

Supercharging the Web: Using On-Device AI In Your Browser

Modern browsers are evolving fast and newly shipped browser-native, on-device capabilities open the door to a new class of AI-powered experiences—automatic page summarisation, contextual insights, content extraction, and more.

In this webinar, we'll explore what native browser APIs (like Chrome's Prompt API) and powerful client-side frameworks like WebLLM can do, how developers can integrate them into real applications, and what this means for the future of browsing.

Expect live examples, and guidance on how to start building with these tools today.

Whether you build customer-facing web apps or internal tools, this session will show how AI-assisted browsing can transform user workflows.

Paul Harrison



Version 2.0a, 24th September 2026

About Paul Harrison

- Developer at FoCul, focusing on Front-end development with Angular
- Over 25 years experience with HCL Notes/Domino - everything from support and administration, to infrastructure, migrations and development



Email: paul.harrison@focul.net

TwitterX: @PaulHarrison



Agenda



- Introduction
- Chrome AI APIs
- Chrome AI API XPages Integration Demo
- WebLLM
- WebLLM Chat Demo
- Chrome APIs vs WebLLM
- Best Practices
- Summary

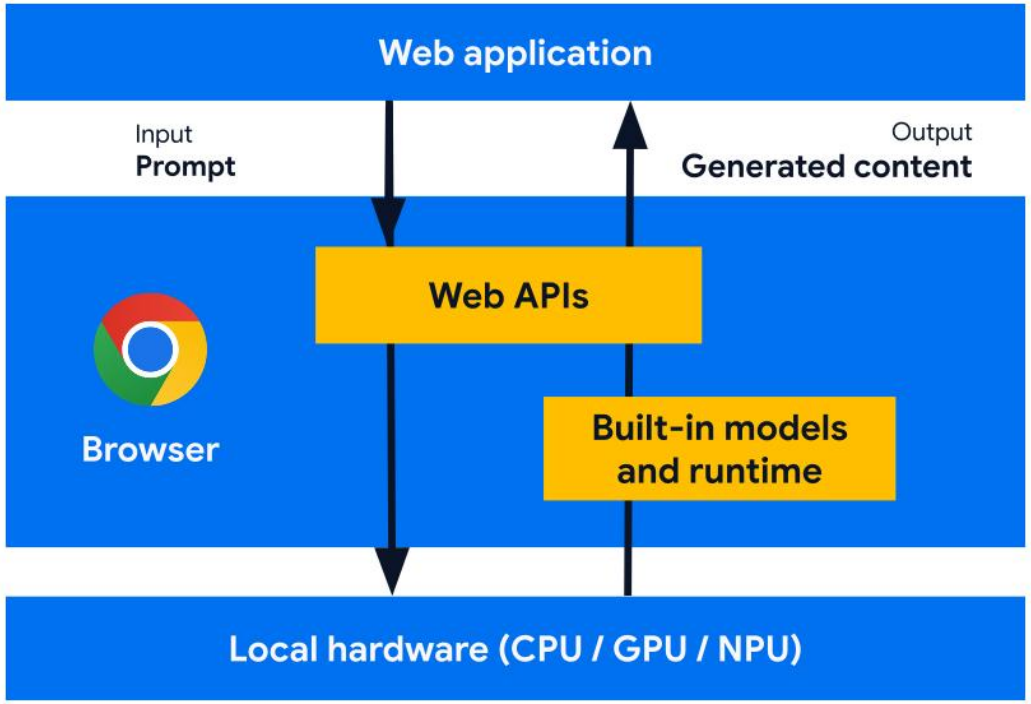
Traditional Approach to Providing AI



- Third-party cloud-based AI offering, using proprietary LLMs:
  - Potential high cost (either tokens and/or per-user basis)
  - Data privacy and sovereignty concerns
  - Variable performance
  - Your data may be used to train their model
- Self-hosted open-source LLMs on dedicated in-house servers:
  - Reduced Data privacy and sovereignty concerns
  - Additional hardware (may need GPU capabilities)
  - Additional workload to setup, manage and maintain
  - Could potentially use Domino IQ (14.5.0+) via custom APIs
- Locally hosted open-source LLMs on customer machines (e.g. Ollama, LM Studio):
  - Little or no Data privacy and sovereignty concerns
  - Support
  - Responsibility
  - Cost effective, but unlikely to be acceptable for most customers

Browser-based Approach

- Privacy:
  - Data stays on the user's device, retains full data privately and sovereignty
- Performance:
  - Local execution can consistently provide quick results by avoiding network latency
- Cost-Effective:
  - Developers can offer AI features without high inference costs, no Cloud AI, no API keys, no cost!
- Offline Functionality:
  - Features can continue to work without an internet connection
- Ease of deployment and use:
  - No additional software or infrastructure to deploy
  - Availability checks -> Create request session-> Perform the request -> Destroy the session



With built-in AI, your website connects with browser APIs to the local processor (CPU, GPU, or NPU). Then it communicates with a local model, which sends a response. The API returns the response.

