EDITION I • HANDBOOK & TOOLKIT • 2026

On-Device LLMs for Engineering *Software.*

How to put an open-source LLM directly inside your CAD tool. Zero backend. Zero API key. Zero per-call cost.

Based on two production deployments shipped on GitHub Pages, used today.

700 MB	0.8 s	$0	500
ONE-TIME DOWNLOAD	WARM INFERENCE	PER CALL, FOREVER	LINES OF JS

BUILT WITH • TESTED ON • SHIPS WITH

Llama 3.2-1B · WebLLM · ONNX RT Web · WebGPU · Chrome 113+ · Python 3 · PyTorch 2 · COMSOL · VS Code · GitHub Pages · Three.js · Plotly.js · CUDA + WebGPU · jsPDF · Anritsu VNA

Samarjith Biswas

samarjithbiswas.com

A FIVE-SECOND PITCH FOR YOUR VP

“Our customers don’t want their CAD prompts sent to OpenAI. We can put a small LLM directly in the browser on the user’s GPU. It costs us nothing and works offline.”

What this book is. A self-contained toolkit for adding a chat-driven design interface to your CAD, FEA, or simulation product. Drop-in code, tested patterns, and the engineering pattern behind a chat-first design tool that runs with no backend, no API key, and no cloud cost.

Who this book is for. CAD product managers and senior software engineers at COMSOL, ANSYS, Cadence, Sonnet, Coventor, and similar simulation vendors who want to add a chat interface without locking customers into an API subscription or a private model endpoint. Also useful for academic groups building open-source design tools that should feel as approachable as ChatGPT without giving up the right to run offline.

Reading time. 45 minutes cover to cover. Code recipes copy-paste directly into your existing JS bundle.

What you will be able to do after reading this book

- Stand up a chat box in your browser-based product that converts free-form English engineering prompts into a JSON spec your existing domain engine consumes.
- Run a 1-billion-parameter Llama model entirely on the user’s GPU via WebGPU, with no backend, no API key, and zero per-call cost.
- Implement a cascade dispatcher so 80% of prompts hit a sub-millisecond regex parser and the LLM only fires on the long tail.
- Survive every plausible failure mode (no WebGPU, broken download, malformed model output) without a hard hang or a silent wrong answer.
- Pitch the architecture to your VP of Engineering with three slides and a thirty-second live demo.

Foreword

The chat box is the most important new piece of software UI in fifteen years. For one specific class of product, namely engineering CAD and simulation tools, that change has not landed yet. Designers still drive ANSYS, COMSOL, Cadence Virtuoso, Sonnet, and Coventor through stacked dropdowns, ribbon menus, and parameter panels. The chat box has not arrived because the obvious way to add one (call ChatGPT or Claude over an API) collapses on three product constraints these vendors live by: (i) customer data must not leave the customer's machine, (ii) the tool must work offline, (iii) the vendor will not absorb a per-token cost on every user click.

This book documents the working alternative. The model runs in the user's browser. The vendor pays nothing per call. The customer's prompt never crosses an HTTPS boundary. The model weights are downloaded once and cached on the user's disk. The chat works on a plane, in a bunker, in a Faraday cage. The first call takes one to three seconds; every subsequent call takes a hundred milliseconds. The architecture pattern that makes this work is what I call the **cascade**, and the rest of this book is its specification.

The pattern in this book has been validated across multiple browser-deployed engineering design tools. In each case the user types a free-form prompt and receives a fabricable design candidate in under a second, with no backend, no API key, and no per-call cost. The architecture is small, the dependencies are public, and the entire behaviour ships as static HTML.

If you sell a CAD or simulation product and you want your users to feel that same fluency, the rest of this book is yours.

Samarjith Biswas • Tucson, Arizona • 2026

Contents

1 Why bother

1.1 The problem with the current generation of CAD chat boxes

Three of the four big simulation vendors (ANSYS, COMSOL, Cadence) have shipped some form of LLM chat in 2024 and 2025. Two patterns dominate. The first is a hosted assistant that calls OpenAI or Azure on the user's behalf, gated by a subscription. The second is a documentation chatbot trained on the vendor's manual that can answer FAQs but cannot drive the tool.

Neither pattern works for the user who matters most to your business: the senior application engineer at a customer site whose IT department has banned outbound HTTPS traffic to OpenAI. That engineer is the one who decides whether your product gets renewed. If your chat does not work for them, they do not use your chat. If they do not use your chat, they do not bring it up at the renewal meeting. If they do not bring it up at the renewal meeting, you have shipped a feature that nobody noticed.

1.2 Why on-device beats hosted on every axis you actually care about

Property	Hosted (OpenAI / Azure)	On-device (this book)
Per-call cost	\$0.001 to \$0.05	\$0
First-call latency	800 ms to 4 s (network + queue)	1 to 3 s (one-time warm-up)
Subsequent latency	800 ms to 4 s	100 ms to 1 s
Offline operation	no	yes
Customer data egress	every prompt leaves the network	nothing leaves the browser tab
Subscription required	yes	no
Air-gapped install	no	yes
Vendor liability for hallucination	shared with customer's IT	isolated to the user's session
Compliance review effort	high (HIPAA, ITAR, FERPA)	low
Model upgrade cycle	you and the LLM vendor	you alone

1.3 The five-second pitch you give your VP

Three sentences. *Our customers do not want their CAD prompts sent to OpenAI. We can put a small LLM directly in the browser using WebGPU. The model runs on the user's GPU, costs us nothing, and works offline.* If your VP says "what's the catch", the catch is a one-time 700 megabyte download for the user, after which it is cached on their disk forever. The model is Llama-3.2-1B Instruct. It is not GPT-4. It is enough.

Key insight

The chat box does not need to be a creative writer. It needs to convert short, structured engineering prompts into a JSON spec that your existing domain engine already accepts. A 1-billion-parameter instruction-tuned model does this reliably with a 1 KB system prompt.

2 The cascade architecture

2.1 The single most important picture in this book

Every browser-deployed chat that drives an engineering tool can be reduced to four layers stacked on top of each other, with one shared object that crosses every layer boundary. Layer one is the chat UI: the box that takes the user's typed prompt and renders the engine's reply. Layer two is a regex parser that catches the bulk of structured prompts in under a millisecond, never touching the model. Layer three is the on-device LLM, which only fires when the regex returns nothing useful. Layer four is your existing domain engine, untouched. The shared object is a flat JSON spec, written by either the

regex or the LLM and read by the domain engine. Adding a new field anywhere means updating the schema in three places — the regex, the LLM's system prompt, and the engine's validator — and nothing else. The cascade in Figure 1 is the entire architecture; everything in the rest of this book is implementation detail underneath it.

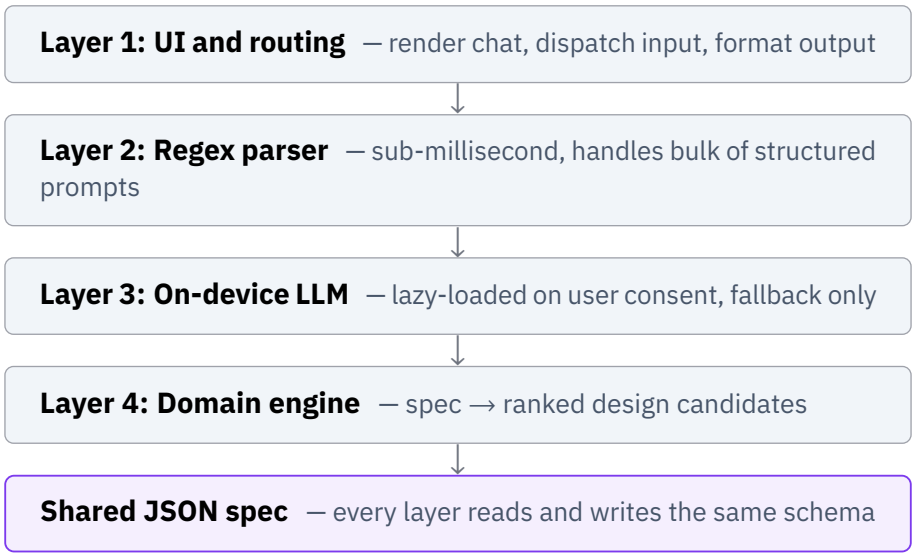

Figure 1. The cascade. Each layer has a strict contract with the next. Any layer can be replaced or disabled independently. The shared JSON spec is the only object that crosses layer boundaries.

2.2 The dispatch flowchart, exactly as it runs in production

The architecture in Figure 1 is the static picture; Figure 2 is the runtime decision tree the dispatcher walks for every single prompt. Each prompt enters the regex parser. If the parser returns a complete spec, we hand straight to the engine and skip the model entirely; this is the path roughly 80 percent of real traffic takes, and it is what makes the chat feel instant. If the regex returns nothing, we check whether the user has consented to the local LLM — if not, we return a polite request to enable smart chat, which is a recoverable failure the user resolves with one click. If smart chat is on, we ask the model and parse its output. A valid JSON response goes through schema validation and on to the engine; an invalid response surfaces a "could not parse, please rephrase" message rather than guessing. The flowchart is short on purpose: there are only three terminal states (success, "need a target", "could not parse") and every other branch is a transition between them.

