

Khairon User Manual

Khairon is a SpiderGen-based Electron app — an AI development assistant that lets you talk to local/remote LLMs while it reads, searches, and modifies your project code, and even runs builds and tests.

This document is written against the code as of 2026-10-14. Actual UI labels and behavior may differ depending on your build version.

1. Screen Layout

When you launch the app, you will see the following structure.

- **Session List (left docking panel)** — All sessions you have created so far are shown as a tree, grouped by project path. Toggle its visibility with the `View > Session List` menu.
- **Chat Window (center, tabbed)** — One tab is opened per session.
 - Top: session name, token usage, and the AI Provider (model) selection dropdown
 - Center: the conversation area where user/AI/system messages accumulate. When the AI modifies files, an "N files modified" indicator appears together with an **Undo** button, and clicking the indicator expands the list of changed files (see section 3).
 - Bottom: the working folder (cwd) bar and the message input box (including attach and send/stop buttons)
- **Code viewer/editor tabs (MDI)** — Clicking a file path during a chat (on a tool card or an edit card) opens a Monaco-based viewer in a separate tab (see section 5).
- **PreChat List (right docking panel)** — A list where you draft messages before sending them to the chat. Toggle it with the `View > PreChat List` menu (see section 3-1).

Menu Bar

Menu	Item	Description
File	New Session	Create a new session (conversation)
	Open File... / Save File / Close File	Open, save, and close files (used by the code viewer/editor in section 5)
	Exit	Quit the app
View	Session List	Show/hide the session list panel

Menu	Item	Description
	PreChat List	Show/hide the PreChat list panel (see section 3-1)
Build	Build Project / Simulate Project	Build a SpiderGen project and run the simulator
Option	Settings...	App and server settings on the General / Lint / Indexing / Logging tabs (including the screen theme — see below)
	AI Model...	Register/edit LLM Providers (models), OAuth sign-in
	MCP Servers...	Register/edit external MCP servers — the external tools the AI may use (see section 14)
Help	About	Version check — shows the installer Version and <code>Release : AS-builddate-patch</code> . Please include that Release line when reporting an issue (see section 15)

Screen Theme

Pick the screen colors under `Option > Settings...` → **General** tab → `테마` (Theme): `black` (default · dark background) or `green` (green tones on a light background). Press **OK** and it switches right away without a restart; the code viewer/editor (section 5) follows. If you press OK without touching the theme, your current choice stays as it is.

2. Sessions and the Working Folder (cwd)

One session = one tab = one working folder (cwd) + one conversation history. This is the same model as Claude Code's "one cwd per session".

1. Create a new session with `File > New Session` .
2. Click the  **Select Folder** button at the bottom of the chat tab to designate the project/code folder you want to work in.
3. The first time you select a folder, a **trust confirmation dialog** appears. If you choose "Yes", the folder is recorded in `trusted-folders.json` so that the same folder can be used from then on without another trust prompt.
4. Once the folder is trusted, indexing of `.aiassist/samples` begins if you have turned indexing on under `Option > Settings... > Indexing` (see section 4). **This works regardless of project type.** Project source code is not indexed; the AI finds it directly with the search tools.
5. **Note:** Once a conversation has started (once the cwd is locked), the working folder cannot be changed within that session. To work on a different folder, create a new session.

6. If the folder contains a `*.prj` file, it is automatically recognized as a SpiderGen project, a `[SpiderGen]` mark is added to the folder bar, and the `Build` / `Simulate` menus become enabled.
7. **Folders without code work too.** You can pick a folder that holds only documents — meeting notes, specs, definitions (`.md` / `.txt` / `.csv` plus `.xlsx` / `.pptx` / `.docx`) — and ask things like "read every document in this folder and summarize". Text search inside those documents works as well (see section 5-1).

Double-clicking a session entry in the session list (left panel) loads that session's conversation history so you can continue where you left off. Every message is automatically saved locally as it is sent, so no separate "save" action is needed.

Renaming a session — right-click a session entry and choose **Rename** (or right-click a chat tab and choose **Rename Session**) to give the session a name of your own. The name then appears in the session list, the chat tab, and the session label. Saving an **empty** name removes the custom name and restores the automatic one (the first-message preview).

2-1. Archiving Sessions

This folds a session you are done with out of the list instead of deleting it. An archived session keeps its whole conversation and can be reopened at any time.

- **Archiving** — right-click a session in the session list and choose `Archive` . Right-clicking a folder (project) node and choosing `Archive All Sessions` archives every session in that folder at once. There is no confirmation dialog — it happens immediately and can always be undone.
- **Finding archived sessions** — they collect under the `Archived` node at the bottom of the tree, grouped by project. It stays collapsed by default; expand it and double-click a session to open it as usual.
- **Search** — type in the `Search sessions` box above the tree and only sessions whose title (or, if you never named one, the first-message preview) contains that text remain. If an archived session matches, the `Archived` side expands automatically. Clearing the box restores the full tree.
- **Unarchiving** — right-click an archived session and choose `Unarchive` . Or just open the archived session and send one message: **it is unarchived automatically**.
- **A session with an answer in flight cannot be archived** — you get a notice saying so. Try again once the answer finishes (or after you press Stop).
- `Delete All Sessions` on a folder node **does not delete archived sessions** (it only deletes what is visible under that folder). To delete archived ones too, right-click the top `Sessions` root and use `Clear All Sessions` .
- Nothing is archived or deleted on its own — archiving is something you turn on and off yourself.

