Generative Production

A Study Guide for Filmmakers Learning Generative AI

2026-04-21

Table of contents

Introduction

This is a single-document version of the Generative Production curriculum — foundations, generative media, vibe-coding for filmmakers, and the practical matters that sit under both. It is assembled from the same markdown source that powers the website at generative-production.com. Read straight through for a complete pass, or jump to any section via the table of contents.

A note on the reading order: this bundle is flat on purpose. Each chapter below is the same short explainer you would find on a topic page of the site, in the order a teacher would assign them. Internal cross-references use the site URLs, so some links will take you out of this file to the web version.

Part I — Foundations. The umbrella claim: media generation and code generation are the same discipline. The following four chapters earn that claim.

1 What is Generative Production?

In one sentence: Generative Production is the discipline of making creative work — films, images, sound, and the code that assembles them — by collaborating with models that generate candidates from your prompts.

Why it matters for filmmakers: Generative tools are not a new kind of camera or a new kind of codec. They're a new kind of collaborator, and the skills that make the collaboration work are the same across media and code. Treating them as one discipline keeps you from re-learning the same lessons in four different contexts.

1.1 Key ideas

- **Media generation and code generation are the same move.** Generating an image with a diffusion model and generating a function with a coding model differ in output, not in method. The filmmaker prompts, the model produces candidates, the filmmaker selects and iterates.
- **The meta-skills are universal.** Specificity in prompts, disciplined iteration, keeping track of what you made, and knowing what you're allowed to do with the result — these apply to every generative tool.
- **Outputs differ, workflows rhyme.** A video model and a coding agent look nothing alike on the surface. But the rhythm of *try, read, refine, commit* is the same.
- **The collapsing middle.** Traditional film-production stages were defined by who did what and when. Generative tools dissolve these boundaries: previs and final pixel become the same generation; the person writing the script is the person directing the shot. Organizing your learning around the tools, not the stages, ages better.

1.2 Common mistakes

- Treating media generation and code generation as unrelated hobbies. They reinforce each other.
- Assuming the skill is "getting the tool to work." The skill is getting the *collaboration* to work — the tool is a starting point.
- Expecting a single prompt to produce a final product. Generative Production is iterative; your first output is almost never your best.

1.3 Related pages

- `prompting-principles.md` — the universal skill.
- `iteration-and-selection.md` — how to work with the fact that the tool gives you many candidates.
- `tracking-your-work.md` — staying reproducible across months.

2 Prompting Principles

In one sentence: A prompt is a set of instructions that tells a generative model what to make; prompting well is the difference between "the model won't cooperate" and "the model is a collaborator."

Why it matters for filmmakers: Every generative tool you use — image, video, audio, code — takes a prompt of some kind. The specifics vary wildly between tools, but the underlying skill transfers. A filmmaker who prompts well in Midjourney will prompt better in Claude Code on day one.

2.1 Key ideas

- **Specificity beats cleverness.** The most common prompt mistake is vagueness. "A cinematic shot" gives the model permission to do anything; "a medium shot, 35mm, golden-hour backlight, shallow depth of field, subject in profile" narrows the space. Narrow spaces produce useful outputs.
- **Tell it what, not how, unless the how matters.** For open creative work, describe the outcome you want and let the model choose the approach. For technical work (code, specific camera moves), describe the approach because the outcome depends on it.
- **Show, don't only tell.** Reference images, code examples, or prior outputs often teach the model faster than adjectives. Modern tools accept multiple modalities of reference — use them.
- **Constrain to create.** Counter-intuitively, adding constraints (a time budget, a palette, a rule like "no text in image") usually improves output. Constraints give the model edges to push against.
- **Iterate in small moves.** Change one thing per attempt. If you rewrite half the prompt between generations, you won't know which change helped. This is the same discipline as debugging code.
- **The prompt is not the whole context.** In media tools, seeds, samplers, and reference images shape the output alongside the prompt. In coding tools, project files, `CLAUDE.md` / `AGENTS.md`, and conversation history do the same. The prompt is one input among several.

2.2 Common mistakes

- **Prompt-and-pray.** Writing one long prompt, hitting generate, being disappointed, and writing a completely different long prompt. Iterate on a working base instead.
- **Over-stuffing adjectives.** Piling on descriptors ("epic, cinematic, masterpiece, 8K, trending") usually adds noise, not signal. Keep the prompt specific instead.

- **Ignoring the tool's own prompting guide.** Every major tool publishes one. They aren't generic — a Midjourney prompt doesn't translate directly to Runway, and neither translates directly to Claude. Read the guide.

2.3 Try this

Pick a tool you've used and write three prompts for the same intended outcome: one vague, one specific, one specific-with-constraints. Compare results. Do this before writing more elaborate prompts — the exercise recalibrates what "specific" actually means.

2.4 Related pages

- `iteration-and-selection.md` — what to do once you have candidates.
- `../generative-media/craft/` — prompting refinements for media (planned).
- `../vibe-coding/working-with-models/` — prompting refinements for code (planned).

