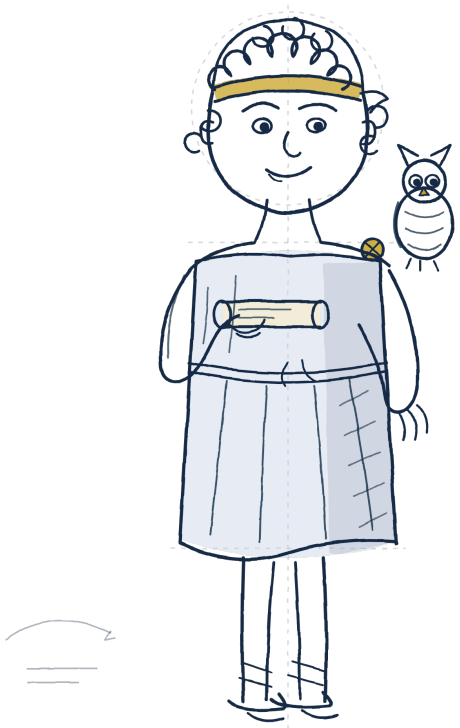
Telemachus for Developers

A small book about a Racket-first, self-hosted team AI platform



IoTone, Inc.

September 2026 — version 0.2.0

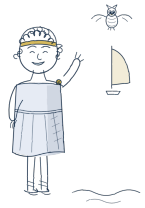
Telemachus, a concept sketch: a young Ithacan with a scroll, Mentor's owl on his shoulder.

How to read this

Telemachus is a self-hosted platform for a team's AI tools and applications. This book is for a developer who has cloned the repository and wants to understand how it is built before changing it. It is short on purpose: the generated reference under `docs/reference/` documents every tool, workflow, permission and HTTP route from the source of truth, and the design documents under `docs/design/` record why each subsystem is shaped as it is. This book is the connective tissue between them — the ideas that recur, the decisions that were expensive to learn, and the map you need before the code makes sense.

Each chapter ends with a short list of the files to open. Read the first four chapters in order; after that, go where your task takes you.

Contents



Chapter 1

What Telemachus is

1.1 The one-paragraph version

A team uploads its documents, talks to a model about them, runs workflows over them, shares the results with exactly the people who should see them, and does all of it on a machine the team controls. The platform is a single Racket program with an HTTP API, a browser console, an S3-compatible endpoint, a workflow engine, a scheduler, a quota ledger, and a plugin system. Every feature above the substrate — localization, the document pipeline, the knowledge graph — is built out of the substrate's own primitives, which is both the architecture and the proof that the architecture works.

1.2 Tenets

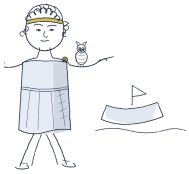
These are not slogans; each one has changed a design at least once.

- **Clean MIT, pure open source.** No open core, no held-back tier. Concepts may be reused from the earlier Python system, but only owner-authored Racket and owner-authored documents were carried over.
- **Racket first.** The reference implementation is Racket 9.2 CS. Non-Racket plugin authoring is explicitly not a requirement; where a data format is the contract (workflow specs, tool schemas), it is because the format earns its place, not because Racket is a barrier.
- **Deterministic dependencies.** Nix is the toolchain, and there is no second one. No Homebrew, no conda, no "python spice kitchen" on the deploy host. A PDF renderer is a dependency; so is a graph-visualization library; both were declined until someone needs them.
- **Contract first, backends swappable.** The durable product is the SDK contract, the HTTP API, the security model and the protocols. A second implementation would target those, not the Racket code. The generated reference exists so that the contract is written down by the code itself.
- **Self-hosted and privacy-first.** Bytes never leave the instance unless the operator points the model URL at something external. The blob store is content-addressed inside an org's namespace precisely so that two tenants can never learn about each other's files.

- **Refuse rather than flag.** A wrong result that looks finished is worse than a failed step. The localization drafter, the field extractor and the knowledge-graph extractor all refuse a model reply that does not validate. This shows up so often that it has its own chapter section.

1.3 What is where

```
docs/design/        design documents and the decision log
docs/reference/     GENERATED: tools, workflows, permissions, API, plugins, SDK
docs/ops/           operator runbooks
docs/integrators-guide.md  putting your own product on the platform
docs/book/          this book
refimpl/racketmaximus/
  server/main.rkt   the HTTP server: handlers, boot, plugin loading
  server/routes.rkt the declared route table
  domain/           the model: authz, repo, flow, sched, quota, i18n, kg, ...
  pkgs/             db-kit, web-kit, cli-kit: small libraries with no platform
    knowledge
  plugins/          shipped plugins: doc-indexer, doc-pipeline, knowledge-graph, rs3,
     ...
  cli/              telemachus-localize, telemachus-docs, telemachus-worker,
                    telemachus-secrets
  static/           the console (one HTML file) and its English strings
  locales/          the catalogs: en, ja, nl, es-419
  test/             unit suites, smoke scripts, the e2e gate, mock servers
```



Chapter 2

Getting it running

2.1 The toolchain

```
nix develop                          # from the repo root; pins Racket 9.2, exports
    PLTCOLLECTS
cd refimpl/racketmaximus
raco make server/main.rkt            # precompile; startup is slow otherwise
raco test test/*-tests.rkt           # the unit suite
racket server/main.rkt               # http://127.0.0.1:8835
```

Enter the shell from the repository root, not from inside `refimpl/racketmaximus`: the shell hook builds `PLTCOLLECTS` from the current directory, and entering it one level down doubles the path and breaks every `db-kit` import with "collection not found".

Never run `raco test test/*.rkt`. The glob includes `test/mock-*.rkt`, which are mock servers that block forever. The unit suite is `test/*-tests.rkt`.

