Big picture (one-sentence)

The browser extension (frontend) scans the page, sends a DOM snapshot to your backend, the backend uses Gemini to generate a structured tour (steps with selector, narrative, action), and the frontend executes those steps with highlights, speech, and a safety-confirmation modal before any click.

Data flow (high level)

1. User triggers “Start Tour” in extension UI.
2. Extension scans the page and sends JSON (title + simplified DOM) to backend `/api/analyze`.
3. Backend calls the LLM and returns a structured tour: ordered steps with `{ element_selector, narrative, action }`.
4. Extension receives tour, displays overlay UI, highlights each step, speaks the narrative, scrolls/clicks as commanded (clicks gated by modal).
5. User can control playback (play/pause/next/prev/stop), and can ask chat-like questions (extension → backend `/api/chat`).

Implementation phases & order (recommended)

1. Prepare backend environment & key** (ensure env var loads).
2. Implement minimal analyze/chat endpoints** (return mock data first).
3. Build frontend overlay UI and domScanner** (use mock responses to verify UI).
4. Integrate real LLM in backend** and update endpoints to return structured steps.
5. Add safety modal for click actions** and wire it to step execution.
6. End-to-end testing on multiple websites** and iterate selectors/fallback heuristics.
7. Harden security and deploy** (auth, rate limits, HTTPS).

Files to create / update (organized by area)
A — Backend (FastAPI)

These files run the Gemini calls and serve /api/analyze and /api/chat.

main.py — (update) application entrypoint. Responsibilities:

Load environment variables (.env) before importing modules that use them.
Create FastAPI app and add CORS middleware (allow local dev origins).
Define routes: POST /api/analyze and POST /api/chat.
Print debug info at startup (key presence) during dev.
Graceful error handling and logging for tracebacks.

chains.py or llm_client.py — (update/create) chain builder module. Responsibilities:

Assemble prompt templates.
Create LLM client wrapper (e.g., LangChain wrapper or directly google-genai client).

Provide a function `get_tour_generator_chain()` that exposes prompt + parser + llm components OR a function that accepts input and returns parsed TourPlan.
Ensure pydantic schema objects are available (TourStep, TourPlan, DOMElement).

* `schemas.py` — **(create optional)** Pydantic models for requests/responses:

  * `PageContent` (title, url, elements).
  * `TourStep`, `TourPlan` (for consistent API payloads).
  * `ChatRequest` (for /api/chat).

* `.env` — **(create)** environment file (local dev). Contains:

  * `GOOGLE_API_KEY` (or `GEMINI_API_KEY`).
  * Optional other keys (rate-limit token secret, allowed origins).

* `requirements.txt` — **(update)** list required Python packages (fastapi, uvicorn, google-genai or google-generativeai wrapper, python-dotenv, pydantic, langchain-google-genai if used).

utils.py — **(optional)** helpers: structured logging, error formatting, rate limiting hooks.

tests/test_analyze_endpoint.py — **(optional)** unit/integration tests that call `/api/analyze` with a small sample payload and assert the returned shape.

B — Frontend / Extension UI (React + content script)

These files build the overlay UI and provide DOM scanning + API calls.

Project structure suggestion:

```
src/
  components/
    TourOverlay.tsx        <-- PATCH/REPLACE (main overlay UI + executor)
  content/
    domScanner.ts          <-- CREATE (scanPage function)
  services/
    api.ts                 <-- CREATE (generateTour, sendChatMessage wrappers)
  manifest.json            <-- CREATE (extension manifest for MV3)
  content_script.js        <-- CREATE (if not using React build for direct injection)
  background.js            <-- CREATE (optional background/service worker)
  popup.html / popup.js    <-- optional
```

Files and roles:

* `src/components/TourOverlay.tsx` — **(patch existing)** the main overlay:

  * Maintains tour state (tourData), playback state (isPlaying), current index.
  * Calls `scanPage()` then `generateTour()` to request backend.
  * Displays overlay UI (header, status, step preview, controls, chat area).
  * Step execution: highlight, scroll, speak, click (click gated by safety modal).
  * Cleanup on stop/unmount.

