

## 4.5 — Build Our Own Tiny Codex

Gavin · AI-assisted course design

## Contents

|                                                             |    |
|-------------------------------------------------------------|----|
| 1 Build the smallest complete loop .....                    | 3  |
| 2 Decide what would count as an agent .....                 | 3  |
| 3 Draw the trust boundaries first .....                     | 4  |
| 4 Choose a model small enough to inspect .....              | 6  |
| 5 Make loading reproducible and bounded .....               | 6  |
| 6 Give the model a small comprehensible job .....           | 7  |
| 7 Define actions before writing the loop .....              | 7  |
| 8 Test expressions without executing arbitrary Python ..... | 8  |
| 9 Make edits recoverable and conflict-aware .....           | 9  |
| 10 Assemble context from authoritative state .....          | 9  |
| 11 Stop on evidence and enforce cancellation .....          | 9  |
| 12 Measure what the trace actually supports .....           | 10 |
| 13 Check your understanding .....                           | 11 |
| 13.1 Suggested answers .....                                | 11 |
| 14 More Learning .....                                      | 12 |

## 1 Build the smallest complete loop

A coding model can suggest a correct replacement expression without changing a file. A file editor can change a file without understanding it. A test runner can report a failure without deciding what to do next. A coding agent connects these components through a controller that repeatedly obtains a proposal, checks it, executes an allowed operation, and returns an observation.

In this lesson we build that connection around a real, downloadable coding-instruction model. The task is deliberately small: repair two functions in a fresh teaching workspace. The model reads source, proposes expression replacements, and receives test results. Its weights remain fixed. The controller supplies tools, remembers file versions, enforces permissions, and ends the run.

The chapter title names a teaching project, not a reproduction of a commercial coding assistant. Our program handles a restricted subset of Python expressions. It cannot explore arbitrary repositories or run arbitrary Python. That narrow boundary makes the experiment inspectable on an ordinary computer and lets us distinguish a model's mistakes from failures in the software around it.

By the end, you should be able to:

- connect an open model to a bounded editing and testing loop;
- explain which component proposes, authorizes, edits, and verifies;
- preserve model failures rather than silently replacing them with correct answers;
- implement a test tool that does not execute model-written Python;
- measure resource use and diagnose where a run stopped;
- state what a successful toy repair does, and does not, establish.

### Math to know / refresh

**Needed now:** integer arithmetic, comparisons, function inputs and outputs, and counting tokens and operations.

**Useful refresh:** a finite-state machine records which transitions are allowed next. A budget is a constraint, not a prediction of the amount of work required.

**Side trail:** program semantics, property-based testing, capability security, and the statistics of benchmark comparisons.

## 2 Decide what would count as an agent

Our central experiment has two proposal sources. A **scripted baseline** supplies known valid actions, including the correct repairs. It verifies the reader, editor, interpreter, and controller. The **model trial** supplies generated actions from the frozen checkpoint. Both travel through the same dispatcher and start from separate, identical broken workspaces.

The baseline is intentionally given the solution. It is a control for harness correctness, not a competing intelligence. If it fails, do not interpret the model trial until the harness problem is understood. If it passes and the model emits malformed JSON, that is evidence about this model and interface under this

