HANDBOOK

Chainable Stories · Handbook

Follow a product commitment into executable structure.

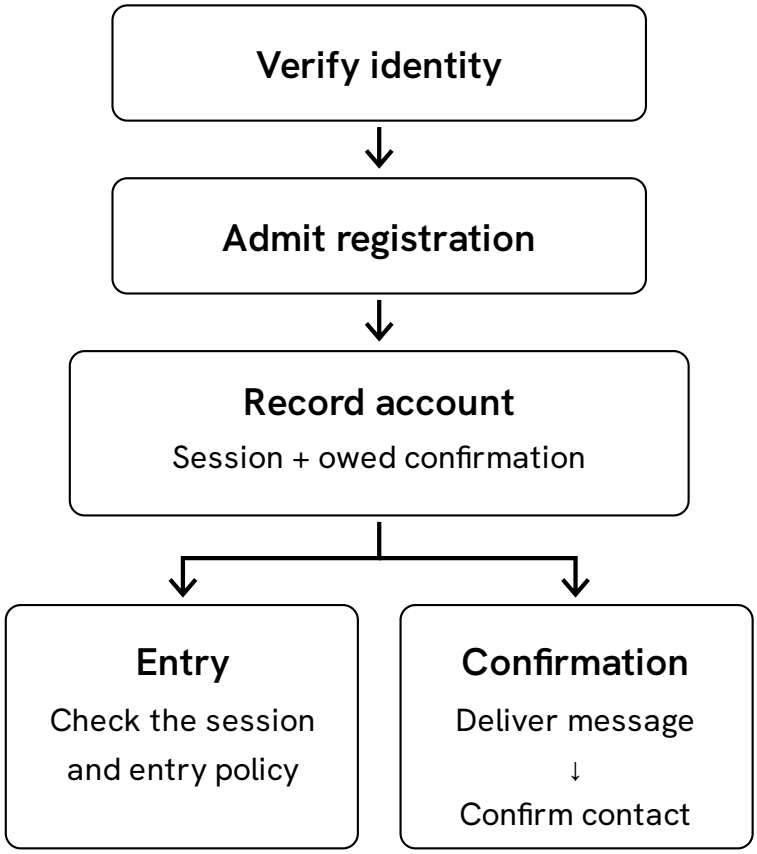
Contents

START HERE

Start with a Promise

A workshop makes three promises:

Registration	Entry	Follow-up
Sign in with Google; collect no password.	Allow entry while the session is valid, even before email confirmation.	Send a confirmation message.



Chainable Stories, also called **USaT**, connects a product promise to the Rust contracts that enforce it.

A change arrives	**What should change?**	**What should stay?**
Require confirmation before entry.	The entry rule.	Google identity and the owed message.
Replace the identity provider's network client.	How verification runs.	Who may register and enter.

> **Tip**
>
> Use those two questions throughout the course: **what changes, and what stays true?**

Read at your own pace. Predict an outcome before opening **Worked discussion**. Open **Engineering depth** for code. Choose a reading path, or continue to the next lesson.

Use private notes or your own notebook. The workshop is a fictional example; its policies belong to the example.

▼ **Engineering Depth**

A story is a **compile-time abstraction**. The compiler checks the selected relationships. Each registration still supplies real input and resources at runtime.

The code examples come from the complete Rust reference. Its small in-memory model makes the relationships visible.

Reflect

> A participant enters, then closes the browser. What does the workshop still owe?

▼ **Worked Discussion**

The confirmation message remains owed. Entry and delivery can proceed independently under this policy. The workshop must retain the pending work after the browser leaves.

START HERE

Choose Your Reading Depth

Start with the question you need to answer. Every path uses the same doctrine; you can move between them.

Perspective	Start here	Useful background
Product / delivery	Story → Telescoping → Policy changes	A product decision you understand
Quality / assurance	Same work → Outcomes → Lost responses	Expected behavior and evidence
Software engineering	Core lessons → Rust reference → Your design	Rust ownership, borrowing, traits and `Result`
AI agents	Field guide → relevant section IDs	The current task and its governing story

The subject is advanced software design. Product and SQA readers can examine promises and consequences without writing Rust.

For each lesson: make a prediction, reveal the discussion, then try a changed condition. Resume whenever ready. Export browser notes before moving devices.