Browser Coverage and Requirements



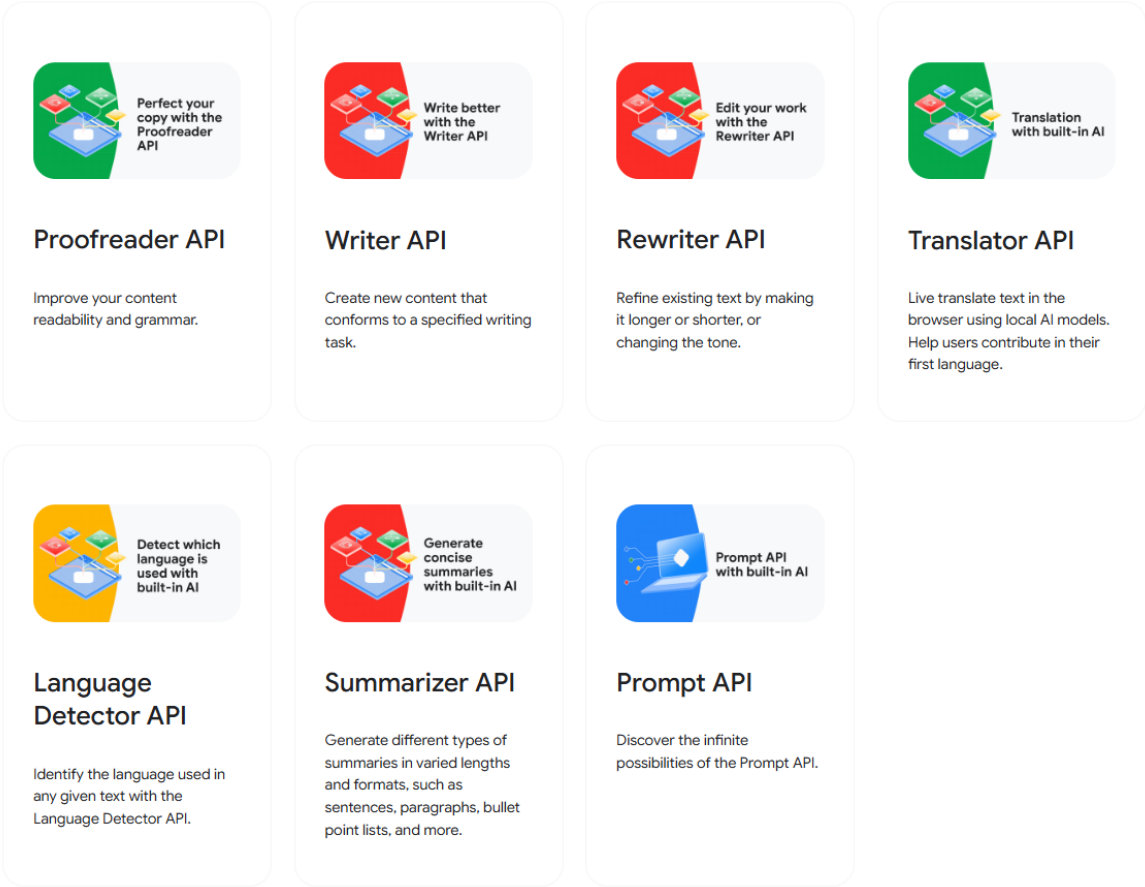
- Chrome:
  - 4 APIs shipped, plus 3 in Origin Trial
- Edge:
  - 2 APIs shipped
- Safari:
  - None available for third-party developers
- Firefox:
  - None publicly available

- Feature Flags may need enabling

- PC Requirements (typical):
  - Windows 11, macOS 13+ or Linux
  - At least 16 GB of RAM and 4 CPU cores, or a GPU with more than 4 GB of VRAM
  - Approximately 22 GB of free space for the initial Chrome profile

Available Chrome APIs

- Language Detector:
  - Identifies the language of a given text
- Translator:
  - Enables on-device translation between multiple languages
- Summarizer:
  - Condenses content into shorter formats
- Prompt:
  - A general interface that allows developers to use natural language
- Writer*:
  - Assist with content creation by helping users write new text
- Rewriter*:
  - Assist with content refinement by helping users refine existing content
- Proofreader*:
  - Focused on grammar correction and improving text quality



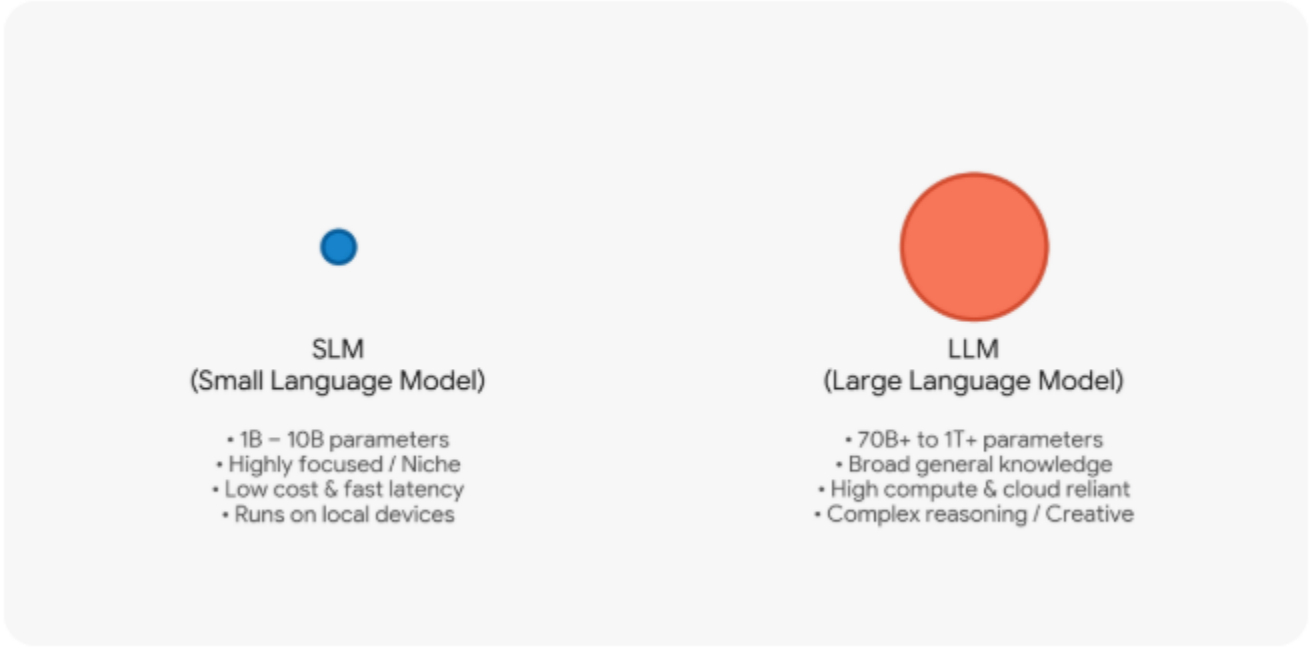
* Still in Origin trials as of September 2026

Language Model Overview



- Chrome uses Gemini Nano, whilst Edge uses Phi-4-mini
- There are both lightweight Models, classed as Small Language Models (SLM):
  - Optimised for on-device processing
  - Smaller file size (4Gb)
  - Lower compute and memory needs
  - Task-focused efficiency
  - Fewer Parameters
- Why SLMs matter for developers:
  - Architectural benefits:
    - Zero setup
    - Models are downloaded on-demand
    - Models are persisted
  - Abstracted APIs

