Requirements Engineering

Oscar Chaparro

In Sprint 0, you implemented an issue someone else wrote.

What did you have to ask the author?

The agent did what the issue said

ISSUE

“Limit the length of budget category names.”

Open **enhancement**

No description provided.

What the agent built: input silently stops at 50 characters. No message. Existing longer names are cut on the next edit.

```
- <Input value={name} onChange={setName} />
+ <Input value={name} onChange={setName} maxLength ={50} />
```

What the author meant: a visible counter, a clear message at the limit, and existing names left alone.

With a human developer, ambiguity produces questions.

With an agent, ambiguity produces code.

Agenda

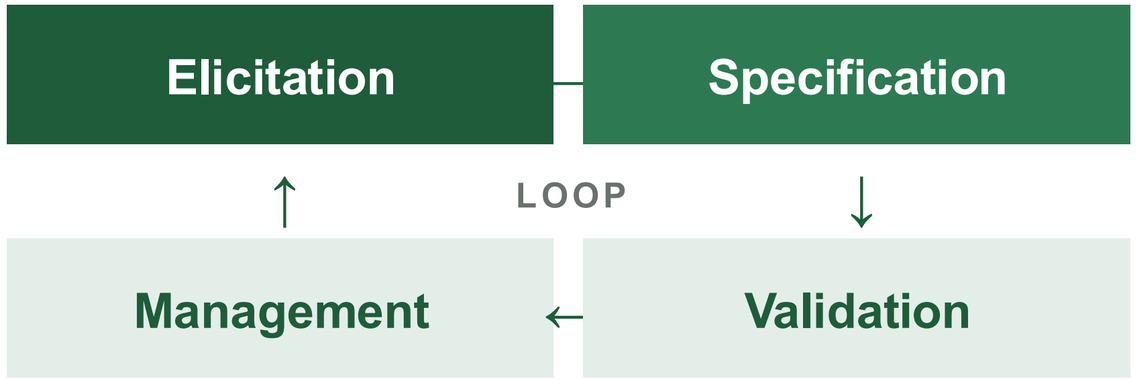
1. What a requirement is
2. Eliciting requirements from an existing system
3. Writing good specifications
4. Validating requirements
5. AI-assisted requirements work
6. When requirements fail: Therac-25
7. Traceability and change
8. Workshop: review your own D2 specs

What a requirement is

What vs. how, functional vs. non-functional

Requirement

- A **requirement** states what the system must do, or a quality it must have from a stakeholder's point of view.
- It says **what** and **why**, not **how**.
- **Requirements engineering (RE)** : discovering, writing, checking, and managing requirements.
- Four activities, repeated every sprint: a loop, not a phase.



Requirements engineering in your sprint

Elicitation

feeds D2

Learn current behavior (app, code); read the issue thread and upstream tracker; ask the author, maintainers, or instructor/TA (the customer). Record answers in the issue.

→

Specification

D2

D2: user story, scenario, acceptance criteria, out of scope, open questions

↓

Validation

D2 D4

teammate review of D2; PRs evaluated against the acceptance criteria (D4)

←

Management

D1

backlog refinement (D1), decomposing issues, instructor-added priorities

↑

Requirement vs. design/implementation

Requirement (D2)

"Users are told when a category name is too long."

"Clearing the canvas cannot delete work by accident."

→

one requirement → many possible designs

Design/implementation decision (D3)

"Add maxLength and a counter to InputCell."

"Show a confirm dialog using the existing Dialog component."

Test: a requirement can be met by many different implementations. If the statement already picks one, it's a design/implementation decision.

State the need, not the mechanism

Requirements state conditions stakeholders need. **Design decisions** state how the team meets them.

THE “HOW” AS THE REQUIREMENT

“Add a Redis cache to the search endpoint.”

- Passes as soon as Redis is added
- Search can still be slow — nothing fails
- No alternatives considered (an index?)
- New dependency nobody evaluated

THE NEED AS THE REQUIREMENT

“Search returns results in < 500 ms for 10k documents.”

- Any design must prove the speed
- The team compares options and records its choice in D3

If the “how” is in the requirement, the need is hidden: nobody can check it, and the implementer — human or agent — builds the mechanism, not the outcome.

Need or solution?