CORE IDEAS

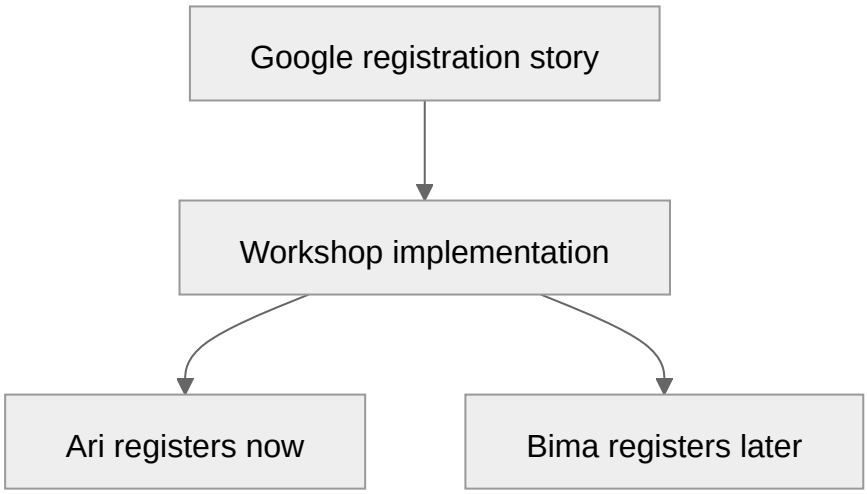
The Story and One Registration

A **story** expresses a product promise as a compile-time abstraction. Its code implements that promise; a particular registration supplies the actual data.

Term	In the workshop
Story	Google registration, no registration password, a selected entry policy, and owed confirmation.
Realization	The Rust code and provider that implement those choices.
Occurrence	This person's registration, using this provider and store now.

The compiler can reject an unsupported combination. The running program must check whether this person's identity and session are valid.

Scope stays specific: `WithoutPassword` describes registration. A separate login story may have its own password rules.



> ⚠ **Warning**
>
> “The type says Google” does not mean “Ari's identity is verified.” The provider must check Ari's actual assertion.

These two reference selections differ only in entry policy:

RUST

```
pub type OpenWorkshop<P> = Workshop<Google<P>, WithoutPassword, CookieValid>;
pub type ConfirmedWorkshop<P> = Workshop<Google<P>, WithoutPassword, ConfirmedOnly>;
```

▼ **Engineering Depth**

`P` names the provider's type. The workshop object holds the actual provider. `CookieValid` permits entry before confirmation; `ConfirmedOnly` waits for confirmation.

Both choices still require an active, unexpired session. See CS-01 and CS-04.

Reflect

> The type selects Google. What must happen before this person's registration proceeds?

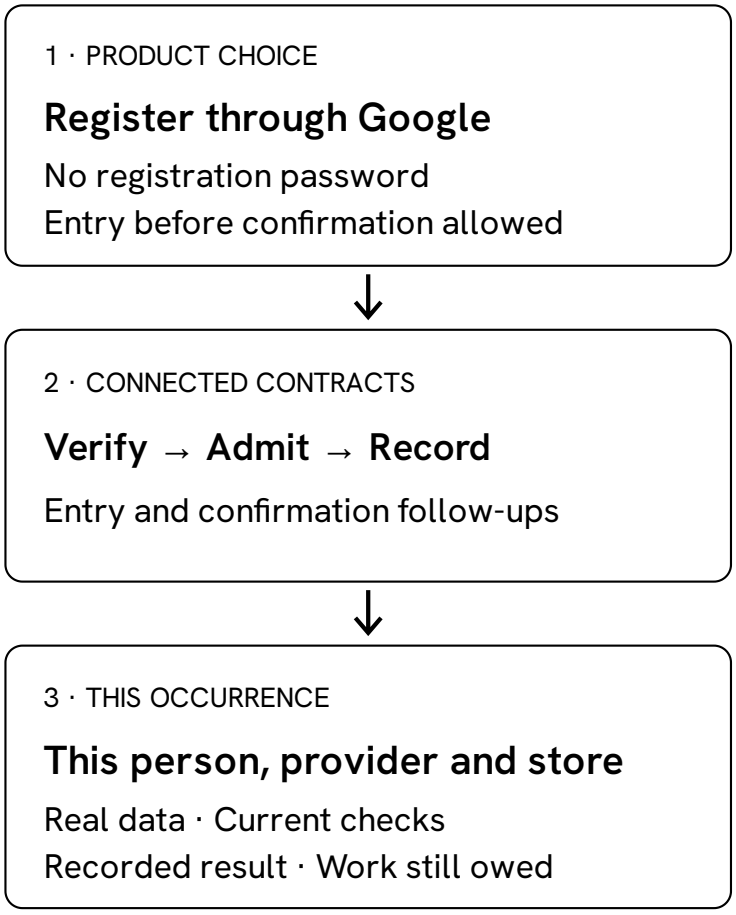
▼ **Worked Discussion**

The configured provider must verify the submitted assertion. Admission then checks the current records. Selecting a provider type alone establishes neither fact.

CORE IDEAS

Follow the Same Promise Inward

Telescoping lets a reader move from a product choice to the code that enforces it.



Reading depth	Question
Product choice	May someone enter before confirmation?
Contract	What must one step establish for the next?
Occurrence	Which provider, person and store does this work use?

A PM can review the first choice. SQA can trace its consequences. An engineer can follow the same names into code. Each view describes the same promise.

Surface	What the reader can learn
`run(mode, flags, payload)`	Little until the implementation is opened.
`OpenWorkshop → Signup → VerifyIdentity → AdmitRegistration`	The selected policy, first work and legitimate next steps.

> **Tip**
>
> Read the names first. Then open one contract to answer the next question. If the names promise more than the code enforces, fix that connection.

