ChatGPT

Converting BDD Gherkin Feature Files to EARS Requirements

EARS (Easy Approach to Requirements Syntax) is a lightweight pattern-based notation for writing clear, consistent requirements. It uses fixed clause keywords (e.g. *While, When, Where, If…then*) in a prescribed order, following a simple temporal logic. This constrains natural-language requirements to be more precise and unambiguous [1] . For example, a generic EARS requirement has the form:

> *While* (precondition), *When* (trigger), *the* <*system*> shall <*system response*> [1] [2] .

EARS defines six requirement types (patterns), each with a specific clause or keyword. These include **Ubiquitous**, **Event-driven**, **State-driven**, **Optional Feature**, **Unwanted-Behavior**, and **Complex**. Each type roughly maps to parts of a system's behavior or conditions:

- **Ubiquitous (Always Active)**: No keyword. Always-on requirement of the system.
  Format: "The `<system>` shall `<system response>`."
  Example: "The mobile phone shall have a mass of less than XX grams." [3] .

- **Event-Driven**: Triggered by an event; uses *When*.
  Format: "When `<trigger>` occurs, the `<system>` shall `<response>`."
  Example: "When 'mute' is selected, the laptop shall suppress all audio output." [4] .

- **State-Driven**: Active in a specified state; uses *While*.
  Format: "While `<state>` is true, the `<system>` shall `<response>`."
  Example: "While there is no card in the ATM, the ATM shall display 'insert card to begin'." [5] .

- **Optional Feature**: Applies only if a feature is present; uses *Where*.
  Format: "Where `<feature>` is included, the `<system>` shall `<response>`."
  Example: "Where the car has a sunroof, the car shall have a sunroof control panel on the driver door." [6] .

- **Unwanted-Behavior**: Defines response to an undesirable condition; uses *If…then*.
  Format: "If `<undesired condition>` occurs, then the `<system>` shall `<response>`."
  Example: "If an invalid credit card number is entered, then the website shall display 'please re-enter credit card details'." [7] .

- **Complex**: Combines multiple EARS clauses (e.g. *While*…*When*, or *If*…*While*).
  Format: "While `<precondition>`, when `<trigger>` (or if `<condition>`), the `<system>` shall `<response>`."
  Example: "While the aircraft is on ground, when reverse thrust is commanded, the engine control system shall enable reverse thrust." [8] .

These patterns guarantee that each requirement has at most one trigger and one response, making statements clear and consistent [2] [1]. Complex behaviors can be handled by combining clauses (e.g. "While *..., when ... shall ...*"), but simple requirements are preferred where possible [8].

Table: EARS Patterns (source: Mavin [9] [6], Intel [10]):

Pattern Type	Clause / Keywords	Example (EARS sentence)
Ubiquitous	***** (none)	"The *system* shall <*action*>."
Event-driven	*When* ... (trigger)	"When `<trigger>`, the *system* shall <*response*>."
State-driven	*While* ... (state)	"While `<state>` is true, the *system* shall <*response*>."
Optional feature	*Where* ... (feature present)	"Where `<feature>` is included, the *system* shall <*response*>."
Unwanted-Behavior	*If ... then ...*	"If `<undesired condition>`, then the *system* shall <*response*>."
Complex	Combinations of above	"While `<state>`, when `<trigger>`, the *system* shall <*response*>."

Each pattern ensures a single, testable statement. Using EARS tends to reduce ambiguity and hidden preconditions [1] [2]. The strict order of clauses (precondition–trigger–system–response) mirrors natural English and enforces consistency in writing requirements [1].

Translating Gherkin Scenarios to EARS

BDD feature files (Gherkin) describe behavior with *Given/When/Then* steps [11]. To convert these into EARS requirements, follow these best practices:

- **Focus on *behavior*, not implementation details.** Write requirements that describe *what* the system does, not *how*. For example, Cucumber's guidance is: "Your scenarios should describe the intended *behaviour* of the system, not the implementation" [12]. So instead of concrete UI steps like "Given I enter username and password", prefer "When <user> logs in", which is the actual functional requirement [12]. This yields concise, declarative requirements.

