TPAC 2024
WebML WG

CHROME BUILT-IN AI TEAM

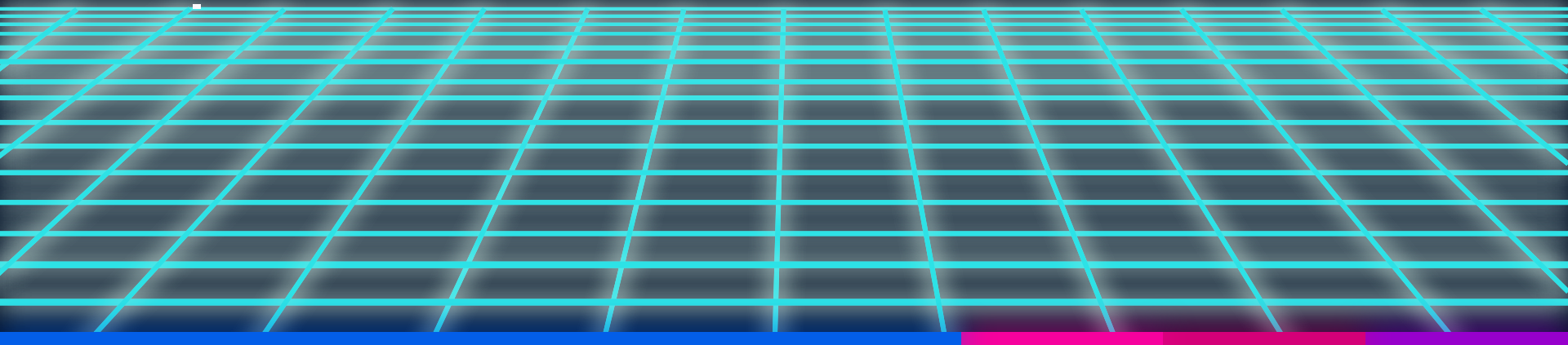
BUILT-IN AI APIs

Domenic Denicola
domenic@chromium.org

Agenda

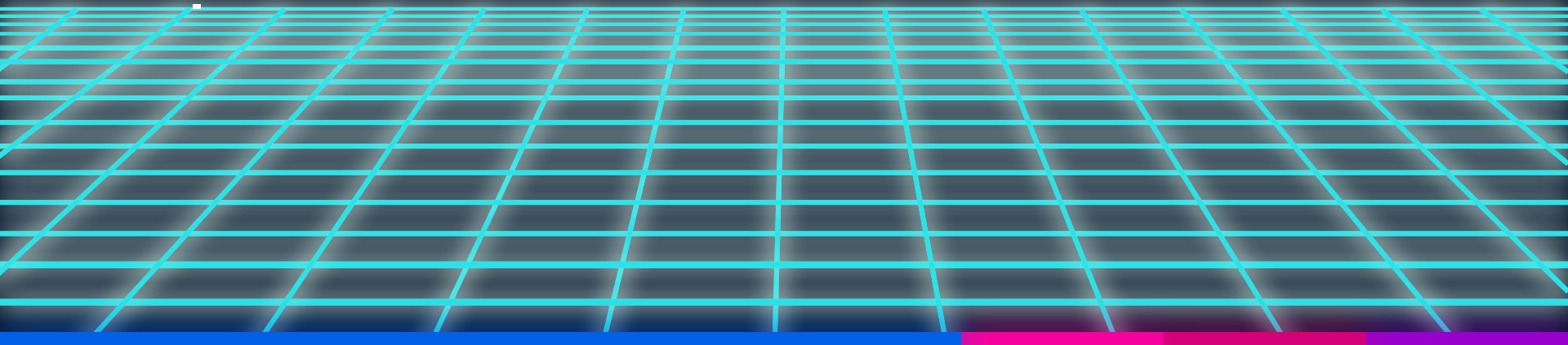
- What is “built-in AI”?
- Why work on this?
- Our current proposals
- Open questions and discussion areas
- Next steps

What is “built-in AI”?