▼ **Engineering Depth**

Follow `Signup` in the root `WorkshopStories` contract to this successor:

```rust
pub trait VerifyIdentity: private::Stage + Sized {
    type Registered: RecordedRegistration;
    type Verified: AdmitRegistration<Registered = Self::Registered>;
    fn verify(self) → Result<Self::Verified, IdentityFailure>;
}
```

`Verified` must support `AdmitRegistration`. `verify(self)` consumes the submitted stage and returns either that successor or an identity failure.

The root also exposes visits, delivery and confirmation. Those continuations remain discoverable without making every step own the whole account lifecycle. CS-02.

Reflect

> Where would you look to learn what happens when the person returns tomorrow?

▼ **Worked Discussion**

Follow the root's `Visit` relationship to `EnterWorkshop`. It checks current session facts and applies the selected entry policy again.

CORE IDEAS

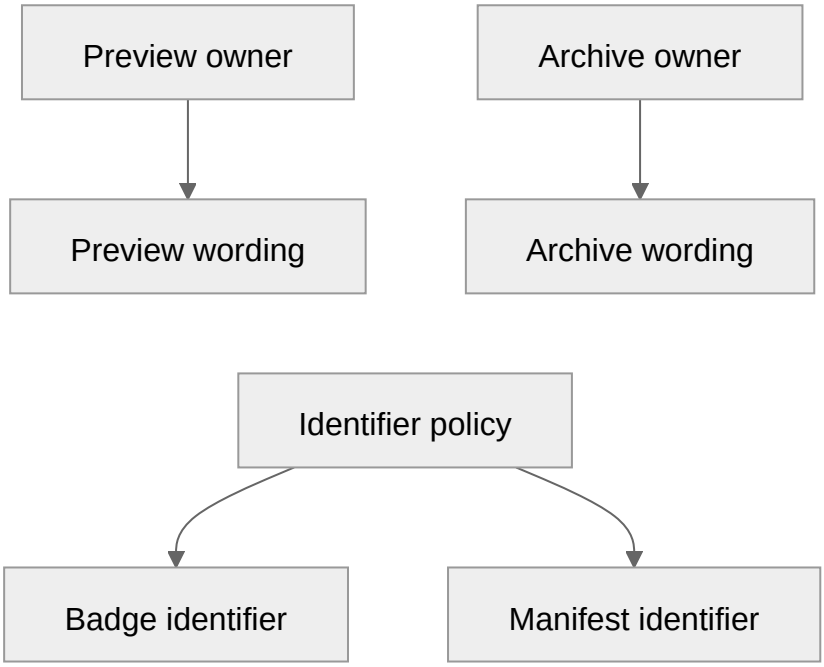
Similar Code, Different Reasons to Change

Preview and archive labels both say `Registration open`. The preview owner now wants an invitation; the archive must preserve its wording.

Elsewhere, badges and manifests share one organization-owned identifier rule. When that rule changes, both outputs must follow it.

Change	Keep together?	Why?
Preview wording	No	Preview and archive have separate owners and purposes.
Official identifier rule	Yes	Both consumers must follow the same policy.

Share the rule that is jointly owned. Keep unrelated presentation choices separate. Identical lines alone do not decide ownership.



▼ **Engineering Depth**

Separate functions can keep preview and archive independent. A shared policy contract can serve both identifier consumers.

For the wording change, two small functions are enough:

```rust
fn preview_label() -> &'static str { "Join the workshop" }
fn archive_label() -> &'static str { "Registration open" }
```

Compare the change's reach:

Design	Preview changes	Archive changes
Both call one `registration_label()`	Yes	Yes, accidentally
Separate wording functions	Yes	No

> **Important**
> This example separates two wording decisions. It is not a rule to duplicate an official identifier policy.

Before adding a mode flag to a helper, ask which product requirement needs that shared behavior. A current boundary may justify an abstraction with one implementation; hypothetical reuse does not. CS-05.

Reflect

> The badge layout changes. Must the identifier policy or manifest change too?

▼ Worked Discussion

Only if the requirement changes them. Sharing the identifier rule does not make the consumers' layouts jointly owned.

CORE IDEAS

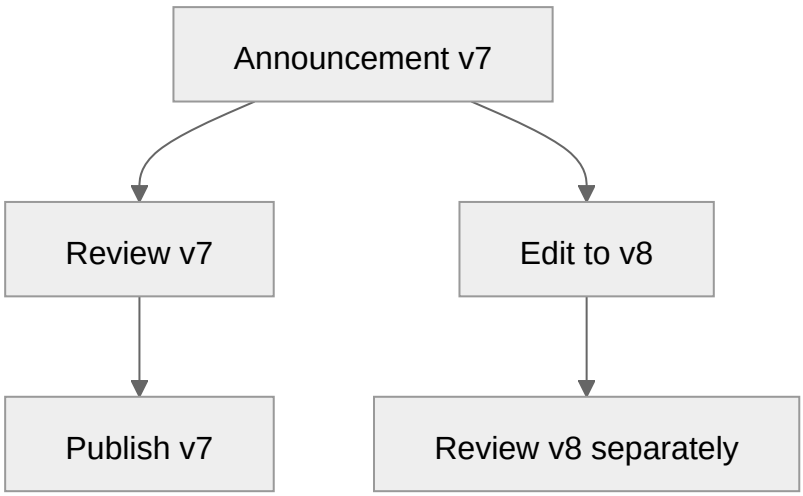
Continue the Same Work

A reviewer approves announcement version 7. Publication receives version 8 instead. Both are announcements, but only version 7 was reviewed.

