High-level checklist

1. Package skeleton

- Create a new file, e.g. `ai-review.el`.
- Add basic headers (`;;; ai-review.el --- ...` etc.).
- Require dependencies:

```
(require 'gptel)
(require 'cl-lib)
```

- Define a minor mode:

```
(define-minor-mode ai-review-mode
  "LLM over-the-shoulder reviewer."
  :init-value nil
  :lighter " AIRev"
  :keymap (make-sparse-keymap)
  (if ai-review-mode
      (ai-review--enable)
    (ai-review--disable)))
```

2. Configure `gptel` backend for Ollama

- Define a backend (for now Ollama, but make it swappable later):

```
(defcustom ai-review-backend nil
  "gptel backend used by ai-review."
  :type 'symbol)

(defun ai-review-setup-ollama ()
  (setq ai-review-backend
        (gptel-make-openai-backend
         "ollama"
         :host "localhost:11434"
         :protocol "http"
         :models '("qwen2.5-coder" "llama3.1" "deepseek-coder"))))
```

- Provide a `defcustom` for the model name:

```
(defcustom ai-review-model "qwen2.5-coder"
  "Model to use with ai-review."
  :type 'string)
```

- In your review function, pass `:backend ai-review-backend :model ai-review-model` to `gptel-request` / `gptel-stream`.

3. Prompt configuration (where does the prompt live?)

You basically have three layers of prompt:

1. **Global/base prompt** – "What does this mode do in general?"
2. **Mode/project-specific flavor** – "For Clojure projects do X, for prose do Y."
3. **Per-call dynamic context** – file name, surrounding defs, story summary, etc.

I'd structure it like this:

- **Global base prompt** as `defcustom`:

```
(defcustom ai-review-base-prompt
  "You are an over-the-shoulder reviewer for Emacs..."
  "Base system prompt for ai-review."
  :type 'string)
```

- **Mode-specific prompts** as an alist:

```
(defcustom ai-review-mode-prompts
  '((prog-mode . "Focus on bugs, refactors, reuse and library suggestic
    (text-mode . "Focus on clarity, grammar, and narrative continuity.'
  "Extra prompt text keyed by parent major mode."
  :type '(alist :key-type symbol :value-type string))
```

- Optionally allow **dir-locals** to override the prompt per project:

```
(defvar-local ai-review-local-prompt nil
  "Optional buffer-local extra instructions for ai-review.")
```

You then build the final system/user prompt in a helper:

```
(defun ai-review--compose-prompt ()
```

```
  (concat ai-review-base-prompt "\n\n"
          (when-let* ((parent (cl-find-if #'derived-mode-p
                                          (mapcar #'car ai-review-mode-pro
                      (mode-extra (cdr (assq parent ai-review-mode-prompts
            (concat mode-extra "\n\n"))
          (when ai-review-local-prompt
            (concat ai-review-local-prompt "\n\n"))))
```

4. Where to *edit* the prompt?

You asked: prompt in the suggestions buffer vs a separate buffer.

My recommendation:

- Keep the **LLM suggestions buffer clean** – mostly output + a little metadata.
- Maintain prompt(s) in:
  - Custom variables (global + mode-specific), and
  - An explicit **"prompt editor" buffer** when you want to tweak them interactively.

Concretely:

- A command to edit the effective prompt in a dedicated buffer:

```
(defun ai-review-edit-prompt ()
  "Edit the effective prompt for this buffer."
  (interactive)
  (let* ((buf (get-buffer-create "*AI Review Prompt*"))
         (current (or ai-review-local-prompt
                      (ai-review--compose-prompt))))
    (with-current-buffer buf
      (erase-buffer)
      (insert current)
      (org-mode)       ;; Optional: for folding/headings
      (goto-char (point-min))
      (setq-local ai-review--source-buffer (current-buffer)))
    (display-buffer buf)))
```

- A command to **save** from that prompt buffer back into `ai-review-local-prompt`

or the global defcustom (your choice):

```
(defun ai-review-save-prompt ()
  "Save prompt from *AI Review Prompt* into this buffer's local setting
  (interactive)
  (unless (eq major-mode 'org-mode)
    (user-error "This command expects the *AI Review Prompt* buffer"))
  (let ((text (buffer-substring-no-properties (point-min) (point-max)))
        (src (buffer-local-value 'ai-review--source-buffer (current-buf
    (with-current-buffer src
      (setq ai-review-local-prompt text))
    (message "ai-review local prompt updated.")))
```

This gives you:

- Clean suggestions window.
- A *separate* space to iterate on prompt, with folding if you want (`org-mode` , `outline-minor-mode` , etc.).

If you *also* want the prompt visible near the suggestions, you can:

- Insert a **read-only header** at the top of the suggestions buffer like:

```
;; in suggestions buffer:
(insert (propertize "Prompt: (press P to view/edit)\n\n"
                    'face 'shadow))
```

and bind `P` to `ai-review-edit-prompt` .