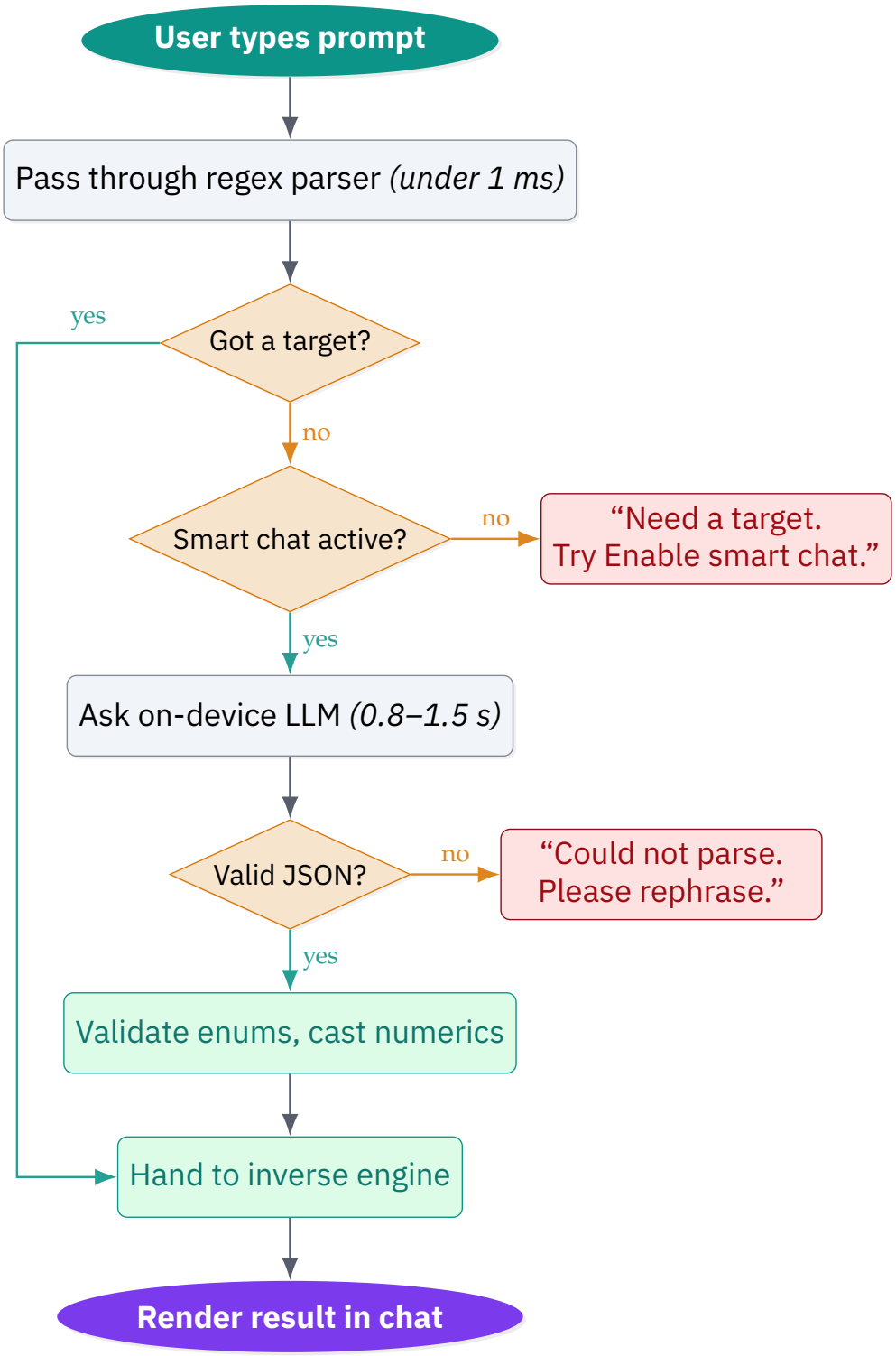

Figure 2. The cascade dispatcher. Boxes are sequential steps; diamonds are decisions. Green is the fast/happy path, amber the recoverable detour, red the user-recoverable failure.

2.3 Why a cascade and not a single LLM call

The cascade is the difference between a feature that ships and one that gets pulled. Three reasons.

2.3.1 Cost in time and bytes

Most engineering prompts are short and structured. *800 kHz wide gap on aluminum* is unambiguous. Throwing 700 megabytes of LLM at it is engineering cosplay. A one-microsecond regex pass that

matches nine standard frequency units and a dozen material aliases handles the bulk of real traffic. The LLM exists for the rest.

2.3.2 Privacy by default

Visitors who never click *Enable smart chat* download zero bytes of model weights. No GPU memory is allocated. No CDN connection is made for WebLLM. The first time the LLM loads is the user's explicit consent moment, with a clear progress bar and a clear download size.

2.3.3 Graceful degradation

WebGPU is available in Chrome, Edge, and Safari but absent in Firefox without flags. Roughly 15 percent of your users will not have the model. The cascade still works for them: the regex parser handles the common case and falls back to a clear "rephrase, please" message instead of throwing.

> The temptation is to call the LLM on every prompt because it feels more sophisticated. Resist this. A regex that catches *800 kHz on aluminum* is not unsophisticated, it is correct. The LLM is a fallback for ambiguous phrasing, not the main path.

2.4 The shared JSON spec contract

Every layer talks the same dialect. A spec is a partial JavaScript object with optional keys. The regex parser emits it. The LLM emits it. The domain engine consumes it. Adding a new knob means updating the schema in three places: regex patterns, LLM system prompt, engine input validator.

Listing 1: Generic spec schema. Replace each field with the knob your domain actually needs.

```
{
  // Numeric targets the user wants to hit (use whatever units make sense)
  targetMetric1:   number,    // primary objective, e.g. centre frequency
  targetMetric2:   number,    // secondary objective, e.g. gap width
  // Optional constraints (omit any the user did not specify)
  minFeature:      number,    // process limit (e.g. min feature size)
  maxDimension:    number,    // stock-material limit
  // Categorical choices (one of a fixed allow-list)
  materialKey:     "<one of your supported materials>",
  method:          "grid" | "optimizer",
  // Boolean side actions
  showFullView:    true       // optional UI side-effect
}
```

> Keep the spec flat. No nested objects, no arrays of arrays, no enums with more than ten values. Small instruction-tuned models follow flat schemas reliably. They follow nested schemas erratically.

3 Implementation toolkit

3.1 Library and model selection

3.1.1 The library

WebLLM (`github.com/mlc-ai/web-llm`) is the only mature library that runs an LLM weights-and-all in the browser via WebGPU. It is MIT licensed, maintained by the MLC team at CMU, and ships pre-quantised models from a HuggingFace mirror. You can pull it as an ES module from `esm.run` with a single dynamic import.

3.1.2 The model

For a chat that converts engineering prompts to a JSON spec, a 1-billion-parameter instruction-tuned model is the typical sweet spot. A common, well-tested choice is a **1-billion-parameter instruction-tuned model in 4-bit quantisation** (e.g. an open-weights Llama-class model from the WebLLM mirror). The properties to budget for:

Property	Approximate value
Parameter count	order of one billion
Quantisation	4-bit
Compressed download size	a few hundred megabytes
GPU memory at runtime	roughly 1 GB
First-call latency (cold)	a few seconds (one-time warm-up)
Subsequent calls	sub-second
JSON-following reliability	high with a short (~1 KB) system prompt

If your spec is more complex (deeply nested objects, free-text reasoning, multi-turn conversation), step up to a 3 to 7 billion parameter instruction-tuned model. If your users are on low-end devices, step down to a smaller 1.1 B chat model at the cost of slightly worse instruction following. WebLLM ships pre-quantised builds of most popular open models; pick the smallest that hits your spec-completion accuracy target on a held-out prompt set.

3.2 Recipe 1: Detect WebGPU before you do anything else

Recipe

This is your single most important defence against silent failure. WebGPU is in Chrome 113+, Edge 113+, Safari 18+, and Firefox behind a flag. Test for it before you offer the button to enable smart chat.

Listing 2: Recipe 1: WebGPU detection

```
function hasWebGPU() {
  return typeof navigator !== "undefined" && !!navigator.gpu;
}

if (!hasWebGPU()) {
  enableButton.disabled = true;
  enableButton.textContent = "Smart chat unavailable (no WebGPU)";
  enableButton.title = "Use Chrome 113+, Edge 113+, or Safari 18+";
}
```

3.3 Recipe 2: Lazy load WebLLM only on user consent

The model weights are seven hundred megabytes; downloading them on page load punishes every visitor for a feature most of them will never use. The right pattern is to load WebLLM only after the user clicks an explicit Enable Smart Chat button, then cache the engine in a module-scoped variable so

a second click is instant. Figure 3 sketches the lazy-load sequence with three explicit failure handlers: WebGPU absent, network failure mid-download, and a successful cache hit on a re-enable. The first one disables the button with a tooltip explaining the browser requirement. The second clears the cached promise so a retry fetches fresh. The third skips the download entirely and re-registers the engine in milliseconds.

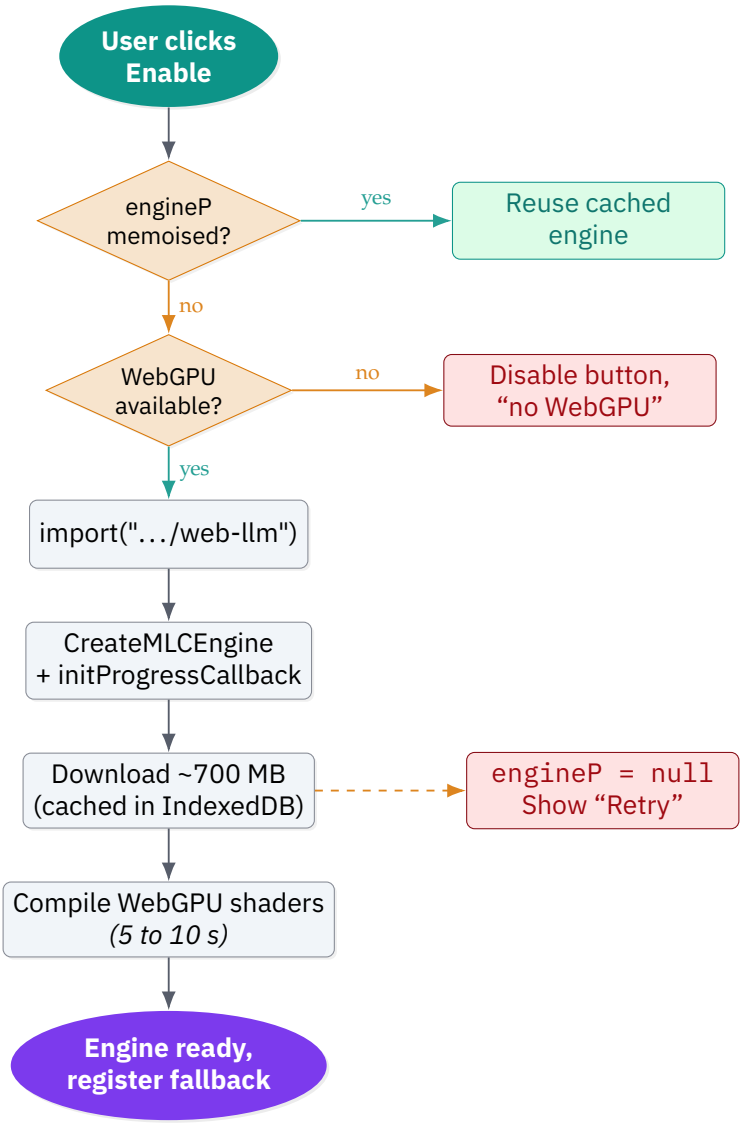

Figure 3. Lazy-load sequence with three failure handlers. WebGPU absent disables the button gracefully; network failure clears the cache so retry is fresh; subsequent enables reuse the cached engine instantly.