- **Several sessions at once** — pick multiple sessions with **Ctrl+click** (add/remove one) or **Shift+click** (range), then right-click one of them: `Archive (N)` · `Unarchive (N)` · `Delete N Sessions` apply to the whole selection. Sessions hidden in a collapsed folder or by the search box are not included.
-

3. Basic Chat Usage

- Type a message in the input box at the bottom and press **Send** to submit it to the server; the AI response is displayed as it streams in.
- `Shift+Enter` (or `Ctrl+Enter` / `Alt+Enter`): insert a line break in the input box (Enter sends)
- The input box **grows with its content**. As you add lines it expands up to about three times its default height, then scrolls inside the box. It returns to the default height when you send or clear the text.
- **Input history** (`↑` / `↓`): press `↑` in the input box to recall the **message you sent last** in this session; keep pressing to go further back (like a command prompt). `↓` walks forward again, and past the newest entry you get back whatever you were typing. While composing a multi-line message, `↑` only recalls history when the caret is on the first line and `↓` only on the last line — otherwise they move between lines as usual. Editing a recalled message turns it into a new draft. Reopening an earlier session refills the history with that session's messages.
- **Stop button**: interrupt response generation
- **Sending messages while the AI is working**: you can send a message with Enter (or **Send**) even while the AI is in the middle of a task.
 - The message goes into a **queue** and the input box is cleared. A queued message appears in the chat as your bubble, with **지금 보내기** (Send now) and **×** buttons on it, and the progress line also shows `· 대기 메시지 N` (N queued messages).
 - Queued messages are handed to the AI **when it moves on to its next step** (the bubble's tag changes to `전달됨`, "delivered"), and the AI takes them into account as it carries on. Anything still undelivered when the task ends is **sent automatically right after it ends** (several messages are combined into one).
 - **지금 보내기 (Send now)**: stops the current task and sends the queued messages right away as the next request.
 - **×**: cancels the queued message — its text goes back into the input box, so you can edit it and send it again. If it has already been delivered, you get a `이미 전달됐습니다` ("already delivered") notice.
 - Pressing **Stop** puts any undelivered messages back into the input box.
 - **Slash commands and image attachments cannot be sent while a task is running** — a notice appears and your input stays as it is, so send them after the task finishes. Only `/open` runs immediately even during a task. Sending from the PreChat list (section 3-1) follows the same rules.

- If a response grows abnormally long, the server cuts it off at its **generation cap** and shows a notice that generation was stopped at the server's limit. Split the request or try again in a new chat.
- **Attaching files** ( button · drag and drop · `Ctrl+V`): what happens depends on the file type.
 - **Images:** attached as thumbnails and sent with the message. The model you are using must support vision (see the `vision` option under Provider settings in section 9). If your model has no vision, set a **Vision Broker** (section 9) so another model reads the image on its behalf and passes along a description — without one, the image is dropped and the AI is only told that it was. There is a maximum number of images. `Ctrl+V` paste accepts images only.
 - **Text files and documents** (`.xlsx` / `.docx` / `.pptx` / `.pdf`): instead of sending the contents, the **file path is inserted into the input box** (relative to the working folder when inside it). Add what you want done and send — the AI reads it with `read_file`. You can drop several files at once.
 - **Other binaries** (legacy `.xls` / `.doc` / `.ppt`, archives, executables, ...): the AI cannot read them, so they are not attached and a notice is shown.
- Use the **AI Provider dropdown** at the top to switch the LLM used by the current session on the fly. (Whether switching clears the conversation is controlled by an option under `Option > Settings... > General`.)
- **Reasoning level button:** the **reasoning level button** on the folder row (current level name + ) opens a small popup above it — a `Reasoning level · <current level>` header, a  help toggle, and a slider for picking the level. Dragging the slider updates the header, and the choice is applied the moment you release it. Clicking outside the popup, pressing `Esc`, or clicking the button again closes it.
 - A higher level means deeper reasoning, at the cost of slower responses.
 - **Level names:** local models use `Light` · `Medium` · `High` · `Extra High` · `Max` (each step raises the reasoning budget), while cloud models such as Claude, ChatGPT and Gemini keep their own wording (`effort: high`, `reasoning: medium`, and so on).
 - **The number of stops depends on the model you selected.** Local vLLM/Ollama, Claude and Gemini offer 5 stops; local OpenAI-compatible servers and OpenAI/Codex offer 4. Switching models redraws the slider accordingly.
 - **The level is remembered per model.** If you set, say, High for a local model and `effort: medium` for Claude, each model returns to the level you last chose for it as you switch back and forth. The first time you use a model it starts at the default level (the 1K-budget step for local models, the third step for cloud models). On a model with fewer stops the display simply folds to the last stop, and your saved level comes back when you return to the original model.
 - If no model is registered at all, the button stays disabled (`Reasoning level`) and the popup does not open — register a model under `Option > AI Model...` first.

- On local models, **at every level**, once the work is done the AI **cross-checks the result against your request**, and on SpiderGen projects it also **verifies the build and that the screen loads** before reporting.
- **Delegating large work:** for work that takes a while, the AI can hand the job to a background worker and keep talking to you — you can also ask for it directly ("delegate this and keep going") (section 13).
- **How responses are shown (conversation view):** each request, its work process and the answer are shown as **one block**.
 - **While the AI is working**, only a single **progress line** at the bottom of the block (`작업 중` `· elapsed time · what it is doing now`) and the AI's interim notes are visible; the process itself (tool runs, command cards) is hidden. Click the progress line (`▶`) to unfold the process.
 - **When the task ends**, the process folds away and a **summary line** (e.g. `▶ 작업 1분 32초 · 스텝 7 · 도구 12회 · 파일 3개` — work time · steps · tool calls · files) appears with the final answer below it. Click the summary line to unfold the process (`▼`). What stays visible outside the fold: the AI's interim notes, the final answer, file-edit cards with the Undo bar, background task cards that are still running (section 13), and warnings and errors. A request answered directly, with no work process, has no summary line.
 - **Command runs** that follow one another are grouped into **one card** — header `AI Command` `· 명령 N · 실패 M` (N commands · M failed), collapsed by default. Click the header (`[show]`) to list one line per command, and click a line to unfold that command's output.
 - When a tool failed and the AI tried again, the progress line and the summary line show `재시도 N` (retry N).
 - Reopening an earlier conversation shows it the same way (with everything folded).
 - Code blocks (collapse/copy), tables (Markdown tables render as tables), and file-path links (click to open the code viewer) are displayed inline in the chat.
- **Undo:** when the AI modifies files, an `N files modified` indicator and an **Undo** button appear at the bottom of the message.
 - Clicking the indicator expands **the list of files changed this time**. Clicking a file name opens it in the code viewer, so you can check what actually changed yourself instead of relying on the AI's description.
 - Pressing **Undo** reverts the modification by one step (back to the "immediately previous state"), and the fact that an undo occurred is also passed to the AI with your next message so that the context stays consistent.
 - Newly **created** files are not undone (only modified existing files are restored) — they do appear in the list, so you can delete them yourself if needed.
- **In-chat search (`Ctrl+F`):** press `Ctrl+F` in the chat screen to open a search box in the top-right corner.
 - Matches in the conversation are highlighted as you type (case-insensitive), and your current position is shown as `1/3` .