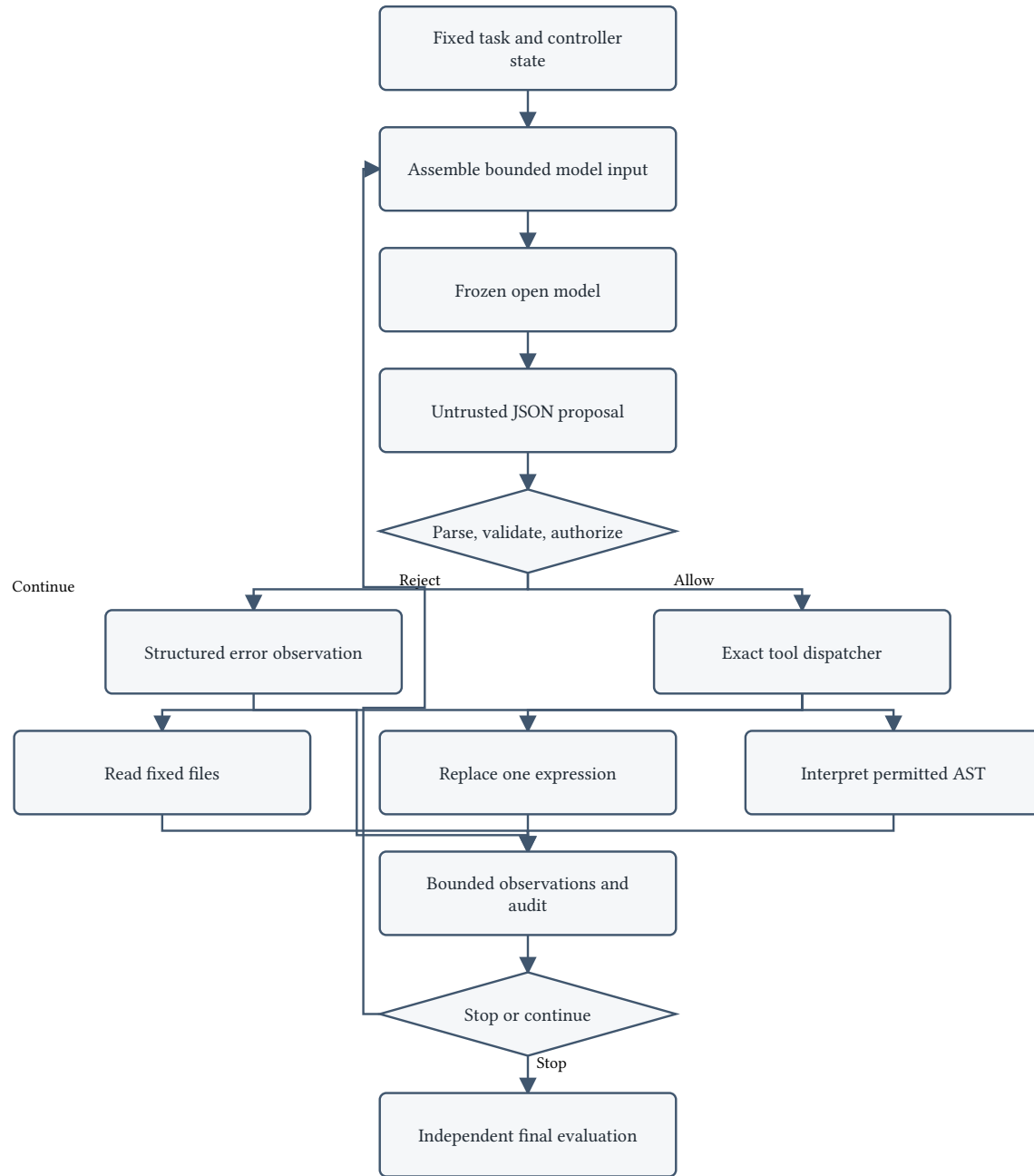
budget. Replacing the malformed answer with the baseline’s action would erase precisely the behavior we intended to observe.

A real-model trial is required to complete the actual-agent part of the Lab. A paper trace or a baseline-only implementation is useful partial work. Neither is a completed model-driven agent. A failed real-model trial still provides a valid experimental result when the model actually generated the proposals and the failure is recorded honestly.

This separation also prevents an attractive demonstration from making an unsupported claim. The program may be able to apply a correct patch even when this particular model cannot reliably produce one. Conversely, the model may suggest the right mathematics while an editor bug corrupts the file.

