Building Chrome Extensions with Claude

Talk Outline — Claude Desktop / Cowork Edition

Prepared for Bo's presentation · August 2026

This outline walks through a live, end-to-end demo: installing the Claude desktop app, setting up a working folder, prompting Claude to build a real Chrome extension, loading and testing it, iterating on it, and (optionally) wiring the extension itself to call the Claude API for live, on-demand AI features. Each section below includes the key talking points and suggested demo actions.

1. Installing Claude on Mac or PC

Goal: get the Claude desktop app installed and Cowork mode (Claude's file-and-task workspace) turned on.

Steps

- Go to claude.com/download and choose the installer for your OS.
- System requirements: macOS 13 (Ventura) or later on Apple Silicon or Intel; Windows 10/11, 64-bit; Linux is available in beta.
- Run the installer and sign in with your Anthropic/Claude account.
- Cowork mode (the file-access, multi-step "agent" workspace used for building things like extensions) requires a paid plan — Pro, Max, Team, or Enterprise.
- Open Cowork from the sidebar of the desktop app once signed in.

Speaker note: Show the download page live, install if not already done, and point out Cowork in the sidebar.

2. Set Up a Claude Folder and Begin Learning

Goal: give Claude a dedicated project folder it can read from and write to, and prime it with the right context before asking it to build anything.

Steps

- Create a folder on your computer, e.g. "Chrome Extension Projects."
- In Cowork, connect that folder so Claude can read and write files directly into it (this becomes the extension's project directory).
- Ask Claude to read up first: have it summarize Chrome's Manifest V3 rules, required files (manifest.json, background service worker, content scripts, popup), and current best practices before writing code.
- Optional: write a short project brief in the folder (one page) describing the extension's purpose, audience, and constraints — Claude will use this as ongoing context across the session.
- Check what skills/plugins are available (Cowork supports installable skills) in case any speed up the build — not required for a basic extension.

Speaker note: Emphasize that a few minutes of setup and context-gathering up front produces much better first drafts than jumping straight to 'build me an extension.'

3. Instructing Claude to Set Up a New Extension

Goal: give Claude enough specification to produce a working Manifest V3 extension on the first pass.

What to tell Claude

- **Purpose, in one sentence:** what problem does this extension solve for the user?
- **Where it lives:** toolbar popup, a side panel, a content script injected into pages, a background service worker, and/or an options page.
- **Permissions, kept minimal:** e.g. activeTab, storage, scripting — ask Claude to justify each permission it requests.
- **Data handling:** does it need to store data locally (chrome.storage.local) or sync across devices (chrome.storage.sync)?
- **Look and feel:** icon set, colors, simple UI mockup description (Claude can generate placeholder icons).
- **Deliverables:** ask explicitly for manifest.json plus all HTML/CSS/JS files, comments in the code, and a short README with install and test steps.

What Claude needs to succeed

- A clear, single-purpose feature description — avoid asking for five features in one prompt.
- Confirmation of Manifest V3 (not the deprecated V2) and target Chrome version.
- File/folder conventions to follow if you have preferences (e.g. src/ folder, naming).
- Permission to iterate — tell Claude you'll test and report back errors, rather than expecting one perfect pass.

Speaker note: Live-prompt Claude here: describe a simple extension (e.g. a page-highlighter or a quick-notes side panel) and let it scaffold the files in real time.

4. How to Install the New Extension

Goal: load the unpacked extension Claude just built into Chrome for testing.

Steps

- Open Chrome and go to chrome://extensions.
- Toggle “Developer mode” on (top-right corner).
- Click “Load unpacked” and select the project folder Claude wrote the files into.
- Chrome will flag any manifest errors immediately — read these back to Claude verbatim if something fails to load.
- Pin the extension to the toolbar (puzzle-piece icon → pin) so it's visible during the demo.
- Walk through the README Claude generated to confirm each feature works as intended.