2.2 Environment

```
DATABASE_URL=sqlite:///$PWD/data/telemachus.db   # or postgres://user:pass@host:port/
    db
TELEMACHUS_MODEL_URL=http://127.0.0.1:11434/v1/chat/completions   # any OpenAI-
    compatible endpoint
TELEMACHUS_MODEL=qwen2.5:7b
TELEMACHUS_HOME=login          # or ‘beta‘ to serve the onboarding funnel at /
TELEMACHUS_S3_PORT=8836        # turns on the S3 endpoint, a second listener
TELEMACHUS_MULTITENANT=1       # several companies on one instance
PORT=8835
```

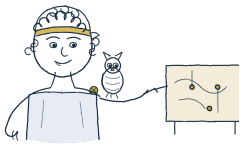
Without `TELEMACHUS_MODEL_URL` the platform runs a simulated model that echoes its input in upper case. Chat works, the smoke suites work, and nothing that needs real inference does. The tools that need a model — the localization drafter, the document pipeline, the knowledge-graph extractor — refuse to run rather than fail every validation with a misleading message. This is deliberate: an operator needs to read "no model configured", not "the reply was not JSON".

2.3 First run

`POST /api/bootstrap` with a username and password creates the operator, the first organization and the first team, and returns the operator's token. It works exactly once. After that, sign in at `/` or with `POST /api/login`.

2.4 Files to open

`CLAUDE.md` at the repository root is the working notebook: every gotcha that cost an afternoon is written there, by subsystem. `flake.nix` is the toolchain. `refimpl/racketmaximus/config.rkt` is the environment.



Chapter 3

A map of the code

3.1 Layers

The server is one module, `server/main.rkt`, and it is deliberately thin: it parses requests, resolves the principal, calls into `domain/`, and encodes the response. Everything that could be wrong about the product lives in `domain/`, which knows nothing about HTTP and is what the unit suites exercise directly.

Module	Owns
`domain/authz`	principals, roles, the permission catalog, `can?`, grants, tokens, audit
`domain/db`	migrations, applied at startup, dialect-neutral
`domain/repo`	the document repository: objects, versions, blobs, sharing, provenance, triggers, the pipeline tools
`domain/flow`	the workflow engine: spec validation, bindings, the run reducer
`domain/sched`	the job scheduler and the concurrency governor
`domain/quota`	the usage ledger and limits
`domain/agent`	the tool registry, the agent loop, the plugin loader
`domain/ai`	the model executor: one call, or a stream, or the simulated fallback
`domain/i18n`	catalogs, ICU formatting, the lint, the Localization Manager
`domain/kg`	the knowledge graph
`domain/apps`	search, translation — applications composed from the rest
`domain/orgs`	multi-tenancy: organizations above teams
`domain/beta`	the onboarding funnel

3.2 The three small libraries

`pkgs/` holds code that knows nothing about Telemachus and could be lifted out.

`db-kit/portable` is a drop-in for Racket's `db` that rewrites `?` placeholders to `$n` on PostgreSQL. Always require it instead of `db`. Forgetting is invisible on SQLite and fails on PostgreSQL with `syntax error at or near "AND"` — and test files issue raw SQL too, so the rule applies to them.

`web-kit` wraps Racket's `web-server` for the JSON control plane and adds `web-kit/http1`,

an HTTP/1.1 listener that treats request bodies as ports. The second exists because the S3 endpoint moves multi-gigabyte files and because `Expect: 100-continue` has to be answered lazily, on the first read of the body, so that a handler refusing before it reads never receives the bytes at all.

`cli-kit` is argument parsing for the two command-line tools.

3.3 One principal, one check

Every request resolves to a *principal*: a user id, a team id, an operator flag, an optional org role, and — for API tokens and S3 keys — a scope list that caps what the issuer could do. Every authorization decision in the platform is one function, `can?`, described in the next chapter. Handlers call `require-perm`; a refusal raises, and the server turns it into a localized 403.

3.4 Files to open

`server/main.rkt` (skim the requires and the `HANDLERS` table), `domain/authz/authz.rkt`, `pkgs/db-kit/portable.rkt`.

Chapter 4

The authorization model

4.1 Permissions and roles

A permission is a string `resource:action`. A role is a named set of them; the built-in team roles are owner, admin, member and viewer, and two org roles sit above them. Wildcards exist (`files:*`, `*:read`, `*:*`) but two tiers are unreachable from any team role whatever it grants: `instance:*` belongs to the operator alone, and `org:*` to a company's org role alone. Every permission carries a one-line description beside its declaration; a built-in role that grants an undescribed permission fails at load. The generated `permissions.md` shows the catalog and the role matrix.

4.2 `can?`, in order

1. **The org gate.** If the resource belongs to another organization, deny, unconditionally, before anything else is consulted. This is what makes multi-tenancy sound: no share, no grant, no scope can tunnel across companies, because the check that would honour them never runs.

2. **The tier.** An `instance:` permission needs the operator flag; an `org:` permission needs the org role.

3. **The role, or a grant.** The principal's team role must cover the permission — *or* the principal must hold a resource grant naming it on this resource. A grant is a delegation: sharing an editable document with a viewer means the viewer can edit that one document, and handing a member the `manage` capability means they can share it onward. Before the sharing work, a grant only widened reachability and could not confer a permission the role lacked, which made "share for editing" a lie.

4. **Owner-ok.** The owner of a resource holds every team-tier permission on it. A member who could not delete a colleague's document can delete their own.

5. **Scopes.** An API token or S3 key caps its issuer: the scope list must also cover the permission. Scopes narrow; they never widen.

6. **Reachability.** A private resource is reachable by its owner, the operator, or a grantee; a team-visible one by anyone in the team; a shared one by owner and grantees.