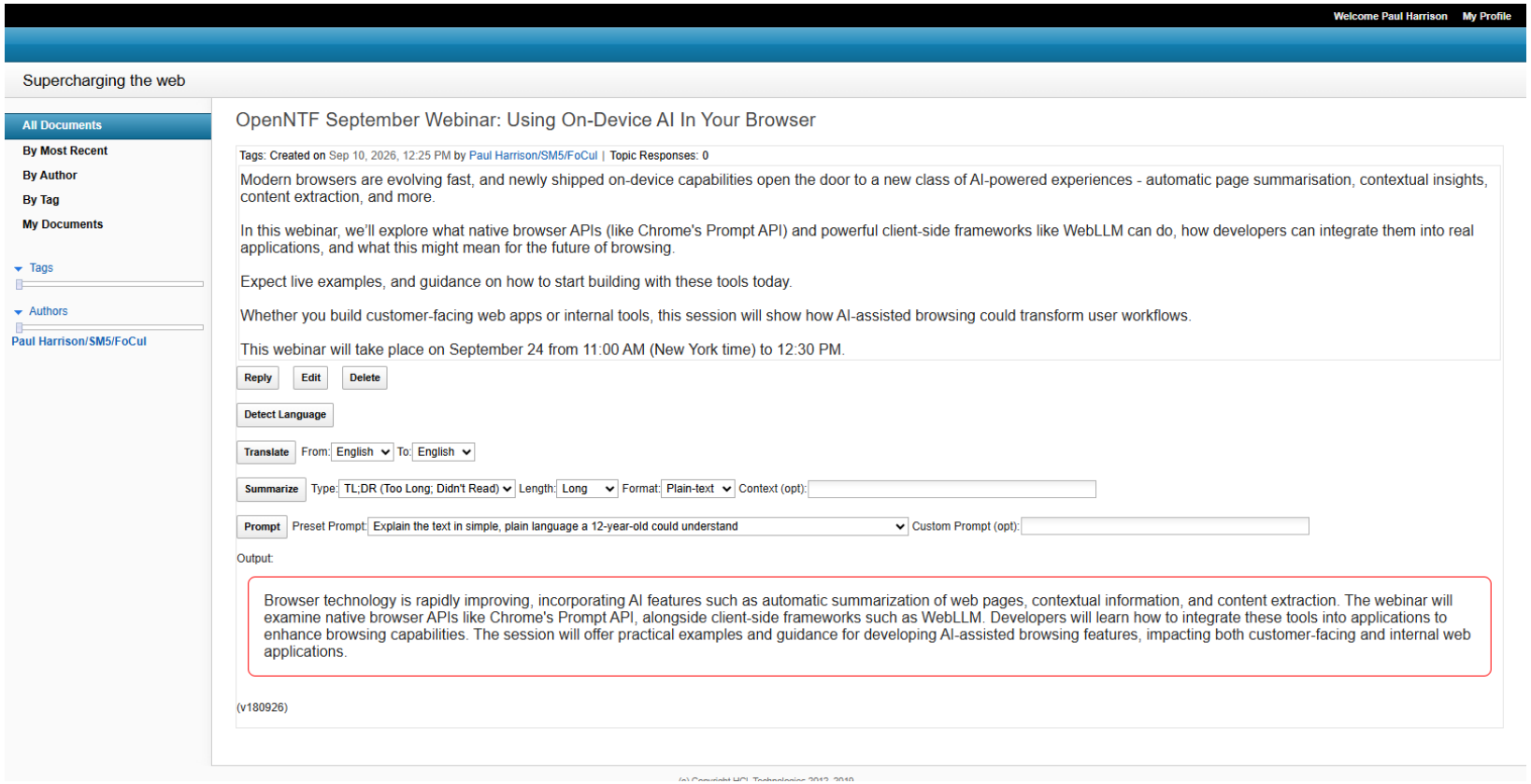
SLM vs. LLM



Metric	Small Language Model (SLM)	Large Language Model (LLM)
Parameter Count	Typically **10M to 10B** parameters	Ranging from **70B to over 1T** parameters
Core Strength	**Narrow focus** & domain specialization	**Broad versatility** & multi-step reasoning
Inference Cost	Ultra-low (~$0.05–$0.50 per M tokens)	High (~$5–$60 per M tokens)
Deployment	Local/On-device & Edge computing	Massive GPU clusters in the Cloud

Chrome AI API Demo

- Language Detector
- Translator
- Summarizer
- Prompt



Chrome AI API Implementation



- Code walkthrough
- The API abstracts the underlying SLM
- All APIs follow a similar pattern:
  - Check for API availability in the browser
  - Check for Model availability and device suitability
  - Create the API session (automatically downloads required support files)
  - Action the API request
  - Destroy the API session

Language Detector




```
// Define the required input/output elements...

var sourceTextElement = document.querySelector('[data-role="sourceText"]');
var streamingOutputElement = document.querySelector('[data-role="outputText"]');
var languageFromElement = document.querySelector('[data-role="languageFrom"]');


streamingOutputElement.textContent = "Processing Language Detector...";


// Check for browser API availability by testing for the presence of the LanguageDetector object...

if (LanguageDetector) {
  detectLanguage();
} else {
  alert("The Language Detector API is not available in this browser");
}
```

```
async function detectLanguage() {

  try {

    // Check model availability...

    const availability = await LanguageDetector.availability();
    if (availability == "unavailable") {
      alert("The model is not available");
      return;
    }

    // Create the session...

    const detectorSession = await LanguageDetector.create({
      expectedInputLanguages: ['en', 'de', 'fr', 'it', 'es']
    });

    // Action the request...

    const detected = await detectorSession.detect(sourceTextElement.textContent);

    const detectedLanguage = detected[0]["detectedLanguage"];
    streamingOutputElement.textContent = "Detected language is most likely: " + detectedLanguage;
    languageFromElement.value = detectedLanguage;

    // destroy any existing session instance to minimise memory leaks and maintain stateless between requests...

    if (detectorSession) detectorSession.destroy();

  } catch (err) {
    alert("Detection failed: " + err);
  }

}
```