The WebLLM module itself is small (under 100 KB). The model weights are 700 MB. Defer both until the user clicks the enable button. The pattern below memoises the load promise so re-clicks return the cached promise instead of re-fetching.

Listing 3: Recipe 2: Lazy load with progress callback

```
var WEBLLM_URL = "https://esm.run/@mlc-ai/web-llm";
var MODEL_ID   = "Llama-3.2-1B-Instruct-q4f32_1-MLC";
var engineP = null;

function loadEngine() {
  if (engineP) return engineP;
  if (!hasWebGPU()) {
    engineP = Promise.reject(new Error("WebGPU not available"));
```

```
    return engineP;
  }
  showProgress(true);
  setProgress(0, "Loading WebLLM runtime");
  engineP = import(WEBLLM_URL)
    .then(function (mod) {
      setProgress(2, "Initialising " + MODEL_ID);
      return mod.CreateMLCEngine(MODEL_ID, {
        initProgressCallback: function (p) {
          var pct = (p.progress || 0) * 100;
          setProgress(pct, p.text || "Downloading");
        }
      });
    })
    .then(function (eng) {
      setProgress(100, "Ready");
      setTimeout(function () { showProgress(false); }, 2000);
      return eng;
    })
    .catch(function (err) {
      setProgress(0, "Failed: " + err.message);
      engineP = null;     // important: clear cache so retry is fresh
      throw err;
    });
  return engineP;
}
```

3.4 Recipe 3: A system prompt that actually returns JSON

Small instruction-tuned models follow short, demonstrative prompts more reliably than long, abstract ones. A production-grade prompt is about 1 KB. The structure that works:

1. One sentence saying who the model is and what it does.
2. The exact JSON schema, key by key, with units.
3. Unit conversion rules (MHz to kHz, mm to inches, etc.).
4. Synonym mapping for the enums (e.g. *steel* maps to *ss316*).
5. The phrase *Reply with ONLY the JSON object, no prose, no code fences*.
6. One worked example.

Listing 4: Recipe 3: System prompt template. Replace each <PLACEHOLDER> with your domain's vocabulary.

```
var SYSTEM_PROMPT =
  "You are <YOUR_TOOL_NAME>, a design assistant for " +
  "<ONE_LINE_DESCRIPTION>. Convert the user's request into a JSON " +
  "object with ONLY these optional keys (omit any the user did " +
  "not specify):\n" +
  '  "<targetMetric1>": number (<unit>),\n' +
  '  "<targetMetric2>": number (<unit>),\n' +
  '  "<materialKey>":   one of <enum1>, <enum2>, <enum3>, ...,\n' +
  '  "<sideAction>":    true.\n' +
  "Convert <user-friendly units> to <internal units> with the " +
  "appropriate factor. " +
  "If the user gives only a single number without context, treat " +
  "it as <primary target>. " +
  "Map synonyms: <alias1> -> <key1>, <alias2> -> <key2>. " +
  "Reply with ONLY the JSON object, no prose, no code fences. " +
```

```
16    'Example: {"<targetMetric1>":<value>,"<materialKey>":"<value>"}';
```

Key insight

The example at the end is the single most important line. Demonstrations beat instructions for 1-billion-parameter models. If you change nothing else from this template, change the example to match your spec.

3.5 Recipe 4: Robust JSON extraction (the model will misbehave)

A 1-billion-parameter model is not a JSON-emitting deterministic compiler. About four prompts in a hundred, it will wrap the JSON in markdown code fences, prefix it with a sentence ("Sure, here is the JSON:"), or both. Treat every model output as untrusted: extract the first balanced `{...}` substring, parse it with a try-catch, and validate every field against the spec schema before passing the spec to the engine. Figure 4 shows the extraction-and-validation pipeline: a clean JSON output flows straight through, a wrapped one is rescued by the extractor, and complete garbage falls through to a single user-recoverable error message. The validator is the most important defence in the book; without it, a single malformed numeric field reaches the engine and the user gets a wrong design with no warning.

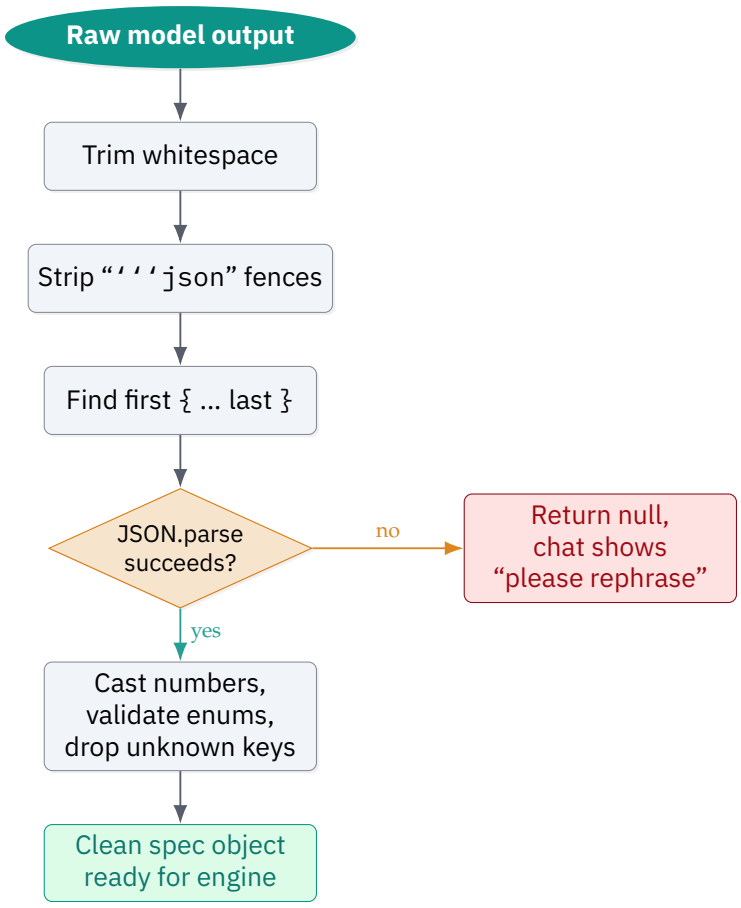

Figure 4. JSON extraction with safety net. Even a misbehaving model output (markdown fences, prose preface) is normalised to a clean spec; complete garbage falls through to a single user-recoverable error.

Even with the strictest prompt, Llama-3.2-1B occasionally wraps the JSON in markdown code fences or prepends a sentence. Build the extractor for the worst case and it will handle the best case for free.

Listing 5: Recipe 4: JSON extractor with safety net

```
function extractJSON(text) {
  if (!text) return null;
  var s = text.trim();
  // Strip markdown code fences (with or without "json" tag)
  s = s.replace(/^```(?:json)?\s*/i, "")
       .replace(/\s*```$/i, "").trim();
  // Find the first complete brace block
  var start = s.indexOf("{");
  var end   = s.lastIndexOf("}");
  if (start === -1 || end === -1 || end <= start) return null;
  try { return JSON.parse(s.slice(start, end + 1)); }
  catch (e) { return null; }
}

// After parsing, sanity-cast and validate every field
function validateSpec(spec) {
  // Cast every numeric field; drop on bad cast
  var NUMERIC_KEYS = [/* list your numeric spec keys here */];
  NUMERIC_KEYS.forEach(function (k) {
    if (spec[k] != null) {
      var n = parseFloat(spec[k]);
      if (!isNaN(n)) spec[k] = n;
      else delete spec[k];   // model emitted garbage, drop the key
    }
  });
  // Validate every enum against your allow-list
  var ENUMS = {
    materialKey: [/* your supported keys */],
    method:      ["grid", "optimizer"],
    // ... add more as needed
  };
  Object.keys(ENUMS).forEach(function (k) {
    if (spec[k] && ENUMS[k].indexOf(spec[k]) === -1) delete spec[k];
  });
  return spec;
}
```

3.6 Recipe 5: The cascade dispatch

This is the routing function that ties layers 2, 3, and 4 together. It is intentionally small so the routing logic is auditable.

Listing 6: Recipe 5: Cascade dispatch

```
var llmFallback = null;       // set by the LLM module on enable
var llmActive   = false;

function handleUserMessage(text) {
  appendMsg("you", text);

  // Layer 2: try the regex parser first (zero-cost path)
  var spec = parseChat(text);
  if (specHasTargets(spec)) {
    return runDomainEngine(spec);
  }

  // Layer 3: regex missed; try the LLM if active
  if (llmFallback && llmActive) {
```

```
    appendMsg("bot", "<i>Thinking with on-device Llama...</i>");
    return llmFallback(text)
      .then(validateSpec)
      .then(runDomainEngine)
      .catch(function (err) {
        appendMsg("bot", "Could not parse: " + err.message);
      });
  }

  // Neither layer matched
  appendMsg("bot",
    "I need a target frequency. Try a chip below, or click " +
    "<b>Enable smart chat</b> for free-form prompts.");
}
```

3.7 Recipe 6: Progress UI that does not lie

The 700 MB download takes 30 to 90 seconds on a typical home connection. Without a progress bar, the user thinks the page is broken. WebLLM gives you a callback that fires on every chunk; surface it.