A	*“Users can find a document by a word in its title.”*
B	*“Use a modal to confirm clearing the canvas.”*
C	*“Admins can see which users haven’t logged in for 90 days.”*
D	*“Store all dates in UTC.”*

Need or solution? For each solution: what is the need behind it?

Need or solution? — answers

A	*“Users can find a document by a word in its title.”*	**Requirement**	Functional: says what users can do; how search works is left open.
B	*“Use a modal to confirm clearing the canvas.”*	**Solution**	Need: users don’t lose a drawing by accident. Undo might meet it better than a modal.
C	*“Admins can see which users haven’t logged in for 90 days.”*	**Requirement**	Functional: who, what, and a clear condition, with no implementation.
D	*“Store all dates in UTC.”*	**Solution**	A design/implementation decision (or a project constraint). Need: dates are correct in every time zone.

Functional requirements

Describe **behavior**: inputs, outputs, rules, states, what the user can do.

Gitea

When a repository has no releases, the Releases page shows a message and a link to create one.

Cal.diy

Event type titles longer than the limit are rejected with an error message.

Excalidraw

Clearing the canvas asks for confirmation before deleting elements.

Usually what a user story + acceptance criteria capture.

Business rules

Many requirements are **business rules**: policies of the domain the system must enforce.

Structural

Define domain concepts and the relationships between them.

“An account is checking, savings, or credit.”

“Every account belongs to exactly one user.”

Behavioral

Support business operations: what is allowed, and when.

“If the balance is below the minimum, the user cannot withdraw.”

“Withdrawals over the daily limit are rejected.”

In an existing system: structural rules live in the data schema and models; behavioral rules in validation and code logic.

Non-functional requirements

Quality attributes — **how well** the system does things.

ATTRIBUTE	MEASURABLE EXAMPLE
Performance	“Search returns results in < 500 ms for 10k issues.”
Security	“Only board admins can delete a board; others get 403.”
Usability / accessibility	“Every new control is reachable by keyboard and has an accessible name.”
Reliability	“A failed sync never loses local edits.”
Compatibility	“Budgets from the previous release open without migration errors.”
Maintainability	“New strings go through the i18n function.”

ISO/IEC 25010 quality model

Make it measurable

“The export should be fast.”

“The new page should be accessible.”

“The feature should be secure.”

“Don’t break existing users.”

Rewrite each so someone could check it pass/fail.

Make it measurable — answers

“The export should be fast.”	**Performance**	Exporting a 1,000-row report finishes in under 3 s on the demo dataset.
“The new page should be accessible.”	**Accessibility**	All new controls are keyboard-reachable and have accessible names; the *axe* tool reports 0 new violations.
“The feature should be secure.”	**Security**	Only users with write access see and can call “Delete”; others get no button and a 403.
“Don’t break existing users.”	**Compatibility**	Data created in the previous release opens without errors; existing tests pass unchanged.

A number or a concrete check makes it testable. Guessed numbers go in Open Questions.

Constraints

Limits on the solution that come from outside the feature.

- Tech stack and frameworks already in the project
- Backward compatibility with existing data and APIs
- Project conventions — CONTRIBUTING.md, linters, i18n, UI component library
- Licenses, dependency policy (“no new deps without discussion”)
- Deadlines — the sprint ends Oct 8

WHERE

CONTRIBUTING.md your D3 standards doc

AGENTS.md

Constraints limit the design, but stakeholders impose them, so they’re requirements.

Why NFRs and constraints get lost

WHY THEY GET LOST

- Nobody states them — “everybody knows.”
- Tests rarely check them.
- Agents optimize for what is stated and tested (L3: intent blindness).

WHERE THEY GO

Project-wide NFRs/constraints

→ CONTRIBUTING.md, AGENTS.md / CLAUDE.md, your standards doc

Issue-specific NFRs

→ an acceptance criterion in the issue

You already know the formats

User story L4

As a [role], I want [function], so that [value]