* `src/content/domScanner.ts` — **(create)** DOM scanning:

  * Traverse the page for relevant tags (h1,h2,p,a,button,img,section,li,span).
  * Return simplified objects: `{ tagName, text, id, className, selector? }`.
  * Limit items and trim text to keep prompts efficient.

* `src/services/api.ts` — **(create)** API client functions:

  * `generateTour(title, elements)` → POST to `/api/analyze`, normalize to `{ steps: [...] }`.
  * `sendChatMessage(query, title, elements)` → POST to `/api/chat`.
  * Handle network errors gracefully and return friendly messages.

* `manifest.json` — **(create)** extension manifest for development:

  * Request `host_permissions` (e.g., `<all_urls>` for dev), `scripting`, `activeTab`, `storage`.
  * Register `content_scripts` to inject overlay or include content script to inject the built React bundle.
  * Add `background.service_worker` (optional) and `action` (popup button).

* `content_script.js` — **(create)** or built bundle:

  * Inject the React app or mount `TourOverlay` into the DOM.
  * Provide a small bootstrap that inserts a container `<div id="interactgen-root"></div>` and mounts React.

* `background.js` — **(optional)** extension background for long-running tasks or central auth.

* `popup.html` / `popup.js` — **(optional)** a quick UI to toggle the overlay, authorize, or show status.

* `assets/` or `styles/` — styles/graphics used in the overlay.

C — Safety modal (UI + state)

This is mostly inside the overlay but conceptually separate.

* Incorporated in `TourOverlay.tsx`:

  * State variables: `showConfirm`, `clickPendingElement`.
  * `requestClickPermission(element)` sets the pending element and shows modal.
  * `confirmClick()` executes the click and resolves the step; `cancelClick()` skips it.
  * Modal UI element sits on top of page (full-screen translucent backdrop) and clearly describes the action.

* Optionally: `components/ConfirmModal.tsx` — **(create)** a small reusable modal component used for click confirmation.

How each file interacts (mapping)

* `TourOverlay.tsx` → uses `scanPage()` (domScanner) to capture DOM, calls `generateTour()` (services/api) → backend `main.py:/api/analyze` → backend `chains.py` calls Gemini → returns `tour_script` → `TourOverlay` sets `tourData` and executes steps (with modal gating for clicks) → updates UI and TTS.

* `TourOverlay.tsx` → uses `sendChatMessage()` for ad-hoc page Q&A → backend `/api/chat` returns text reply.

Testing & debugging checklist (what to check at each step)

Backend

* Confirm `.env` loads before anything else imports it. On startup, print whether key exists.
* Test `/api/analyze` with a small curl/postman payload → server should return valid JSON with `tour_script` array even when LLM is not wired (use mock).
* When wired to LLM, ensure you use a generation-capable model from ListModels and handle parsing errors (log raw LLM output for debugging).

Frontend / Extension

* Start with mock tour response (hardcoded steps) so you can iterate UI & executor without backend dependency.
* Verify highlight and tooltip render correctly across multiple pages.
* Ensure `scanPage()` returns a manageable number of elements (limit 100) and trimmed text.
* Confirm cross-origin frames are handled (iframe limitations exist — you may need content script in frame).

Click safety

* Test that click permission modal blocks the click until user confirms.
* Test skip behavior (No → skip click and advance).
* Avoid executing clicks on destructive elements (form submits, external links) without additional checks.

Security, deploy & production considerations

Never embed API key in client code or extension package. Keys must remain server-side.
* Add authentication to your backend so only authorized extension instances can call it (e.g., signed token, API key exchange, or OAuth).
* Add rate limiting and request quotas to protect your billing.
* Serve backend over HTTPS in production and restrict allowed origins.
* Add logging and alerts for unusual usage.

UX considerations & safety UX

Always show a visible, persistent overlay so users know automated actions are happening.
For click actions, show element preview + small screenshot (if possible), or the element text, so users can make informed decisions in the modal.
Offer a setting to default to “always ask before clicking” vs “auto click with confirmation on critical selectors”.

Deployment & dev notes

Local dev: run backend with `uvicorn main:app --reload --port 8000` in same terminal where you export the environment key.
Extension dev: load unpacked extension in Chrome/Edge from the build output (React build / bundled content script).
QA: test on 10–15 diverse websites (simple static pages, ecommerce, dynamic SPA) to verify fallback selector heuristics.