4.3 Capabilities

A person sharing a document does not pick permission strings. They pick a capability — view, edit or manage — and the platform writes the corresponding set of grant rows: `files:read`; plus `files:write`; plus `files:delete` and `files:manage`. Never a wildcard; a grant row says exactly what it says. Only manage can share onward. A grant may name a user or a team in the same organization, may carry an expiry (epoch seconds — `BIGINT`, because a 32-bit column overflowed on a year-2099 expiry in the PostgreSQL smoke), and an expired row is ignored, not deleted, because it is the audit trail of who was given what.

4.4 Multi-tenancy

Off by default. With the flag on, an *org* sits above teams, a superadmin runs the instance and an org admin runs one company. The org admin manages but does not read: no `documents:read`, no `chat:use`. A company admin who needs a team's data joins that team as a member, and the join is audited. The gate at step 0 never reads the feature flag, so turning the flag off on a populated instance hides the management planes and keeps enforcing.

4.5 Files to open

`domain/authz/permissions.rkt`, `domain/authz/authz.rkt` (read `can?` and `has-grant?`), `docs/design/rbac-and-teams.md`, `docs/design/document-sharing.md`, `test/authz-tests.rkt`, `test/tenancy-tests.rkt`.

Chapter 5

Data and migrations

Migrations live in `domain/db/migrations.rkt` as a list applied at startup. Every statement must be dialect-neutral: SQLite now, PostgreSQL as the second target, and the unit suite and both smoke suites honour a pre-set `DATABASE_URL` so that everything runs against either. Verify on PostgreSQL before calling anything done; this book's authors did, and it caught an int4 overflow, a missing placeholder rewrite and a timestamp format that broke every S3 listing.

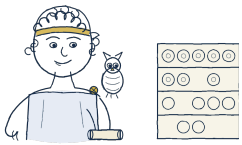
Some rules that came from those runs:

- Quote reserved words (`"window"`).
- Time windows are epoch integer columns, not timestamp arithmetic. `CURRENT_TIMESTAMP` has second resolution and its text form differs by dialect; anything that must order "newest first" within a second carries an epoch-millisecond column.
- Use `ON CONFLICT ... DO NOTHING` or `DO UPDATE` for upserts; both dialects support the syntax with a named conflict target.
- A `hasheq`'s iteration order is not stable across Racket processes. Never depend on it for anything that must be reproducible — JSON written to disk, the order of columns in a generated table.

Unit fixtures come from `test/db-fixture.rkt`: `(fresh-db)` gives a migrated, isolated database — in memory on SQLite, a private schema per fixture on PostgreSQL, because test files run concurrently.

5.1 Files to open

`domain/db/migrations.rkt`, `test/db-fixture.rkt`.



Chapter 6

The document repository

6.1 Objects, versions, blobs

A repository object is an ownable resource with the same three fields notes carry — team, owner, visibility — so `can?` governs it with no new authorization code. Writing the same key twice appends a version rather than replacing one; the object id is the stable thing a grant or a URL points at. Bytes live in a content-addressed store keyed by SHA-256, namespaced by *organization*: global deduplication across tenants would be an existence oracle. The store is never handed a principal — only a namespace and a digest — so a storage backend has no authorization to get wrong.

Uploads are a raw `PUT` body, never base64 in JSON. Every download is served as an attachment with `nosniff` and a denying CSP unless the type is on a short inline allowlist; SVG and HTML are never inline, because they share an origin with the console.

6.2 The S3 endpoint

A second listener speaks enough S3 that `aws`, `rclone`, `boto3` and Cyberduck work: SigV4 verified from the specification and pinned to AWS's published vectors, path-style buckets named by team slug, ranged GETs (mandatory — the CLI downloads large objects in parallel ranges), multipart uploads. Access keys are ordinary credentials with scopes; the secret is stored in the clear because SigV4 needs it to verify. Unimplemented sub-resources answer 501, never a silent success. Presigned links are signed with the caller's own newest key, so revoking the key kills the link.

6.3 Sharing and provenance

Sharing is the capability model of the previous chapter, over the grants table that already existed. Provenance is `repo_derivations`: when a workflow writes a document *from* another, the output takes the source's visibility and live grants at the moment of creation — private from its first byte if the source is — and a derivation row names the source version, the run and the step. After that the two documents are independent. "Where did this form come from" is a click.

6.4 Files to open

`domain/repo/repo.rkt`, `domain/repo/blobs.rkt`, `plugins/rs3/`, `domain/s3/sigv4.rkt`, `docs/design/doc`
`test/repo-tests.rkt`, `test/s3-smoke.sh`.

Chapter 7

Tools and the agent

7.1 Declaring a tool

```
(define-tool doc_text
  #:description "Return the text of a repository document."
  (object string #:description "The repository object id")
  (version string #:optional #:description "A specific version id"))

(register-tool! "doc_text" doc_text "files:read"
                (lambda (conn principal args) ...))
```

`define-tool` expands to the OpenAI-compatible function schema the model sees, so the declaration and the contract are one thing. The handler gets the database connection, the calling principal and the parsed arguments; it checks its own permission and meters its own AI spend. A result that is not a string is JSON-encoded at the agent boundary and kept as a value inside a workflow — a list-returning tool feeds a `map` step directly.

7.2 The registry

Built-in tools, plugin tools and workflow tools all land in one registry, tagged with their source. A team may disable any tool; the agent offers only enabled tools to the model and dispatch refuses disabled ones. Plugins run in-process with platform privileges — placing one in `plugins/` is the consent — and sandboxed out-of-process plugins exist as a separate, capability-scoped mechanism.