3 C's L4

Card, Conversation, Confirmation

INVEST L4

split or rewrite stories that fail several letters

Acceptance criteria L4, glossary

testable pass/fail conditions

Bugs L4

observed / expected behavior, steps to reproduce

TODAY ADDS

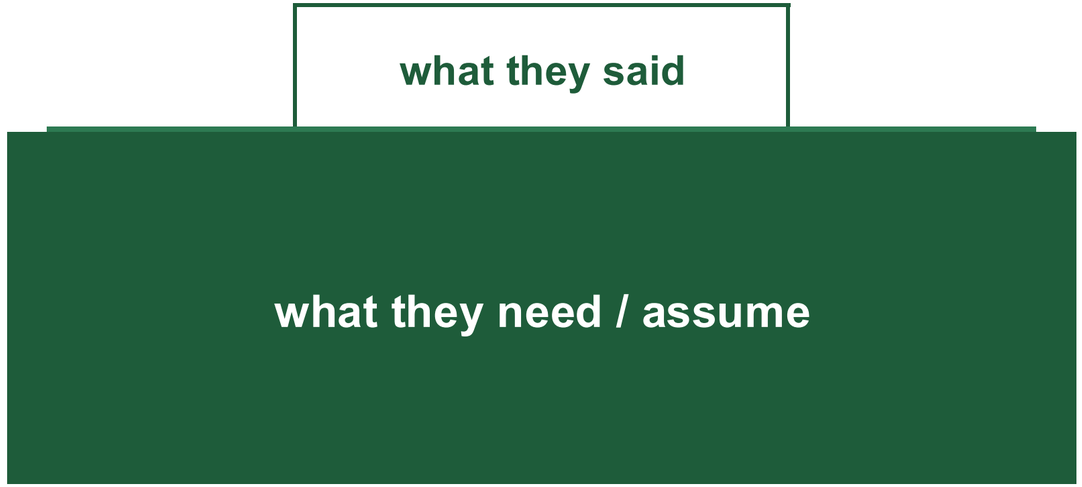
where requirements come from, and what makes them good.

Elicitation

Finding out what is actually needed — in a system that already exists

Elicitation is discovery

- Needs are **hidden** (tacit), **incomplete**, and **conflicting**.
- Users describe **solutions** and **symptoms** — your job is to find the need.
- Stakeholders assume context you don’t have.
- Ambiguity is a normal input, not an error.



Stakeholders in open source

End users
use the product

Community
issue threads, discussions, +1s

Self-hosters / operators
run it (Gitea, Discourse, Chatwoot…)

Upstream maintainers
decide what gets merged; set conventions

Your team
builds and maintains your fork

Instructor & TA
the “customer”: may add issues and set priorities

They often disagree: users want features, maintainers want small scope · users want change, self-hosters want stability · the customer’s priorities vs. your plan

Every issue is a change to a system

Example: **“Limit the length of budget category names.”**

CURRENT BEHAVIOR

Names accept any length.

Found in the app and code — not in the issue.

+

REQUESTED CHANGE

Stop at 50 characters and tell the user.

From the stakeholder, in the issue.

+

MUST NOT CHANGE

Existing longer names still display and can be edited.

Nobody asks for it; users rely on it.

The spec = acceptance criteria for the change + for what must be preserved.

Where to learn current behavior

SOURCE	WHAT IT TELLS YOU
Running app	current behavior: the de facto spec
Tests	executable requirements: each assertion states expected behavior
Code	validation rules, limits, permissions, edge-case handling
Issue & PR history	what was requested, rejected, and why
Docs, CONTRIBUTING, release notes	intended behavior and rules
Maintainers & discussions	priorities and tacit knowledge

From request to ready issue

1

Request

posted by a customer, support, or the PO — often a solution or a symptom

→

2

Triage

valid? duplicate? in scope?

→

3

Refinement

elicitation happens here: questions, answers, examples

→

4

Ready

story, AC, out of scope — answers recorded in the issue

OPEN SOURCE

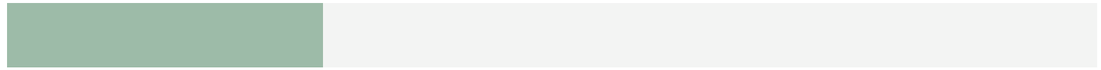
Mostly asynchronous: maintainers ask in the comments, label “needs info”, the requester replies.

COMPANY

Calls with the requester, backlog refinement meetings, quick demos.

The issue records elicitation. It doesn’t replace it.