- High-level, task-focused APIs (translate, summarize, etc.)
- Prepackaged browser-provided ML models and fine-tunings
- Browser-mediated, e.g. model updates and storage, permissions, ...
- Complementary to:
  - On-device “bring your own AI” / WebNN stacks
  - AI access via API calls

Why work on this?



Because it's there

- Browser/OS combos already ship language models
  - Apple Intelligence on macOS
  - Firefox AI platform
  - Gemini Nano on Chrome and Android
  - Windows Copilot
- They also have special-purpose ML models, e.g. language detection, translation, face/QR code recognition, ...

Because developers want it

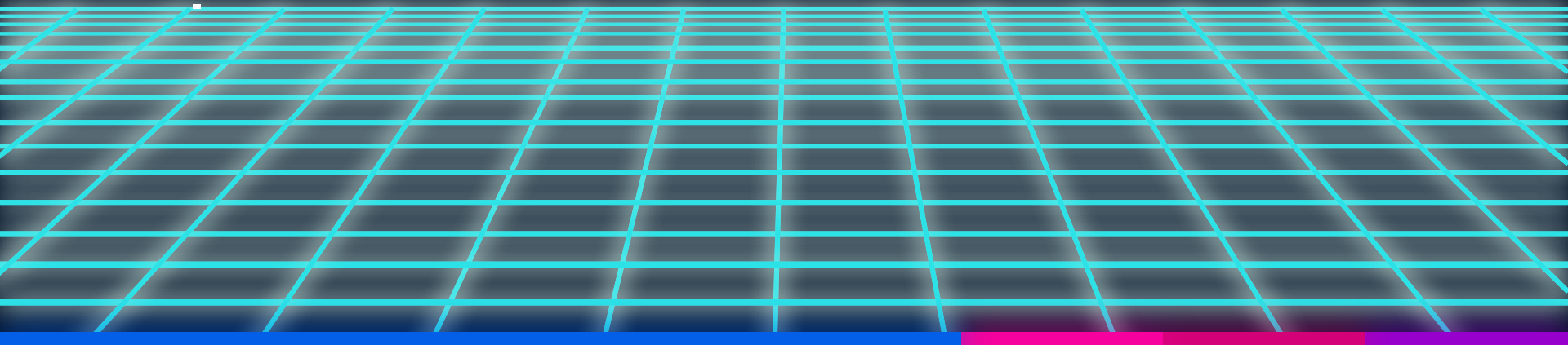
Results from Chrome's Early Preview Program

- Behind-a-flag testing using the prompt API
- 13 partners, 1 week, 8 countries, 50 prototypes
- Top use cases:
  - Summarization
  - Writing
  - Rewriting
  - Translation
- But others too, e.g. proofreading, sentiment analysis, toxicity detection

Because we can abstract away complexity

- Many developers are excited about built-in AI APIs but not ready to learn about HuggingFace, MediaPipe, ONNX, WebAssembly, ...
- Purpose-built models and fine-tunings can exceed generic model and API performance and get better over time.
- We can incorporate more advanced techniques like hierarchical summarization or prefix caching.
- Sharing multi-GiB binary blobs across sites is as-yet unsolved.

Our current proposals



- Language Detector
- Translator
- Summarizer
- Writer
- Rewriter
- Prompt? (it's complicated)

} Writing Assistance APIs

Language Detector

Detect language with confidence levels

```
const detector = await ai.languageDetector.create();

const results = await detector.detect(someUserText);
for (const result of results) {
  console.log(result.detectedLanguage, result.confidence);
}
```

Translator

Translate text from one language to another

```
const translator = await ai.translator.create({
  sourceLanguage: "en",
  targetLanguage: "ja"
});

const text = await translator.translate("Hello, world!");

const readableStream = await
  translator.translateStreaming(aLotOfText);
```