7.3 The model seam

Everything that calls a model goes through `run-chat` in `domain/ai/executor.rkt`, and the tools that need structured output take the model as a parameter (`current-doc-chat`). Unit tests script it; the HTTP smokes run `test/mock-llm.rkt`, a deterministic OpenAI-compatible server whose chat mode answers from a file of `needle: reply` pairs, longest needle winning. The same smoke runs against a live model when the model URL is set. This is how a pipeline that needs a model is tested in CI in seconds and verified against qwen2.5:7b on a developer's machine.

7.4 What a message may carry

A message's content is a string *or* a list of parts, the OpenAI-compatible `[{type: "text", text: ...}]` shape that gateways and multimodal servers send. One module decides this for every path — the agent loop, the chat endpoint, the stream and the pull wire — so they cannot disagree. Text parts are joined; content that carries parts and *no* text is a named error rather than a quiet empty string. That distinction is the whole point: an empty reply must never be manufactured, because it is indistinguishable from a model that said nothing. Content that is absent or empty is still legitimately empty — that is what a tool-call-only turn looks like — so the agent parsers raise only when the turn has no tool calls either. This was a silent bug for a year: `(if (string? c) c "")` read every reply, and a provider that answered in parts produced an empty message and no complaint.

A caller may also ask for a shape. `response_format` — `text`, `json_object` or `json_schema` — rides to the provider *and* is checked against the reply here, because most local servers ignore the field entirely and a schema request would otherwise come back as prose the caller parses as though it had conformed. Validation is the same subset validator, and the same `$.total: expected number, got string` refusal, as the document pipeline's extraction. The simulated no-model fallback refuses a format rather than passing an upper-cased echo off as conforming JSON. Streaming judges the format once the stream has ended and reports the verdict in its final event; raising after the answer has already gone out would be worse.

7.5 Files to open

`domain/tools/dsl.rkt`, `domain/agent/registry.rkt`, `domain/agent/plugins.rkt`, `domain/ai/executor.`
`docs/reference/tools.md`.

Chapter 8

Workflows

8.1 The spec is the contract

A workflow is a validated data document: a slug, typed inputs, and steps that use a tool, branch on a predicate, or fan out over a list. `define-workflow` is a macro that emits that document; a JSON document posted to the API goes through the same validator, and there is no second path. Unknown fields are rejected, including a newer spec version and a step kind this build lacks. The binding sublanguage is frozen — references and seven predicates, no arithmetic, no evaluation — and the escape hatch is "write a tool".

```
(define-workflow process-upload
  #:input ([object_id string] [schema object] [template string] [locales array])
  (step text     (tool doc_text #:object (in object_id)))
  (step fields   (tool doc_extract_fields #:text (out text result) #:schema (in
    schema)
                                          #:object (in object_id) #:run (run-id) #:
    step "fields")
                #:retry 1)
  (step has_form (choice (empty (in template)) #:then translate_source #:else form))
  (step form     (tool doc_render #:template (in template)
                                  #:data (out fields result fields) ...))
  (step translate_form (map #:over (in locales)
                            (tool doc_translate #:object (out form result object_id)
                                                #:locale (item) ...))
        #:end)
  ...)
```

8.2 Execution is a reducer

The engine keeps nothing in memory. `flow-advance!` reads a run from the database and enqueues its next step as a scheduler job; when the job finishes, the step's output is written and the cursor moves. Durability, cancellation, quota admission, per-team concurrency caps and the org gate are therefore inherited from the scheduler rather than reimplemented. A fan-out is one parent row and N child jobs; the last child to finish advances the cursor under a lock, or two children would each enqueue the next step.

Declared inputs are required and typed; a run that omits one is refused at start rather than resolving a binding to null three steps in. Plugin-contributed workflows are materialized into

a team's definitions on first lookup; that insert is idempotent because the console fires two lookups in parallel and the second once produced a 500 — found by the end-to-end gate, not by any smoke that looked things up one at a time.

8.3 Files to open

`domain/flow/spec.rkt` (normative), `domain/flow/dsl.rkt`, `domain/flow/bind.rkt`, `domain/flow/run.rkt`
`docs/design/workflow-engine.md`, `docs/reference/workflows.md`.

Chapter 9

The document pipeline

9.1 The first-user path

A team uploads a file and a workflow processes it: extract its text, pull structured fields against a JSON schema, fill a form template, translate the result. Four core tools compose into the shipped `process-upload` workflow; schema, template and locales are inputs, so one workflow serves invoices and intake forms with no new code. Every output is a repository document beside its source, with the source's grants and a provenance row.

9.2 Refuse, do not flag

The field extractor validates the model's reply against the schema and refuses a missing required field, a string where a number was asked for, and an invented key — an object schema with no `additionalProperties` is *closed*, a deliberate departure from JSON Schema's default. The template renderer refuses a placeholder the data cannot satisfy rather than rendering a blank: a form with an empty Total that looks finished is the failure the whole design exists to prevent. The step retries once, because a schema mismatch is the model's mistake and a second reading at temperature zero often conforms.

9.3 Templates

Markdown and HTML templates use `{{field}}`, dotted paths and `{{#each items}}` blocks. A DOCX template is a zip whose `word/document.xml` carries the same placeholders, and two things make that harder than it sounds: Word splits a placeholder across runs the moment the author pauses or the spell-checker looks at it, so every tag between a `{{` and its `}}` is dropped before rendering; and line items want a table row per item, so a row whose only text is `{{#each items}}` opens a block and a row that is only `{{/each}}` closes it. PDF rendering is deferred: every PDF renderer is a dependency, and a DOCX or HTML form is what a person edits anyway.