- `Enter` or `↓` jumps to the next match, `Shift+Enter` or `↑` to the previous one (wrapping around at the ends). The chat scrolls to each match automatically.
- Close with `Esc` or `×` and all highlights disappear. If you have text selected in the chat when you press `Ctrl+F`, it is prefilled as the search term.
- The search box works per chat tab — switching to another tab hides it along with the tab, and it is available again when you return.

3-1. PreChat List

A panel for **drafting messages in advance** — frequently used instructions, or messages you are not ready to send yet — and sending them to the chat when needed. Open it via `View > PreChat List`.

- **Add:** type a message in the input box at the bottom of the panel and press `Enter` (`Shift+Enter` for a newline). It is added to the list as a bubble.
- **Send (double-click):** double-clicking an item **sends it immediately** to the active chat session. The item stays in the list, so you can reuse the same instruction repeatedly. If you had a draft in the chat input box, it is restored after sending. If the AI is in the middle of a task, the message is queued (see "Sending messages while the AI is working" in section 3).
- **Copy (📄):** puts the message into the chat input box (appended after a newline if the box already has content) and also copies it to the system clipboard. The item stays in the list.
- **Delete (🗑️):** removes that item. The **Clear All** button empties the current folder's list after confirmation.
- **Edit:** click an item to select it, then press `F2` or the **Edit (F2)** button — the bubble becomes editable in place. `Enter` saves, `Shift+Enter` adds a newline, `Esc` cancels.
- **Context menu: right-click** an item to get a **Copy / Delete / Send to chat** menu — the same actions as the buttons above, usable even when the panel is too narrow to show the buttons (the menu does not appear while an item is being edited).
- **Per-folder lists:** the list is stored **per working folder (cwd)**. Switching chat tabs or changing the working folder automatically switches to that folder's list, and the panel header shows the current folder path (last two segments). Sessions without a working folder (No Project) use their own list. Lists survive app restarts.
- **Target chat:** the active chat tab → otherwise the first open chat tab → otherwise a new session is created.

Slash (/) Commands

These are commands **you run directly**, without going through the AI's judgment. Type them in the chat input box in the form `/command` and they run immediately. You can list them all at any time with `/help`.

Command	Description
<code>/find <part of a file name> [-n count]</code>	Searches the whole project by file name (runs <code>find_file</code> directly; case-insensitive, partial match). Shows up to 50 results by default and says <code>limit reached</code> when there are more — use <code>-n</code> to change the count (max 500). — Example: <code>/find manager</code> , <code>/find .lay -n 200</code>
<code>/grep <query> [-n count]</code>	Searches the whole project for text (runs <code>search_project</code> directly). Shows up to 20 results by default; raise it with <code>-n</code> (maximum 500). If the limit is reached it is shown as <code>N+ matches (limit reached)</code> . — Examples: <code>/grep loadConfig</code> , <code>/grep sendResult -n 100</code>
<code>/outline <file path></code>	Shows the file's function/class list (headings for markdown) with line numbers (runs <code>file_outline</code> directly). — Example: <code>/outline Source/mdi/AlChatView.js</code>
<code>/tree [path] [depth]</code>	Shows the project folder structure as an indented tree (runs <code>project_tree</code> directly). Both path and depth can be omitted (defaults: root, depth 3, maximum 5). — Examples: <code>/tree</code> , <code>/tree Source 2</code>
<code>/open <file path></code>	Opens a file immediately in the code viewer (Monaco, read-only) — if there is no split, it opens in a new split on the right. — Example: <code>/open Source/MenuInfo.js</code>
<code>/simulate</code>	Runs the built SpiderGen project in the simulator window (runs <code>spidergen_simulate</code> directly). If <code>bin/index.html</code> does not exist, you are prompted to build first.
<code>/skill <skill name> [request]</code>	Force-triggers the specified skill (section 6). Unlike the other slash commands, it does not bypass the AI — the skill body (instructions) is injected into this request, and the AI continues the work following those instructions. Useful when the AI fails to trigger a skill automatically. Typing just <code>/skill</code> with no name lists the available skills. The injection applies only to that turn. — Example: <code>/skill my_skill apply it to the login screen</code>
<code>/<skill name> [request]</code>	You can use a skill name directly as a slash command — <code>/my_skill apply it to the login screen</code> is identical to <code>/skill my_skill apply it to the login screen</code> . This works for every skill (built-in or project) whose name does not collide with a basic command above; basic commands always take precedence. A name that matches no skill is treated as normal chat text.
<code>/help</code>	Immediately lists every available slash command and skill (with source tags <code>[내장]</code> = built-in / <code>[프로젝트]</code> = project).

4. Indexing (RAG)

- **Off by default** — this feature needs an embedding server, so you must **turn indexing on and enter the embedding server** under `Option > Settings... > Indexing`. While it is off, conversations and tools work normally.
- **Automatic reference material**: once it is on, the files in `{project-folder}/.aiassist/samples` are indexed in advance, and material similar to your request is found and referenced automatically. **This works for any kind of project** — SpiderGen or not, it operates as soon as you put files in that folder. If the folder does not exist, only this feature is off and the conversation works normally.
- **The content is not limited to UI samples either** — code examples, convention documents, design notes, frequently used patterns, or anything else you want the AI to consult.
- **Project source code is not indexed** — the AI finds it directly with the search tools (`search_project` and so on) when needed.

Adding Samples

To give the AI more material to consult, create a folder under `.aiassist/samples/` and put files in it. The example below adds a SpiderGen UI pattern, but there is no required file type or folder structure.

```
ButtonSample/  
  _desc.md          (optional) sample description  
  ButtonSample.lay  layout  
  ButtonSample.js   script
```

- If the folder does not exist you must create it first for indexing to start, and only material that is sufficiently similar to the request is adopted automatically.
- Indexing results are cached, so files that have not changed are not re-indexed.
- The `.aiassist` folder itself is excluded from general search tools such as `search_project` and `find_file`.

5. Code Viewer / Editor (Monaco)

Click a file path shown on the AI's tool call card or edit card during a chat and that file opens in a Monaco-based viewer in an MDI tab. You can click to open it the same way after opening a session anew and loading an earlier conversation. File paths shown as **inline code in chat text** (e.g. a file list the AI mentions in its report) can be clicked and opened the same way.