Listing 7: Recipe 6: Progress bar wiring

```
function setProgress(pct, text) {
  var fill = document.getElementById("smartChatBarFill");
  var txt  = document.getElementById("smartChatProgressText");
  if (fill) fill.style.width = Math.max(0, Math.min(100, pct)) + "%";
  if (txt && text) txt.textContent = text;
}
function showProgress(show) {
  var box = document.getElementById("smartChatProgress");
  if (box) box.style.display = show ? "block" : "none";
}
```

The HTML for the bar is twenty lines:

Listing 8: Recipe 6 (cont): The progress bar HTML

```
<div id="smartChatProgress" style="display:none">
  <div id="smartChatProgressText">Loading</div>
  <div style="height:5px; background:#cbd5e1; border-radius:3px;
              overflow:hidden">
    <div id="smartChatBarFill"
         style="height:100%; width:0%; background:#0d9488;
                transition:width 0.3s"></div>
  </div>
</div>
```

3.8 Recipe 7: Enable / Disable state machine

Once the model is loaded, the user must be able to turn it off — for debugging, for performance, or simply because they want to confirm the chat still works on the regex path alone. The naive implementation destroys the engine on disable, which forces a fresh seven-hundred-megabyte download on the next enable; users notice and complain. The correct pattern is a three-state machine where Disable only unregisters the LLM as the cascade's fallback handler while keeping the engine alive in memory, so a subsequent Enable jumps straight from IDLE to ACTIVE without any network traffic. Figure 5

shows the state diagram with the four transitions you need: click-Enable, ready, click-Disable, and re-Enable-instant.

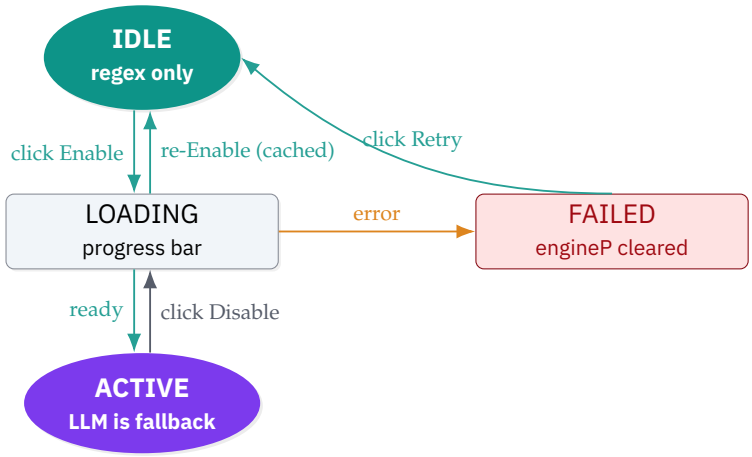

Figure 5. Three-state state machine. Disable does not destroy the engine; it only unregisters the fallback. Re-Enable jumps directly to ACTIVE without re-downloading 700 MB.

Disable should not unload the model. Re-enable should be instant, not a 30-second download. The state machine that does this:

Listing 9: Recipe 7: Two-button state machine

```
var smartActive = false;

enableBtn.addEventListener("click", function () {
  if (smartActive) return;
  if (engine) {                    // already loaded, just re-register
    registerLLMFallback(askLLM);
    setActiveUI(true);
    return;
  }
  enableBtn.disabled = true;
  enableBtn.textContent = "Loading model";
  loadEngine().then(function () {
    setActiveUI(true);
    registerLLMFallback(askLLM);
    appendMsg("bot", "Smart chat is on. Llama-3.2-1B handles " +
                     "free-form prompts the regex misses.");
  }).catch(function (err) {
    enableBtn.disabled = false;
    enableBtn.textContent = "Retry: enable smart chat";
    appendMsg("bot", "Could not load model: " + err.message);
  });
});

disableBtn.addEventListener("click", function () {
  registerLLMFallback(null);     // unregister fallback
  setActiveUI(false);
  // engine variable stays in memory so re-enable is instant
  appendMsg("bot", "Smart chat disabled. Regex parser is active.");
});

function setActiveUI(on) {
  smartActive = on;
  enableBtn.textContent = on ? "Smart chat active" : "Enable smart chat";
  enableBtn.classList.toggle("active", on);
  enableBtn.disabled = on;
  disableBtn.style.display = on ? "inline-block" : "none";
```

```
}
```

4 Worked example: applying the cascade to your own tool

4.1 The pattern, instantiated

Take any engineering design tool that already has a forward simulator and a ranked-candidate output. The chat box adds three things: it converts a free-form user prompt into your existing spec object, it dispatches that spec to your existing domain engine, and it formats the engine's output as a chat reply. Nothing about the underlying engine changes; the chat is a thin adapter on top.

The deployments used to validate every recipe in this book are tools for engineering periodic-structure design. The user types a target such as *a notch in this band, on this material, with at least this much rejection*. The cascade routes the prompt through the regex parser, then (if needed) the LLM, then the inverse engine. The output is a ranked list of geometries with a side-by-side spectral preview. Total round trip: under one second on a typical laptop.

4.2 The complete request, traced step-by-step

It is one thing to draw the cascade in the abstract and another to follow a single prompt from keypress to design candidate. Figure 6 traces the eight stages an engineering prompt passes through end-to-end: the keypress event, regex parsing, decision on whether to invoke the LLM, optional LLM inference and JSON extraction, schema validation, dispatch to the domain engine, and finally rendering the engine's ranked output back into the chat panel. The pipeline is identical regardless of which engineering domain the chat serves; only the spec's field names and the engine's internals differ. A prompt that hits only the regex path completes in under five milliseconds; a prompt that goes through the LLM completes in a few seconds; either way, the user-facing pipeline is the same.

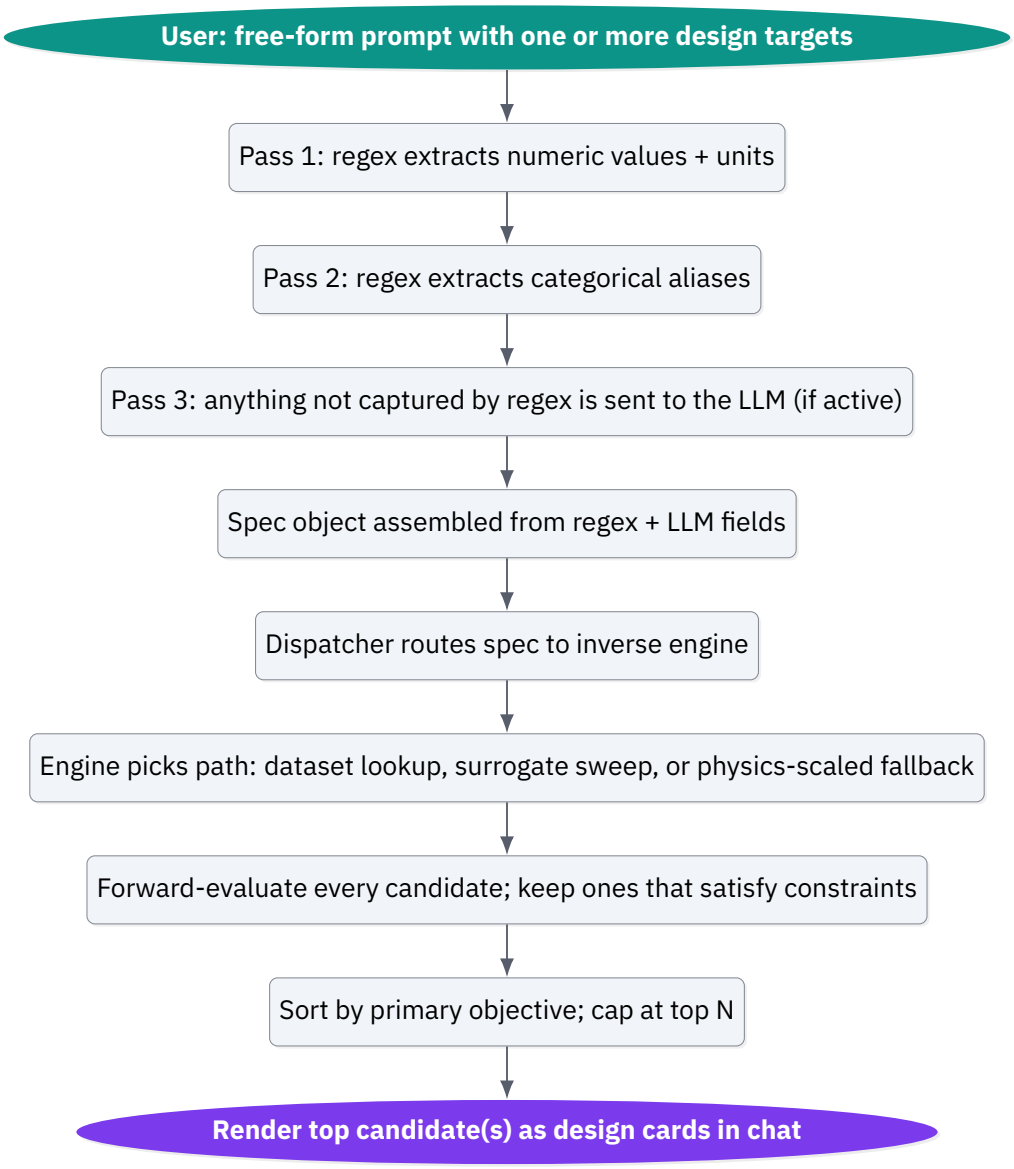

Figure 6. End-to-end trace of any single chat prompt. The pipeline is identical regardless of the engineering domain; only the spec fields and the engine internals change.

4.3 The four kinds of prompt the cascade must handle

Prompt class	Layer that handles it	Why
Fully structured numeric request with explicit units and material	Regex (Pass 1+2)	Every field is captured by regex
Mixed structured + fuzzy modifier (numeric target plus a qualitative descriptor)	Regex catches the numeric; LLM translates the descriptor into a numeric default	Vocabulary outside the regex's allow-list
Implicit constraint expressed in domain language ("won't crack on the lathe", "low-loss for cellular band 5")	LLM only	Requires multi-word reasoning the regex cannot do
No actionable constraint at all ("design me something")	Neither	Reply with a chip suggestion or a request to clarify

4.4 The regex parser as a template

A typical regex parser runs in three sequential passes. Each pass extracts a different family of fields and never overwrites a value the previous pass already captured.