9.4 Triggers

A trigger is a team-scoped subscription: when a document matching this prefix and these content types lands, run that workflow with it. Triggers are evaluated in exactly one place — a hook at the end of `repo-put!`, after the version commits — which covers the console upload, the text-document shim and an S3 `PUT` by construction. A match enqueues a run as the *uploader*, scopes and all; a run can only read what its uploader could read, and its outputs are owned by someone real. Fires are exactly once per version. A pipeline's own outputs never re-fire the trigger that produced them, and an opted-in trigger never fires on the output of a run it started itself; without that rule an opted-in trigger ran itself fifty times in a test before the drain limit stopped it. An S3 key issued with the default `files:*` scopes cannot start a workflow — the trigger's history says so plainly — and needs `workflows:run` for a prefix meant to fire.

9.5 Files to open

`domain/repo/doc-tools.rkt`, `domain/tools/jsonschema.rkt`, `domain/repo/triggers.rkt`, `plugins/doc-pipeline/main.rkt`, `docs/design/document-workflows.md`, `test/doc-pipeline-tests.rkt` `test/doc-triggers-tests.rkt`, `test/doc-pipeline-smoke.sh`.

Chapter 10

Localization

10.1 Three homes

Server refusals live in `surface/messages.rkt`; the console's chrome in `static/ui-strings.json`; the generated documentation's prose in `docs/reference/strings.json`. All three are *surfaces* for one extractor, which folds them into `locales/en.json`, the base catalog. Translations live beside it, each string pinned to the hash of the English it was made against: editing the English makes the translation stale automatically, in every language, with nothing rewriting a status. The funnel's operator-authored copy is the exception — an overlay on the experience document, not a catalog — because it is content, not code.

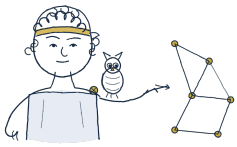
Resolution is the request's locale header, then the instance default. An unknown locale falls back to the instance default, not to English; a Japanese-default instance is possible.

10.2 The Localization Manager

The flagship: the platform building a real tool out of its own primitives. Catalogs on disk stay the shipping artifact; two tables are the workflow around them. *Missing* and *stale* are derived, never stored. A translator cannot approve their own string. AI drafting is a queue of scheduler jobs, twenty strings each, quota-metered — and a bulk draft stops when the team is over budget, which looks like a hang and is the platform working. A draft that loses, invents or mangles a placeholder is refused; the check counts braces, because the formatter is lenient by design and accepts exactly the drafts that need refusing. Dutch and Latin American Spanish were produced with this tool, every string reviewed, and pass the tool's own gate.

10.3 Files to open

`domain/i18n/`, `cli/telemachus-localize.rkt`, `locales/`, `docs/design/localization.md`.



Chapter 11

The knowledge graph

Entities and the relations between them, extracted from the team's documents, every fact traceable to the document and version that asserted it. Nothing enters the graph without a source, and that one rule makes the rest fall out.

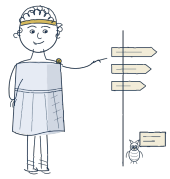
Visibility is inherited, never assigned. A fact is visible to a principal if at least one of its mentions is on a document they can read; mentions are filtered per row, so a private memo's snippet stays private even when the team can see the fact through a report. An entity a caller can see no mention of does not exist for them, and the API does not distinguish that from a missing id.

Validation is the provenance. The extractor asks the model for a strict shape and refuses a snippet that is not a verbatim substring of the text, a relation to an entity not in the same reply, or a nameless entity. A new version of a document supersedes its old mentions; deleting a document forgets its facts; an entity or relation with no mention left is pruned.

Deliberately narrow. Deduplication is by type and normalized name; cross-type merging is where knowledge graphs go to die. There is no graph query language — two hops from a named entity covers the questions people have — and no visualization library: the tab is lists with citations, and the agent's `kg_query` tool answers in text with the documents that said so.

11.1 Files to open

`domain/kg/kg.rkt`, `domain/kg/kg-tools.rkt`, `plugins/knowledge-graph/main.rkt`, `docs/design/knowle`
`test/kg-tests.rkt`.



Chapter 12

The route table and the generated reference

12.1 Declared, not enacted

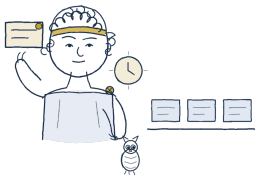
Every HTTP route is one entry in `server/routes.rkt`: method, path pattern, handler key, how it authenticates, the permission it enforces, the feature flag it sits behind, and one line of documentation. The server binds keys to handlers in one table and refuses to boot if either side names something the other lacks. An endpoint cannot exist undocumented, and a documented endpoint cannot fail to exist. Public-ness is declared rather than a comment beside a `cond` clause, which is what lets a test assert it.

12.2 Generated and gated

`telemachus-docs describe` evaluates the tool registry, the plugin loader, the workflow specs, the permission catalog and the route table into one JSON model; `render` writes `docs/reference/`; `check` regenerates and fails on a byte of difference. The output is committed, so a pull request that changes a tool's parameters shows the documentation change in the same diff. Every description is also a `doc.*` message in the catalogs, translated by the same Manager as everything else.

```
racket cli/telemachus-docs.rkt render      # regenerate docs/reference/
racket cli/telemachus-docs.rkt check       # the CI gate
racket cli/telemachus-docs.rkt render --locale ja
```

Adding an endpoint is therefore: one entry in the table, one handler in the map, one render, one commit. Forgetting the render fails CI; forgetting either half refuses to boot.



Chapter 13

The job queue

13.1 What a job is

A job is a row: a team, a user, a kind, a JSON payload, a priority, a status. Nothing about a running job lives in memory, which is why a restart loses no work and why a workflow step, a bulk translation and a model call routed to another machine can all be the same thing. A bounded pool of worker threads drains the queue; each thread claims one job, runs its kind's handler, and writes the result back to the row.