- HTML-family files such as `.html` and `.lay` are automatically beautified for readability.
- `.md` **(Markdown) files open as a preview** — headings, tables, lists, and code blocks are rendered as a document, just like in chat, and the "View source" button at the top right switches to the raw text. This is meant for reading document deliverables such as analysis reports. (If the "Edit mode when opening source" setting is on, the file opens in source/edit mode first.)
- Files open **read-only** by default. If you enable the "Edit mode when opening source" checkbox under `Option > Settings... > General`, files will open in an **editable** state from then on.
- In edit mode, changing the content adds a modified mark to the tab title; saving with `Ctrl+S` or `File > Save File` removes the mark.
- **Git change gutter markers:** if local git is available in the Electron environment, changed lines are marked in the left gutter when you open a file — added (green bar) / modified (blue bar) / deleted (red triangle). The comparison baseline is the working tree vs HEAD when there are uncommitted changes, and the last commit (HEAD vs HEAD~1) otherwise. Clicking a gutter marker opens a full-view overlay where you can compare the original (base) with the current content (peek).

5-1. Documents (Excel / PowerPoint / Word / PDF)

The AI can **read Office documents and PDFs, and edit Office documents**, the same way it handles code files.

- **Reading:** `.xlsx` (one table per sheet), `.pptx` (text and notes per slide) and `.docx` (paragraphs and tables) are read as text. Examples: "read spec/plan.pptx and build the screen", "show me only the W1805A01 sheet of messages.xlsx" — for Excel you can name a sheet to read just that part; charts and images are reported by count only.
- **Reading PDFs:** `.pdf` files are read as text per page (`## Page N`). For long PDFs you can ask for a page range ("just pages 3-5"). This works for PDFs that contain text (exported from Office, word processors, browser print, ...); **scanned-image PDFs have no extractable text** and the AI tells you so. Password-protected PDFs cannot be read either.
- **Editing:** changes an Excel **cell value**, the **text on a PowerPoint slide**, or a **Word paragraph** in place. Only the affected part is patched, so formatting, formulas elsewhere, charts and images stay untouched. Adding rows/columns/sheets or changing formatting is not supported.
- **Undo:** exactly like code files — **Undo** restores the original bytes.
- **Note:** legacy formats (`.xls` / `.ppt` / `.doc`) cannot be read — save them as `.xlsx` / `.pptx` / `.docx` first. Non-text files such as archives and executables cannot be opened by the AI, and it tells you so.

- **Dropping a document into the chat or picking it with the  button inserts its path into the input box** (see section 3). Files outside the working folder can be read by absolute path as well.

6. Skills (Per-Project Custom Guides)

A skill is a work guide (a markdown document) that says "when a request like this comes in, handle it like this". The AI is aware of the skill catalog (name + description) on every request, and when your request matches a particular skill's description, it loads the full body with the `use_skill` tool and follows those instructions.

Built-in Skills

Built-in skills are **merged into the same catalog** as your project skills (tagged `[내장]` (built-in) in `/help`). If you create a project skill with the same `name`, **the project one wins** — this is how you override a built-in skill to fit your project.

There is currently one built-in skill, `create_agent` — ask something like "make me an agent that only investigates code" and the AI reads this skill and creates the agent definition file of section 13 in the right place and format.

Storage Location

The following folder is created automatically the first time you select a working folder.

```
<project-folder>/aiassist/skills/  
my_skill.md          (a skill you created – this project only)
```

Skills you want in every project go in `skills/` under the user-data folder (`<userData>` of section 8): `<userData>/skills/my_skill.md`. This folder is not created automatically — create it when you need it. When names collide, the first of **project > global (`<userData>/skills`) > built-in** wins, and `/help` shows the source as `[프로젝트]` (project) · `[전역]` (global) · `[내장]` (built-in). Global skills are read when you select (or re-select) the working folder.

Skill File Format

```
---  
name: my_skill  
description: one line describing when to use it (max 300 characters)  
---  
# Body  
  
Write step-by-step instructions, examples, cautions and anything else you need here.
```

- `name` and `description` are **recommended** (if missing, they fall back to the file name and the first paragraph of the body).
- `description` is what the AI uses to decide whether to use this skill, so writing one line yourself makes it trigger more accurately.

How to Use

1. **Ask the AI to create it:** ask something like "save this way of working as a skill" in the chat and the AI creates the file in the correct location and format for you.
2. **A newly created skill is usable right away** — no re-indexing needed. Only when you edit an existing skill's `name` / `description` yourself do you need to **select the working folder again** for the change to be picked up (body edits take effect immediately). If a new skill's `name` matches a skill that already exists (including global and built-in ones), the AI tells you — rename it to use it right away (when it collides with a global or built-in skill, the new project skill wins after you re-select the working folder).
3. From then on, when you make a request that matches the skill description, the AI automatically loads that skill and works according to its instructions. If the AI fails to trigger it automatically, you can trigger it yourself with `/skill <skill name> [request]` (see the slash commands in section 3).

7. Memory

A per-project persistent memory feature is implemented. Facts and preferences you saved earlier are retained even when you open a new session on the same project folder.

- Storage location: `{project-folder}/.aiassist/memory.md` (a separate file per project — it is not a global memory shared by all projects).

- How to save: when you state a new lasting preference or explicitly ask the AI to "remember this", the AI records it one line at a time with the `save_memory` tool (saved immediately, without a separate approval).
- Usage: saved content is automatically applied to all later conversations, and the AI refers to it like a standing instruction. You may also edit or delete the file directly; changes take effect from the next chat.
- If you want to reuse common rules that span projects (coding conventions and the like) across several projects, managing them as **Skills** (section 6) is a better fit — global skills in `<userData>/skills/` in particular are visible in every project. (Skills keep separate files and load the full body only when needed, which suits longer procedures and rule sets.) Memory suits short fact/preference bullets that keep accumulating for a single project.

8. Where Session Data Is Stored

All conversations are saved automatically to local storage (Electron `userData`).

```
<userData>/aiassist/
sessions.json      (session index: sessionId, projectPath, preview, created/modified time,
data/
  <sessionId>.jsonl  (conversation content per session, append-only)
archive/data/
  <sessionId>.jsonl.gz (conversation content of sessions archived more than 7 days ago – comp
```

Once a session has been archived (section 2-1) for 7 days, its conversation is compressed and moved to `archive/data/<sessionId>.jsonl.gz` . You open it exactly the same way (it is decompressed for you), and unarchiving moves it back, so there is never any need to touch these files yourself.