### **3 Draw the trust boundaries first**

The following is an original diagram of our design. The model sees a bounded request and returns text. It has no direct handle to the editor, filesystem, or evaluator.



The **request assembler** selects the evidence the model receives. The **validator** checks the shape of generated data. The **authorizer** checks whether the operation is allowed in the current state. The **dispatcher** maps a small, fixed set of names to trusted implementations. The **evaluator** checks the resulting expressions against synthetic fixtures.

No generated tool name is passed to dynamic attribute lookup, an import statement, or a shell. The mapping is literal application code. A proposal containing `"action": "shell"` is rejected because there is no such operation. Writing “permission granted” in a model answer changes no permission state.

ReAct studied language-model trajectories that interleave reasoning and actions with observations. SWE-agent investigated how the interface provided to a software agent affects its behavior. Our exercise borrows the general insight that interaction design matters; it does not reproduce either paper’s benchmark, tools, model, or reported scores. ReAct, SWE-agent

## 4 Choose a model small enough to inspect

The reference checkpoint is **Qwen/Qwen2.5-Coder-0.5B-Instruct**, an official Qwen coding-instruction model distributed under Apache 2.0. Pin revision `ea3f2471cf1b1f0db85067f1ef93848e38e88c25`, rather than a moving main branch. The published configuration specifies 24 layers, hidden size 896, 14 attention heads, two key/value heads, and vocabulary size 151,936. It uses the Qwen2 causal-language-model implementation. These are checkpoint facts, not performance predictions. Official model repository, pinned configuration

The model’s safetensors file is 988,097,824 bytes. The Lab pins that file’s digest and an explicit manifest of tokenizer, configuration, and license files. The selected download is about 1.00 GB in decimal units, below a 1.2 GB model-artifact ceiling. This excludes Python packages. The checkpoint is small by current model standards, but it is not a tiny text file. Official pinned weight metadata

For the reference experiment, use CPU inference, float32 parameters, eager attention, at most two CPU threads, and no compilation or training. Roughly half a billion float32 parameters require about 2 GB just for parameter storage. Loading buffers, the inference runtime, activations, tokenizer, and operating system need additional memory. Allow several GB of working headroom; an 8 GB machine may still be tight. Disk needs include both model files and the installed runtime. These are planning estimates, not measured resource results for the learner’s computer.

Ask the learner to opt in before downloading. If the machine or connection is unsuitable, stop at the harness and label the real-model stage unperformed. Do not quietly switch to paid inference, a much larger checkpoint, or an unrelated tiny base model and pretend the experiment is unchanged.

## 5 Make loading reproducible and bounded

Download only named files from the pinned revision, enforce the aggregate byte limit, and verify sizes and hashes before loading. Keep that directory separate from the editable toy files. A complete manifest identifies both the intended revision and the actual local bytes. Record the runtime’s Python, PyTorch, Transformers, tokenizers, safetensors, and Hub-library versions too.

Load locally with `local_files_only=True`, `trust_remote_code=False`, and `use_safetensors=True`. Reject unexpected custom-code configuration or extra executable artifacts. Use the installed library’s Qwen2 implementation. Safetensors avoids the pickle-based model-loading path; it does not prove that the entire runtime or every input parser is invulnerable. Hugging Face specifically recommends safetensors and warns about remote model code. Transformers security policy

The acquisition step and the experiment are separate. Once the verified files exist, the experiment should not need the network. Hub offline settings and local-only loading prevent intended library fetches; they are not an operating-system network sandbox. Our model-facing tools simply contain no networking operation.

Fix generation settings explicitly: greedy decoding, one beam, no sampling, a repetition penalty of 1.0, and at most 128 new tokens per proposal. Do not accidentally inherit sampling defaults from the checkpoint's generation configuration. Record the effective settings rather than only the command the learner intended to run. The Transformers generation interface distinguishes these controls. Generation configuration

Fixed settings make comparisons clearer. They do not establish identical results across every library version, CPU architecture, or numerical implementation.