The same problem occurs when two stores both contain record 7.

What matches?	What still needs checking?
Record type	Actual record and scope
Provider type	Configured provider instance
Proposal ID	Exact reviewed version
Department role	Actual person and current authority

Carry the work and resources the next step relies on. If a later boundary reloads them, check the binding there.



> ⚠ **Warning**
>
> A matching record number is not enough. Store A's record 7 and store B's record 7 are different work.

The reference carries real data inside its protected state:

```rust
pub struct Verified<'a, P, E> {
    workshop: &'a mut Workshop<Google<P>, WithoutPassword, E>,
    subject: String,
}
```

▼ **Engineering Depth**

The fields are private. The mutable borrow retains this workshop through the registration steps. Other code cannot swap its store or provider during that borrow.

Later handles check the workshop instance at runtime:

```rust
if !Arc::ptr_eq(&self.workshop.identity, &self.session.workshop) {
    return Err(EntryFailure::ForeignSession);
}
```

A zero-sized marker fits a data-free choice. This state needs both a subject and a resource borrow. CS-06, CS-07.

Reflect

> Can a later step accept new input without replacing the original work?

▼ **Worked Discussion**

Yes. A delivery observation can add evidence about the already-bound operation. It must identify that operation. Replacing the reviewed version is a different change.

CORE IDEAS

Say What Happened

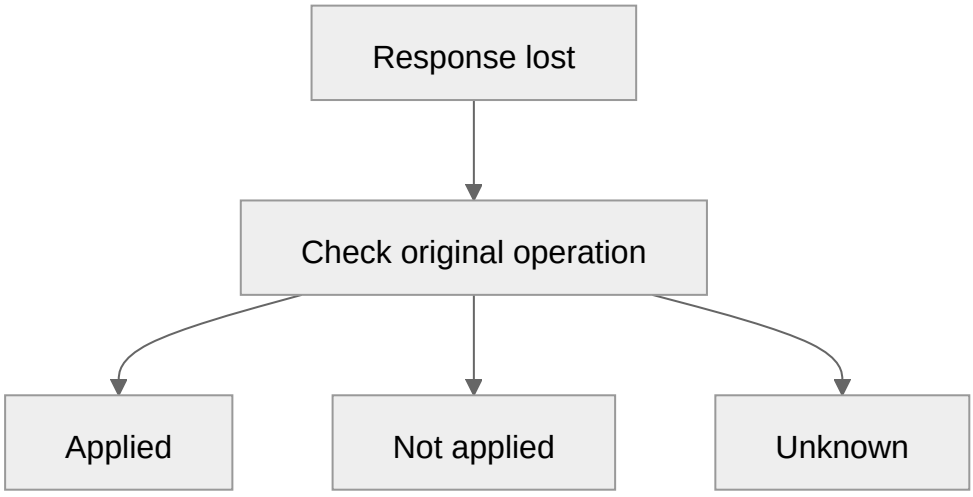
Registration is recorded, confirmation is queued, and the account screen fails to refresh.

Operation	Known result	Next step
Registration	Recorded	Keep that result.
Confirmation	Pending	Continue the owed delivery.
Account view	Unavailable	Refresh the view.

Calling all three “registration failed” would encourage duplicate work.

Misleading message	Useful message
“Registration failed. Try again.”	“Registered. Confirmation is pending. Reload your account view.”
“Send failed.” after a timeout	“Send outcome unknown. Check operation 42 before resending.”

A lost response after a remote send is different: the send may have succeeded. Keep the original operation ID and ask the source what happened before retrying unsafely.



> ⚠ **Warning**
>
> A second send is a second possible effect. A fresh request ID does not make that safe.

▼ **Engineering Depth**

The reference separates these identity outcomes:

```rust
pub enum IdentityFailure {
    Rejected,
    Unavailable,
    InvalidObservation,
}
```

A rejected assertion and an unavailable provider call for different responses. Neither records an account.

Represent the alternatives the operation really has. A pure label formatter can return its label directly. Remote delivery needs its own outcome and recovery contract. CS-08, CS-09.

Reflect

> A reviewer records “reject this proposal.” Did the review operation fail?

▼ **Worked Discussion**

The review completed with a rejection decision. Refusing an unauthorized person's attempt to review is a separate outcome.

CORE IDEAS

Change the Promise Deliberately

A developer replaces 300 lines of storage code. Another changes one entry rule.