You can delete and manage sessions from the session list panel (right-click menu).

9. AI Model (Provider) Settings

Register and edit the LLM Providers you want to use from the `Option > AI Model...` menu.

Field	Description
Name	The identifier shown in the dropdown

Field	Description
API Type	The kind of model server — vLLM / Ollama / Claude / OpenAI / Gemini / OpenAI-compatible
model	Model name (e.g. <code>qwen3-coder-next</code>)
host / port	API endpoint
chatPath	Chat API path
Max Context	The model's context limit (in tokens, <code>max_model_len</code>) — detected automatically if left empty (see below)
(advanced) <code>max_output_tokens</code>	Maximum output tokens per response. Normally left unset — it is chosen automatically from the context size (8K–32K). Set it directly in the config file (<code>ai-client-config.json</code> / <code>ai-config.json</code>) if you need to
https	Whether to call the endpoint over HTTPS
Authentication method	Choose either API Key or OAuth (account sign-in) (see below)
apiKey	Authentication key (shown only for the API Key method)
headers	Custom headers to add to requests (one per line, in <code>"name": "value"</code> format)
vision	Whether image input (attachments) is supported
Vision Broker	A model that reads images on this model's behalf when this model cannot accept them (vision unchecked) — see below
Streaming	Whether to show the response as it arrives. Unchecked, the response is shown all at once when it is finished. Claude and a ChatGPT subscription (OAuth) always stream

The built-in Providers (server-managed, `ai-config.json`) and the Providers you add yourself (`ai-client-config.json`) are shown together in the dropdown. An embedding model (for sample indexing) is also configured separately.

Vision Broker — a "reader" model for models that cannot accept images

Use this when you work with a model that has no vision (e.g., a text-only GLM) but still want to use images. Once set, whenever an attached image or an automatic UI-test screen capture arrives, the broker model first describes the image in detail, and your current model receives that description and carries on "as if it had seen it".

1. In `Option > AI Model...`, select the model that cannot accept images and **unchecked vision** — the select box below only becomes enabled when vision is unchecked.

2. In the **Vision Broker** select box, choose the model that will read images instead. The list shows only **user-added models with vision checked** (the model itself and built-in models are excluded); "None" turns it off. Choosing a cloud model sends your images to an external service and may incur charges.
3. Press OK — the change applies immediately (no restart).

Notices you may see during use (System lines in the chat):

- **"* The current model has ... interpret images on its behalf."** — shown once per conversation in which the broker was used (local or cloud alike). The same image is never read twice.
- **"▲ This model (...) does not accept image input (rejected by the backend)"** — vision was checked, but the actual model refused images. The broker does **not** switch on automatically in this case — uncheck vision for that model and set a broker.
- **"▲ The vision broker (...) failed to interpret image ..."** — an error on the broker model's side. The image is left out; check the broker model's settings.

Keep in mind: what gets passed along is a text description, so details such as "the exact color of the top-left button" may be lost. If the model you chose as broker is later deleted or its vision is unchecked, the setting falls back to "None".

Specifying the Context Limit Manually (`max_model_len`)

A model's context limit (total token count) is normally detected automatically. In environments where automatic detection fails (in-house gateways, proxies, and so on), long conversations may be truncated early or raise context-overflow errors, so specify the limit yourself in that case.

- **Providers you added:** enter it in the **Max Context** field of the `Option > AI Model...` settings window (stored as `max_model_len` in `ai-client-config.json`)
- **Built-in Providers:** write `max_model_len` directly into the provider entry in `ai-config.json` :

```
"my-gateway": {
  "host": "gateway.example.com",
  "port": 443,
  "chatPath": "/v1/chat/completions",
  "model": "OpenAPI",
  "max_model_len": 65536
}
```

It can be used in both `ai-config.json` and `ai-client-config.json` , and takes effect as soon as you save the file.

OAuth (Account) Sign-In

Instead of an API key, you can **sign in with a Claude or ChatGPT account** and send requests using that subscription.

1. On the Provider edit screen, switch the authentication method to **OAuth**.
2. Select the provider type (`claude` | `chatgpt`).
3. Click the **Sign in** button; a browser opens and takes you to that service's sign-in/authorization screen.
 - Claude: copy the code displayed after signing in, paste it into the input box, and click the **Apply Code** button to exchange it.
 - ChatGPT: the authorization code is received automatically over the local loopback address and exchanged without any pasting.
4. When the connection succeeds, the status indicator changes to "Connected".
5. The issued tokens are stored **encrypted** and are retained even if you reinstall the app.

`Option > Settings...` manages general options (whether to clear the conversation when switching models, edit mode when opening source, and so on) and Lint-related settings. Lint's **Auto Lint** (automatically running ESLint on just the edited file after a successful `edit_file`) requires the project's `package.json` to have an `eslint` dependency; if it is enabled but `eslint` is missing, a skip notice is shown once per session with the edit result. The `lint` tool that the AI calls directly likewise requires `eslint/prettier` to be installed in the project, and returns a notice suggesting installation if they are not.

10. Command Execution Approval and Safeguards

- When the AI tries to run a shell command, an **approval request popup** appears and you decide with **Deny** or **Allow**.
- The approval popup appears **inside the chat tab that asked**. If several requests pile up in one tab (for example the chat and a delegated worker asking at the same time), they appear **one at a time, in order**, and a request from another tab **highlights that tab's button with a background color** (it blinks once when it appears and returns to normal when you open the tab or the request ends). **There is no time limit** — it waits until you answer.
- While an approval popup is up it covers that tab's Stop button. To stop the work, press **Deny, then Stop**. Stop also cancels the approval requests the chat was waiting on (requests from a background delegated worker remain while that delegation keeps running — section 13).
- At the bottom of the approval popup there is an **"Auto-approve for this session"** checkbox. If you check it and then press **Allow**, later `run_command` / `process_op` (starting a background process) requests in the same session (connection) are approved automatically without a

popup. Even then, however, the blocklist check below is always performed first. (If you choose Deny, auto-approval is not applied regardless of the checkbox.)