## 6 Give the model a small comprehensible job

The workspace contains `TASK.txt` and `toy_math.py`. The task asks for two integer functions:

- `clamp(x, lo, hi)` returns the nearest endpoint when `x` lies outside the inclusive interval, and otherwise returns `x`. Inputs satisfy `lo <= hi`.
- `total(unit, qty, fee)` returns the cost of `qty` units plus a single fee.

The deliberately broken file is ordinary readable Python:

```
def clamp(x, lo, hi):  
    return x  
  
def total(unit, qty, fee):  
    return unit + qty + fee
```

We do not ask the agent to install dependencies, inspect a home directory, discover credentials, or modify its tests. Its authority is limited to replacing the returned expression of either named function. Function names, parameters, file names, wrappers, and fixtures belong to the harness.

The exact instruction tells the model to produce one JSON object, choose from four actions, and treat file contents and observations as data. It also states the expression grammar, budgets, and success rule. This improves the chance of useful behavior. It is not where permissions are enforced.

The distinction matters when a file contains an instruction-like sentence. Quoting “ignore the tool restrictions” in a read result does not promote it to a system instruction. Preserve its provenance in the assembled request, but expect that the model may still follow it. The dispatcher must remain safe when it does.

## 7 Define actions before writing the loop

The Lab specifies four actions: read, replace, test, and finish. There is no general-purpose write or shell action.

A replacement identifies a function, its expected current expression, the replacement expression, and the workspace version:

```
{"action": "replace", "function": "total", "old": "unit + qty + fee", "new": "unit * qty  
+ fee", "version": 0}
```

This is a contract example, not an observed model response. The edit succeeds only if every precondition matches. If another accepted edit has advanced the version, the proposal is stale even if its mathematics is correct. The model must use the latest successful source read or committed edit receipt rather than overwrite from stale state. An edit receipt includes both current expressions and the new version, so it can support another replacement without an extra read.

Parse a single JSON object. Reject duplicate keys, unexpected fields, trailing prose, non-finite values, wrong types, and oversized output. Do not hunt for a plausible JSON substring inside a paragraph. That would silently add a repair algorithm whose contribution is difficult to separate from the model's ability.

Schema validation and authorization answer different questions. `"version":false` fails type validation even though Python treats Boolean values as an integer subtype. A schema-valid replacement attempted before any successful source read fails authorization. A permitted replacement containing a function call fails expression validation. All three return distinct errors and leave the workspace unchanged.

The observation records what happened: for example, `stale_version`, `expression_rejected`, or `edit_committed`. "The model intended to fix it" is not an execution result.

## 8 Test expressions without executing arbitrary Python

A fixed command such as `python tests.py` is not a sandbox. If that test script imports an edited module, the module can execute code with the subprocess's privileges. An allowlisted command name does not constrain all behavior behind that name.

Instead, our test tool parses the two function definitions and interprets only a deliberately small expression language. It permits bounded integer constants, approved parameter names, addition, subtraction, multiplication, unary signs, single comparisons, and conditional expressions. Every unknown syntax node is rejected. There are no calls, imports, attributes, indexing, comprehensions, loops, classes, decorators, strings, or reflection.

The interpreter recursively inspects a node's type and applies a corresponding trusted operation to already validated scalar values. It never invokes `eval`, `exec`, `compile`, or imports the edited file. The fact that Python can parse source into an abstract syntax tree does not itself make that source safe to execute. Python's documentation also warns that even ostensibly limited parsing and literal handling can exhaust resources on hostile inputs. Python AST documentation

Therefore, bounds precede and follow parsing: source bytes, lexical nesting, numeric-token size, tree depth, node count, evaluation steps, and intermediate integer magnitude. Conditional evaluation visits only the selected branch, but validation examines both branches before any evaluation. Rejecting a forbidden call only when execution happens to reach it would leave a hole in the language boundary.