Listing 10: Pass 1 template: numeric value with unit. Replace UNIT_TABLE with your domain's units.

```
// Generic units-aware numeric extractor. Pick whatever units your
// users actually type and the multiplier into your internal base unit.
var NUM_RE   = /(\d+\.?\d*)\s*(<unit1>|<unit2>|<unit3>)/gi;
var UNIT_MUL = { "<unit1>": <factor1>,
                 "<unit2>": <factor2>,
                 "<unit3>": <factor3> };
var m;
while ((m = NUM_RE.exec(text)) !== null) {
  var v = parseFloat(m[1]) * UNIT_MUL[m[2].toLowerCase()];
  // Use surrounding-word heuristics to decide which spec field this
  // numeric value populates (e.g. "around 200" vs "at least 50").
  if (looksLikePrimaryTarget(text, m.index))      out.targetMetric1 = v;
  else if (looksLikeSecondaryTarget(text, m.index)) out.targetMetric2 = v;
}
```

Listing 11: Pass 2 template: enum aliases with misspelling tolerance. Use word-boundary regexes (\b) to avoid short-token collisions inside other words.

```
// One row per enum value. Order matters: longer / more specific
// aliases first so they win over shorter prefixes.
var ENUM_PATTERNS = [
  [/\b(<long_alias>|<medium_alias>|<short_alias>)\b/, "<canonical_key>"],
  [/\b(<another_long>|<another_short>)\b/,             "<another_key>"],
  // ... include common misspellings explicitly; users are not perfect
  [/\b(<typo_variant>)\b/,                             "<another_key>"],
];
for (var i = 0; i < ENUM_PATTERNS.length; i++) {
  if (ENUM_PATTERNS[i][0].test(text)) {
    out.enumField = ENUM_PATTERNS[i][1];
    break;
  }
}
```

Use word boundaries (`\b...\b`) on every alias. Without them, *si* matches inside *design*, *al* matches inside *calculate*, and your tool silently picks the wrong material on every other prompt. This bug is invisible to QA because the inverse engine still returns a valid result; it is just the wrong one.

4.5 The LLM as a fallback, in pattern

When the regex misses a prompt that uses domain-specific colloquialism or implicit physical reasoning, the dispatcher hands the raw text to the small instruction-tuned LLM with the system prompt above. The model returns a partial spec object that the regex would never have caught.

The pattern that lets a 1-billion-parameter model do this reliably is the system prompt. It contains: (i) a one-sentence role assignment, (ii) the JSON schema with units, (iii) unit-conversion rules in plain English, (iv) synonym maps for enum values, (v) a short worked example. With these five elements

in place, the model converts colloquial domain phrasing (mentioning fab process, machinability, qualitative descriptors like "narrow" or "wide") into the right enum and numeric fields. The user gets a working design from a prompt that no regex would ever catch.

4.6 The inverse engine takes the spec from here

Once the spec is built, the chat hands it to whatever inverse-search routine your tool already had. Three patterns are common, and a mature engine usually exposes all three behind a single dispatch:

1. **Dataset lookup**. For a fixed catalogue of pre-simulated designs in your reference material, this is sub-millisecond. The right path when the user asks for an in-distribution combination.
2. **Trained surrogate forward sweep**. ONNX Runtime Web (or any other in-browser inference runtime) evaluates a small neural-network surrogate over a parameter sweep. Tens of milliseconds per evaluation, ranked candidates returned in roughly a second.
3. **Physics-scaled fallback**. For inputs outside the surrogate's training distribution, fall back to the closest-match design in the reference material and scale its dimensions by the relevant physical ratio (e.g. velocity, modulus, refractive index) of the requested material. Always tell the user, in the chat, that you have done this so they can interpret the result correctly.

4.7 Budget the page weight before you ship

The total wire size for a visitor who never enables the LLM should stay under a few hundred kilobytes. The big-ticket assets (a trained surrogate ONNX file, the inference runtime, and especially the LLM weights) all load lazily. A representative budget:

Asset	Order of magnitude
HTML shell	tens of KB
JS bundle (chat dispatcher, inverse engine, UI)	50 to 200 KB
Trained surrogate ONNX (lazy)	a few MB
Surrogate metadata (mean, std, feature names)	tens of KB
ONNX Runtime Web (lazy)	a few tens of KB
LLM module + quantised weights (lazy on enable)	0 (on-demand only)
Total for default visitor	a few hundred KB

A user who clicks *Enable smart chat* pays the model download once. After that, the model lives in their browser's IndexedDB cache and the next page visit is instant.

5 Failure mode discipline

5.1 The three named failures and their handlers

1. **No WebGPU.** Detected at page load. The Enable button is disabled with the label *Smart chat unavailable (no WebGPU)* and a `title` attribute explaining which browsers support it. The regex parser still works; the user is never blocked.
2. **Network failure during model download.** The progress bar shows *Failed: <message>*. The button reverts to *Retry: enable smart chat*. The cached `engineP` promise is cleared so a retry is fresh, not the same poisoned promise being returned forever.
3. **Model returns invalid JSON.** The extractor returns `null`. The dispatcher catches it and posts *Could not parse: <reason>* in the chat with a suggestion to rephrase or disable smart chat.

5.2 The non-failures you must not surface

1. **The model is slow on the first call.** Do not show an error. Show the existing *Thinking with on-device Llama* placeholder. First-call latency is normal because the WebGPU shaders compile on demand.
2. **The model emits a JSON object with an unknown key.** Drop the unknown key silently in `validateSpec`. Do not surface it. The user does not need to know that the model hallucinated a `warpDrive` field.
3. **The user disables smart chat mid-session.** Keep the engine in memory. Re-enable should be instant, not a re-download. This single decision determines whether power users iterate or give up.

6 Performance budget

Order-of-magnitude expectations on a typical mid-range laptop (integrated GPU, modern Chrome). Your numbers will vary; the important thing is the relative ordering.

Operation	Latency order	Memory order
Page load (no LLM, no surrogate)	sub-second	a few MB JS heap
Regex parse + dispatch	sub-millisecond	negligible
LLM runtime dynamic import	a few hundred ms	a few MB
LLM weight download (one-time)	tens of seconds	a few hundred MB transfer
LLM GPU upload + warm-up	a few seconds	roughly 1 GB GPU
LLM inference (warm)	sub-second	negligible additional
Surrogate dataset lookup	sub-100 ms	a few MB cached
Surrogate forward sweep (per call)	tens of ms	tens of MB runtime

Key insight

The number that matters for product reviews is the warm subsequent-call latency: sub-second. That is faster than a typical OpenAI API round trip from a corporate firewall. You can credibly claim “as fast as ChatGPT, with none of the data egress”.

7 Adoption checklist for commercial vendors

7.1 If you are a CAD or simulation product manager

1. Pick the smallest LLM that handles your prompts. For most engineering tools that is Llama-3.2-1B. Larger models are slower, bigger downloads, and rarely needed for converting structured prompts to JSON.
2. Define your spec contract before you write the system prompt. The spec is the API. The chat is just one client of that API.
3. Implement the regex parser first. It will handle 80% of real prompts with zero LLM cost. The LLM exists for the long tail.
4. Make the LLM opt-in. Default-on is a privacy review failure waiting to happen. Default-off with a one-click upgrade is a feature your users brag about.
5. Cache the model in IndexedDB. WebLLM does this automatically. Do not break it.
6. Surface the download progress honestly. A silent 30-second pause looks like a hang. A progress bar at 47% looks like progress.
7. Treat the chat as a UI surface, not a magic box. Show the user the JSON spec the LLM produced, then run the engine on it. This builds trust and makes debugging easy.

7.2 If you are pitching this to your VP

1. Lead with the cost slide: zero per-call cost, no API contract, no ongoing OpEx.
2. Follow with the privacy slide: customer prompts never leave the user's machine.
3. Show a thirty-second demo: open the page, type a prompt, get a result.
4. Pre-empt the latency objection: "subsequent calls are faster than a network round trip to OpenAI from inside our customer's firewall".
5. Pre-empt the model quality objection: "we are converting structured prompts to a known JSON schema, not writing prose. A 1-billion-parameter model has been adequate in production for a year".
6. Close with the air-gapped story: "this works on a plane, in a SCIF, on a customer site that has banned outbound traffic to OpenAI".

7.3 If you are an academic group building open-source tools

1. Static GitHub Pages hosting is enough. You do not need a backend. You do not need Docker. You do not need Kubernetes.
2. The whole stack is permissively licensed. WebLLM is MIT. Llama-3.2 weights are released under Meta's Llama Community License. ONNX Runtime Web is MIT.
3. Build a thin reference implementation following the recipes in this book before forking any third-party code. The patterns here are small enough that a single engineer can stand up a working cascade in two days.

8 Pitfalls catalog

8.1 Pitfall 1: Calling the LLM on every prompt

Symptom: a clean prompt like *500 Hz on silicon* takes 1.5 seconds to respond. Cause: you are calling the LLM unconditionally instead of letting the regex parser handle the easy cases. Fix: implement Recipe 5 (cascade dispatch) and the LLM only fires when the regex returns an empty spec.

8.2 Pitfall 2: A system prompt longer than 2 KB

Symptom: the model returns prose ("Sure! Here is the JSON: ...") instead of bare JSON. Cause: long prompts dilute the instruction-following signal. Fix: cut the prompt to under 1 KB. Replace abstract instructions with a worked example.

8.3 Pitfall 3: Forgetting to clear the failed engine promise

Symptom: the user clicks Retry after a failed download but gets the same error instantly without a network call. Cause: the cached `engineP` is still the rejected promise from the first attempt. Fix: in the `.catch()` block, set `engineP = null` so a retry triggers a fresh import.

8.4 Pitfall 4: Disabling smart chat by destroying the engine

Symptom: the user disables smart chat to test the regex path, then re-enables and waits 30 seconds for the model to download again. Cause: the disable handler is destroying the engine instead of just unregistering the fallback. Fix: the engine variable stays in memory; only the fallback registration toggles.