Change	Decision to trace
New storage code	Does it preserve behavior, compatibility and pending work?
New entry rule	Who may enter now, and who owns that policy?
New engine	Where does old unfinished work remain?
Retired feature	What happens to existing consumers and obligations?

Review the consequence. Line count is a poor guide to authority.

Before	After	Same promise?
Look up the account in a vector.	Look it up in an index.	Yes, if the same facts and outcomes are preserved.
Require confirmation before entry.	Permit entry without confirmation.	No: a different group may enter.

> **Tip**
>
> Describe the changed behavior in one sentence before deciding which owner or review is needed.

▼ **Engineering Depth**

These reference implementations produce different entry behavior:

```rust
impl EntryRule for CookieValid {
    fn permits(_: bool) → bool {
        true
    }
}
impl EntryRule for ConfirmedOnly {
    fn permits(confirmed: bool) → bool {
        confirmed
    }
}
```

Selecting `ConfirmedOnly` changes a product rule. Replacing how an unchanged rule reads its records may be an implementation change.

If a compiler error exposes a forbidden combination, check the requirement before widening the bound. The project owns its checks and release cadence. CS-11, CS-12.

Reflect

> All tests pass after a bound is relaxed. What decision remains?

▼ **Worked Discussion**

Determine whether the requested change authorizes the newly admitted behavior. Tests can support that decision; they cannot supply the owner's mandate.

WHAT-IF CASES

Which Changes Belong Together?

Both pairs below share code today. Predict which change should affect both consumers.

Case	Requested change	Requirement
A: preview and archive wording	Make the preview more welcoming.	Preserve the archive's original wording.
B: badge and manifest identifiers	Change the official identifier format.	Both must follow the organization's one identifier policy.

▼ **Engineering Depth**

In A, separate leaves allow independent changes. An `is_archive` flag added only to save the helper can create unwanted coupling.

In B, a shared policy keeps both consumers aligned. Their surrounding layouts can still change independently. CS-05.

Reflect

> Name the owner and reason to change in each case. What fact would reverse your choice?

▼ **Worked Discussion**

A has independent wording decisions. B has shared policy authority. Similar code hides that difference.

An owner could later unify A's wording or split B's policies. That would change the agreement, so make the decision explicit.

WHAT-IF CASES

Require Confirmation Before Entry

The owner changes entry from `CookieValid` to `ConfirmedOnly`.

Fill the last column before opening the discussion. Every session below is active and unexpired.

Visit	CookieValid	ConfirmedOnly
Immediately after registration	Allowed	?
Return before confirming	Allowed	?
Return after confirming	Allowed	?

A separate change removes password collection during registration. Decide whether that also changes the separately owned login story.

> ⓘ **Note**
>
> Confirmation, session expiry and revocation are separate facts. Changing one does not erase the others.

▼ **Engineering Depth**

RUST

```
pub type OpenWorkshop<P> = Workshop<Google<P>, WithoutPassword, CookieValid>;
pub type ConfirmedWorkshop<P> = Workshop<Google<P>, WithoutPassword, ConfirmedOnly>;
```

The selected policy reaches the current entry check:

RUST

```
if !account.active {
    return Err(EntryFailure::Revoked);
}
if self.workshop.now ⩾ account.expires {
    return Err(EntryFailure::Expired);
}
if !E::permits(account.confirmed) {
    return Err(EntryFailure::ConfirmationRequired);
}
```

Initial and later visits use this same handler. Delivery remains independently available. CS-04, CS-08.

Reflect

> Fill the table. Which rule governs later password login?

▼ **Worked Discussion**

`ConfirmedOnly` yields **refused, refused, allowed**. An expired or revoked session is still refused after confirmation.

Removing registration password collection does not decide later login. Follow that story's own contract and owner.

WHAT-IF CASES

Two Stores Both Have Record 7

Case	Established fact	Attempted substitution
A	Work belongs to store A, record 7.	Send its handle to store B, record 7.
B	The reviewer approved version 7.	Publish version 8 under that approval.

The values have compatible types. Identify the missing relationship in each case.

▼ **Engineering Depth**

The reference checks the actual workshop instance:

RUST

```
if !Arc::ptr_eq(&self.workshop.identity, &self.session.workshop) {
    return Err(EntryFailure::ForeignSession);
}
```

A proposal story would also retain the approved version or digest. Check it when publishing, using the current stored record. CS-06.

Reflect

> What would you bind in the state? What would you recheck after a human delay?

▼ **Worked Discussion**

A needs the same store instance and record. B needs the exact reviewed content. A new source observation can be valid later input if it identifies the original work; a replacement proposal needs its own decision.

WHAT-IF CASES

Handle a Lost Response

Write a short user-facing result for each case.

Case	Facts
A	Registration was recorded. Confirmation is queued. The account view failed to load.
B	A remote source received a send request. The response was lost; its effect is unknown.
C	An authorized reviewer recorded a rejection. Another person was refused permission to review.

▼ **Engineering Depth**

For B, retain the original operation ID, input and source binding. A retry's correlation ID may change; it does not identify a new business intent by itself.