3 Iteration and Selection

In one sentence: Generative tools produce candidates; the craft is in deciding which candidate to keep, which to iterate on, and when to stop.

Why it matters for filmmakers: Film has always been about selection — you shoot coverage, you cut the film. Generative tools dramatically cheapen the "coverage" step, which means selection becomes the rate-limiting skill. The filmmakers who do well with generative tools are the ones with taste and discipline in culling, not the ones with the best prompts.

3.1 Key ideas

- **The model is a coverage machine.** A text-to-image model gives you four candidates per generation. A coding agent gives you several attempts at the same function. A video model gives you takes. In all three cases, your job is directing the model toward better candidates *and* editing them down.
- **Cull early, cull often.** Keep every generation and you drown. Mark the one or two candidates worth building on, discard the rest, and iterate. This is the opposite of "save everything in case" — you want active selection.
- **Iterate on the best candidate, not the next idea.** The temptation is to start over when the first batch disappoints. Usually the better move is to take the best of what you got and push in the direction it suggests. This is how working with a model becomes a conversation instead of a gamble.
- **Set a budget before you start.** "I'll spend 20 minutes on this shot" or "I'll give this function three attempts." Without a budget, iteration becomes a slot machine.
- **Know when to stop.** Generative tools are especially good at tempting you to keep iterating forever — the next batch might be perfect. A 75%-quality shot that ships beats a 95%-quality shot that never does.

3.2 Common mistakes

- **Keeping every generation.** The archive becomes noise.
- **Starting over after every disappointing batch.** You lose the direction you were building toward.
- **Iterating without recording what changed.** Without a log, you can't reproduce the good attempts. See `tracking-your-work.md`.

3.3 Related pages

- `prompting-principles.md` — how to aim the model in the first place.
- `tracking-your-work.md` — how to remember what worked.

4 Tracking Your Work

In one sentence: Generative work is only as reproducible as your records of it; a prompt book for media and a git history for code are the same discipline with different files.

Why it matters for filmmakers: Without records, a great generation is unrepeatable — you can't extend it, match it, or recover it if a file goes missing. And a project that can't be rebuilt in six months is effectively disposable. Tracking your work is what turns generative output from a trick into a craft.

4.1 Key ideas

- **Capture the inputs, not just the outputs.** For media: prompt, seed, sampler, model, reference images, tool version. For code: the conversation that produced it, the prompt or rules file, the model. The output alone is not enough — you need to know how to get back to it.
- **One source of truth per project.** For media work, keep a prompt book (a spreadsheet or a markdown file) alongside the finished frames. For code, the git repository is the prompt book — commit messages carry the *why*. Either way, the record lives next to the work.
- **Save seeds and commits, not just final picks.** The abandoned attempts carry information about what almost worked. A commit you revert is still a record; a seed you noted is still a recipe.
- **Models change under you.** A model version that produced a look last year may no longer exist. Note the exact version when you use it and, for production work, consider whether you need to archive the model itself.

4.2 Common mistakes

- **Trusting the cloud tool to remember for you.** Most generative services retain history for some window and then discard it — or change the retention policy without warning. Export locally.
- **Losing the prompt when keeping the image.** A folder of PNGs with no prompts attached is a gallery, not a record.
- **Mixing commits and generations.** For code projects, don't let the agent commit on your behalf without reviewing the diff — a forgotten commit of an experimental generation is hard to untangle later.

4.3 Related pages

- `iteration-and-selection.md` — what you're tracking is the iteration history.
- `../practical/` — reproducibility at project scale (planned).

Part II — Generative Media. Pages on producing images, video, audio, and spatial work with generative tools.

5 Concepts

The conceptual spine of this section. These pages describe *what the tools are* and *how they work*, independent of any specific product.

5.1 Contents

- `model-basics/` — diffusion intuition, LoRAs, seeds, CFG, samplers, conditioning, and the terms every media generator encounters.
- `modalities/` — one page per modality (image, video, audio, spatial) with sub-topics inline.
- `tools-and-vendors/` — the landscape: local vs cloud, hardware realities, cloud compute providers, interfaces, model zoos.

6 Model Basics

The foundational vocabulary of generative media models. Read top-to-bottom for a first pass, or dip in for a specific term.

6.1 Planned topics

- **Diffusion intuition** — how generation actually works, no math.
- **Seeds** — the reproducibility lever. (page)
- **Samplers & schedulers** — what they do, which to pick when.
- **Steps & CFG** — how much to think and how hard to listen to the prompt.
- **LoRAs** — lightweight style/subject adapters.
- **Quantization & file types** — why the same model ships in many sizes.
- **Resolution & aspect ratios** — native sizes and why going off-spec breaks things.
- **Conditioning** — ControlNet, IPAdapter, depth/pose/edge. The "steering wheel."
- **Embeddings & textual inversion** — teaching a model a new word.
- **Checkpoints vs LoRAs vs embeddings** — disambiguation.
- **Fine-tuning vs LoRAs vs prompting** — when to reach for which.