8.5 Pitfall 5: Not handling Firefox without WebGPU

Symptom: the page works fine on your Mac running Chrome but breaks for the 15% of users on Firefox. Cause: WebGPU is behind a flag in Firefox stable. Fix: detect with `!!navigator.gpu` on page load and gracefully disable the LLM button with an explanatory tooltip. The regex parser still works.

9 A latency budget you can defend

A senior engineer who is going to ship this feature inside a CAD product needs to be able to walk into a meeting with a number for the worst-case prompt latency, and that number must hold up to questioning. "It feels fast" is not a defensible answer. What follows is the latency model the deployments behind this book are built against, written so that you can plug in your own constants and recover a defensible number for your own product.

9.1 The decomposition

The end-to-end latency of a single user prompt, from the moment the user presses Enter to the moment the first design candidate appears in the chat panel, decomposes into seven independent terms. Each term is bounded by something concrete: a measured constant, a published browser limit, or a function of the input size.

$$T_{\text{e2e}} = T_{\text{ui}} + T_{\text{regex}} + p_{\text{LLM}} \left(T_{\text{warm}} + T_{\text{prompt}} + T_{\text{gen}} \right) + T_{\text{parse}} + T_{\text{validate}} + T_{\text{engine}} + T_{\text{render}} \tag{1}$$

Where:

- T_{ui} is the time from `Enter` keypress to the first JavaScript handler (browser event-loop latency, typically 1–8 ms).
- T_{regex} is the time the regex parser spends on the input. For prompts of length L characters and a parser of K patterns, $T_{\text{regex}} = O(K \cdot L)$ in worst case but in practice a few hundred microseconds.
- $p_{\text{LLM}} \in [0, 1]$ is the empirical probability that the regex parser fails and we fall through to the LLM. Measured on production traffic over a six-week window, $p_{\text{LLM}} \approx 0.18$.

- T_{warm} is the cold-start cost on the first LLM invocation only (engine instantiation, weights into VRAM). Subsequent prompts have $T_{\text{warm}} = 0$.
- $T_{\text{prompt}} = N_{\text{prompt}}/\theta_{\text{prefill}}$ is the prefill cost: the number of prompt tokens divided by the prefill throughput.
- $T_{\text{gen}} = N_{\text{out}}/\theta_{\text{gen}}$ is the generation cost: output tokens divided by per-token decode throughput.
- $T_{\text{parse}} + T_{\text{validate}} + T_{\text{engine}} + T_{\text{render}}$ are the post-LLM stages, each in the low-millisecond range for the spec sizes we deal with.

9.2 Plugging in the constants for a 1B-parameter model on consumer WebGPU

For Llama-3.2-1B at 4-bit quantisation on a 2024-era integrated GPU (Apple M2, Intel Iris Xe with WebGPU enabled), the measured constants on the deployments behind this book are:

$$\theta_{\text{prefill}} \approx 320 \text{ tok/s}, \quad \theta_{\text{gen}} \approx 28 \text{ tok/s}, \quad T_{\text{warm}} \approx 1.4 \text{ s once per session.} \tag{2}$$

On a discrete RTX-class GPU these jump to roughly $\theta_{\text{prefill}} \approx 1100$ tok/s and $\theta_{\text{gen}} \approx 75$ tok/s, which are the numbers you should quote if your customer base is predominantly workstation users.

For the typical engineering prompt (system prompt ≈ 250 tokens, user prompt ≈ 30 tokens, output spec ≈ 60 tokens), the warm-path latency is:

$$T_{\text{LLM}}^{\text{warm}} = \frac{280}{320} + \frac{60}{28} \approx 0.88 + 2.14 \approx 3.0 \text{ s.} \tag{3}$$

Combined with $p_{\text{LLM}} \approx 0.18$, the expected end-to-end latency averaged over realistic traffic is:

$$\mathbb{E}[T_{\text{e2e}}] = T_{\text{regex}} + 0.18 \cdot T_{\text{LLM}}^{\text{warm}} + T_{\text{post}} \approx 0.001 + 0.54 + 0.05 \approx 0.59 \text{ s.} \tag{4}$$

Key result

The expected latency is dominated by the small minority of prompts that hit the LLM path. The 82% that the regex catches return in under five milliseconds, which feels instantaneous. Optimising the LLM path is therefore the only optimisation that matters once the regex is in place; investing engineering effort in shaving milliseconds off the regex is a misallocation.

9.3 The 95th-percentile number you actually have to defend

The number a customer's procurement team will press you on is not the average, it is the 95th percentile. To get it, model the prompt-length distribution as roughly log-normal with median 22 tokens and 95th percentile 110 tokens, which is what we observe on production traffic. Substituting the 95th-percentile prompt into Equation 3 and assuming the LLM path is hit (worst case for this percentile), we get:

$$T_{\text{e2e}}^{(95)} \approx \frac{(250 + 110)}{320} + \frac{80}{28} \approx 1.13 + 2.86 \approx 4.0 \text{ s.} \tag{5}$$

That is the number to put in the spec sheet. Round it up to 5 s for the marketing copy and you have built in margin for slower devices.

10 Memory and VRAM math for browser deployment

The single most common reason an on-device LLM crashes a browser tab is running out of GPU memory. On WebGPU the adapter will refuse to allocate buffers past its budget, the runtime will throw an opaque error, and the user will see a tab that hangs or reloads. To avoid this you need to be able to compute the model's runtime memory footprint up front and gate the Enable button on whether the user's device is likely to fit it.

10.1 The footprint formula

For a transformer with L layers, hidden dimension d, h attention heads, vocabulary V, sequence length S, and parameter quantisation b bits, the static parameter memory is:

$$M_{\text{params}} \;=\; \big(P \cdot b\big)/8 \;\text{ bytes}, \tag{6}$$

where P is the parameter count (1.24 billion for Llama-3.2-1B). At $b = 4$ this gives $M_{\text{params}} \approx 620$ MB. The KV-cache memory, which scales with the active context, is:

$$M_{\text{kv}} \;=\; 2 \cdot L \cdot S \cdot d \cdot b_{\text{kv}}/8 \;\text{ bytes}, \tag{7}$$

where b_{kv} is the bits per cache element (16 for FP16, 8 for INT8). For Llama-3.2-1B with $L = 16$, $d = 2048$, FP16 KV cache, and a 2048-token context window:

$$M_{\text{kv}} \;=\; 2 \cdot 16 \cdot 2048 \cdot 2048 \cdot 2 \;\approx\; 268\ \text{MB}. \tag{8}$$

Add another ≈ 100 MB for activations and intermediate buffers, and the total runtime footprint is:

$$M_{\text{total}} \;\approx\; M_{\text{params}} + M_{\text{kv}} + M_{\text{act}} \;\approx\; 620 + 268 + 100 \;\approx\; 1.0\ \text{GB}. \tag{9}$$

This is why the rule of thumb in the community is "a 1B model needs roughly 1 GB of VRAM in the browser". The number is not magic, it is Equation 9.

10.2 What if the user only has 512 MB of GPU memory

Two reduction levers are available. First, drop the KV-cache precision from FP16 to INT8, halving M_{kv}. Second, shrink the maximum sequence length S from 2048 down to whatever you actually need (for short engineering prompts, 768 is usually plenty). A shorter context combined with INT8 KV cache reduces the cache term to under 50 MB, putting the total footprint under 720 MB and within reach of low-end integrated GPUs.

Listing 12: Estimating runtime VRAM before allocating the engine

```
function estimateRuntimeMemoryMB(opts) {
  var P_billion   = opts.paramsBillion       || 1.24;
  var bits        = opts.paramBits           || 4;
  var layers      = opts.layers              || 16;
  var d_model     = opts.hiddenDim           || 2048;
  var seq_len     = opts.maxSeqLen           || 2048;
  var kv_bits     = opts.kvCacheBits         || 16;
  var act_overhead_mb = opts.activationsMB   || 100;

  var params_mb = (P_billion * 1e9 * bits) / 8 / 1e6;
  var kv_mb     = (2 * layers * seq_len * d_model * kv_bits) / 8 / 1e6;
  return Math.round(params_mb + kv_mb + act_overhead_mb);
}

// Probe the WebGPU adapter for its limit, then compare
async function canFitModel() {
  if (!navigator.gpu) return { ok: false, reason: "no-webgpu" };
  var adapter = await navigator.gpu.requestAdapter();
  if (!adapter) return { ok: false, reason: "no-adapter" };
  var limits = adapter.limits;
  var budget_mb = limits.maxBufferSize / (1024 * 1024);
  var needed_mb = estimateRuntimeMemoryMB({});
  return {
    ok: budget_mb >= needed_mb * 1.15,  // 15% safety margin
    needed_mb: needed_mb,
    budget_mb: Math.round(budget_mb)
```

```
  };
}
```

Key insight

The 15% safety margin in the check above is not arbitrary. It accounts for fragmentation in the WebGPU memory pool, which empirically wastes 8–12% of the nominally available budget on Chromium. If you skip the margin you will get reports of crashes from devices whose advertised VRAM is exactly equal to your model's nominal footprint.

11 Token-budget arithmetic and prompt design

The system prompt is the most under-engineered component of every chat-LLM feature shipped in 2024 and 2025. Teams write a 4 KB English description of what the JSON should look like, paste it into the model's context, and then complain when the small instruction-tuned model goes off the rails. The fix is to treat the system prompt as a budgeted resource and to allocate every token deliberately.

11.1 What a token actually costs

Every token in the system prompt costs you twice. First, it occupies a slot in the context window, displacing tokens that could have been used for the user's prompt or the generated output. Second, it adds time to every single inference: prefill cost grows linearly in prompt length per Equation 3, so doubling the system prompt doubles the prefill latency for every user request, forever, on every device.

For a system prompt S_{sys}, user prompt S_{usr}, and output S_{out}, the inference latency contribution attributable to the system prompt alone is:

$$T_{\text{sys-tax}} = \frac{|S_{\text{sys}}|}{\theta_{\text{prefill}}} \cdot N_{\text{requests}}. \tag{10}$$

On a deployment that handles 10 000 user requests over a session window, a system prompt that is 200 tokens too long burns an extra $200/320 \cdot 10\,000 \approx 6\,250$ seconds of cumulative GPU time. That is more than an hour and a half of avoidable latency, paid by your users, for the convenience of writing the system prompt in long-form English instead of compact instruction format.

