicitly allows AI-written code and publishes the Studio MCP itself. There's nothing to apologize for. Checking it is your job.

WHEN SOMETHING MAKES NO SENSE

Read this once now, and again the first time something is inexplicable.

What you're seeing	What's happening	What to do
The AI says everything is empty while the game plays fine	Its code runs in the *editing* copy, not the running game	Check live state from inside a playtest, not through the bridge
A part exists but players can't see it	Roblox loads distant things on demand and this one's contents arrived late	Add contents before it loads in, or turn that off while prototyping
The player's screen is totally empty	Nothing spawned for the camera to follow, so nothing loads	Spawn something to look at, or turn off on-demand loading for now
The player changes something, the server never notices	Changes on a player's device don't travel to the server by themselves	Send a message to the server, let the server make the change
Something works, then randomly doesn't	Handing control of a part to a player can quietly lose a race	Re-try until it sticks instead of assuming once was enough
Frame rate got *worse* after "optimizing"	A known Roblox bug. The optimization keeps doing the hidden work anyway	Undo it. It can cost more than it saves
A model is see-through from one side	The model has no inside faces	Fix the facing in Blender, preview with one-sided rendering on
A brand-new asset says it failed	Waiting for moderation, not broken	Wait. Upload a day early next time

Check the thing itself, never the label on the reply.

THE WHOLE GUIDE, HONESTLY

- **You describe it in English.** The better you describe, the bigger you can build.
- **Ask for the smallest playable version first**, then fix it by playing it.
- **Write down anything you've said twice.** A sentence buys a sentence. A file buys a project.
- **Make it research, then make it argue** with your design, before you build it.
- **Batch what you notice in a file** instead of sending forty one-line messages.
- **You press Play.** The AI marks its own homework and always gives itself an A.

DO THIS TODAY

Look at that first message on the cover again. Misspelled, rambling, no plan. **That's the bar.**

Grab the starter — `github.com/TerraByte-Dev/ai-roblox-starter` — open Studio, connect the bridge, and send one message describing the smallest playable version of an idea you like. Not a good idea, one you'd want to play. Then press Play and tell it the first thing that's wrong. That's the whole loop, inside an hour.

Come show me — `discord.gg/p9TVXqfjxD`. Post the idea and the first screenshot, however broken. Broken first attempts are my favorite thing in there.

WHAT ELSE THIS UNLOCKS

This guide gets you started. The more you make, the more you'll find you can make.

Here's the part worth noticing. **If you can connect an MCP to an agent and actually use it, you've already done the hard part**, and that skill has nothing to do with Roblox.

Google has an MCP. Want an AI that triages your inbox, reads your Drive, or builds you a spreadsheet from a description? You should be able to set that up right now. Same steps you already followed.

Blender has one, and you met it in Part 4. Push further: write it custom skills and build whole 3D scenes, not just game props.

Hundreds of other apps have one, and more show up every month. Your notes, your calendar, your music library, your code, your smart home.

The barrier to entry is at an all-time low across the board. You just cleared it for one thing. Go clear it for something else.

Go see what I made with only AI tools. ShellShift is live on Roblox under **TerraByteStudios.** Is it perfect? No. I'm a technologist at heart and I bounce between hobby projects. But given it was built entirely with AI, I think it's pretty impressive, and it stands as a proof of concept for anyone thinking about trying this.

And go learn from Jake Van Clief. If you want a grounded framework for building with AI instead of a pile of tips, his teaching is the best free material out there — **Clief Notes**, `skool.com/cliefnotes`. The folder-as-memory idea running through this whole guide comes from his and David McDermott's **Interpretable Context Methodology** (2026), `arxiv.org/abs/2603.16021`.

Find me in the Discord — discord.gg/p9TVXqfjxD · **@terrabyte.dev** · terrabytesolutions.com