Translator

```
// Define the required input/output elements...

var streamingOutputElement = document.querySelector('[data-role="outputText"]');
var sourceTextElement = document.querySelector('[data-role="sourceText"]');
var languageFromElement = document.querySelector('[data-role="languageFrom"]');
var languageToElement = document.querySelector('[data-role="languageTo"]');


streamingOutputElement.textContent = "Generating Translation...";


// Check for browser API availability by testing for the presence of the Translator object...

if (languageFromElement.value == languageToElement.value) {
  alert("From and To languages must differ");
} else if (Translator) {
  translateText();
} else {
  alert("The Translator API is not available in this browser");
}
```

```
async function translateText() {

  try {

    // Check model availability...

    const availability = await Translator.availability({
      sourceLanguage: languageFromElement.value,
      targetLanguage: languageToElement.value,
    });

    if (availability == "unavailable") {
      alert("The model is not available");
      return;
    }

    // Create the session...

    const translatorSession = await Translator.create({
      sourceLanguage: languageFromElement.value,
      targetLanguage: languageToElement.value
    });

    // Action the request...

    const stream = translatorSession.translateStreaming(sourceTextElement.textContent);

    streamingOutputElement.textContent = '';

    for await (const chunk of stream) {
      streamingOutputElement.textContent += chunk;
    }

    // destroy any existing session instance to minimise memory leaks and maintain stateless between requests...

    if (translatorSession) translatorSession.destroy();

  } catch (err) {
    alert("Translation failed: " + err);
  }

}
```

Summarizer




```
// Define the required input/output elements...

var streamingOutputElement = document.querySelector('[data-role="outputText"]');
var sourceTextElement = document.querySelector('[data-role="sourceText"]');
var summarizeTypeElement = document.querySelector('[data-role="summarizeType"]');
var summarizeLengthElement = document.querySelector('[data-role="summarizeLength"]');
var summarizeFormatElement = document.querySelector('[data-role="summarizeFormat"]');
const summarizeContextElement = document.querySelector('[data-
role="summarizeContext"]');


streamingOutputElement.textContent = "Generating Summary...";


// Check for browser API availability by testing for the presence of the Summarizer object...

if (Summarizer) {
  summarizeText();
} else {
  alert("The Summarizer API is not available in this browser");
}
```

```
async function summarizeText() {

  try {

    // Check model availability...

    const availability = await Summarizer.availability({
      outputLanguage: 'en'
    });

    if (availability == "unavailable") {
      alert("The model is not available");
      return;
    }

    // Create the session...

    const summarizerSession = await Summarizer.create({
      sharedContext: summarizeContextElement.value.trim(),
      expectedContextLanguages: ['en'],
      type: summarizeTypeElement.value,
      format: summarizeFormatElement.value,
      length: summarizeLengthElement.value,
      outputLanguage: 'en'
    });

    // Action the request...

    const stream = summarizerSession.summarizeStreaming(sourceTextElement.textContent);

    streamingOutputElement.textContent = '';

    for await (const chunk of stream) {
      streamingOutputElement.textContent += chunk;
    }

    // destroy any existing session instance to minimise memory leaks and maintain stateless between
requests...

    if (summarizerSession) summarizerSession.destroy();

  } catch (err) {
    alert("Summarization failed: " + err);
  }

}
```

Prompt

```
// Define the required input/output elements...

var streamingOutputElement = document.querySelector('[data-role="outputText"]');
var sourceTextElement = document.querySelector('[data-role="sourceText"]');
var promptPresetElement = document.querySelector('[data-role="promptPreset"]');
var promptCustomElement = document.querySelector('[data-role="promptCustom"]');


streamingOutputElement.textContent = "Generating Prompt Response...";


// Check for browser API availability by testing for the presence of the LanguageModel object...

if (LanguageModel) {
  promptText();
} else {
  alert("The Prompt API is not available in this browser");
}
```

```
async function promptText() {

  try {

    // Check model availability...

    const availability = await LanguageModel.availability({
      expectedInputs: [{ type: 'text', languages: ['en'] }],
      expectedOutputs: [{ type: 'text', languages: ['en'] }]
    });

    if (availability == "unavailable") {
      alert("The model is not available");
      return;
    }

    const systemPrompt = promptPresetElement.value === 'custom'
      ? (promptCustomElement.value.trim() || 'You are a helpful assistant. Answer clearly and concisely.')
      : promptPresetElement.value.trim();

    // Create the session...

    const promptSession = await LanguageModel.create({
      initialPrompts: [{ role: 'system', content: systemPrompt }],
      expectedInputs: [{ type: 'text', languages: ['en'] }],
      expectedOutputs: [{ type: 'text', languages: ['en'] }]
    });

    // Action the request...

    const stream = promptSession.promptStreaming(sourceTextElement.textContent);

    streamingOutputElement.textContent = '';

    for await (const chunk of stream) {
      streamingOutputElement.textContent += chunk;
    }

    // destroy any existing session instance to minimise memory leaks and maintain stateless between requests...

    if (promptSession) promptSession.destroy();

  } catch (err) {
    alert("Prompt failed: " + err);
  }

}
```