If the source can establish that the original effect did not occur, follow its retry contract. Otherwise preserve uncertainty. CS-09.

Reflect

> What is known, what is unknown, and what can safely happen next?

▼ **Worked Discussion**

Case	Result and next action
A	Registration recorded; confirmation pending; view unavailable. Retry the view.
B	Send outcome unknown. Reconcile the original operation before an unsafe resend.
C	First review completed with rejection. Second review request was refused.

Now replace a pure label formatter with a network lookup. The lookup story needs failure handling; the pure formatter can remain total.

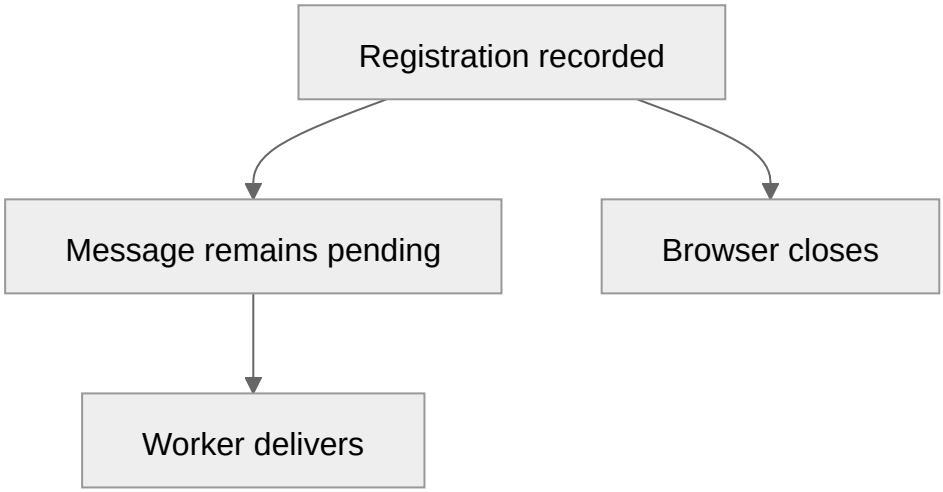
WHAT-IF CASES

Keep Track of Owed Work

An account has been recorded, and confirmation is still pending.

Interruption	What must remain owned?
Participant closes the browser	The owed message
Worker stops before delivery	The unfinished delivery
Process restarts	Any obligation promised to survive restart
New registration is retired	Existing pending work

Identify the owner and storage needed for each row.



> **Important**
>
> In-memory ownership can survive a dropped handle. Surviving a process restart requires storage that outlives that process.

▼ **Engineering Depth**

The reference keeps the obligation in its workshop object:

```rust
drop(work);
let delivery = shop.pending_confirmation().unwrap();
drop(delivery);
assert_eq!(shop.owed_messages(), 1);
```

Dropping either handle leaves one message owed. Destroying the workshop object loses its memory. A production story that promises restart recovery needs durable storage and source-safe recovery. CS-07, CS-09.

Reflect

> Which interruptions does the reference survive? What must a durable version add?

▼ **Worked Discussion**

It survives abandoned handles while the workshop object lives. It does not survive losing the process.

For retirement, decide how existing work will finish, migrate or be cancelled under proper authority. Removing the new-registration route does not settle those obligations.

WHAT-IF CASES

Change Engines and Workers

Case	Proposed shortcut
A: engine B now handles new work; engine A's old operation is unresolved	Ask the currently configured engine B what happened.
B: worker generation 8 took over; generation 7 returns late	Accept generation 7's result because it still owns a Rust value.
C: a private helper was replaced	Preserve its old three-call sequence because a test expects it.

Predict what each shortcut could break.

> ⚠ **Warning**
> “Current engine” describes new routing. It does not establish where an older operation happened.

▼ **Engineering Depth**

A recovery record must retain its source binding. A worker result may need a current claim or generation check before it can settle stored work. These distributed mechanisms lie beyond the in-memory reference.

For C, trace the assertion to the consumer's requirement. Keep required behavior and compatibility; retire obsolete private arrangements. CS-06, CS-11.

Reflect

> Which case changes authority? Which needs a decision about old work?

▼ Worked Discussion

A needs evidence from the original source or an authorized migration. B needs the current right to settle. C needs the actual consumer promise, which may never have required three calls.

APPLY IT

Design a Story of Your Own

Choose familiar work: approve a document, reserve equipment, publish an announcement or resolve a support request. Use invented data.

Prepare	Include
The promise	Actor, beneficiary, mandate, exact work, result and relevant follow-ups
Three views	Product choice → related contracts → actual resources
Two changes	One that preserves the promise; one that changes it
One interruption	What remains known or owed, and who handles it

Words and a diagram are enough for the product view. Engineers can add a small implementation.

For example, start with a document publication story:

Question	Example answer
What is promised?	Publish the exact version approved by the document owner.
Same-promise change?	Replace file storage while keeping versions and decisions bound.
Changed promise?	Permit a different role to approve.
Interruption?	A response is lost after publication; reconcile the original operation.