7 Seeds

In one sentence: A seed is a number that, combined with your prompt and model, makes a generation reproducible — change the seed and you get a different image from the same prompt; keep the seed and you get the same image.

Why it matters for filmmakers: Seeds are the reproducibility lever. Without them, a good generation is a lucky accident. With them, you can iterate on a specific look, match a previous shot, or hand a project to a collaborator and have them reproduce your work exactly.

7.1 Key ideas

- **The seed is a starting point for the noise.** Diffusion models start from random noise and denoise it toward your prompt. The seed controls which random noise — same seed, same starting noise, same output (given identical prompt and settings).
- **Seeds are not magical quality knobs.** A "good seed" for one prompt is arbitrary for another. Don't chase seeds the way you might chase a prompt revision.
- **Locking the seed lets you iterate on the prompt.** This is the main working use: lock the seed, change one word in the prompt, see what that word does. Without a locked seed, you can't tell what's a prompt effect and what's seed variance.
- **Seeds only reproduce if everything else matches.** Same model version, same sampler, same step count, same image size, same conditioning. Change any of these and the seed-level reproducibility breaks. This is why you also track model and settings, not just the seed.
- **Write down the seeds for final picks.** If a generation makes it into the film, its seed belongs in your prompt book alongside the prompt and model version.

7.2 Common mistakes

- **"Seed hunting."** Rolling random seeds with a static prompt, hoping for a magic output. This is the slot-machine failure mode.
- **Not recording seeds on final generations.** You can't extend, match, or recover a look if you didn't write down the seed.
- **Assuming seed reproducibility across services.** Two services can both accept a seed and still produce different output — they likely use different models, samplers, or precision. Reproducibility is per-tool.

7.3 Related pages

- `../../../foundations/tracking-your-work.md` — where seed logs live.
- `../../../foundations/iteration-and-selection.md` — why locking a seed changes the loop.
- *(Planned: samplers-and-schedulers, resolution, model-versions.)*

8 Modalities

One page per modality. Sub-topics live inline because the ideas cluster.

8.1 Planned pages

- **Image** — txt2img, img2img, inpaint/outpaint, upscaling, style transfer.
- **Video** — txt2vid, img2vid, vid2vid, frame interpolation, consistency, clip-length limits.
- **Audio** — dialogue and voice cloning, ADR, lip-sync, Foley/SFX, music/score.
- **Spatial** — 3D generation, NeRF, Gaussian Splatting, camera tracking, plate reconstruction.

9 Tools and Vendors

The landscape, framed categorically so the pages age gracefully. This folder tells you *how to reason about* a tool, not which one to pick today.

9.1 Planned topics

- **Open weights vs closed APIs vs open source** — what each actually lets you do.
- **Service tiers** — infrastructure vs workflow vs full-service, and how to tell which layer you're on.
- **Local vs cloud** — privacy, cost, control, ceiling trade-offs.
- **Hardware realities** — GPU/VRAM, Mac vs PC, minimum viable rig.
- **Cloud compute** — Colab, RunPod, Modal, Replicate, Fal as categories.
- **Interfaces** — ComfyUI, A1111, Forge, Invoke; node-based vs prompt-first.
- **Cloud video tools** — Runway, Kling, Sora, Luma, Pika, Higgsfield framed categorically.
- **Model zoos** — Hugging Face, Civitai; reading a model card; spotting dubious uploads.

10 Craft

Media-specific technique and decision-making. These pages assume you've read the `foundations/` — they refine the shared meta-skills for moving images and sound.

10.1 Planned topics

- **Prompting for visual generation** — positive/negative, weighting, syntax differences. Refines `foundations/prompting-principles.md`.
- **Seeds and reproducibility as craft** — directed versus discovered.
- **Character consistency across shots.**
- **Scene and location consistency.**
- **Shot-to-shot continuity** — lighting, wardrobe, time-of-day.
- **Camera motion control** in video models.
- **Temporal coherence** — flicker, morphing, and how to fight it.
- **Directing AI: intention vs randomness.**
- **Blending generative with traditional** — plates, comps, hybrid pipelines.
- **When *not* to use AI.**

11 Lifecycle Crosswalk

A supplemental view, not the primary spine. Generative tools are collapsing traditional film-production stages — previs becomes production, post becomes production — so this page cross-references concepts to lifecycle stages rather than organizing them that way.

Every bullet below links *into* a page elsewhere in the repo. Nothing is duplicated here.

11.1 Planned structure

1. **Preamble** — why this is a crosswalk, not a spine.
2. **Stages** — Development, Pre-production, Production, Post, Delivery. Each a short paragraph plus a bulleted list of links.
3. **Collapsing seams** — callouts where a tool genuinely dissolves a stage boundary.