Speaker note: Do this live: load-unpacked, show the extension icon appear, and click through its basic function.

5. Making Adjustments, Additions, or Updates While in Use

Goal: show the iterative loop of prompting Claude for a change, reloading, and testing — the normal day-to-day workflow.

Steps

- Describe the change to Claude in plain language, referencing the specific file or feature if you know it (e.g. “in popup.js, add a button that clears saved notes”).
- Let Claude edit the files directly in the connected project folder.
- Back in chrome://extensions, click the reload icon on the extension's card to pick up the change.

- If the manifest itself changed (new permission, new file), do a full reload of the extension, not just the page.
- If a content script changed, also refresh the target web page it runs on.
- Use Chrome DevTools to catch issues: “Inspect” the popup, or click “service worker” on the extension card to see background-script console errors — paste any errors back to Claude to fix.
- Keep a simple changelog and bump the manifest's “version” field with each meaningful update.

Speaker note: Demo a small live change end-to-end: ask Claude for a tweak, reload the extension, verify it worked.

6. Top Ten Things Extensions Can Do for Users

Goal: give the audience a sense of the range of what's possible, beyond the one demo extension.

1. Modify or enhance webpage content in real time — content scripts that inject, highlight, translate, or restyle page content.
2. Block or filter content — ad blockers, distraction blockers, and content filters using the declarativeNetRequest API.
3. Automate repetitive tasks — auto-filling forms, one-click actions, batch operations across tabs.
4. Save and sync data across devices — notes, bookmarks, or settings via chrome.storage.sync.
5. Add custom UI surfaces — toolbar popups, side panels, and options pages for dashboards or tools.
6. Integrate with external services and APIs — weather, translation, project trackers, or AI assistants.
7. Capture and organize information from pages — clippers, highlighters, read-it-later and note-taking tools.
8. Manage tabs, windows, and sessions — tab organizers, session savers, workspace switchers.
9. Add quick-access commands — right-click context menu actions and custom keyboard shortcuts.
10. Send notifications and reminders — alerts tied to browsing activity, schedules, or page changes (chrome.alarms, chrome.notifications).

Speaker note: Pick two or three of these to show real examples of (screenshots or quick live browsing) to make the list concrete.

7. Live Claude in the Extension via API Key

Goal: explain the difference between Claude writing the extension's code once versus the extension calling Claude live, at runtime, for dynamic AI features — and how to wire that up safely.

When you need this

- Not required for most extensions: fixed logic (highlighting, storage, notifications, UI) doesn't need a live API call — Claude just writes the code once.
- You do need it for dynamic, per-request AI behavior: summarizing whatever page the user is currently on, a live chat side panel, real-time translation or rewriting, or answering questions about page content.

Getting an API key

- Go to platform.claude.com (Anthropic Console), sign up or sign in.
- Settings → API keys → Create key; name it and optionally scope it to a workspace.
- The full key (starts with sk-ant-) is shown only once — copy and store it securely.
- Add a payment method to the Console; API usage is billed separately from a Claude subscription.

Wiring it into the extension

- Make the API call from the background service worker, not a content script, to avoid page CORS/CSP restrictions.
- Never hard-code your API key into code that ships to other users — unpacked extension source is fully visible. For a personal/demo extension this is fine; for anything distributed, route calls through your own backend proxy or have each user supply their own key.
- If each user supplies their own key, store it with chrome.storage.local via the options page, and never log it or send it anywhere but Anthropic's API.
- Send a request to the Messages API with the model, a system prompt describing the extension's job, and the page content or user question as input; render the response in the popup or side panel.
- Be mindful of cost and rate limits — cap request frequency and message length, and show usage/cost expectations to the audience.

Demo idea

- A side panel that sends the current page's visible text to Claude and returns a live 3-sentence summary or Q&A on demand.

  Speaker note: This is a good closing demo: it ties the whole talk together by showing Claude both as the tool that built the extension and as a live capability inside it.