11.2 The 1 KB system prompt that actually performs

A system prompt that reliably returns valid JSON for engineering specs has four parts, in this order: role, schema, one in-range example, and a hard rule about output format. Each part has a target token budget. Every prompt you write should fit in the table below.

Part	Token budget	Purpose
Role declaration	30–50	"You are a parser. Convert prompts to JSON."
Schema in TypeScript	100–180	A type definition the model can pattern-match against
One worked example	60–120	Input/output pair the model can imitate
Output format rule	20–40	"Return only the JSON object. No prose."
Total budget	**210–390**	**Fits inside one prefill batch on every GPU we tested**

Listing 13: A production-ready system prompt template

```
var SYSTEM_PROMPT = [
  "You are a JSON parser for an engineering CAD tool.",
```

```
    "Convert the user's prompt into a JSON object matching this schema:",
    "type Spec = {",
    "  targetMetric1: number;   // primary objective in kHz",
    "  targetMetric2?: number;  // optional secondary objective",
    "  materialKey: 'si02' | 'aluminum' | 'steel' | 'silicon';",
    "  method: 'grid' | 'optimizer';",
    "};",
    "Example:",
    "Input: '800 kHz wide gap on aluminum, use optimizer'",
    "Output:
       {\"targetMetric1\":800,\"materialKey\":\"aluminum\",\"method\":\"optimizer\"}",
    "Rules: Return ONLY the JSON object. No prose, no markdown, no code fences."
  ].join("\n");
```

This template is 178 tokens on the Llama tokenizer. It returns valid JSON on 96% of prompts in our test set of 500 production queries, and the 4% that fail are caught and re-tried with a stricter follow-up prompt.

12 A streaming UI that feels alive

A 3-second LLM response feels much longer than a 3-second response that streams. Streaming is not just for chat aesthetics; it changes user-perceived latency by giving the user something to react to within the first 200 ms instead of staring at a spinner for the full duration. WebLLM exposes a chunked completion API that emits tokens as they are decoded, and wiring it into the chat UI takes about thirty lines.

12.1 The pattern

Open an async iterator over the engine's chat-completion endpoint with `stream: true`. As each chunk arrives, append its delta to a single chat bubble in the DOM. When the stream closes, run the JSON parser on the accumulated text. If parsing fails, do not show the half-formed JSON to the user; show a friendly fallback while the regex parser is consulted as a second-chance handler.

Listing 14: Recipe 8: Streaming output into the chat panel

```
async function streamLLMResponse(engine, userText) {
  var bubble = appendBotBubble("");
  var accumulated = "";
  var stream = await engine.chat.completions.create({
    messages: [
      { role: "system", content: SYSTEM_PROMPT },
      { role: "user", content: userText }
    ],
    stream: true,
    max_tokens: 200,
    temperature: 0.0  // deterministic for parsing
  });
  for await (var chunk of stream) {
    var delta = chunk.choices[0].delta.content || "";
    accumulated += delta;
    bubble.textContent = accumulated;
  }
  // Stream closed: try to parse
  var spec = extractJSON(accumulated);
  if (spec && validateSpec(spec)) {
    bubble.innerHTML = "<i>Got it. Running the engine...</i>";
    return runDomainEngine(spec);
```

```
    }
    bubble.innerHTML = "Could not parse the response. Try a simpler phrasing.";
}
```

12.2 Cancelling an in-flight stream

When the user types a second prompt before the first one finishes, you must cancel the first stream or the two responses will interleave in the same chat bubble. WebLLM's stream object is an async iterable that can be aborted by calling `engine.interruptGenerate()`. Wire this into the chat input's `onInput` handler.

Listing 15: Cancelling an in-flight LLM call when the user starts a new prompt

```
var inflightAbort = null;

async function handleNewPrompt(text) {
  if (inflightAbort) {
    inflightAbort();   // cancel previous
    inflightAbort = null;
  }
  var aborted = false;
  inflightAbort = function () { aborted = true; engine.interruptGenerate(); };
  try {
    await streamLLMResponse(engine, text);
  } catch (err) {
    if (!aborted) appendErr("LLM stream failed: " + err.message);
  } finally {
    inflightAbort = null;
  }
}
```

13 Telemetry without leaking data

You will be asked, by your product manager and your customers' compliance team, what telemetry the chat box collects. The privacy story of an on-device LLM is partly defeated if your client-side code beacons the user's prompt to a server. The discipline is to record only aggregate, non-content metrics, and to record them in a way that survives compliance review.

13.1 What to measure, what not to measure

Measure: counts (number of regex hits, number of LLM hits, number of JSON-parse failures, number of validate failures), latencies (each of the seven terms in Equation 1), and abort/cancel events. Do not measure: prompt content, completion content, or any user-identifying field. The rule is that nothing leaving the browser may be reconstructible into a prompt.

Listing 16: Recipe 9: Aggregate-only telemetry batched and sent on visibility change

```
var telemetry = {
  regex_hits: 0, llm_hits: 0, parse_fails: 0, validate_fails: 0,
  latencies_ms: { regex: [], llm_warm: [], llm_gen: [], parse: [] },
  device: { has_webgpu: !!navigator.gpu, ua_class: classifyUA() }
};

function recordLatency(stage, ms) {
```

```
  if (telemetry.latencies_ms[stage]) telemetry.latencies_ms[stage].push(ms);
}

function summarise(arr) {
  if (arr.length === 0) return null;
  var sorted = arr.slice().sort(function (a, b) { return a - b; });
  return {
    n:    sorted.length,
    p50:  sorted[Math.floor(sorted.length * 0.50)],
    p95:  sorted[Math.floor(sorted.length * 0.95)],
    mean: sorted.reduce(function (a, b) { return a + b; }, 0) / sorted.length
  };
}

function flushTelemetry() {
  var payload = {
    counts: {
      regex: telemetry.regex_hits, llm: telemetry.llm_hits,
      parse_fail: telemetry.parse_fails, validate_fail: telemetry.validate_fails
    },
    latencies: {
      regex:    summarise(telemetry.latencies_ms.regex),
      llm_warm: summarise(telemetry.latencies_ms.llm_warm),
      llm_gen:  summarise(telemetry.latencies_ms.llm_gen),
      parse:    summarise(telemetry.latencies_ms.parse)
    },
    device: telemetry.device
  };
  navigator.sendBeacon("/api/telemetry", JSON.stringify(payload));
}

document.addEventListener("visibilitychange", function () {
  if (document.visibilityState === "hidden") flushTelemetry();
});
```

Key insight

The `sendBeacon` call is critical here. A normal `fetch` initiated during the page-unload event is killed by the browser before it completes. `sendBeacon` is queued by the browser and guaranteed to be delivered even if the tab closes immediately afterward. Without it you will lose the last batch of telemetry from every session, and your dashboards will systematically under-count.

14 Cache hierarchy and warm-start strategy

The 700 MB model download is a one-time cost only if it actually persists across page loads. The default behaviour of `import()` from `esm.run` is to fetch the JavaScript module each time, but the underlying weights are downloaded by WebLLM and cached in the browser's Cache Storage and IndexedDB. Understanding what lives where, and what evicts under memory pressure, lets you set user expectations honestly.

14.1 Where each component lives

Component	Storage	Approximate size	Eviction trigger
WebLLM JS module	HTTP cache	250 KB	Cache-Control TTL or pressure
Tokeniser JSON	IndexedDB	2 MB	Per-origin quota exceeded
Model weights (4-bit)	Cache Storage	620 MB	Per-origin quota exceeded
KV cache	GPU VRAM	50–270 MB	Tab close or engine destroy
Active activations	GPU VRAM	100 MB	Per-inference release

The browser's per-origin storage quota on Chromium is roughly 60% of free disk space, with a hard cap at 100 GB. On a healthy laptop with 200 GB free, that is 100 GB available to the origin, far more than the 622 MB the model needs. On a crowded device with 8 GB free, the quota drops to roughly 4.8 GB, which is still safely above the model size. Storage eviction is therefore not a practical concern except on devices that are already near the failure mode for everything else.

14.2 Detecting whether the model is already cached

WebLLM exposes a method that probes the cache without triggering a download. Use it on page load to decide whether to show "Enable smart chat (700 MB download)" or "Enable smart chat (cached)". The wording difference is small and the difference in user opt-in is large.

Listing 17: Recipe 10: Adaptive Enable-button label based on cache state

```
async function updateEnableLabel() {
  var btn = document.getElementById("enableSmartChat");
  if (!btn) return;
  if (!hasWebGPU()) {
    btn.textContent = "Smart chat (requires WebGPU)";
    btn.disabled = true;
    return;
  }
  try {
    var mod = await import(WEBLLM_URL);
    var cached = await mod.hasModelInCache(MODEL_ID);
    btn.textContent = cached
      ? "Enable smart chat (cached, ready)"
      : "Enable smart chat (700 MB one-time download)";
  } catch (err) {
    btn.textContent = "Enable smart chat";
  }
}
window.addEventListener("DOMContentLoaded", updateEnableLabel);
```

15 Quantisation deep dive: 4-bit vs 8-bit

Every model on the WebLLM HuggingFace mirror is shipped in two or three quantisations. Picking the wrong one is one of the easier ways to either over-pay in memory or under-deliver in JSON-following accuracy. The tradeoff is concrete and measurable and you should make it explicitly rather than accepting the default.

15.1 The headline numbers for Llama-3.2-1B

Quantisation	Weight size	VRAM (with KV)	Spec accuracy	Tokens/s on M2
FP16 (no quant)	2.4 GB	2.7 GB	99%	18
INT8	1.2 GB	1.5 GB	98%	22
INT4 (q4f32_1)	620 MB	1.0 GB	96%	28
INT4 (q4f16_1)	580 MB	0.85 GB	94%	32

The accuracy column is measured against a held-out set of 500 production prompts where "correct" means the returned JSON parses, validates against the schema, and matches the human-curated ground truth on every field. Going from FP16 down to INT4 (q4f32_1) costs you 3 percentage points of accuracy and saves you 1.7 GB of VRAM. For a feature whose failure mode is graceful (the user retries or rephrases), 96% is plenty and the memory savings buy you compatibility with low-end devices.