(Content to be written in a follow-on plan once the linked pages exist.)

Part III — Vibe-coding for Filmmakers. Writing code with AI coding tools, treated as a first-class form of generative production.

12 Foundations (Vibe-coding)

Mental models for someone who has coded little or not at all. This is the on-ramp before the setup and safety material — enough to make the rest of the section make sense.

12.0.1 What is an agent?

An "agent" in the coding sense is a model that can take actions on your behalf — read files, run commands, edit code — not just chat about them. This is the crucial distinction from a chat window where the model only produces text. Three shapes are common: **autocomplete** (suggests the next few lines as you type, you stay in control), **chat** (you ask, the model answers, you copy the answer into your project), and **agent** (the model reads and writes your project directly, often running commands in a terminal). Each has different affordances and different failure modes; pick the one that matches how much trust the task deserves.

12.0.2 Where does code actually run?

Code has to run *somewhere* — your laptop, a cloud server, or a hybrid of both — and the difference matters more than it looks. Code that runs locally is fast, free, and private, but limited by your hardware; code that runs in the cloud can scale but costs money and exposes your data. Many AI coding tools also use cloud resources on your behalf (the model itself is in the cloud even when your project is local), which is why a "free" agent can still surprise you with bills. Before starting a project, know where each part of it will execute and what that implies for cost, privacy, and reliability.

12.0.3 Terminal literacy

The terminal is a text-based way of telling your computer what to do — no icons, no mouse. Agents drop you into one constantly because it's the fastest way for a program to manipulate files and run commands. You don't need fluency, but you need comfort with a handful of verbs: `cd` to change directories, `ls` to list files, `cat` to read a file, `git` to track changes, and enough awareness to recognize when a command looks dangerous (anything with `rm -rf`, anything that rewrites history, anything you didn't ask for). When the agent proposes a command, read it before approving it.

12.0.4 Git literacy

Git is the version-control system that remembers every change you commit to a project. For filmmakers this is the equivalent of keeping every take: you can always return to an earlier version, compare two versions, or branch off a new experiment without losing the main line. Git matters *especially* when you're working with AI-generated code, because the agent will sometimes produce changes you don't want — a revert is the undo button, and it only works if you've been committing along the way. The working habit: commit small, commit often, write a one-line message describing the *why*, and never skip the review step before pushing changes public.

13 Platforms

Where you actually write code with a model. Options span a spectrum from fully-hosted tools that run in your browser to command-line agents that work directly on your own machine. The trade-offs are consistent across the spectrum: the more hosted the tool, the less setup you have to do and the less you can customize; the more local the tool, the more control and visibility you have, at the cost of complexity and your own hardware. For a first project, start further up the hosted end — most frustration at the start is setup, not code.

13.0.1 In-browser environments

Tools like **Lovable**, **Bolt**, **v0**, and **Replit** let you describe what you want and watch the model assemble a working app in a browser tab — no install, no setup, no git to learn on day one. They're the fastest path from idea to running code, and the right place to prototype. The trade-off is that you're confined to what the platform supports: particular frameworks, particular deploy targets, particular kinds of customization. When a project outgrows the sandbox, you export the code and move somewhere else.

13.0.2 Native desktop apps from model makers

Anthropic's **Claude** and OpenAI's **ChatGPT** both ship native Mac and Windows apps. These are chat-first tools — the model can read files you drop in, run small bits of code in a sandbox, and answer questions — but they don't manage a project the way an agent or IDE does. Good for explanations, one-off snippets, debugging a small file, and learning. Not where you build a non-trivial project over weeks.

13.0.3 AI-integrated IDEs

An IDE (integrated development environment) is the editor you'd use even without AI — **VS Code**, **JetBrains**, **Xcode**. AI-integrated IDEs bolt a model into that editor. **Cursor** and **Windsurf** are VS Code forks built around an agent; **VS Code + GitHub Copilot** (or Claude's extension) adds similar capabilities to the mainline editor; JetBrains has its own AI integration. This is where most day-to-day coding with AI happens — you see your project, you see the model's proposed changes, and you decide what to accept.

13.0.4 Command-line agents

At the other end of the spectrum, command-line agents like **Claude Code**, **Codex CLI**, and **Aider** run in a terminal, read and write files in your project directly, and execute shell commands. They're less beginner-friendly — you need to be comfortable in a terminal — but they're also the most powerful, because the agent has the same access to your project that you do. The right tool when you're orchestrating multi-step work across many files.

13.0.5 Typical pricing

Most platforms cluster into a few tiers. **Free** usually means limited usage, slower models, or a rate cap — enough to try the tool, not to rely on it. **~$20/month pro subscriptions** are the default for serious single-user work: Claude Pro, ChatGPT Plus, Cursor Pro, GitHub Copilot, Replit Core. **Usage-based API pricing** (Anthropic, OpenAI, Google) charges per token — cheap for occasional scripted calls, potentially expensive for heavy agent use over time. **Team and enterprise plans** stack higher still, typically $50–100+ per seat with admin controls. Before committing to a platform, confirm which tier your actual workflow needs — "the monthly tier is enough for most people" is a common refrain that breaks down fast once you're building something serious.