Summarizer

Summarize text in a variety of styles

```
const summarizer = await ai.summarizer.create({
  sharedContext: "An article from the Daily News magazine",
  type: "headline",
  length: "short"
});

const summary = await summarizer.summarize(article, {
  context: "This article was written 2024-08-07 and " +
           "it's in the World Markets section."
});
```

Writer

Write new text following a prompt

```
const writer = await ai.writer.create({
  tone: "formal"
});

const result = await writer.write(
  "A draft for an inquiry to my bank about how " +
  "to enable wire transfers on my account"
);
```

Rewriter

Transform and rephrase input text as requested

```
const rewriter = await ai.rewriter.create({
  sharedContext: "A review for the Flux Capacitor 3000"
});

const result = await rewriter.rewrite(reviewText, {
  context: "Be as constructive as possible."
});
```

Prompt

Chat directly with an instruction-tuned language model

```
const session = await ai.assistant.create({
  initialPrompts: [
    { role: "system", content: "Predict 2 response emojis" },
    { role: "user", content: "LGTM" },
    { role: "assistant", content: "👍, 🚢" }
  ]
});

const result = session.prompt("Back to the drawing board");
```

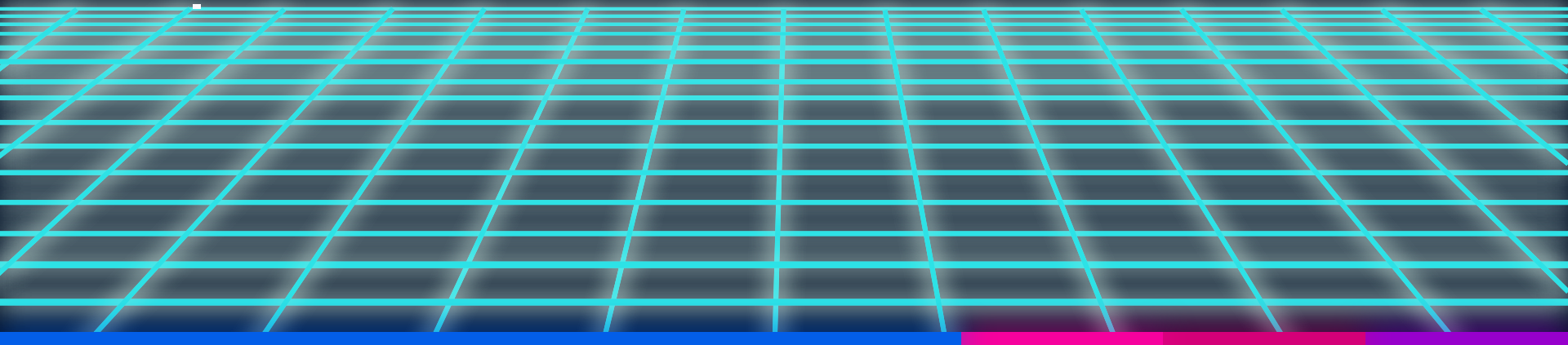
Prompt vs. task-based APIs

- We think task-based APIs have many advantages (including better potential for interoperability). We are focusing our efforts there.
- But designing and building task-based APIs for every use case is hard. Maybe eventually direct prompting is the best route.
- Immediate plans are to origin trial Prompt API *limited to Chrome Extensions*, to further draw out use cases.
- We're not currently incubating prompt in web standards space and would prefer to focus TPAC on task-based APIs.

Cross-cutting design notes

- All APIs have a corresponding capabilities API, exposing e.g.:
  - supported translation languages
  - supported writing tones or rewriting lengths
  - maximum number of tokens
- We want to expose unified designs, e.g. xStreaming(), destroy(), AbortSignal integration, capabilities API shape, ...
- API availability of “no” / “readily” / “after-download”

Open questions and discussion areas



On-device vs. cloud

- We've intentionally designed the APIs so they could be backed by either on-device or cloud models. (Or a hybrid.)
- However, we're not sure any browser is actually interested in a cloud implementation.
- Web developers generally prefer on-device for privacy guarantees.