Two policies are applied at claim time rather than inside the handler, so no kind can forget them: a **per-team concurrency cap**, which stops one team's batch monopolizing the pool, and **quota admission**, which leaves an over-budget team's jobs queued rather than failing them. A queue that has stopped moving with nothing running is usually a team out of AI budget, and that is the platform working.

13.2 Claiming one

Claiming looks trivial and is not. The pool scans for queued rows, picks a candidate, and flips it to running — but the flip must be *conditional* on the row still being queued, and the condition is worthless unless somebody reads the result:

```
UPDATE jobs SET status='running', started_at=CURRENT_TIMESTAMP,
               lease_until=?, executor_id=?, claim_token=?
 WHERE id = ? AND status='queued'
 RETURNING id
```

The claim is believed only when a row comes back. This matters more than it looks: the driver reports affected rows on PostgreSQL and not on SQLite, so a row count would have been a dialect-dependent answer, while `RETURNING` works on both. A claimant whose update matched nothing moves to the next candidate instead of running a job another worker holds. Before this, exclusion rested on a lock inside one process — fine until there are two.

13.3 When a worker dies

Every claim carries a **lease**: a timestamp the holder refreshes while it works. If the holder dies, the lease lapses, and a reaper returns the job to the queue with its attempt count raised,

failing it for good after three. This applies to the in-process pool exactly as it does to a remote worker; when it did not, a pool job whose process died sat `running` forever and kept consuming its team's concurrency slot. A job left behind by an older build — running, with no lease — is swept back to the queue when the pool starts.

Recovery creates a subtler problem, and it is the one worth remembering. Suppose a lease lapses while the original run is still alive; the reaper requeues the job; the *same* holder claims it again. The first run finally finishes and writes its result. Status matches. Holder matches. Nothing in either check can tell the two attempts apart. So each claim also mints a **fencing token**, and every later write about that attempt carries it — the terminal write, the lease refresh, and a remote worker's heartbeat, completion and failure. Requeueing clears the column; so does every terminal write. A token from a previous attempt can never match again, and neither can a second write from the current one.

```
claim  -> id + lease_until + claim_token
heartbeat / complete / fail  -> must carry that claim_token, or 409
```

13.4 Executors that come to the work

An inference host does not have to be callable from the server. A **pull executor** runs a loop that long-polls for a job it can run, heartbeats while it runs it, and posts the result — so it works from behind NAT or on a tailnet, connecting outward only. Its credential is an API token whose only scope is `jobs:execute`, shown once and bound to the executor row; the executor is resolved through the token, never through a field in the request.

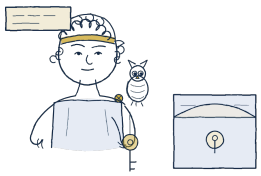
A **remote kind** is registered with no in-process handler: the pool skips it, only a worker can claim it, and its results are validated by a function the kind must supply, because there is no local handler to be strict on its behalf. The shipped one is `infer.chat` — exactly the wire `run-chat` speaks — so a synchronous model call becomes a sub-job and waits for a worker, and every model-using tool works unchanged when its chat is routed elsewhere. A sub-job carries its parent and is exempt from the team cap at claim: two parents at a cap of two would otherwise each wait forever for a sub-job nothing could claim.

13.5 Which model runs what

A *model role* resolves to an executor name: the team's setting, then an environment variable, then the local model. One role ships — `utility` — and it carries the bulk work: knowledge-graph extraction, the pipeline's field extraction and translation, and the Localization Manager's drafts. A person's chat never goes there. The tools set the role around their own model call, so a GPU box can take the batch work without any tool knowing it exists. Setting a role to an executor that does not exist is refused, because a typo would otherwise send every extraction quietly back to the local model.

13.6 Files to open

`domain/sched/scheduler.rkt` (read `claim-job!`, `finish!` and `with-lease`), `domain/exec/pull.rkt`, `domain/ai/roles.rkt`, `cli/telemachus-worker.rkt`, `docs/design/pull-executors.md`, `test/pull-tests` `test/pull-smoke.sh`.



Chapter 14

Secrets and the front door

14.1 What is hashed, and what cannot be

Passwords and API tokens are stored as hashes — a token as a peppered HMAC-SHA-256 with the pepper in the environment, so a database dump without the process's environment cannot be replayed. Rotating the pepper signs everyone out at once, which is a fine emergency action and the reason it is rotatable. Tokens also expire now: a session after thirty days, a console-issued API token after ninety, and a row with no expiry is still "never", so an upgrade signs nobody out.

Three secrets cannot be hashed, because they are not compared — they are *used*. A TOTP seed generates the code; an S3 secret keys the HMAC that verifies a signature; a push executor's key signs its requests. All three used to sit in the clear, which made one dump — a backup, a replica, a `pg_dump` pasted into a ticket, a stolen volume snapshot — enough to replay every one of them.

14.2 Sealing the rest

They are sealed with AES-256-GCM, keyed from the environment and never from the database. The claim is deliberately narrow, and worth stating precisely because the earlier note in the code got it half right: this defends a **dump** and does nothing for a **host**. An attacker running as the server reads the key out of its own environment. But a dump travels without that environment, and a dump is the thing that ends up somewhere it should not.

```
enc:v1:<key-id>:<nonce>:<ciphertext+tag>
```

Anything without that prefix is plaintext and is read as-is, so adopting a key is not a migration and needs no downtime: rows seal as they are rewritten, and a CLI seals the rest. The same CLI rotates — a previous key reads, the current key writes — and it is deliberately not something the server does at boot, because a mistyped key at startup would seal every row with a key nobody has. Running with no key at all remains supported; that is the "the database is the trust boundary" position, and the only change is that it is now visible in the boot line and in the admin status rather than inherited silently.