14 Setup and Safety

The things that will hurt you if you skip them. Credentials, billing, what never goes into git, and how to let an agent run commands without letting it ruin your afternoon.

14.0.1 Managing credentials

Credentials (API keys, passwords, tokens) are the secrets your code uses to access paid services. They get handled differently from every other kind of configuration because a leaked key gets scraped from a public repo within minutes and billed to you until you rotate it. The standard pattern: store real keys in a password manager, copy them into a local `.env` file your project reads from, add `.env` to `.gitignore` so it never gets committed, and ship a `.env.example` with blank values so collaborators know which keys to fill in. Set up `.gitignore` *before* you put a credential in a file — the mistake is almost always committing the file before realizing you should have excluded it.

14.0.2 API keys, billing, and spending caps

Every paid generative tool authenticates you with an API key, and every one of those keys is attached to a billing account. The two things that will bite a beginner: rate limits (you hit a ceiling and requests start failing) and uncapped spend (a runaway script, or a leaked key, burns through your card). Treat an API key like a credit card — one per project so blast radius stays contained, a hard monthly spending limit set at the vendor dashboard before you write a line of code, and email alerts at 50% and 90% of that limit. A leaked uncapped key has cost beginners thousands of dollars in a day; a capped one caps the damage.

14.0.3 Repos and `.gitignore`

A repository ("repo") is the folder git tracks. A public repo is visible to everyone on the internet, including bots that scan GitHub for leaked secrets — so "what goes in the repo" is a load-bearing decision. `.gitignore` is the file that tells git *not* to track certain files: credentials (`.env`), build artifacts (`node_modules/`, `_site/`), generated media, large binaries, and anything personal or secret. Every new project starts with a `.gitignore` tailored to its language or framework (most hosts offer a starter file), and every commit should be preceded by a `git status` to confirm you're about to push what you think you're pushing.

14.0.4 Sandboxing

Sandboxing means running the agent in an environment where it *can't* hurt you — a container, a virtual machine, a project folder it can't escape. This matters because agents execute commands, and an agent that's been tricked, confused, or compromised can do real damage to your machine. The practical move: run agents against a project directory with a clear boundary, never give one full access to your home folder or system directories, use permission prompts for anything that touches your filesystem, and review each command before approving when the tool supports it. Trust the agent to the exact degree you'd trust a well-meaning intern — capable, but supervised.

15 Working with Models

Choosing, prompting, and managing context with AI coding tools. This section refines `foundations/prompting-principles.md` for the coding case, and adds the code-specific environment — rules files, skills, MCP — that shapes what the model actually sees.

15.0.1 Coding models compared

Several model families dominate coding: Anthropic's Claude (Opus / Sonnet / Haiku tiers), OpenAI's GPT family, Google's Gemini, and a growing set of locally-runnable open models. They differ in three ways that matter: raw capability (how hard a problem can it solve), context window (how much of your project it can hold in memory at once), and cost per token. A model that's cheaper per token but needs three attempts to get a function right is not actually cheaper. The right default is the most capable model in the tier that fits your budget; reach for smaller models only when the task is genuinely small (a rename, a syntax question) or when you're running locally for privacy.

15.0.2 Prompting for code

Prompting a coding model refines the universal prompting principles: specificity beats cleverness, constraints improve output, one change per iteration, and reference examples teach faster than adjectives. The specific-to-code moves: name the language, framework, and version (`Rails 8.1`, `Python 3.12`, `Node 22`) so the model doesn't guess; describe the existing code style if you want consistency; show a small example input and expected output; and say what you've already tried if the first attempt didn't work. The anti-pattern is asking for a "complete implementation" with vague requirements — you'll get something plausible that doesn't fit your codebase.

15.0.3 Context management

The chat window is an *environment*, not a prompt box. Everything in it — the files the agent has read, the commands it has run, the earlier questions and answers — shapes the next response. This is why "start a new chat" is useful advice: when the environment drifts or fills with irrelevant detail, a fresh conversation is often faster than trying to correct course. Rules files (`CLAUDE.md`, `AGENTS.md`, or the equivalent for your tool) are how you pin the stable context — conventions, domain knowledge, gotchas — so you don't re-type it every session. The skill is curating the environment, not just writing prompts into it.

15.0.4 Skills

Skills are reusable packages of instructions and examples that extend an agent's behavior for a specific task — "review a pull request," "create a migration," "debug a failing test." They're a step up from rules files: rules live in the project, skills can live in the tool and follow you across projects. Build one when you notice yourself explaining the same workflow to the agent for the third time; don't bother earlier, because premature skills become stale instructions you have to maintain. A good skill is a recipe, not a personality.