Techniques used in practice

TECHNIQUE	SHARE OF PRACTITIONERS USING IT
Interviews — one-on-one conversation with a stakeholder	 73%
Meetings / workshops — facilitated group session, incl. backlog refinement	 67%
Prototyping — show a sketch, mockup, or demo; collect reactions	 58%
Scenarios — walk through a concrete story of a user doing a task	 41%
Observation — watch users do their real work, to see what they don’t say	 29%

NaPiRE survey, Wagner et al., ACM TOSEM 2019 (N = 228).

Interviews

One-on-one conversation with a stakeholder to learn their needs.

HOW TO RUN IT

1. Prepare 5–8 open questions; follow the conversation
2. Ask about past behavior: “Tell me about the last time you…”
3. Ask “why?” until you reach the need
4. Don’t pitch your solution
5. Read back what you heard; post it in the issue

EXAMPLE

Issue: **“Add budget alerts.”**

✗ “Would you like budget alerts?”

✓ **“Tell me about the last time you overspent. When did you notice?”**

PITFALL

“Would you…?” questions get polite yeses, not evidence.

Workshops and meetings

A facilitated group session to reach shared understanding and decisions.

HOW TO RUN IT

1. One goal, a timebox
2. 3–5 people who can answer (product, dev, test)
3. Separate facilitator and note-taker
4. Use a format, e.g. **Example Mapping**
5. End with decisions and open questions recorded

EXAMPLE

Example Mapping — **“Limit category name length”**

RULE	Max 50 characters
EXAMPLE	Paste 60 chars → cut to 50
EXAMPLE	Old 70-char name → still editable
QUESTION	Does the API enforce it too?

PITFALL

Without a facilitator, the loudest voice decides.
Without notes, decisions are lost.

Prototypes

Show a sketch, mockup, or demo and collect reactions.

HOW TO RUN IT

1. Pick the question it should answer
2. Build the lowest fidelity that answers it: sketch → mockup → branch demo
3. Give a task; let the user drive
4. Watch where they hesitate; ask why
5. Record what you learned as AC; throw it away

EXAMPLE

Two mockups for the category name:

A blocks typing at 50, with a counter **B** allows typing, shows an error on save

The user tries both, picks A, and asks: “What about my old long names?” → AC4

PITFALL

A working demo looks finished: its accidental choices become requirements nobody decided.

Scenarios

Walk through a concrete story of one user doing one task, step by step.

HOW TO RUN IT

1. Pick a real user, a goal, and concrete data
2. Tell the story one step at a time
3. At each step ask “what if…?”: empty, limit, error, other role
4. Write the main path and variants as Given / When / Then

EXAMPLE

“Ana renames *Groceries* to a 60-character name and presses Enter.”

What if she pastes it? What if the name already exists? What if she’s offline?

PITFALL

Walking only the happy path.

Observation

Watch users do their real work, in their own setting.

HOW TO RUN IT

1. Pick a real task and a real user
2. Watch in person or by screen-share; ask them to think aloud
3. Note workarounds, hesitations, repeated steps
4. Ask about them afterwards, not during
5. Record needs, not solutions

EXAMPLE

Watch a Gitea maintainer triage 20 open PRs.

They open each one to check the last activity date.

→ **Need: see stale PRs at a glance.**

PITFALL

People act differently when watched; one session is one user.

Asking good questions

- Ask about the **need** , not the solution — “What are you trying to do?”
- Ask for **examples and edge cases** — “What should happen if…?”
- Ask what is **out** — “Does this include X?”
- One specific question per comment; answers go **in the issue** , not DMs
- Who: issue author · upstream maintainers (their tracker) · instructor/TA (Zulip)

✗ **VAGUE**

“Can you clarify?”

✓ **SPECIFIC**

“When the name is exactly at the limit, should typing be blocked or should we show an error on save?”

What would you ask?

Sit with your team. Your issue:

Actual Budget *“Add budget alerts.”*	**Cal.diy** *“Add a waitlist for fully booked time slots.”*	**Excalidraw** *“Let users export only part of the drawing.”*
Gitea *“Notify authors when their PR goes stale.”*	**Medusa** *“Support discounts for loyal customers.”*	**Outline** *“Remind owners when documents get outdated.”*

Write your three best questions, before any acceptance criteria. Then share one per team.