- **Map Gherkin clauses to EARS clauses:**

- *Given* clauses (context/preconditions) become EARS preconditions. Use **While** if the context is an ongoing state, or **Where** if it's a feature-condition. E.g. "Given the user has a valid account" ⇒ "While the user has a valid account..." (State-driven).
- *When* clauses (actions/triggers) become EARS *When*. E.g. "When the user submits a login form" ⇒ "When the login form is submitted..." (Event-driven).
- *Then* clauses (outcomes) become the system response (after "shall"). Use "shall" in place of "should" or "will". For example, "Then the dashboard is displayed" ⇒ "...the system shall display the dashboard."

- *And* steps can often be merged or split. If a scenario has multiple *Then* steps, each can become a separate requirement, or combined if they describe one response. Aim for one EARS sentence per distinct response.

- **Use consistent phrasing and the word "shall."** Every EARS requirement uses an actor ("the system" or specific subsystem) followed by "shall" and a verb. Remove first-person or test-language phrasing. E.g. turn "the user sees 'Welcome'" into "the system shall display 'Welcome'." This aligns with requirement standards [2].

- **Avoid extraneous details.** Remove implementation specifics (GUI actions, button presses) that do not pertain to *what* the system must do. As the Cucumber guide notes, using implementation details ("Given I press the login button") makes scenarios brittle – better to describe behavior ("When the user logs in") [12]. Likewise, shorten or split long scenario steps into clear clauses.

- **Ensure one idea per requirement.** EARS favors one response per requirement. If a scenario describes multiple system outputs ("And I see X on the page"), consider making separate EARS statements for each. Otherwise combine with conjunction if truly one response.

Example translation:
Gherkin scenario:

```
Given the user has a valid account
When the user submits the login form
Then the user is redirected to the dashboard page
And a welcome message is shown
```

EARS requirements:
- "While the user has a valid account, when the login form is submitted, the system shall redirect the user to the dashboard page."
- "While the user has a valid account, when the login form is submitted, the system shall display the welcome message."

Each sentence follows an EARS pattern (state + event + response), replacing *Then* outcomes with "the system shall…". Notice we repeated the precondition for each response to keep sentences simple.

These steps convert acceptance criteria into requirement statements that are clear, grammatical, and unambiguous. By focusing on **what** happens (not how), and using consistent keywords (While, When, If, etc.), the resulting EARS requirements will be more precise and testable [12] [13].

Identifying Key Requirements and Prioritization

Not all extracted requirements will be equally important. To highlight *key (driving) requirements* from the Gherkin set, consider: - **Frequency and centrality:** Requirements that appear in multiple scenarios or in the main (Happy Path) scenario usually represent core functionality. For example, if many scenarios involve "login", then the login requirement is driving.

- **User-story goal alignment:** Link each EARS statement to the original user story or feature goal. If a requirement directly fulfills the story's acceptance, it's high-value; if it's a secondary path, it may be lower.
- **Coverage of success vs. error paths:** Primary success conditions often get higher priority than error handling (though errors are still needed).
- **Stakeholder input or regulatory needs:** If some scenarios reflect critical business or safety conditions (e.g. payment processing, security checks), those requirements should be high-priority.

Once important requirements are identified, assign a **priority score**. A simple method is a 3-point scale (High/Medium/Low) or 1/2/3 rating. Laura Brandenburg suggests giving each requirement a score 1 (high), 2 (medium), or 3 (low) [14] based on its importance to the project's goals. For example: - **High (1):** Essential functionality for core user goals (e.g. user authentication, main data processing).
- **Medium (2):** Important enhancements or alternate flows (e.g. user profile editing, optional filters).
- **Low (3):** Nice-to-have features or edge cases (e.g. non-critical reports, cosmetic settings).

Another common scheme is **MoSCoW** (Must/Should/Could/Won't) categorization [15], which similarly distinguishes mandatory features from optional ones.