15.0.5 MCP

MCP (Model Context Protocol) is a standard way for an agent to talk to external tools — a filesystem, a database, a browser, an image-generation API. Without MCP, each integration is a one-off; with it, an agent can plug into any compatible tool without custom code. For a beginner, the heuristic is simple: if you find yourself copy-pasting between an agent and another service repeatedly, that's the point where wiring up an MCP server saves time. Don't wire up servers speculatively — each one adds surface area for mistakes.

15.0.6 Agents vs chat vs autocomplete

These three shapes of AI coding assistance each have a right job. **Autocomplete** (Copilot-style inline suggestions) is fastest for routine code — you stay in control, the model saves keystrokes. **Chat** is best for explanations, design questions, and small isolated functions you'll paste in yourself. **Agents** shine on multi-file changes, refactors, and debugging sessions where reading and writing the project is half the work. Mixing them is normal: use autocomplete while typing, pop out to chat to understand an error message, dispatch an agent to implement the change. The wrong move is using one shape for every job.

15.0.7 Prompting tips

Models tuned to be helpful can be sycophantic — they'll agree with the framing of your question rather than push back on it, especially when the question implies a preferred answer. "Is this a good idea?" invites a pat on the back; the model reads your framing and tends to confirm it. "What are the pros and cons of this approach?" forces a more balanced analysis. The same move applies elsewhere: instead of "Is this code clean?" try "What would a senior engineer criticize about this code?"; instead of "Should I use Rails here?" try "What are the trade-offs between Rails and a lighter framework for this use case?" Frame questions to invite critique, not agreement, and the model becomes a more useful collaborator.

16 Building Software

Sensible defaults for someone assembling their first non-trivial project. The theme: prefer the boringest credible choice, and read things before pasting them.

16.0.1 Reading API docs

Most services you'll integrate — an image-generation API, a cloud storage bucket, a video-delivery platform — publish documentation that describes what calls are available and what they expect. The skill is skimming docs for the three things you need: the authentication method (how the key goes in), the endpoint structure (what URL to hit and with what shape of data), and a working example (to confirm you read the first two correctly). REST and HTTP are the common vocabulary — a call is almost always a `GET`, `POST`, `PUT`, or `DELETE` to a URL with a JSON body. SDKs wrap this in a language-specific library; raw HTTP calls work when no SDK exists. Either way, read the docs before asking the model — the agent will invent plausible-looking calls that don't exist.

16.0.2 Frameworks vs rolling-your-own

A framework is a pre-built scaffolding — Rails, Next.js, Django, SvelteKit — that decides most of the structural questions for you so you can focus on what's unique about your project. Rolling your own means building the scaffolding yourself, which an agent will gladly do. The trade-off: frameworks give you battle-tested patterns, community support, and a shared vocabulary with other developers, at the cost of conforming to their conventions. Rolling your own gives you freedom at the cost of reinventing problems that have already been solved. For a beginner, the framework almost always wins — you want to spend your learning on *your* problem, not on everyone else's problems that the framework already solved.

16.0.3 Native development

Native development means building software that runs directly on a specific platform — iOS, Android, macOS, Windows — rather than in a web browser. It's more powerful (access to hardware, platform APIs, offline use) but costly in ways beginners don't see up front. There's a literal price of admission to become a developer on each platform: Apple's Developer Program is $99/year, Google Play is a $25 one-time fee, and code-signing certificates for Windows and macOS add their own recurring costs. Then there's the approval cycle: every update to an iOS or Android app goes through platform review, which can take days to weeks and sometimes gets rejected for reasons you didn't anticipate — a gatekeeper your web project simply doesn't have. Each platform also has its own toolchain, language, and hardware requirements (iOS development requires a Mac). Don't pick native for your first project — ship something on the web first, where publishing is free,

updates are instant, and nothing stands between you and your users. Come back to native once you've shipped.

16.0.4 Boring tech wins

The counter-intuitive rule of working with AI coding tools is that older, more established technologies produce better results than trendy ones. Rails, Django, Next.js, plain Postgres, plain HTML — the model has seen these in millions of projects and knows their idioms deeply. The new shiny framework released last quarter has less training data behind it, and the agent will confidently improvise code that looks right but isn't. The move: default to the boringest credible choice; only reach for the cutting edge when the problem genuinely requires it. Boring tech is also better for *you* — it has documentation, tutorials, and Stack Overflow answers to help when the model is wrong.

16.0.5 Dependencies and package managers

A dependency is someone else's code that your project uses — a library, a framework, a utility. A package manager (npm, pip, bundler, cargo) installs and tracks them. Dependencies are powerful because you get to skip work someone else did, and dangerous because you're trusting code you didn't write and often didn't read. Two beginner failure modes: installing random packages the agent suggests without checking who maintains them, and letting versions drift so that "works on my machine" doesn't reproduce anywhere else. The discipline: prefer well-known packages with active maintenance, pin versions in a lockfile, and read what a package does before adding it.