> **Tip**
>
> Start with one familiar promise. Add a continuation only when that promise owes it.

Before changing the design, predict what each change should affect. Return later and explain one prediction without looking at your first answer.

Bring the promise, diagram, changes and unresolved questions to your reviewer. Kresna judges the exercises in the guided learning setting.

▼ **Engineering Depth**

Show the meaningful composition and successors. Keep required data inside protected states, and identify facts that must be checked at runtime.

Explain how the design preserves the original work through the interruption. Use your project's normal checks. A different implementation can satisfy the same story.

Reflect

> What evidence would help your reviewer distinguish a different implementation from a weakened promise?

▼ **Worked Discussion**

Follow both changes back to the original promise. Point to the decision owner, the admitted work, current checks and remaining obligations. Explain any limit your design does not cross.

GO DEEPER

Read the Reference through Its Root

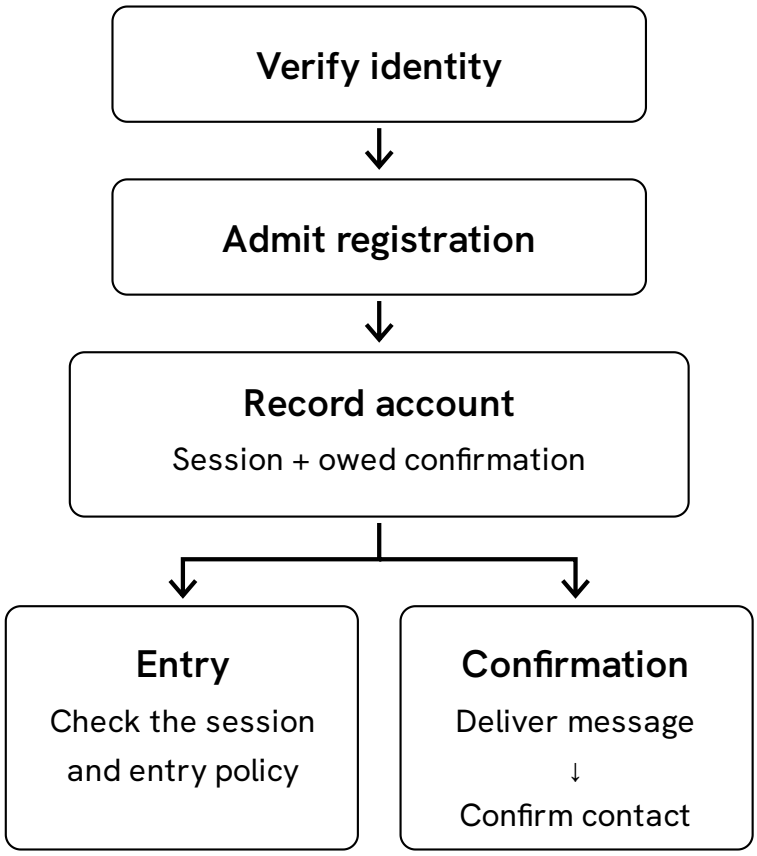
Download the complete Rust reference.

Start with the selected story, then follow its contracts into the data and operations.

RUST

```
pub type OpenWorkshop<P> = Workshop<Google<P>, WithoutPassword, CookieValid>;
pub type ConfirmedWorkshop<P> = Workshop<Google<P>, WithoutPassword, ConfirmedOnly>;
```

File	Read for
`src/contract.rs`	Root choices and successor relationships
`src/model.rs`	Private states, actual resources and effects
`tests/flows.rs`	Worked uses and observed policy differences



The example models one in-memory workshop. It demonstrates registration, both entry policies, confirmation and recovery of dropped handles.

> ⚠ **Warning**
>
> Identity checks and delivery are fixtures. This is not an authentication service: it provides no real OAuth, email delivery or recovery after process loss.

▼ **Engineering Depth**

This generic consumer follows the root contract:

```rust
fn register<S: WorkshopStories>(shop: &mut S, assertion: &str) → (S::Session, S::Confirmation) {
    shop.signup(assertion.into())
        .verify()
        .unwrap()
        .admit()
        .unwrap()
        .record()
        .into_followups()
}
```

Here, `unwrap()` belongs to a known-valid fixture. An application must handle the declared refusal and availability outcomes.

The two follow-ups are exposed together:

```rust
pub trait RecordedRegistration: private::Stage + Sized {
    type Session;
    type Confirmation;
    fn into_followups(self) → (Self::Session, Self::Confirmation);
}
```

The workshop keeps owed work independently of either returned handle. The entry policy is sealed; the identity adapter is an open trusted port. Both choices have specific purposes in this example.

This attempted shortcut is rejected by the compiler:

```rust
shop.signup("a".into()).record();
```

`Submitted` has no `record()` method. Verification and admission produce the required successor.

Reflect

> Find one compile-time relationship, one runtime check and one limit of this model.

▼ Worked Discussion

`Verified: AdmitRegistration` constrains the next capability. Entry still checks current expiry and revocation. Pending work survives dropped handles, but only while the in-memory workshop lives.