Two details carry weight. Each value is sealed with its own column and row id as additional authenticated data, so a ciphertext lifted out of one row and pasted into another does not open — someone with write access cannot move a known secret onto another principal. And a sealed

value that will not open *raises*: it must never degrade to an empty string, because an empty TOTP seed would silently disable somebody's second factor.

14.3 The second factor, and the way back in

A password can be changed and an access key re-issued. A TOTP seed cannot be rotated by using it, which is what made a leaked one permanent — so it is revocable: a user can turn their own off, and an operator can revoke anyone's, both audited. That covers the leak. It does not cover the lost phone, so enrolment also hands out ten single-use **recovery codes**, and a code is accepted anywhere a TOTP code is. Unlike the seed, a code *is* compared, so it is hashed like a token.

The small decisions are where the value is: matching forgives case and dashes, and the alphabet drops the characters people confuse, because these are read aloud and typed off paper; spending a code is the same statement that matches it, so two simultaneous sign-ins cannot both spend one; a spent row is kept rather than deleted, so "already used" stays distinguishable from "never issued"; re-issuing replaces the whole set, because a set on paper should be the whole truth about what opens the account; and revoking the second factor deletes the codes with it.

14.4 The console's policy

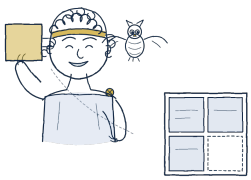
Every response carries a baseline set of headers, and the console carries a Content-Security-Policy that is same-origin for everything. Its script source is now a **per-request nonce** rather than `'unsafe-inline'` — which is the directive that lets an injected script tag execute.

Earning that cost more than the policy line. A nonce cannot authorize an inline event handler, and the console was built from a hundred and forty-one of them. They are all delegated listeners now: a render writes a data attribute holding an index into a registry of closures, and one listener per event type dispatches to it. The call expressions stayed where they were readable, and their arguments are now real values instead of being escaped into a string and parsed back out — which also removed a class of quoting bug where a title containing an apostrophe could break its own button. Style attributes keep their allowance: two hundred and fifty of them remain, and injected CSS is a far smaller prize than injected script.

One handler sat inside a single-quoted string rather than a template literal, so its registry call never interpolated and the funnel's submit button shipped as dead text. No static check saw it; a browser test pressed the button and timed out. There are now static checks for both the inline handlers and that exact stranding, because the policy holds only while they hold.

14.5 Files to open

`domain/authz/secretbox.rkt`, `domain/authz/authz.rkt` (tokens, 2FA, recovery codes), `cli/telemachus-se`
`docs/ops/secrets-at-rest-runbook.md`, `test/secretbox-tests.rkt`, `test/console-tests.rkt`.



Chapter 15

Extending Telemachus

15.1 Four seams, and one prefix

A plugin is a directory with a manifest and an entry module, loaded at boot. It may fill four seams, and the whole extension contract is those four:

- **Tools** — `(provide tools)`. A capability the model may call. It registers through the same registry as the built-ins, so per-tool permissions and per-team activation apply without the plugin doing anything.
- **Workflows** — `(provide workflows)`, or spec files in a directory. Validated exactly as an API-published spec is, and materialized into a team on first lookup, because a plugin has no team at load time.
- **HTTP routes** — `(provide routes)`. Mounted at `/api/x/<plugin-id>/...`. The prefix is platform-fixed, so a plugin cannot shadow a core route or another plugin's; every route requires a bearer token and the permission it names is checked before the handler runs; and a malformed entry fails *that plugin's load* rather than a request an hour later.
- **Job kinds** — registered from `init!`, named `x.<plugin-id>.<name>`. A plugin kind is an ordinary job: the enqueuing team and user, the concurrency cap, quota admission, the org gate, cancellation and the lease all apply, and none of it is inherited from the plugin. A remote kind must supply a validator, since there is no in-process handler to be strict for it.

15.2 Screens of your own

A plugin serves its own pages from two directories, and the difference between them is who may read them. `landing/` is public at `/beta/bundle/<id>/` and publicly cached — it is a marketing funnel a prospect is linked to. `bundle/` is authenticated at `/api/x/<id>/bundle/`, requires a bearer token, and is answered `private, no-store`. Customer screens belong in the second; before it existed there was only the first, and serving a signed-in screen set from it would have put customer pages behind a public cache.

Assets ship inside the bundle. The console's policy is same-origin, so a CDN font or script is blocked — deliberately.

15.3 Wearing the customer's colours

The branding document carries a *theme*: background, surface, ink, muted, brand, the ink that sits on the brand, a corner radius, a mode and a font. It is the same token vocabulary the onboarding funnel already used, so a palette is written once and worn by both, and it is per-organization for free, because a company's branding is one more document under the same key — a company on its own hostname is themed before anyone signs in.

Two rules keep it honest. An unknown token is refused rather than ignored, because a token that silently does nothing is how an integrator concludes the theming is broken. And contrast is *enforced* by the server, not merely warned about: WCAG AA for text, muted text, text on a panel and a button label, and a lower bar for the brand against its ground. This is the screen people sign in on, and an instance that themed its own sign-in link into invisibility would have no way back through the UI. The button's ground is a colour the console mixes rather than a token, so that mix is what gets checked — checking the brand itself would fail themes that render perfectly and pass ones that do not — and which side of the palette counts as dark is measured, not read from the mode token.

15.4 What is not available

Stated plainly, so nobody designs around a hope: a plugin cannot add a tab to the core console — it serves its own screens instead; feature flags are per team, with no instance-wide switch, so "this deployment has no chat" means doing it for every team; and there is no hot reload, so adding a plugin means a restart.