The functions look like Python because they are valid examples of a narrow Python subset. The interpreter does not implement general Python semantics. That tradeoff is explicit: the model still reads code, changes a real file, receives failures, and iterates, while its generated expressions cannot request host operations through this interface.

## 9 Make edits recoverable and conflict-aware

Create a new, private run directory. Refuse to reuse an existing output directory or initialize inside an existing repository. Keep fixtures, logs, and the model outside the editable workspace. The reader accepts exact logical filenames, not arbitrary paths.

An accepted edit follows a transaction-like sequence: verify current version and file digest, validate the proposed expression, generate the complete two-function file, preserve the previous bytes in an exclusive backup, write a temporary file, and atomically replace the target. Recheck the resulting bytes and record the new version and digest. If a precondition fails, do not write.

Reject symbolic links and unexpected hard links in managed files. Use directory-relative, no-follow operations where supported, and fail closed if the implementation cannot enforce its chosen filesystem policy. Merely testing whether a string starts with the workspace path is insufficient.

Atomic replacement prevents observers from seeing a partially written file under normal filesystem semantics. It is not a universal compare-and-swap or protection against an attacker racing with the same user privileges. This teaching workspace assumes one controller and no hostile host process. Detect ordinary external edits through hashes and stop on conflict. Python documents the conditions and platform behavior of file replacement; implementation tests must cover the supported system. Python filesystem operations

## 10 Assemble context from authoritative state

Use the pinned tokenizer's chat template, including its generation prefix. A chat-oriented causal model still consumes tokens; message dictionaries need the separators expected by that checkpoint. Formatting a chat template and then adding special tokens again can introduce an unintended input. Transformers chat templates

Our model request contains the fixed system instruction, task, tool contract, current budget and version, and a bounded record of observations. A read result exposes source only after a successful read. The model cannot access the fixture file merely because it exists elsewhere on the computer.

Allow at most 2,048 input tokens after chat formatting, plus the separate 128-token generation allowance. This is our resource policy, far below the checkpoint's advertised context capacity. Count actual tokens before each call. Never trim away permissions or the task to squeeze in another error log.

For this small experiment, deterministic compaction is enough: preserve the most recent successful read, all edit receipts, the latest public test result, and the latest validation error. Omit older repeated read/test observations first, recording omitted event IDs. If essential context still exceeds the cap, stop with `context_limit`. The complete audit remains on disk.

A summary generated by the same model could help in a larger system, but it would add another fallible component. In particular, a summary must not replace the controller's version number or invent an approval. Here, authoritative state is small enough to retain directly.

## 11 Stop on evidence and enforce cancellation

Permit at most six generated proposals in the core trial. Every generation attempt counts, including invalid output. An error can inform the next attempt; it does not create free extra calls. Two consec-

utive invalid or denied proposals end the run. This keeps a weak format follower from consuming an unbounded budget.

Use a 120-second per-call ceiling and a ten-minute experiment ceiling covering loading, generation, and final evaluation. Enforce the deadline through a trusted supervising process that can terminate the model worker. Wait only until the earlier of the call and experiment deadlines, and recheck cancellation and expiry after generation, immediately before dispatch, and immediately before an edit commits. A cooperative generation callback is useful but cannot guarantee that a hung inference call returns promptly. This worker executes the installed inference stack, never model-written code; it is a cancellation mechanism, not a security sandbox.

On cancellation, stop scheduling actions and reject late proposals. Preserve committed edits and their backups rather than claiming cancellation reversed them. Record whether a replacement committed before cancellation. A model worker cannot commit edits directly, so killing it cannot interrupt an editor halfway through a generated instruction.

`finish` is a request to stop. It is not proof of success. The controller accepts a completion request only after a passing public test bound to the current file version and SHA256. Every tool checks the managed files against their recorded digests; an external change stops the run rather than silently refreshing a passing-test flag. After termination, an independent evaluator checks that same verified final snapshot against all fixtures, including cases never returned to the model, if the deadline permits. Record both the termination reason and the final pass counts. A budget-exhausted run might leave correct code; that is a useful distinction, not a reason to rewrite its history.