Chrome API Limits & Constraints

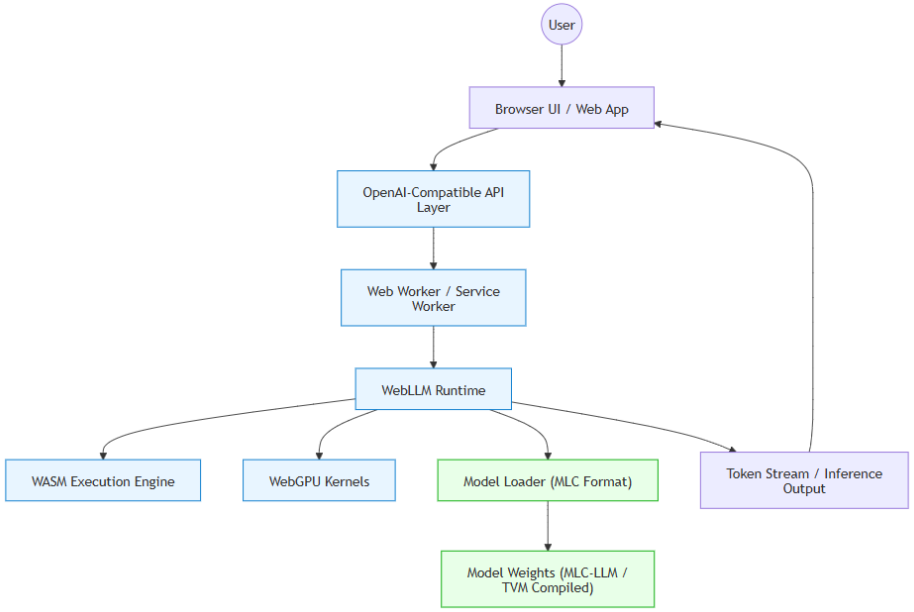


- Tight Context Window & Input Limits:
  - Small Context Window (c.~4,096 tokens)
  - Per-Prompt Limit (c. 1,024 tokens)
  - Chunking May Be Required
- Deployment and Model Friction:
  - Cold Start Download
  - Non-Deterministic Updates
- Model Capabilities & Rigidness:
  - Reduced Reasoning
  - No Web Workers
  - Iframe Restrictions
  - Incomplete Language Set
  - Initialization Latency
  - The 1-Minute Idle Eviction
  - May require browser Feature Flags
  - Inconsistency between Chrome & Edge

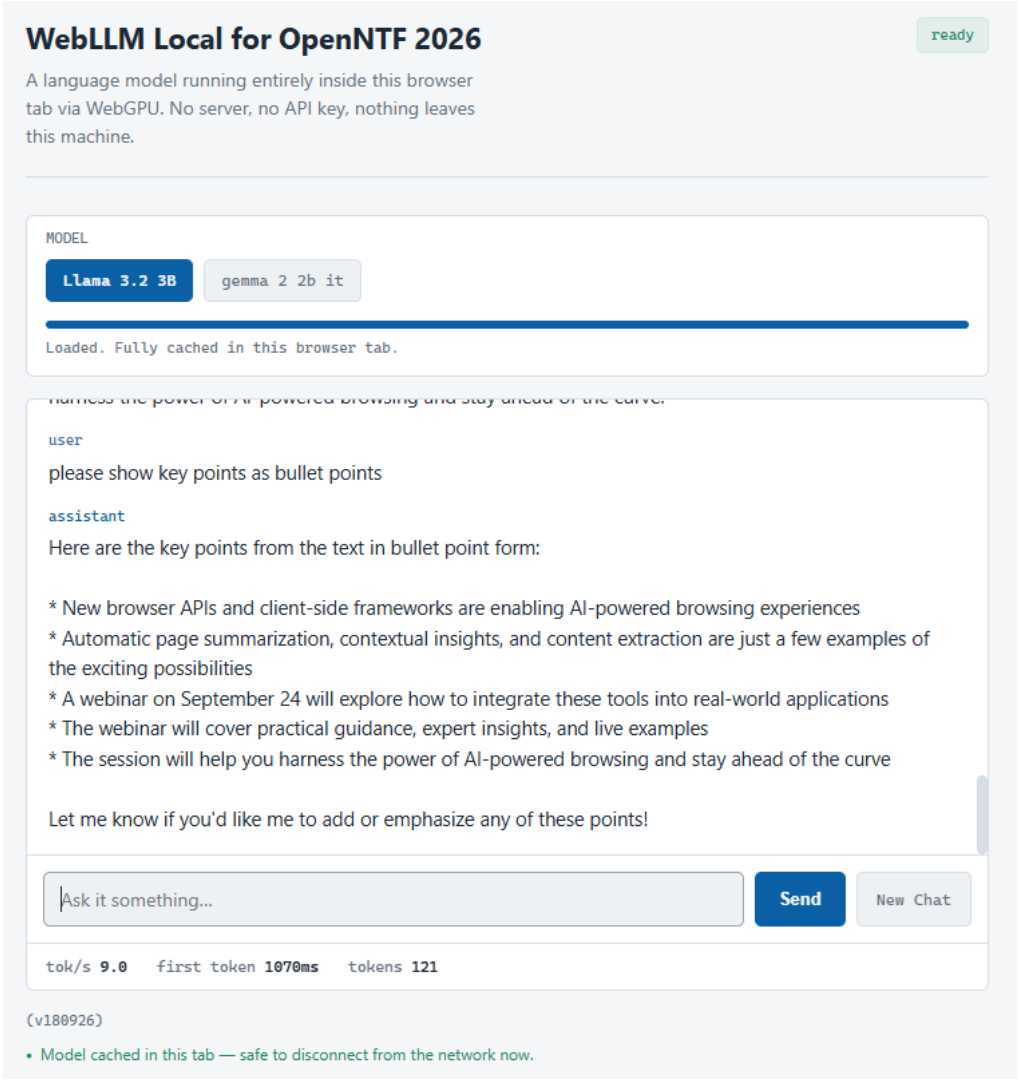
WebLLM

WebLLM is a high-performance, in-browser LLM inference engine that brings language model inference directly onto web browsers with hardware acceleration. Everything runs inside the browser with no server support.

- Uses WebGPU acceleration
- Supports Web Worker and/or Service Worker for best performance
- Supports Model caching
- Small and Medium sized Models such as:
  - Llama
  - Gemma
  - Phi
- Also supports custom and Hugging Face-style LLMs (once converted into MLC format)
- Cross-platform (Windows/MacOS/Linux plus mobile) and wider browser support - subject to WebGPU and WebAssembly (WASM) requirements



WebLLM Demo


WebLLM Local for OpenNTF 2026
ready
A language model running entirely inside this browser tab via WebGPU. No server, no API key, nothing leaves this machine.
MODEL
Llama 3.2 3B
gemma 2 2b it
Loaded. Fully cached in this browser tab.
user
please show key points as bullet points
assistant
Here are the key points from the text in bullet point form:
* New browser APIs and client-side frameworks are enabling AI-powered browsing experiences
* Automatic page summarization, contextual insights, and content extraction are just a few examples of the exciting possibilities
* A webinar on September 24 will explore how to integrate these tools into real-world applications
* The webinar will cover practical guidance, expert insights, and live examples
* The session will help you harness the power of AI-powered browsing and stay ahead of the curve
Let me know if you'd like me to add or emphasize any of these points!
Ask it something...
Send
New Chat
tok/s 9.0 first token 1070ms tokens 121
(v180926)
• Model cached in this tab — safe to disconnect from the network now.