16.0.6 Reading error messages

Error messages look intimidating but they're usually specific and helpful — the first line often tells you exactly what went wrong and which file to look at. The beginner move is to paste the error into the agent and ask for a fix; the better move is to read it first, form a hypothesis, *then* consult the agent with "I think X is happening because Y — is that right?" This produces better help because you're collaborating on a diagnosis instead of asking the model to guess. Stack traces read bottom-up: the deepest call is usually a library you don't control; the call near the top in *your* code is almost always where the real bug lives.

17 Common Traps

A named gallery of specific beginner mistakes, each with the fix. Each trap is its own short page — a cautionary tale you can hand to someone about to make the same mistake. Cross-linked to `../../practical/` where topics overlap (licenses, reproducibility, cost).

17.1 The gallery

- **Committing secrets to git.** — the single most common way beginners get burned.
- **Blindly running agent-suggested shell commands.** — most are fine; once in a while, catastrophic.
- **Force-push disasters.** — one flag, whole branch gone.
- **Letting the agent refactor without review.** — a 300-line diff you didn't read.
- **Accepting the first thing that compiles.** — "compiles" ≠ "works."
- **Scope-creep prompts.** — you asked for one change; you got six.
- **Building on sand.** — unreliable deps, beta APIs, deprecated models.
- **Runaway API bills.** — loops, retries, leaks — always set a ceiling.
- **Deploying to production without testing.** — works-on-my-machine, at scale.
- **Trusting a local-running model demo.** — a clever trick is not a pipeline.
- **"Works on my machine" illusions.** — reproducibility is a practice, not a hope.

18 Committing Secrets to Git

The single most common way beginner coders get burned. You add an API key to a config file, commit it, push to a public repo, and within minutes a bot has scraped the key and started using it on your account. The fix is preventive: set up `.gitignore` before you put a credential anywhere near the project, verify with `git status` before every commit, and if it happens anyway, rotate the key at the vendor immediately — don't just delete the commit. Git history is distributed; you cannot unshare a leaked secret.

See also: `../setup-and-safety/` for the managing-credentials pattern that prevents this.

19 Blindly Running Agent-Suggested Shell Commands

An agent proposes a command; you hit approve without reading it. Usually fine. Once in a while, catastrophic — a command that deletes a folder you cared about, overwrites a file, or runs something dangerous with your credentials. The fix is a habit: read every command before approving, flag anything with `rm -rf`, `sudo`, or a redirect into a file you didn't expect, and never run an unfamiliar command in a directory you don't want to lose. The agent will propose the right command most of the time; the cost of reading it is trivial compared to the cost of a mistake.

See also: `../setup-and-safety/` on sandboxing — another layer of defense when you can't read every command.

20 Force-Push Disasters

`git push --force` overwrites the remote branch with your local state, discarding whatever was there before. Beginners use it to "fix" a messy commit history, and end up erasing work — sometimes their own, sometimes a collaborator's. The fix: never force-push unless you explicitly need to and understand exactly what you're overwriting, prefer `--force-with-lease` (which at least checks someone else hasn't pushed in the meantime), and never force-push to `main`. If you need to undo a commit, use `revert` (adds a new commit that cancels the old one) instead.

See also: `../foundations/` on git literacy — the habit that makes force-push unnecessary.

21 Letting the Agent Refactor Without Review

You ask the agent to “clean up” a module, it rewrites three hundred lines, and you approve the diff without reading it because it’s too long. Two weeks later a subtle bug surfaces, and you can’t tell what changed vs. what was always wrong. The fix: treat a large refactor as three small ones. Ask the agent to do one thing at a time — rename this, extract that, inline the other — and review each step. If the agent proposes a sweeping change, ask for a summary of what’s moving where *before* approving the diff.

22 Accepting the First Thing That Compiles

The code compiles, the test passes, the feature appears to work — so you ship it. Then an edge case blows up in production. "Compiles" means the syntax is correct; "works for my input" means nothing about the inputs you didn't try. The fix is a small checklist: try the empty input, the wrong type, the oversized input, and the obvious adversarial input before declaring a function done. Coverage of the golden path alone has always been a trap; with AI-generated code, where the model optimizes for "looks plausible," it's worse.

23 Scope-Creep Prompts

You ask for a small change; the agent takes the opportunity to reformat six other files, rename three functions, and "improve" a module you didn't mention. Now the diff is unreadable and you can't tell what was actually asked for. The fix: constrain the agent in the prompt ("change only the `authenticate` function; do not modify other files") and review the diff before committing. When the agent oversteps, reject the whole change and re-prompt rather than trying to cherry-pick — the cheap move is to redo it, not to untangle it.