## 12 Measure what the trace actually supports

Record generated text before parsing, validation decisions, dispatched actions, expression changes, version hashes, test observations, input/output token counts, and elapsed times. Keep the source of every proposal explicit: `scripted` or `model`. Log synthetic task content only; no credentials or private repository data are needed.

Separate model loading, first-call latency, subsequent generation, validation, editing, and testing. A long first call might reflect loading or initialization rather than difficult planning. Tokens per second is meaningful only when the measurement boundary is defined. A six-step task may fail because its model exhausted the output allowance halfway through valid JSON, not because it lacked the correct expression.

Useful outcomes include: the baseline repaired both functions; the real model made two invalid proposals; an edit succeeded but failed boundary tests; all tests passed but no valid finish arrived; or the model could not load within the resource policy. These are possible outcomes, not published observations from a run of this chapter.

The held-out fixtures are hidden from the agent's input and tools, not from the learner reading the Lab. They diagnose obvious overfitting to visible examples. They are too few to prove correctness over every integer input, and one task cannot measure general coding ability.

### Build and inspect the loop

Lab 16 — Build a Tiny Coding Agent provides the exact files, action contract, model manifest, fixtures, budgets, and acceptance checks. Complete the scripted harness checks first, then conduct a separate real-model trial. Preserve unsuccessful outputs in the submission.

## 13 Check your understanding

1. The scripted baseline passes, but the model produces prose instead of JSON. What has been verified, and what has failed?
2. Why does importing an edited module invalidate the proposed narrow test boundary?
3. A correct patch carries an old version number. Why should the editor reject it?
4. Why must both branches of a conditional expression be validated?
5. What changes when an input file contains an instruction to add a shell tool?
6. All fixtures pass after the sixth action, but the model never requests completion. Which facts should the report preserve?
7. Which additional isolation controls would be necessary before allowing arbitrary code execution?

### 13.1 Suggested answers

1. The baseline provides evidence that the tested harness path works. The model failed the required output contract in this trial; the scripted solution does not establish the model's ability to produce one.
2. Importing a module can execute its top-level code with the importing process's privileges. That exceeds the restricted expression interpreter's boundary.
3. A stale version no longer identifies the state on which the edit is authorized. Use a current source read or committed edit receipt, together with the matching digest and exact old expression.
4. An unselected branch still belongs to the proposed program. Validating both branches prevents forbidden syntax from being accepted merely because the current fixtures do not reach it.
5. The text supplies no new authority. Preserve its provenance as untrusted data; the closed dispatcher still rejects a shell proposal, even if the model follows the instruction.
6. Record proposal-limit termination, no accepted finish, the final version and digest, and the observed public and held-out pass counts. Passing fixtures alone does not satisfy this protocol's `verified_success` rule.
7. Independently configured OS or container isolation, an unprivileged identity, restricted mounts and network access, no host credentials or sockets, CPU/memory/process/time limits, and tested cleanup. A fixed command or a separate worker process is insufficient.

A useful agent combines learned suggestions with deliberately engineered state transitions. The model need not be reliable for the controller to reject forbidden operations reliably within its tested scope. And a carefully governed controller does not make a weak model good at coding. Keeping both statements true is the main accomplishment of this build.

## 14 More Learning

- Qwen2.5-Coder model repository: inspect the official model card, license, configuration, and pinned file history.
- Transformers chat templates: trace message structure into the actual token input.
- Transformers security policy: understand model serialization and remote-code boundaries.
- Python AST documentation: compare Python’s full grammar with the small language this Lab permits.
- ReAct: examine a published action–observation design and its evaluation tasks.
- SWE-agent: investigate why an agent’s computer interface belongs in the experimental description.