Device reach

- Especially for language model-backed APIs, multiple gigabytes of VRAM is not always available.
- We're designing the API surfaces to be obviously fallible, with capabilities testing on the golden path.
- Still unclear how this will play out in the real world. Origin Trials should give more data!

Privacy and fingerprinting

- Downloaded models / fine-tunings / language packs are envisioned to be stored without partitioning.
- Unmitigated, this is a fingerprinting vector. Various mitigations under consideration, e.g.:
  - Prompts (sometimes auto-granted?)
  - Grouping downloads
  - Causing probing for status to start the download

Interoperability

- Task-based APIs are better for interop than direct prompting, but could still be tricky.
- This ground has been lightly explored by other high-level ML APIs such as shape detection.
- Can we write web platform tests? Or something more like ML evals/benchmarks? Specify invariants, e.g. “shorter” vs. “longer”?

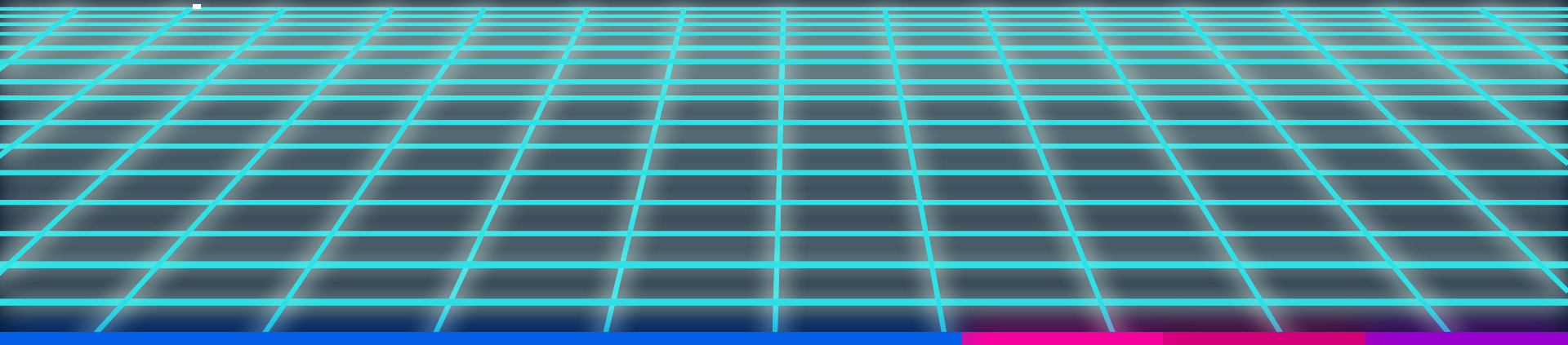
Quality

- Can browsers provide models that are reliably good enough for developers to want to use?
- Can they provide guarantees of quality over time?
- Will varying quality across browsers, OSes, or releases make the technology too unreliable?

Internationalization

- Developers want to know if a model “supports Japanese”, but this is a complex question.
- If a model is not well-tested or safe to use in a language, but can kind of work anyway, how do we handle that?
- Also, ongoing discussion about language tag details (WICG/translation-api#11): en, en-US, en-GB-oed, en-Dsrt, …

Next steps



Origin Trials

- Origin Trials for everything except the prompt API, staggered as we finish implementations
- Origin Trial *for Chrome Extensions* for the prompt API

Web standards engagement

- Chat with us at TPAC!
- Various reviews requested: W3C TAG, W3C i18n, Mozilla + WebKit standards positions repositories
- Explainer repositories open and ready for your issues
- Currently in WICG, but would love to end up in the WebML WG, if you'll have us.

Links

- Explainers
  - WICG/translation-api
  - WICG/writing-assistance-apis
  - explainers-by-googlers/prompt-api
- https://goo.gle/chrome-ai-dev-preview