GO DEEPER

Words You Can Navigate by

Word	Meaning here
Commitment	Work the product actually owes within a named scope.
Story	The compile-time abstraction expressing that commitment and its relationships.
Realization	Code and selected collaborators implementing the story.
Occurrence	A particular piece of work bound to actual input, actors and resources.
Composition	Meaningful constituent commitments selected together under compatible relationships.
Successor	A legitimate continuation that consumes what earlier work established.
Telescoping	Following the same commitment from its readable selection into relevant detail.
Jurisdiction	The decisions a story owns and the boundary it deliberately respects.
Admission	Establishing that this actual occurrence may proceed under current facts.
Observation	What a source or check reports; its trust and scope still matter.
Obligation	Work still owed, whether or not a caller remains present.
Recovery	Re-establishing knowledge and safe progression for the original work.
Inertia	Deliberate resistance to changing a meaningful commitment.

The historical search term **USaT** leads to the full method: **Chainable + Composable Traits as Stories with Telescopic Organization**. States may carry data; see the example.

▼ **Engineering Depth**

Rust term	Reading aid
Trait	An executable contract for capabilities and relationships. A name alone proves nothing.
Bound	A condition a selected type must satisfy before certain construction or use is permitted.
Associated type	A type a contract names, such as the particular successor it produces.

Rust term	Reading aid
Generic parameter	A selected type; useful when it communicates a meaningful choice.
Lifetime/borrow	A relationship constraining how long actual data or a resource may be used.
Consuming method	Takes ownership of a value; a non-copyable value cannot then be reused.
Private field	Prevents ordinary external callers from constructing or changing a protected carrier directly.
Sealed trait	A contract external callers cannot freely implement. Intended adapter ports can remain open.
ZST	A zero-sized type, useful for a data-free distinction. It also implements Rust's `Sized`.
Enum / Result	A way to represent genuine runtime alternatives at their owning boundary.

GO DEEPER

AI Guide and Memory Exports

Download	Use
Agent start	Exact Commons entry and a short task handoff
Single Markdown file	Complete compact doctrine
Section records	Exact bodies, IDs, aliases, related IDs, edition and source hash
Index	Exact-ID and keyword lookup

In AMem, search `RUSTCSINDEX`, request `mode: "full"`, and follow the linked record IDs. Search aliases such as `RUSTCS07` keep the section identifier in one token; `CS-07` alone is unreliable in the current tokenizer. Offline, use the Markdown headings directly.

Start with `CS-01` through `CS-04`, then retrieve the sections governing the Rust task. Keep each rule with its qualifications. Follow related IDs when an excerpt seems absolute.

The section export preserves exact text and source hashes. The Commons index links the admitted records. Project-specific business rules and release workflows still belong to their owners.

When an edition changes, identify which records it replaces. Read the source when retrieval is ambiguous. The hash identifies the file; it does not establish approval.

All three downloads work offline. The guide includes a Mermaid relationship map for text-based readers and agents.

ABOUT THE MATERIALS

Learning Design: Sources and Limits

The lessons use contrasts, worked examples and short explanations. These sources informed those choices.

Source	Finding or recommendation	Use here
Schwartz & Bransford, 1998	College psychology students benefited from comparing cases before an explanation.	Predict a difference, then read the discussion.
Chi and colleagues, 1994	Prompted self-explanations helped pupils learning the circulatory system.	Explain why an outcome follows.
Mayer & Chandler, 2001	Learner-controlled segments improved transfer in two narrated-animation experiments.	Let readers pause and choose depth.
Kalyuga and colleagues, 2003	Their review describes guidance becoming redundant as expertise grows.	Make technical detail optional.
IES practice guide, 2007	Connect concrete and abstract examples; combine useful graphics with words; revisit learning.	Use one recurring example, diagrams, code and later variations.
Liu and colleagues, 2024	The tested language models used long contexts unevenly across positions.	Give agents exact section IDs and selective retrieval.

The first two papers' methods and discussion were examined. Mayer/Chandler and Kalyuga were used through their author/publisher abstracts. The IES recommendations have differing evidence ratings.

These studies concern other learners, subjects, media and models. Applying them here is a design judgment; they do not prove this course effective or establish a fixed human attention span. Human review should identify the particular misunderstanding so the lesson can improve.

ABOUT THE MATERIALS

Edition and Offline Copies

Chainable Stories (USaT), edition 0.2.1. The implementation guide concerns Rust code. The source package retains its full version identifier, `0.2.1-review`.

Copy	How to use it
Offline reader	Unzip, then open the top-level `index.html`.
PDF handbook	Read or print; discussions follow their prompts.
Complete handbook	Read every lesson in one view; find the PDF in Downloads.
AI guide and reference	Keep the Markdown, section records and Rust source together.

The interactive reader stores notes in your browser. Export them before changing devices or clearing browser data. In the PDF or without JavaScript, use your own notebook. External research links need internet access.

Exercises are self-paced. Bring your reasoning to your reviewer; the course does not award a score. Use your project's own release workflow when applying the method.