- An approval popup also appears when the AI tries to send an **HTTP request to an external address** (local addresses such as `localhost` run immediately). External requests are confirmed **every time**, and that popup has no auto-approve checkbox.
- **Project commands** (npm, npx, pytest) also run through the same shell. Scripts listed in `package.json` (`npm run <name>` , `npm test`), `npm install` / `npm ci` without package names, `npx jest|vitest|mocha|eslint|prettier|tsc` and `pytest` run without approval; `npm install <package>` (**adding a dependency**) and anything else show the approval popup. Commands that do not exit on their own, such as a dev server, are started in the background with `process_op` , which asks once when starting.
- **git commands** run through the same shell (`run_command`) — there are no separate git tools. Read-only commands such as `git status` , `diff` and `log` , and local work such as `add` , `commit` and switching branches run without approval; `git push` and destructive commands (`reset --hard` , `checkout -- <path>` , `clean` , `branch -D` , `stash drop` , `rebase` , `commit -- amend`) ask **every time, regardless of auto-approval** (that popup has no auto-approve checkbox). The output of `git diff` / `git show` is shown in the chat as a colored diff.
- **Process-killing commands** (`taskkill` , `kill` , `pkill` , `Stop-Process` and the like) also ask **every time, regardless of auto-approval** (no checkbox). This keeps the AI from accidentally terminating Khairon itself or another job.
- On Windows the AI's shell is **Git Bash** (PowerShell if Git for Windows is not installed). The user terminal tabs stay on PowerShell.
- Destructive or dangerous commands such as `rm -rf /` , `git push --force` , `sudo` , and `curl | bash` are blocked outright by the server, regardless of auto-approval.
- Command output (standard output) is streamed in real time so you can follow it on screen.

11. Building and Simulating SpiderGen Projects

This chapter applies only when you are working on a SpiderGen project. Once the working folder is recognized as a SpiderGen project (containing a `*.prj` file), build and simulation become available.

- `Build > Build Project` : builds the project.
- `Build > Simulate Project` : runs the build output in the simulator window.

To verify screen behavior automatically, use the **automated UI test** in section 12.

12. Automated UI Test (run_test)

This chapter also applies only to SpiderGen projects.

Ask **"test ~ for me"** in the chat and the AI plans a test scenario, drives the simulator itself, checks each step, and reports the outcome. No mode switch or preparation is needed.

Making the request

- Examples: "test that pressing Find ID on the login screen opens the dialog", "test the sign-up input validation".
- **Put the values to be used right in the request** — accounts, verification codes, and the like: "test login with ID asoosoft, password winway1, verification code 123456". The AI types the values you specified even if the fields already look filled in (so a remembered value cannot make the test pass with something you did not intend).
- How it works: the AI reads the code to work out the entry path → writes a scenario file → runs the simulator (building first if needed) → clicks and types through the screen step by step, confirming each step's success condition → reports per-step pass/fail with evidence in the chat. You can follow it in the chat's tool-execution display, and watch it directly in the simulator window (which is left open after the test).
- When the AI finds a defect it **only reports it — it does not modify your project code**. Ask separately if you want a fix.

Writing a scenario without running it

Use this when you want to review the plan first instead of running the test right away.

- Examples: **"write a login test scenario named order_test"**, "just write the scenario for the sign-up test".
- The AI checks the code and **writes the file only** — with the steps and each step's success condition. It does not build and does not launch the simulator.
- If a scenario with that name already exists, it **updates that file** instead of writing a second one.
- Read it, edit it if you like, then run it with "test with that scenario" (or "run order_test").

The scenario file

- Scenarios are kept as `.md` files under `.aiassist/tests/` in the working folder. They list the steps and each step's success condition, so you can read and edit them yourself.
- When a test finishes, the outcome (pass/fail and a summary) is **appended to the end of that same file with a timestamp** — history accumulates with each run.

Re-running a saved test

- **The file name alone is enough**: "run order_test again", "test with order_test.md". It is looked up under `.aiassist/tests/`, ignoring the extension, letter case, and `-` versus `_`.
- A re-run skips the code investigation and scenario writing and executes straight away, so it is much faster than the first run.

- If several files share the name, the AI lists the candidates and asks which one you mean — answer with the exact path (`.aiassist/tests/order_test.md`).
- If no scenario matches the name, it is treated as a **new test**: a fresh scenario is written and saved under that name. If you meant to re-run an existing one, check the file name.

Recording your own actions into a test (● Rec)

You can record manual interactions with the simulator and turn the recording into a test.

- Start recording with the **● Rec toggle button** in the top bar (the simulator window must be open). Use the app as usual — clicks and inputs are recorded.
- Press the button again to stop: a **name dialog** appears, and saving stores the recording as `test-scenario/records/<name>.json` under the working folder. Cancelling the dialog keeps the recording in memory (it is lost if the app reloads).
- Then ask in chat: **"run a test following the records/ recording"** — the AI reads the recording and converts it into a scenario (`.md`) with the same name, then runs it. The action steps come from the recording; the success conditions (verification) are derived by the AI from the screen context. The original `.json` recording is preserved as material for re-conversion.
- To fold a recording into an existing scenario, ask something like **"update using the records/ recording"**.

Turn frequent tests into skills — for a test you run repeatedly, make it a skill (section 6) and run it with `/name`. Keep only "run the test with the `.aiassist/tests/<file>` scenario" plus parameters such as the account in the skill body, and leave the steps and expected values in the scenario file — that way results accumulate in the scenario file rather than in the skill.

Limits: the simulator always starts from the app's first screen, so the AI navigates to the target screen as part of the test. Screens that need a separate backend server cannot be tested unless that server is running (the AI reports that it could not reach them).

13. Background Delegation (Sub-agents)

This hands a large piece of work to **another AI worker while you keep talking**. There is no button or menu — the AI uses it on its own when it judges the work warrants it, and you can also ask for it directly ("delegate this and keep going"). It is available on every model (local, Claude, ChatGPT).

How it flows

1. When the AI hands work off, a **DELEGATE: d1g1 card** appears in the chat and streams what the worker is reading and changing, live.

2. The AI **does not wait — it ends its turn**. The card keeps progressing after the conversation ends, and **stopping the conversation with the Stop button does not stop the delegated work**.
3. When the worker finishes, **the AI picks it up on its own — you do not have to say anything**. A line appears in the chat ( 백그라운드 위임이 끝났습니다 - 완료 보고를 받아 이어갑니다.), then the AI reads the completion report and carries on with your original request — it checks the deliverables, applies them to the real project and tells you what it applied, or hands the next piece of work to a worker.
4. In a SpiderGen project, when code or screen files were applied, the AI **follows up by verifying the build — and for screens (`.lay`), that they actually load** — and reports the results along the way.
5. If it judges the deliverables to be faulty, it does not apply them and says so.
6. If you were in the middle of another exchange while the delegation was running, the AI picks it up once that answer finishes. When a completion report is folded into an ongoing exchange, you see one line:  백그라운드 위임 완료 보고를 이번 메시지에 첨부했습니다.