WebLLM Implementation (1/2)

openntf

- **Import the module -** No build step needed, loads from CDN

```
import * as webllm from "https://esm.run/@mlc-ai/web-llm";
```

- **Pick a model -** From WebLLM's prebuilt catalogue

```
webllm.prebuiltAppConfig.model_list // array of {model_id, ...} entries
```

- **Create the engine -** Passing a progress callback to track download/compile (This downloads the model's weights (cached by the browser afterward) and loads it into a WebGPU runtime)

```
const engine = await webllm.CreateMLCEngine(modelId, {
  initProgressCallback: (report) => console.log(report.progress, report.text),
});
```

WebLLM Implementation (2/2)

- **Maintain your own chat history -** As a plain array

```
const history = [{ role: "user", content: "Hello" }];
```

- **Run inference -** Via an OpenAI-compatible chat completions call

```
const stream = await engine.chat.completions.create({
  messages: history,
  stream: true,
});
for await (const chunk of stream) {
  const delta = chunk.choices?.[0]?.delta?.content || "";
  // append delta to UI
}
```

- **Switch models -** By reloading the same engine instead of recreating it

```
engine.setInitProgressCallback(initProgressCallback);
await engine.reload(newModelId);
```

WebLLM Limits & Constraints



- Memory and Hardware Restrictions:
  - WebAssembly (WASM) 4GB Buffer Limit
  - WebGPU Memory Allocation
  - Mobile Restrictions
- Context Window:
  - Context Length Barriers
- Model Size and Download Overhead:
  - Gigabyte Downloads
  - Accuracy vs Size Trade-off
- Browser & Compatibility Constraints:
  - Hard WebGPU Requirement
  - Ephemeral Lifecycle

Chrome AI APIs vs. WebLLM



Feature	Chrome/Edge Browser built-in AI APIs	WebLLM
Model Selection	Fixed (e.g., Gemini Nano)	Flexible (Llama, Phi, Mistral)
User Download	Zero download required	Requires downloading models on first load
Cross-Browser	Vendor-locked (Chrome/Edge APIs)	Universal (Works anywhere with WebGPU & WebAssembly)
Control	Browser vendor manages updates	Developer controls version and setup
Recommended for	Quick, lightweight tasks	Custom, heavy, or specialized tasks
Execution environment	Runs natively in the browser's C++ engine	Runs as standard web code using WebAssembly (Wasm) and WebGPU
Performance and hardware	The browser vendor tunes it automatically for the host operating system	Utilises local graphics cards directly through WebGPU for maximum performance

Best Practices



- Prepare the model at a reasonable time
- Clone sessions for repetitive tasks
- Destroy unused sessions
- Render streaming responses safely and efficiently
- Optimize input for speed
- Cloud Fallback
- Model Language

Summary



- Some AI APIs are already built into modern browsers, though with constraints and limits
- Chrome's AI APIs are arriving slowly compared with the pace of AI elsewhere, so they still feel a little experimental
- More Chrome APIs have been promised
- If your web app needs basic, task-specific AI and data privacy is important, Chrome's AI APIs may be a good fit, given their ease of deployment and use
- Chrome's APIs aren't the only way to run AI locally in the browser. WebLLM offers similar or complementary functionality with more flexibility, but you'll need to write extra code to get the best performance from it
- If you want the privacy of local AI, or Cloud AI costs are too prohibitive, perhaps consider a hybrid approach, using Chrome's APIs for language detection & translation, and WebLLM for more general-purpose AI, with models such as Llama 3.2 3B or Gemma 2 2B IT



Thank You!

Do we have time for any questions?

References



- Chrome Ai APIs
  - https://developer.chrome.com/docs/ai/built-in
- Edge Ai APIs
  - https://learn.microsoft.com/en-us/microsoft-edge/web-platform/prompt-api
- WebLLM
  - https://webllm.mlc.ai/