Priority	Criteria / Examples
High (1)	Core user-facing requirements that fulfill main objectives (critical features present in most scenarios) [14].
Medium (2)	Supporting or secondary features (alternative or "nice-to-have" flows, appear in some scenarios).
Low (3)	Optional or edge-case behaviors (rarely needed, future enhancements).

This table (inspired by a 1-2-3 rating approach [14]) can be used to tag each EARS requirement. In practice, high-priority requirements get implemented and tested first. You may refine priorities by stakeholder review or by considering risk/complexity (for example, urgent bug fixes might boost a formerly "low" requirement to higher).

Automation and Pipeline Tips

You can automate much of this translation in a CLI pipeline. Some suggestions:

- **Parsing tools:** Write or use existing scripts to parse `.feature` files. For example, Unix tools (`grep`, `awk`) or a Python script can extract lines starting with `Given`, `When`, `Then`, then apply regex or string templates to form EARS sentences. For instance, a script might convert `^When (.*)` to `When \1, the system shall ...`.

- **LLM/Codex CLI assistance:** Modern AI coding assistants can automate natural-language tasks. Codex CLI is an OpenAI tool that "runs in a terminal" and can "read your codebase, make edits and run commands" [16] [17]. For example, you could prompt Codex to process feature files:

```
codex run "Convert feature/*.feature to EARS requirements"
```

The agent could read each scenario, identify Given/When/Then parts, and output corresponding "shall" statements. Because Codex can interact with files and execute code (with approval), you can integrate it into a CI pipeline. (For instance, in `auto` mode Codex uses GPT-5-Codex optimized for coding [17].) This approach requires review of the AI output to ensure accuracy.

- **Custom CLI tools:** You could also write a custom CLI program (in Python, Node, etc.) that accepts a feature file and prints EARS statements. This tool can be plugged into CI/CD (e.g. GitHub Actions). For example, a Node.js script might use the gherkin parser to load scenarios and then apply mapping rules to output `.req` files.

- **Preserve traceability:** If automating, include references back to the original feature (e.g. via comments or requirement IDs) so reviewers know which scenario gave rise to each requirement. This can aid maintenance and validation.

Regardless of tools, always **validate** the generated requirements. Automation can speed up the process, but a human should verify that the EARS statements correctly capture the intended behavior without omitting conditions.

References

This report is based on published EARS guidelines and best practices in BDD. Key sources include Alistair Mavin's EARS description [18] [8] and Intel's use of EARS patterns [10], along with recognized Gherkin writing advice [12] [13]. Practical priority methods are drawn from industry guidance [14] [15], and automation uses Codex CLI documentation [16] [17].

---

[1] [18] Adopting EARS Notation for Requirements Engineering - Jama Software
https://www.jamasoftware.com/requirements-management-guide/writing-requirements/adopting-the-ears-notation-to-improve-requirements-engineering/

[2] [3] [4] [5] [6] [7] [8] [9] Alistair Mavin EARS - Alistair Mavin
https://alistairmavin.com/ears/

[10] Microsoft PowerPoint - ICCGI 2013 EARS.pptx [Read-Only]
https://www.iaria.org/conferences2013/filesICCGI13/ICCGI_2013_Tutorial_Terzakis.pdf

[11] What is "Given - When - Then"? | Agile Alliance
https://agilealliance.org/glossary/given-when-then/

[12] [13] Writing better Gherkin | Cucumber
https://cucumber.io/docs/bdd/better-gherkin/

[14] Requirements Prioritization Made Simple
https://www.bridging-the-gap.com/requirements-prioritization/

[15] What Are Requirement Prioritization Techniques? - Techcanvass
https://techcanvass.com/blogs/requirement-prioritization-techniques.aspx?srsltid=AfmBOopypWLB7o3CB27LqLwMd3U7Te0jAQn_Dx9A1jhlWPWzb_2cnWLJ

16 17 Codex CLI

https://developers.openai.com/codex/cli/