15.5 Files to open

`docs/integrators-guide.md`; `plugins/integrator-demo/`, the worked example — a tool, two routes, a job kind and a themed screen; `domain/agent/plugins.rkt`; `domain/branding/branding.rkt`; `docs/reference/sdk.md` and `docs/reference/plugins.md`.

Chapter 16

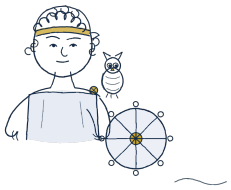
How the project tests

- **Unit suites** (`test/*-tests.rkt`) exercise `domain/` directly against a fresh database, on both dialects. Whole pipelines run through the real scheduler by draining its queue in the test process.
- **Smoke scripts** (`test/server-smoke.sh` and friends) boot a real server and assert over HTTP with `curl`. *Test the endpoints, not just the model*: the bug that actually bit the Localization Manager lived in the HTTP layer, where a query filter compared a string against symbol keys and silently fell back to its default.
- **The end-to-end gate** (`test/e2e/validate.sh`) drives the console in a headless browser, fails on any uncaught page error and any 5xx, and runs the whole document pipeline against the scripted mock model. It is the deploy gate; the screenshot tours do not assert and will photograph a broken page.
- **Mocks are servers.** `test/mock-llm.rkt` is a deterministic OpenAI-compatible model with a tool-call mode for the agent loop and a chat mode for the pipeline.
- **Verify on PostgreSQL** before calling anything done. Everything honours `DATABASE_URL`.
- **The gates in CI**: unit, smoke, multi-tenancy, the S3 endpoint against the real `aws` client, the localization gate (English required, other locales advisory), the documentation drift gate, the end-to-end gate, and the funnel's browser test.

A suite that can skip is a suite that can hide. The S3 smoke exits cleanly when the `aws` CLI is absent — so when the credential's secret became encrypted at rest, the path that decrypts it on every signed request went unexercised, because no developer box and no CI runner had the client. The client is in the development shell and installed in CI now, and the suite runs with the secret sealed, which is how a real deployment runs it. The same suite had never run outside the Nix shell, so it had never needed to set its own collection path; its first run in CI could not load a library and reported that the server never came up.

Some things only a browser can see. Converting the console's inline handlers left exactly one of them inside a quoted string rather than a template literal, so it rendered as literal text and the funnel's submit button did nothing. Every static check passed. A browser test pressed the button and timed out. Where a static check *is* possible, write it afterwards: the two invariants that conversion depends on are now pinned in a unit test that reads the console as a file.

Some habits the tests taught: never assert "newest first" on two rows created in the same second; kill a background server by PID, never by pattern (`pkill -f` matches its own command line); a test that creates documents must point the blob root at a temporary directory or it writes into the checkout.

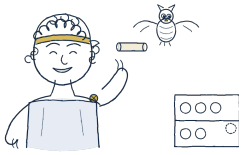


Chapter 17

Operating notes

- **Paths are anchored at definition.** The web server repoints the current directory at its own web root while it handles a request — inside the read-only Nix store on a packaged install. A relative path resolved lazily aims at the store. The blob root is absolute by construction, a unit test asserts it, and the server refuses to boot with a relative one. This shipped once: startup was healthy, every unit test passed, and the first document save failed with “Permission denied”.
- **Quotas stall queues.** A workflow step is a scheduler job admitted only when the team is under its AI budget. With the default two thousand tokens a day, a bulk draft or a batch extraction stops after the first jobs and looks like a hang. Raise `ai.tokens.total` first; the Automations card shows the remaining budget for this reason.
- **Uploads above one megabyte need the listener’s limit raised**, or the underlying web server drops the connection with no response at all.
- **The console caches its HTML at startup**; restart to pick up edits.
- **Multi-tenancy needs one restart** to set the flag; onboarding a company after that needs none.
- **PDF text extraction needs poppler-utils**; a missing extractor fails the indexing run deliberately, so that no PDF is silently unsearchable for months.
- **Decide about secrets at rest before you take a backup.** With no key set, a dump of the database carries every TOTP seed and S3 secret in the clear — a supported choice, and now a visible one. Adopting a key is one command and no downtime; keeping the key in the same backup as the database buys nothing. Losing it means re-enrolling every second factor and re-issuing every access key.
- **Upgrading the server upgrades the workers.** A pull worker must return the claim’s fencing token with every heartbeat and result; one that does not gets a 409. The reference worker sends it, but a third-party worker has to be updated in step.
- **A theme can be refused.** The console’s palette is validated and contrast-gated when it is written, so an operator who asks for an unreadable combination is told which pair and by how much, rather than discovering it on the sign-in screen.

The runbooks under `docs/ops/` cover the document repository, the workflow engine and multi-tenancy in operator terms.



Chapter 18

Contributing

- Read `CLAUDE.md` before starting; add to it when you learn something that cost you time. It is the project's memory.
- Keep SQL dialect-neutral and run the suites on both dialects.
- Every model-facing feature validates and refuses; every AI call is metered; every resource is checked through `can?`.
- Refuse loudly rather than degrade quietly. An unreadable reply, an unknown theme token, a secret that will not decrypt and a schema the model did not honour are all errors that name themselves — never an empty string, a dropped field or a silent default. Most of the bugs this project has had to hunt were something failing politely.
- Adding a route, a tool, a workflow or a permission changes the generated reference: regenerate and commit it in the same change.
- New user-facing strings go into the right home of the three, then through the extractor. English is required by the gate; other locales are coverage.
- Design first for anything with a policy fork: a short document under `docs/design/` with data shapes, a contract any backend could implement, and the decisions to confirm. The decision log is `docs/design/decisions.md`.
- Commit messages say what was verified and how. A slice is done when the unit suites, the smokes, the gate and `nix build` agree, on both databases.