Feature Flags, Settings & Internals

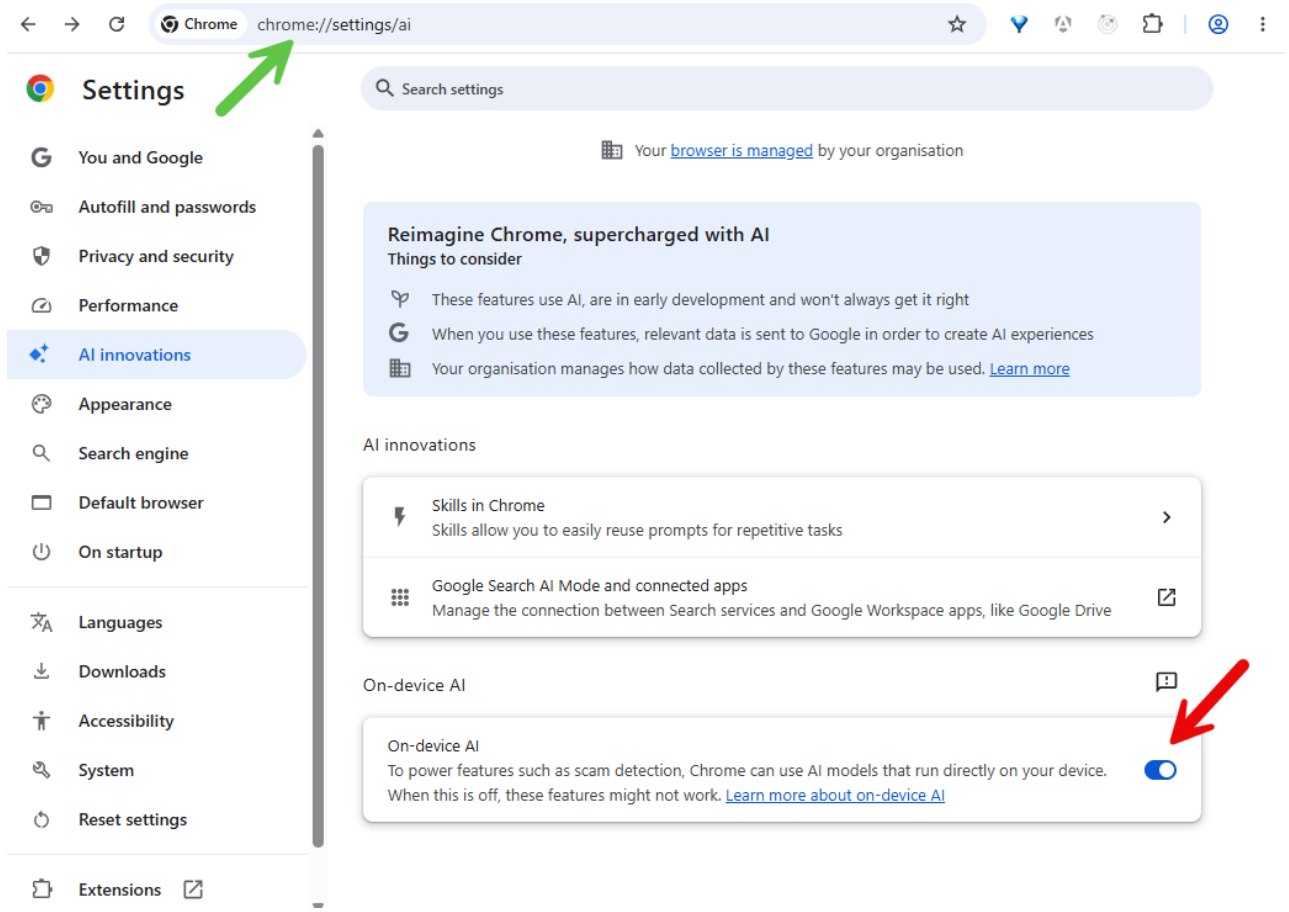


- For all APIs (enable as required)*:
  - chrome://flags/#enable-ai-features
  - chrome://flags/#prompt-api-for-gemini-nano
  - edge://flags/#edge-llm-prompt-api-for-phi-mini
- For the Prompt API (enable as required)*:
  - chrome://flags/#optimization-guide-on-device-model
- Chrome on-device AI Feature enable/disable*:
  - chrome://settings/ai
- Manage Internal Chrome Components*:
  - chrome://components/
- Manage Chrome Model status and logs*:
  - chrome://on-device-internals/
- Manage and view Chrome AI Translation files*:
  - chrome://on-device-translation-internals/

** Actual feature names may differ between Chrome and Edge browsers (and releases)*

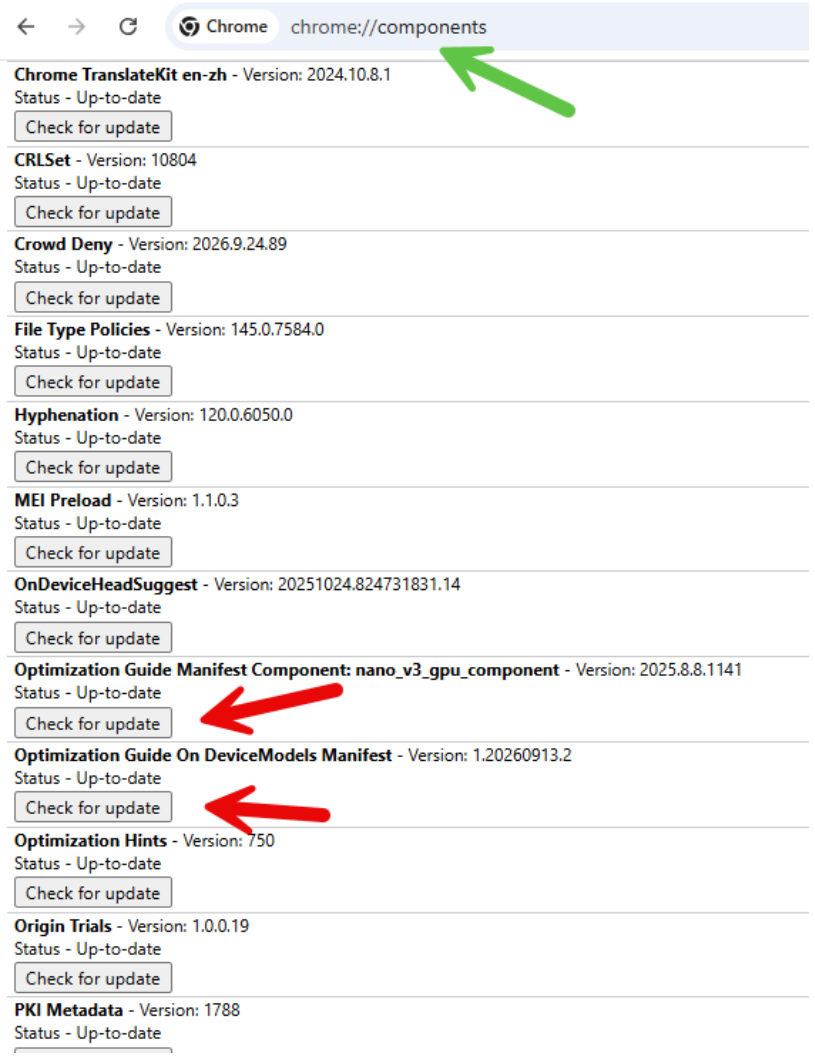
Chrome on-device AI Feature enable/disable

- chrome://flags/#enable-ai-features



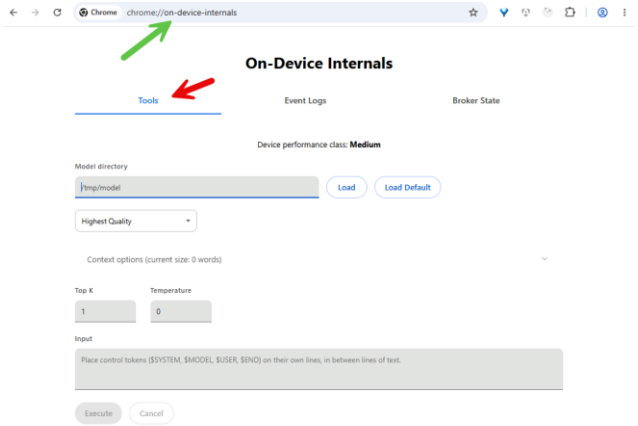
Chrome AI API Manage Internal Components