5. Suggestions buffer & mode

- When a review runs, create or reuse a buffer like `*AI Review: foo.clj*` .
- Define a special mode:

```
(define-derived-mode ai-review-suggestions-mode special-mode "AI-Review
  "Mode for displaying AI review suggestions."
  (setq buffer-read-only t))
```

- Store a pointer to the source buffer:

```
(setq-local ai-review--source-buffer (current-buffer)) ; when creating
```

- Keybindings in suggestions buffer:
  - `a` – apply patch/diff to source buffer
  - `g` – rerun review
  - `TAB` – cycle visibility (if you decide to fold sections)
  - `q` – quit window

6. State & idle timer

- Per-buffer state:

```
(defvar-local ai-review--last-hash nil)
(defvar-local ai-review--last-review-time 0)
(defvar-local ai-review--suggestions-buffer nil)
```

- Global idle timer:

```
(defvar ai-review--idle-timer nil)

(defcustom ai-review-idle-seconds 10
  "Idle time before automatically running a review."
  :type 'number)

(defun ai-review--enable ()
  (unless ai-review--idle-timer
    (setq ai-review--idle-timer
          (run-with-idle-timer ai-review-idle-seconds t
                               #'ai-review--maybe-review-current-buffer

(defun ai-review--disable ()
  ;; only turn off timer if no buffers still have ai-review-mode
  (when (and ai-review--idle-timer
             (not (cl-some (lambda (b)
                             (with-current-buffer b ai-review-mode))
                           (buffer-list))))
    (cancel-timer ai-review--idle-timer)
    (setq ai-review--idle-timer nil)))
```

- In `ai-review--maybe-review-current-buffer`, check:
  - Is `ai-review-mode` enabled?
  - Is buffer visiting a file?
  - Has the buffer changed since `ai-review--last-hash`?
  - Has enough time passed since `ai-review--last-review-time`?

  Then call `ai-review-review-now`.

7. The review function (`gptel` integration)

- Gather content:

```
(defun ai-review--buffer-hash ()
  (secure-hash 'sha1 (current-buffer)))

(defun ai-review--region-or-defun ()
  (if (use-region-p)
      (buffer-substring-no-properties (region-beginning) (region-end))
    (save-excursion
      (mark-defun)
      (buffer-substring-no-properties (region-beginning) (region-end)))
```

- Call `gptel-request` (or `gptel-stream`) with your composed prompt and snippet:

```
(defun ai-review-review-now ()
  (interactive)
  (let* ((snippet (ai-review--region-or-defun))
         (system-prompt (ai-review--compose-prompt))
         (src (current-buffer)))
    (setq ai-review--last-hash (ai-review--buffer-hash)
          ai-review--last-review-time (float-time))
    (gptel-request
     (format "Here is some code/text:\n\n%s\n\nPlease respond with a ur
             snippet)
     :system system-prompt
     :backend ai-review-backend
     :model ai-review-model
     :callback
```

```
     (lambda (response)
       (ai-review--display-suggestions src response)))))
```

8. Displaying & applying suggestions

- In `ai-review--display-suggestions`:

```
(defun ai-review--display-suggestions (source-buffer response)
  (let* ((bufname (format "*AI Review: %s*"
                          (buffer-name source-buffer)))
         (buf (or (buffer-live-p ai-review--suggestions-buffer)
                  (get-buffer-create bufname))))
    (with-current-buffer buf
      (ai-review-suggestions-mode)
      (let ((inhibit-read-only t))
        (erase-buffer)
        ;; optional header
        (insert (propertize
                 (format "Review of %s\n\n" (buffer-name source-buffer))
                 'face 'bold))
        (insert response)
        (goto-char (point-min)))
      (setq-local ai-review--source-buffer source-buffer)
      (setq ai-review--suggestions-buffer buf))
    (display-buffer buf '(display-buffer-in-side-window
                          (side . right)
                          (slot . 0)))))
```

- Decide on your wire format:
  - If the model returns a **unified diff**, you can parse and apply it with something like `diff-mode` + a custom "apply all hunks" function.
  - Or tell the model: "Respond with just the corrected code, nothing else," and diff the original vs the new version yourself, then apply.

That's a slightly bigger chunk, but it's mostly plumbing and can be added after the basic roundtrip works.

Prompt placement: final recommendation

Given what you said:

> I could put the prompt there, but it would need to be separated from the LLM feedback, and ideally 'folded' or something. Or, I could create a different buffer for the prompt.

I'd do this:

1. **Primary editable prompt lives in a dedicated buffer**, `*AI Review Prompt*`, opened by a command, using `org-mode` so you can fold headings ("General instructions", "For this project", etc.).
2. **Effective prompt is stored in Lisp variables** (global + mode-specific + optional buffer-local override).
3. **Suggestions buffer** only displays:
   - A one-line "Prompt: (press P to view/edit)" header.
   - Then the model's suggestions/diff.
4. A key in the suggestions buffer (`P`) calls `ai-review-edit-prompt` to jump to the editor buffer.

That keeps:

- UI clean while you're coding/writing.
- Prompt easy to tweak, version, and comment.
- Implementation relatively simple.

If you want, next we can nail down one small piece (e.g., exact Ollama backend definition, or the suggestions buffer mode + keymap) and get that working as a first milestone.