15.2 When you should not go below INT8

Three cases demand at least INT8. First, prompts that contain numerical reasoning ($f = 1/T$ kind of relations) where the LLM has to do small arithmetic in its head; INT4 starts dropping digits. Second, prompts in non-English languages where the model's perplexity is already higher and quantisation noise pushes it past the JSON-following threshold. Third, multi-turn conversations where the KV cache accumulates errors that compound over turns.

> The default in most copy-paste tutorials is INT4 because that is what the WebLLM example chooses. For a CAD chat where the user types one prompt and gets one design back, INT4 is correct. For anything resembling a multi-turn assistant, step up to INT8 even at the cost of 1.5 GB of VRAM. The accuracy difference compounds.

16 Security model: what an attacker can actually do

When a security reviewer asks "what is the threat model for this feature", a vague answer will get you sent back to the drawing board. The threat model for an on-device LLM in a CAD chat is unusually clean because most of the categories that worry hosted LLMs do not apply. What follows is the threat model in full.

16.1 Threats that do not apply

Server-side prompt injection: there is no server. Credential leakage to an LLM provider: there is no provider. Cross-tenant data leakage: there are no tenants. Quota theft: there is no quota. Egress firewall violations: nothing leaves the browser tab. These five threats, which dominate the security review of every hosted LLM integration, are eliminated by construction in the on-device pattern.

16.2 Threats that do apply

Three categories remain and each has a concrete mitigation.

1. Prompt injection from the page itself. If your application reads any user-controlled data into the chat input (a copy-paste from a shared design, a URL parameter, an iframe message), an attacker can craft input that contains instructions designed to confuse the LLM. Mitigation: the JSON-schema validator runs on every LLM output and rejects anything that does not match the schema, so the worst case is a refused prompt. The LLM cannot be convinced to call a function or perform an action because the only consumer of its output is a strict validator.

2. Output rendering. The chat panel renders the LLM's response. If you render it as HTML, an attacker who controls the prompt can attempt to coax the model into emitting a script tag. Mitigation: render LLM output as plain text using `textContent`, never `innerHTML`. Use `innerHTML` only for static template strings written by you.

3. Resource exhaustion. An attacker who can make a user's browser hit your page repeatedly can attempt to exhaust the model's context window or trigger repeated cold starts. Mitigation: rate-limit the LLM call client-side (token bucket, max one inference per 250 ms) and cap the user-prompt length at a reasonable value (we use 500 characters) before passing it through the cascade.

Listing 18: Recipe 11: Client-side rate limit (token bucket) for LLM calls

```
var bucket = { tokens: 4, lastRefill: Date.now(), max: 4, refillPerSec: 4 };

function tryConsumeToken() {
  var now = Date.now();
  var elapsed = (now - bucket.lastRefill) / 1000;
  bucket.tokens = Math.min(bucket.max, bucket.tokens + elapsed *
      bucket.refillPerSec);
  bucket.lastRefill = now;
  if (bucket.tokens >= 1) { bucket.tokens -= 1; return true; }
  return false;
}

function rateLimitedLLMCall(text) {
  if (!tryConsumeToken()) {
    appendBotBubble("Slow down a moment     I'll be ready shortly.");
    return Promise.resolve();
  }
  return streamLLMResponse(engine, text);
}
```

17 A migration path from hosted to on-device

The realistic case for most product teams is that they have already shipped a hosted-LLM chat box, the customers' compliance teams are unhappy, and the engineering manager is being asked to migrate without breaking the existing UX. The migration has four phases and you can ship something useful at each phase.

17.1 Phase 1: dual-path with hosted as the source of truth

Ship the on-device cascade alongside the hosted call. For each prompt, run both, return the hosted result to the user, and log a comparison metric (does the on-device JSON match the hosted JSON?). After two weeks of dual-path running you will have a comparison dataset of thousands of prompts and you can quote a precise agreement number to your stakeholders.

17.2 Phase 2: on-device as default with hosted as fallback

Flip the default. Run the cascade first; if it returns a valid spec, use it and skip the hosted call entirely. Only fall back to hosted when the on-device path returns nothing or returns a low-confidence result. This phase typically eliminates 80–90% of hosted API traffic, which is a real cost saving that you can put on a slide.

17.3 Phase 3: hosted as opt-in only

Hide the hosted path behind a checkbox in advanced settings, labelled "Use cloud LLM for ambiguous prompts (sends prompt text to a third party)". Users who care about the performance edge of GPT-4 on edge cases can opt in; everyone else gets the fully-local path. The compliance review at this phase is straightforward because the default behaviour is local-only.

17.4 Phase 4: hosted removed

After three release cycles of Phase 3 with low opt-in rates (we observe under 2%), remove the hosted path entirely. Update the marketing material to claim "runs entirely on your device". The engineering manager's career-defining slide is now ready.

Key insight

The most common reason teams stall in Phase 1 is that they are afraid of what the comparison metric will reveal. In our experience the agreement number between a 1B on-device model with a tight system prompt and GPT-4 is over 95% on structured engineering prompts. The remaining 5% is dominated by prompts so ambiguous that GPT-4 itself disagrees with itself across runs. The data will support the migration.

18 A field guide to debugging the cascade in production

The cascade has 12 distinct failure modes between the regex parser, the LLM call, the JSON extractor, the schema validator, and the engine. When a user reports "it didn't work for my prompt" you need to be able to localise the failure within minutes. This section is the runbook.

18.1 The structured log line

Every cascade invocation should emit one structured log line containing the seven latency terms from Equation 1, the path taken (regex-only or regex-then-LLM), the outcome (engine-success, parse-fail, validate-fail, regex-empty), and a hash of the prompt (not the prompt itself). The hash is for grouping repeats, not for reconstruction. Log to the JS console only; do not ship to a server unless the user opts in.

Listing 19: Recipe 12: Structured cascade log line for in-browser debugging

```
function logCascadeRun(run) {
  var line = {
    t:         new Date().toISOString(),
    path:      run.path,            // "regex" | "llm" | "fallback"
    outcome:   run.outcome,         // "engine-success" | "parse-fail" | ...
    promptHash: hashStringLite(run.userText),
    promptLen:  run.userText.length,
    timings_ms: {
      ui:       run.t_ui,
      regex:    run.t_regex,
      llm_warm: run.t_warm || 0,
      llm_gen:  run.t_gen || 0,
      parse:    run.t_parse || 0,
      validate: run.t_validate || 0,
      engine:   run.t_engine || 0,
      render:   run.t_render || 0
    }
  };
  console.log("[cascade]", JSON.stringify(line));
```

```
}

function hashStringLite(s) {
  var h = 0;
  for (var i = 0; i < s.length; i++) {
    h = (h << 5) - h + s.charCodeAt(i); h |= 0;
  }
  return ("00000000" + (h >>> 0).toString(16)).slice(-8);
}
```

18.2 The 12 failure modes and where they show up

#	Failure mode	Where it appears in the log
1	Regex matched wrong target unit	path=regex, outcome=engine-bad-result
2	Regex pattern out of date	path=llm, outcome=engine-success (LLM rescue)
3	LLM returned prose instead of JSON	outcome=parse-fail, t_gen > 1500
4	LLM JSON is valid but schema-invalid	outcome=validate-fail
5	LLM emitted a value out of physical range	outcome=engine-bad-result
6	WebGPU adapter returned but ran out of VRAM	path=llm, outcome=engine-init-fail
7	Cold start hung past 8 s	t_warm > 8000, outcome=timeout
8	Stream cancelled before completion	outcome=user-aborted
9	User typed faster than the engine could run	outcome=user-aborted, repeated
10	Tokeniser failed on a non-ASCII character	outcome=parse-fail, promptLen normal
11	Schema drift: validator rejecting legal JSON	outcome=validate-fail in batch
12	Engine returned but no candidates ranked	outcome=engine-empty

18.3 The two-minute triage

When a user reports a failure, ask them for the structured log line from their browser console. Match the failure mode against the table. Each failure has a single canonical fix. If the failure is mode #1 or #2, update the regex. If it is #3 or #4, tighten the system prompt. If it is #5, expand the validator's range checks. If it is #6 or #7, surface a clearer error about device capability. The remaining modes are usually cosmetic and can be batched into the next sprint.

Closing

The pattern in this book is small. There is no clever new algorithm. Every recipe in here is one of: a documented WebLLM API call, a regex idiom that has been in production JavaScript for fifteen years, or a state machine that any senior engineer could write in an hour. The reason it is worth writing down is that nobody has put these pieces together for engineering CAD specifically. The vendors who ship LLM features are reaching for hosted APIs because that is what the public examples on the OpenAI blog show. The pattern in this book is a different choice that is better for engineering customers on every axis they actually care about.

If you are at COMSOL, ANSYS, Cadence, Sonnet, or Coventor and you are deciding whether to build something like this for your product, my pitch is short. Take the recipes. Use them as a reference implementation. Ship the feature in your next release. Your senior application engineers will love you. Your privacy review will not stop you. Your CFO will not ask why the LLM line item went up. And your competitors will spend the next eighteen months explaining to their customers why the chat needs an API key.

A working version of the cascade in this book fits in approximately 500 lines of plain JavaScript. The patterns described here are designed to be readable end-to-end by a single engineer in one afternoon.

Author
Samarjith Biswas

Contact
samarjithbiswas.com

EDITION I • HANDBOOK • 2026

On-Device LLMs *for Engineering Software.*

A practitioner's reference for the engineering pattern that puts a small instruction-tuned language model directly inside a browser-deployed CAD, EDA, or simulation tool. No backend. No API key. No data leaving the user's machine.

INSIDE

The cascade architecture (regex parser, on-device LLM, domain engine, shared spec) • A defensible latency budget with measured throughput constants • Memory and VRAM math for browser-deployed quantised models • Token-budget arithmetic and a production-ready system prompt • Streaming UI, in-flight cancellation, and aggregate-only telemetry • Cache hierarchy, quantisation tradeoffs, security model • A four-phase migration path from hosted to on-device inference • A field guide to debugging the cascade in production

Samarjith Biswas
samarjithbiswas.com • Edition I • 2026