Do not edit the delegated files yourself while the work is running. If the same file changes in your editor meanwhile, the AI does **not** overwrite it — it holds that file back and tells you, so your edit is never lost. It then re-reads the current file to merge, or waits for your call.

Deliverables and applying them

Files produced by the worker are stored **only in the work folder** (`.aiassist/quality-work/<delegation id>/`). The worker cannot touch real project files; those change only when the AI applies the deliverables (`apply_files`) after reviewing them. Once applied, you can revert them with **Undo**, exactly like any other edit.

You do not need to approve each apply. If you dislike the result, revert it with **Undo** or ask for a rework. The Undo button only shows for that turn, so to revert an older apply just say "revert the files you applied earlier".

The worker's **final answer (report)** is always saved in full as `report.md` in the work folder. For research-style delegations where the answer itself is the deliverable, the AI picks up that content as-is; if the answer is very long, the AI reads `report.md` to continue. To keep answers short, say so when delegating (e.g. "key points only, under 2,000 characters").

The work folder is never cleaned up automatically, so you can still open deliverables you chose not to apply.

Agent definition files (choosing the worker)

You can pin down the **model, thinking level, tools and extra instructions** a worker uses by putting a file in your working folder at `.aiassist/agents/<name>.md`, in the same format as the Skills of section 6 (frontmatter + body). When the AI delegates, it reads the `description` and picks the fitting worker. You can also name one yourself ("hand this to `grep_local`").

Workers you want in every project go in `<userData>/agents/` under the user-data folder (same rule as global skills — create it yourself; on a name collision the project one wins). A global worker's `model` must be a name registered in `AI Model...` on that PC.

```
---
name: grep_local
description: Investigation-only worker. Searches the code and reports findings without editing a
model: qwen-local
quality: 1
tools: read_file, search_project, list_files, find_file
maxRounds: 40
---

Do not modify project files. Find where and how the requested feature is implemented and report
```

Field	Description
<code>name</code> · <code>description</code>	Required. <code>description</code> is what the AI uses to pick this worker, so make that one line clear
<code>model</code>	The name of a model you registered under <code>Option > AI Model...</code> . Leave it out to use the same model as the chat
<code>quality</code>	Thinking level <code>1</code> – <code>5</code> . Leave it out to use the same level as the chat
<code>tools</code>	The tools the worker may use, comma-separated. Pick from the default tools, or list MCP tools you connected in section 14 (<code>mcp__<server>__<tool></code>) to hand them to the worker. An MCP tool needs its server connected at delegation time (if it is disconnected, the delegation does not start and the AI tells you). Leave it empty for all default tools
<code>maxRounds</code>	Cap on how many work rounds the worker takes. If you write more than the model's own cap, the model's cap is used
<code>wait</code>	<code>true</code> makes it a waiting delegation — instead of ending its turn, the AI waits for the worker's result and carries on within the same answer (below)

Field	Description
<code>profile</code>	<code>plain</code> makes it a worker for non-coding roles (review, judgement, summaries...) — the coding rules and work-folder instructions are left out and it follows the output format written in the body as is. Its default tools are read-only, and listing a file-editing tool in <code>tools</code> makes the definition invalid. Omit it for the usual coding worker
Body	Extra instructions for the worker. The default instructions (work-folder confinement, report format) still apply (with <code>profile: plain</code> , the body's format takes precedence)

Waiting delegation (`wait: true`)

A normal delegation runs in the background: the AI ends its turn and picks the result up on its own later (see "How it flows"). When you hand work to a worker with `wait: true`, **the AI waits for the result right there and moves on to the next step within the same answer** — useful for skills (section 6) that chain steps such as "design review → implementation → tests". Several workers called at once run **in parallel**, and the `DELEGATE` cards show progress while it waits. In this mode **Stop also stops the workers** (background delegations keep running regardless of Stop). A stopped delegation can be resumed later, e.g. "continue dlg3".

- When a delegation runs with an agent, the `DELEGATE` card title and the completion report both carry `agent=<name> · model=<model>`.
- Ask in the chat ("make me an agent file that only investigates code") and the AI reads the built-in skill `create_agent` (section 6) and creates it in the right place and format (it asks you first if it does not know which model to use), and **a newly created definition is usable in that same session** (the same rule as Skills — but if you **rewrite a definition under an existing name**, you have to select the working folder again for it to be picked up).
- A malformed definition (missing required fields, a tool name it cannot use, and so on) is dropped from the catalog, and the reason is logged in one line on the server. If you had the AI create it, the AI tells you the reason right away — and also tells you when the name collides with an existing one.
- **With no definition files at all, everything behaves exactly as before** — the same model and thinking level as the chat, with the full tool set.

Things worth knowing

- By default **two delegations run at a time**; if you hand over more, they are not refused but **wait their turn** (up to 8 by default) and start as earlier ones finish. Only when the queue is full is a request refused. You can change both numbers under `options.delegate` in `ai-config.json` (`maxConcurrent` 1–8 · `maxQueued` 0–64 — match them to the model's throughput, e.g. 2 for a local LLM and 4 for a cloud one).

- A delegation is stopped automatically when it makes **no progress for 15 minutes** (a single long step such as running a command does not trigger this, and time spent waiting for your answer on an approval card does not count) or **2 hours after it starts** (time spent waiting in the queue does not count).
- When a worker reports completion, the AI **carries on with the next step without waiting for your input**. It continues on its own like this **up to 20 times in a row**; after that a notice appears in the chat, and it carries on once you send a message (any remaining reports go along with that message).
- A completion report carries **only facts** — what the worker found (with assumptions and unverified points marked separately) and where its deliverables are. Whether to apply them, hand the work back, or ask you is decided by the AI based on your original request.
- Ask "show me the delegation status" to check progress, or "stop the delegation" to end one. Files created up to that point remain in the work folder.
- If the worker tries to run a shell command, the usual **approval card** appears. When several workers ask at once, the cards appear one at a time (section 10).
- By default the worker uses **the same model and thinking level as your session** (an agent definition file, above, can change that) — on cloud models (Claude/ChatGPT) you are billed for both the conversation and the delegated work.
- **If the app or the server restarts**, delegations in flight stop and their completion reports are never delivered. Whatever was produced remains in the work folder for you to use directly (asking the AI to apply it will be refused in that case — delegate again, or copy the files yourself).