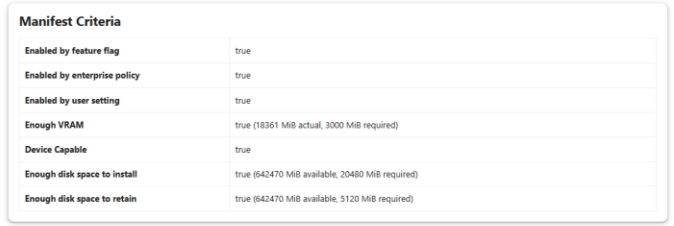
- chrome://components/

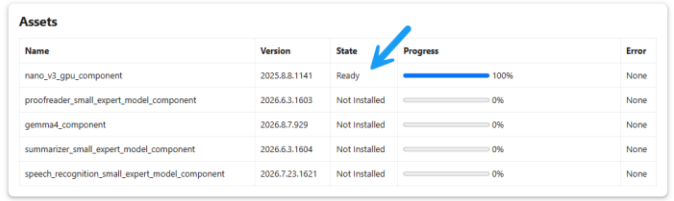


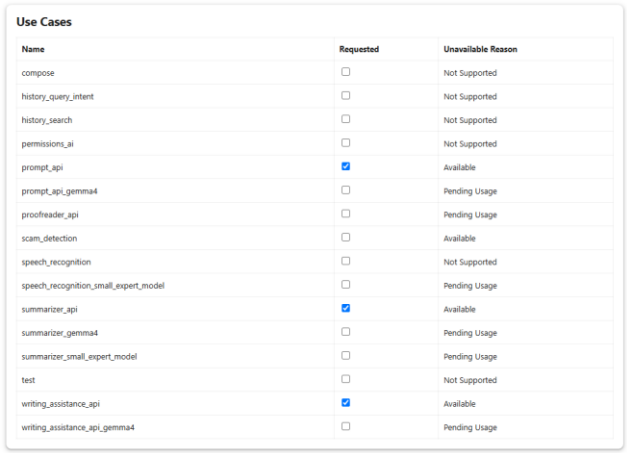
Chrome AI API Manage Model status and logs

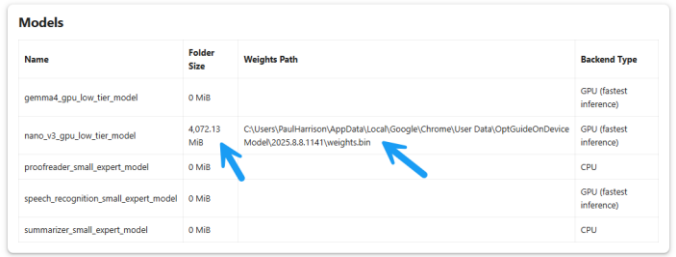
- chrome://on-device-internals/

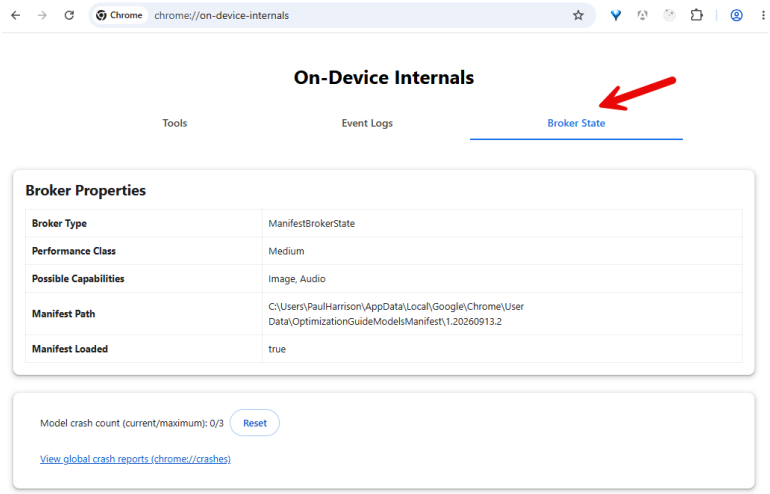






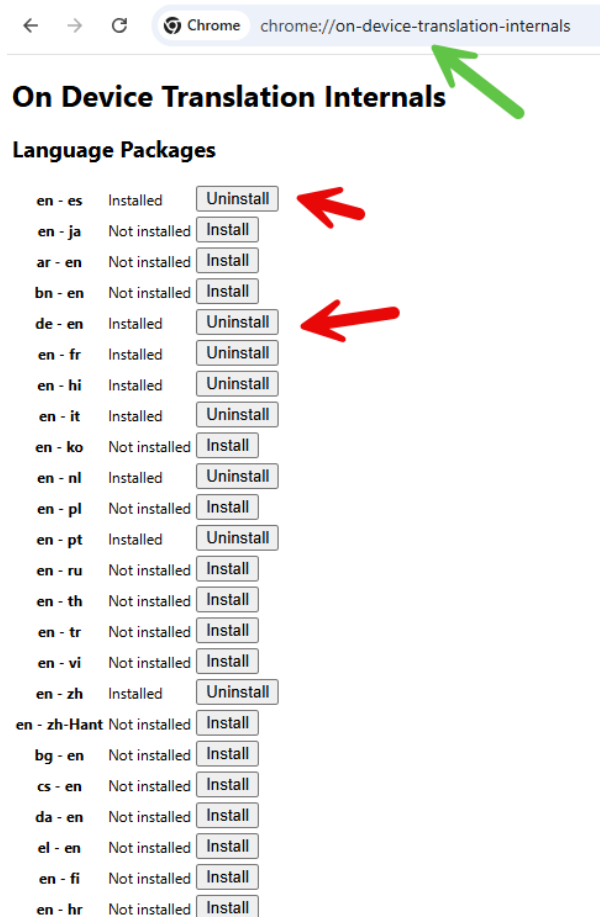






Chrome AI API Manage and view AI Translation files

- chrome://flags/#enable-ai-features



Demo Code

- Xpages
  - https://github.com/8dot3/OpenNTF2026-Supercharging-XPages

- WebLLM
  - https://github.com/8dot3/OpenNTF2026-Supercharging-WebLLM