Agents as elicitation assistants

✓ Good at

- Searching code/tests: “Does this already exist? Where?”
- Summarizing long issue threads
- Explaining current behavior
- Listing edge cases; drafting Open Questions

Keep human

- Answering the questions
- Talking to maintainers and users
- Deciding priority and scope

Rule: **the agent drafts the questions; stakeholders answer them.**

A prompt for elicitation

Read-only — do not change any files.

I want to add a character limit to budget category names.

1. Does any limit already exist (UI, API, database)? Show file and line.

2. Where else are category names displayed, imported, or truncated?

3. List edge cases and open questions I should ask the maintainers.

Do not propose an implementation yet.

Then open each file:line yourself.

Where agents fail at elicitation

- They retrieve **what was written, not what was left unsaid** (Lyu et al., 2026).
- They **fill gaps silently**, turning one interpretation into code before intent is settled.
- They can be wrong about what exists — verify every claim.
- No access to tacit knowledge, priorities, politics.

Lyu et al., “How Do Practitioners Build SE Agents?”, arXiv:2607.10856, 2026.

Specification

Writing requirements someone else can build and test

Qualities of a good requirement

QUALITY	MEANS	✗ BAD EXAMPLE
Unambiguous	one reading	“show a reasonable limit”
Complete	covers errors and edge cases	no word on existing data
Consistent	no contradictions	AC1 blocks typing, AC3 shows an error on save
Verifiable	pass/fail check exists	“feels responsive”
Feasible	can be built with the constraints	“works offline” in a server-only feature
Necessary & traceable	tied to a need/source	extras nobody asked for

After IEEE 830 / ISO/IEC/IEEE 29148.

Words that signal trouble

fast **easy** **intuitive** **user-friendly** **reasonable** **appropriate** **flexible**

support **handle** **manage** **improve**

and/or **etc.** **as needed** **if possible**

all **never** **always** *(really?)*

TBD in an acceptance criterion

Test: **would two developers — or two agents — build the same thing?**

Find the ambiguities

Limit the length of budget category names Open Actual Budget

“When creating or renaming a budget category, the name field accepts any number of characters without restriction. The app should prevent names that exceed a reasonable maximum and inform users when they reach the limit.”

Rewritten

Limit the length of budget category names Open enhancement

STORY

As a **budget user**, I want **to know when a category name is too long** , so that **names display fully in the budget table**.

ACCEPTANCE CRITERIA

- **AC 1** Typing stops at 50 characters; pasted text is cut to 50.
- **AC 2** A counter (e.g., “48/50”) appears once 40 characters are typed.
- **AC 3** Renaming follows the same rule as creating.
- **AC 4** Existing names over 50 still display and can be edited down.

OUT OF SCOPE

group names, notes, API-level validation.

OPEN QUESTIONS

Is 50 right? (ask maintainers) Should the API reject longer names?

Scenarios and edge cases

Scenario = one concrete path: **Given / When / Then** (L4).

Always ask about:

empty	**at the limit**	**just over**	**paste vs. type**
unicode / emoji	**long translations (i18n)**	**existing data**	**other roles and permissions**
errors and offline	**narrow / mobile screens**		

Use cases

User story

short; starts a conversation

Use case

a structured flow:

- actor
- precondition
- main flow
- alternative flows
- result

Use case: Clear canvas (Excalidraw)

Actor: user **Pre:** canvas has elements

1. User selects “Clear canvas”
2. System asks for confirmation
3. User confirms
4. System removes all elements
5. User can undo

3a. User cancels → nothing changes

Use when a feature has several steps or many alternatives. **Each alternative flow → a test.**

EARS: precise requirements

Ubiquitous always true	The <system> **shall** <response>. *“The category editor shall accept names of up to 50 characters.”*
Event triggered	**When** <trigger>, the <system> **shall** <response>. *“When the name reaches 50 characters, the editor shall show “Maximum 50 characters”.”*
State while in a state	**While** <state>, the <system> **shall** <response>. *“While offline, the app shall save edits locally and sync them when reconnected.”*
Unwanted errors, failures	**If** <condition>, **then** the <system> **shall** <response>. *“If a sync fails, then the app shall keep local edits and show a Retry option.”*
Optional feature present	**Where** <feature is included>, the <system> **shall** <response>. *“Where email is configured, Gitea shall email the author when their PR goes stale.”*

Optional tool: one requirement per sentence, with an explicit trigger and response.

Mavin et al., EARS, IEEE RE 2009 (Rolls-Royce).