14. Connecting MCP Servers (Adding External Tools)

An MCP (Model Context Protocol) server is a **set of tools provided by another program** — a design tool, an in-house system, a file server, and so on. Connect one here and the AI calls those tools during a conversation just like its own.

Open the registration window with `Option > MCP Servers...`. The list of registered servers is on the left, and the settings for the selected server are on the right.

Registering a Server

1. Press **Add** to create an entry and type the server name in **Name**. Only letters, digits, `_` and `-` are allowed, and this name is prefixed to the tool names.
2. Choose the **transport**. Only the fields that transport needs are shown.
 - **stdio (local process)** — runs the server program on this PC. **Command** (`npx`, `node`, `python`, or the path to an executable), **Args** (separated by spaces; wrap an argument containing spaces in `"quotes"`), **Cwd** (working folder, optional), **Env** (environment

variables, one `"name": "value"` per line).

- **http (Streamable HTTP) · sse (legacy HTTP+SSE)** — connects by address to a server that is already running. **URL** (must start with `http://` or `https://`), **Headers** (request headers, one `"Authorization": "Bearer ..."` per line).
3. Clearing the **Enabled** checkbox stops the app from connecting to that server (the settings are kept).
 4. Turning on **Approve every call** makes an approval card appear each time the AI uses a tool from that server (see below).
 5. **Timeout** is the limit for a single tool call, in milliseconds. Left empty it is 120000 (2 minutes).
 6. **Delete** removes the selected entry from the list.
 7. Pressing **OK** saves and connects at once — no app or server restart is needed.

Checking the Connection

- The **Status** line shows the current state of the selected server — `Connected - N tools`, `Connecting`, `Error: ...`, `Disabled`, or `Not saved - press OK and the server will connect`.
- Once connected, the box below lists the **tools** the server offers (name and description).
- The **Test Connection** button tries the values you just typed once, without saving. On success it shows `Connection succeeded - N tools` and the tool list.

Servers That Require an Account Sign-In (OAuth)

Some http/sse servers ask for an **account sign-in** instead of a token header.

1. Check **Use OAuth sign-in** — the sign-in and sign-out buttons are only enabled while it is checked.
2. Press **Sign in**; a browser opens and takes you to that service's authorization screen. When you approve, the app takes over, connects automatically, and the status text changes to "Signed in".
3. The issued tokens are stored **encrypted** and are refreshed automatically when they expire. **Sign out** deletes the stored tokens.
4. If the connection test reports `Sign-in required`, do this sign-in first.

How It Looks in the Chat

- Tools from a connected server appear in the chat under the name `mcp__<server>__<tool>` — the `read_file` tool of a `files` server is `mcp__files__read_file`, for example. When the AI calls one, it shows up as a tool-execution card like any other tool.
- On a server with **Approve every call** turned on, an approval card showing `MCP <server> / <tool>` and the values being passed appears before the tool runs. Tools the server reports as read-only run without approval. If you check **"Auto-approve for this session"** on this card and

press Allow, MCP tool calls are no longer asked about for the rest of that session (connection); a new session asks again. This checkbox applies to MCP calls only and is separate from the shell-command auto-approval in section 10.

- Only tools from servers that **connected successfully** are offered to the AI. Tools of a disabled server, or one that failed to connect, simply do not exist as far as the AI is concerned.

Things Worth Knowing

- Registrations are saved in `<userData>/ai-mcp-config.json`, the same user-data folder as your session data (see section 8). The format is the same `mcpServers` map that Claude Code and Cursor use, so you can paste an example from another tool's instructions as-is, and editing the file directly takes effect as soon as you save it.
- Tokens you put in Headers are kept at the same level of protection as the API keys in the model settings.
- Attaching several servers with long tool descriptions adds to what is sent with every request, which makes responses slower and more expensive. Clear the **Enabled** checkbox on servers you are not using.
- The background worker in section 13 is not given MCP tools by default. To let a worker use one, list its `mcp__<server>__<tool>` name under `tools` in the agent definition file (see section 13).

15. Automatic Updates

Khairon checks for new builds on its own while it is running, and tells you inside the app when one is available.

- The app **opens first and checks 12 seconds later**, then checks again **every 4 hours**. There is no window that waits for the check at startup.
- When a new build is available, a **notice card appears in the lower-right corner** of the screen.
 - `새 버전 x.y.z 이 있습니다` ("version x.y.z is available") — a build with a new installer.
 - `화면 업데이트가 있습니다 (콘텐츠 N)` ("a screen update is available (content N)") — a build that changes only the screens.
- Press **[업데이트]** (Update) to start the download; the card shows the progress. An installer build is installed once downloaded and the app restarts; a screen build applies with a single restart.
- Press **[나중에]** (Later) to close the card; you are told again at the next check.
- If the download fails, the card shows `업데이트를 받지 못했습니다 · 네트워크를 확인한 뒤 다시 시도하세요` ("could not get the update — check your network and try again"). Press **[다시 시도]** (Retry) to download again, or **[닫기]** (Close) to dismiss it.

- An installation built for closed networks (with update checks turned off) does not check at all. To get updates from an in-house server, put in place the configuration file your administrator provides.
 - **Checking which build you have:** `Help > About` shows something like `Version : 1.5.2` and `Release : AS-20261014-54` ; the last number is the patch number. To see what changed in each build, open the **What's New** menu on the Khairon homepage.
-

Summary: Your First Run

1. Create a session with `File > New Session`
2. Select the working folder (cwd) at the bottom of the chat window → confirm trust
3. If needed, register and select the model to use in `Option > AI Model...` (API key or account OAuth sign-in)
4. Enter your request in the chat window → the AI automatically references code/samples when needed and calls tools when file changes or command execution are required (dangerous commands ask for approval). If you do not like the result, revert it with **Undo**.
5. Click a file path to view/edit it in the Monaco viewer (Git changes are shown in the gutter)
6. Ask "save this as a skill" to accumulate recurring work rules in `.aiassist/skills/` → they are used automatically from then on. Ask "remember this" for short facts and preferences and they accumulate in `.aiassist/memory.md` , applied automatically in later sessions as well.
7. Return to an earlier conversation from the session list and continue at any time