24 Building on Sand

Unreliable dependencies, beta APIs, deprecated models, preview-tier services — building on any of them means your project can stop working without warning. A beta video-generation API changes its response format; your pipeline breaks. A model gets deprecated; your shot no longer renders. The fix is awareness: when you depend on something unstable, know it, document it, and have a fallback plan. For anything production-critical, prefer the oldest stable version of the tool that does what you need.

See also: `../../practical/` on reproducibility — how to keep a project working six months from now.

25 Runaway API Bills

A script gets stuck in a loop and calls the API a million times overnight; a misconfigured job keeps retrying a failed request; a leaked key is used by someone else. The common thread: no ceiling. The fix is layered — per-key spending caps at the vendor, per-request timeouts in your code, a circuit breaker that stops calling an API after N consecutive failures, and email alerts when spend exceeds a threshold. You set these up *before* you need them, because the time you need them is at 3am when the damage is already happening.

See also: `../setup-and-safety/` on API keys and spending caps.

26 Deploying to Production Without Testing

It works on your machine, so you push it to the live server, and the live server has different environment variables, different database state, and different traffic patterns. Features that "worked" in development break under real conditions. The fix: a staging environment that mirrors production, an automated test suite that runs before every deploy, and a deploy that's reversible (you can roll back in one command). Small projects don't need elaborate infrastructure, but every project needs *a way* to deploy safely and *a way* to undo a deploy.

27 Trusting a Local-Running Model Demo

You run an open-weight video model on your laptop, it produces a beautiful shot, you plan the whole project around that capability. Then the project grows: you need a hundred shots, you need consistency across them, you need reliability. Local-model demos don't scale — what feels like a clever trick on one shot becomes a serious infrastructure project at production volume. The fix is honesty about the gap: a demo proves a look is possible, not that a pipeline is viable. Before committing, budget the real cost of scaling the approach.

See also: `../../practical/` on cost management and reproducibility.

28 “Works on My Machine” Illusions

Your project runs fine locally. On a collaborator’s machine, or on the server, or on GitHub Actions, it breaks — because your machine has a specific Python version, a specific env var, a file that exists only locally, or a tool you installed and forgot. The fix is reproducibility as a practice: pin every version in a lockfile, document every required env var in `.env.example`, check in a working setup script, and test the “fresh clone” scenario at least once before calling a project done. If a project can’t run on a new machine from a git clone plus a documented setup step, it’s not really shareable.

Part IV — Practical. The non-creative matters that sit under both pillars: licenses, cost, reproducibility, provenance, delivery.

29 Practical

The non-creative stuff that sinks projects. This folder sits at the top level because it applies across the whole site — licenses, cost, and reproducibility matter as much for code generation as for media generation.

29.1 Planned topics

- **Model and tool licenses** — non-commercial vs permissive vs OSS; what "open weights" actually lets you do.
- **Copyright and training-data debates** — written to age gracefully.
- **Likeness and consent** — actors, real people, estates.
- **Provenance and labeling** — C2PA, visible disclosure norms, platform requirements.
- **Cost management** — API spend, cloud GPU hours, iteration budgets.
- **Reproducibility** — archiving prompts, seeds, model versions, and for code: lockfiles and commits.
- **Delivery considerations** — codecs, color space, frame rates.

---

30 Glossary

An A–Z index of terms used across the repo. Each entry is a one-line definition and a link to the fuller page.

(This glossary will grow as pages are written. It is currently a stub with the terms already covered; new entries should be added as each page is authored.)

30.1 A

30.2 C

- **CFG (Classifier-Free Guidance)** — a knob controlling how strictly the model follows the prompt. See `generative-media/concepts/model-basics/` *(page planned)*.

30.3 D

- **Diffusion** — the process by which many image/video models turn noise into coherent output. See `generative-media/concepts/model-basics/` *(page planned)*.

30.4 G

- **Generative Production** — the discipline this repo teaches. See `foundations/what-is-generative-production.md`.

30.5 I

- **Iteration and selection** — the core loop of working with generative tools. See `foundations/iteration-and-selection.md`.

30.6 L

- **LoRA (Low-Rank Adaptation)** — a lightweight adapter that teaches a base model a style, subject, or concept. See `generative-media/concepts/model-basics/` *(page planned)*.

30.7 P

- **Prompt** — the instruction you give a generative model. See `foundations/prompting-principles.md`.

30.8 S

- **Seed** — a number that makes a generation reproducible. See `generative-media/concepts/model-basics/seeds.md`.

31 Learning Sources

Curated places to keep learning outside this repo.

31.1 YouTube

- **Curious Refuge** — ongoing coverage of generative tooling for film.

31.2 Discord

- **Banodoco** — community for open-weight video tooling.

31.3 In person

- **Local meetups** — search for "AI filmmaking," "generative film," or "AI video" in your city.

31.4 Model and tool directories

- **Hugging Face** — models, datasets, demos. Read the model card before using anything.
- **Civitai** — image-model hub with user-uploaded checkpoints and LoRAs. Higher signal-to-noise than it looks, but read the cards.