Leanexe's verification knowledge bases: reviewed entries whose cited declarations a checker resolves

exaspec@morphism.com

27 November 2026

Abstract

Leanexe compiles programs written in a first-order subset of Lean and proves that the compiled code computes the Lean definition. Lean's kernel checks the proof for each program. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines. This report describes the verification knowledge bases (VKBs) that record what agents and people learn while proving theorems about such programs.

A VKB records each recurring problem of proofs about compiled programs once, as a reviewed entry whose cited Lean declarations a checker resolves against leanexe's library. An entry states when it applies, what to do, what fails, and which finished proofs used the method. Its metadata names the Lean modules, declarations, and goal shapes that a search can match. The checker confirms that each cited name exists and that the entry's own modules import it, and it does not judge the guidance. No theorem imports a VKB, so a wrong entry can mislead a proof and cannot make a theorem false.

Leanexe's public VKB is a directory of its repository, and the internal VKB of exaspec, a service that runs agents on leanexe, is private. The internal VKB's review admits an entry only after removing the project's names, domain, numbers, and code, and it sends each defect of leanexe to leanexe as a reproduced issue. A rule of 27 November 2026 sets the conditions under which an internal entry moves to the public VKB. Agents have read entries and credited them in their journals, and no controlled measurement shows how much entries shorten proofs. The report compares the VKBs with Mathlib, the Archive of Formal Proofs, LeanDojo and its prover ReProver, LEGO-Prover, and a catalog of property specification patterns.

1 Introduction

A proof that a compiled program computes its specification repeats steps that earlier proofs have taken. Leanexe's verified compiler gives each program a theorem that the program's WebAssembly module computes the Lean definition, and the work that remains comes after that theorem (`VKB/README.md:3`, [10]). That work takes the same forms across programs, such as an invariant of a loop or a bound in closed form on the memory that a call allocates. An agent is a language model that reads files and runs tools in a session. An agent knows of earlier sessions only what their files record, so it solves again each recurring problem that no file records.

A verification knowledge base (VKB) is a collection of written entries, each about one recurring problem of proofs about compiled programs. An entry is a directory that holds a README of up to four paragraphs and a file of metadata that names the Lean declarations the entry cites (`VKB/README.md:7`). A checker resolves each cited name in the Lean environment that leanexe's full check builds, where the full check is the script `tools/full-check` that runs leanexe's builds and checks in order. A review admits, revises, or retires each entry (`VKB/README.md:121`). Agents search the entries before a proof task and record their lessons after it (`AGENTS.md:55`).

A VKB records each recurring problem of proofs about compiled programs once, as a reviewed entry whose cited Lean declarations a checker resolves against leanexe's library. Agents have read entries and credited them in their journals, and no controlled measurement shows how much entries shorten proofs. Leanexe keeps a public VKB in its repository [21]. exaspec, a service that runs agents on leanexe in its users' repositories, keeps an internal VKB that is private [7, 9].

The report describes the following parts of the two VKBs.

1. The form of an entry, the checker's tests, and the defects that the check leaves to review (Section 3).
2. The rules that admit, revise, and retire public entries, and the review that removes a project's details from internal entries (Section 4).
3. The two VKBs, the issues and fixes by which lessons pass from the internal VKB to leanexe, and the rule for moving an entry (Section 5).
4. The record of agents' use of entries and the measurements that it lacks (Section 6).

Section 7 compares the VKBs with libraries of formal proofs, with the libraries of machine provers, and with a catalog of specification patterns.

This report describes leanexe at commit `4556fa84` of 26 November 2026 and the internal VKB as of 27 November 2026. Every link to a leanexe file points to that commit. The report describes the internal VKB in general terms and names none of its projects, sessions, repositories, files, or entries. Companion reports describe leanexe's compiler [10], its targets [12], its numerical library [11], and its guarantees and their assumptions [13]. The author read the VKB's files at the pinned commit and did not run the checker.

2 Recurring proof work after the compiler's theorem

For each function `f` of a program `p` in the dialect, the total, first-order subset of Lean that leanexe compiles, the command `verified_compile` states the theorems `p.f.implements` and `p.f.trapFree` [10]. The first states that the program's WebAssembly module computes `f` or traps, and the second excludes the trap under a bound on allocation that a caller proves [13, §2.3]. A user's proof relates `f` to an independent specification and composes that equation with the generated theorems into a call theorem or a master theorem [10, 11]. Both state that an export returns without a trap, a call theorem with the specification's value and a master theorem with a value in a stated relation to its argument (`VKB/entries/spec-call-theorem:3`, `VKB/entries/spec-call-theorem:27`). Proofs for C, bare-metal programs, AArch64 Images, Cortex-M7 boards, and GPU kernels add the theorems of each back end [12].

The public VKB covers the proof obligations of every artifact and property that leanexe's index names by a code, the specification proofs of the library, and the proofs that check the back ends' models (`VKB/README.md:3`). The index, `index.md`, uses codes such as A64 for an AArch64 Image and M7 for a Cortex-M7 image. Its entries also give guidelines for the work around a proof, such as testing a statement before its proof and showing that a check can fail. Table 1 lists kinds of proof work with one public entry for each.

3 Entries and the checker's tests

A VKB records each recurring problem of proofs about compiled programs once, as a reviewed entry whose cited Lean declarations a checker resolves against leanexe's library. The entry's README gives the method in prose, and its metadata names the declarations that the prose relies on. The

Table 1: Kinds of proof work after the compiler's theorem, with one public entry for each and the entry's use as the VKB's index states it (VKB/README.md:11–117). Each entry's name links its README.

Work	Public entry	Use
Invariants of the dialect's loops	Loops and builds	Invariants of `LeanExe.loop` and element equations of `LeanExe.build`
Call and master theorems	Specification call theorems	A specification equality combined with the generated execution and allocation theorems
Allocation bounds	Closed-form allocation bounds	The generated allocation term bounded in input sizes or counts of loop passes
Numerical accuracy	Accuracy against real functions	A binary64 function's distance from the rounding of a Mathlib function, carried to the WebAssembly module
Emitted files	File theorems	A statement that an emitted `.wasm` file holds the encoding of a WebAssembly module
Bare-metal C	Bare-metal C programs	Whole-program theorems for fixed arguments, loaded inputs, request loops, and controllers
AArch64 Images	AArch64 Image theorems	An Image's run from boot, from the compiler's theorems
Cortex-M7 images	Cortex-M7 board theorems	A step's run from reset through every window, from its compiler theorem
GPU kernels	GPU kernels	WGSL dispatch and PTX launch theorems, composed into a host program
Kernel evaluation	Kernel evaluation costs	The comparison that makes the kernel evaluate a large literal or a projected term
Statements	Testing a statement before its proof	Evaluation of a new statement on an executable model before its proof
Evidence from checks	Checks that can fail	A seeded fault that a check must detect before its passes count as evidence

checker establishes that those names exist and that the entry's own modules import them. It establishes nothing about the method, which only review judges. Figure 1 shows the two files and the checker's tests.

3.1 The two files of an entry

An entry is a directory `VKB/entries/NAME` that holds `README.md` and `entry.json` (VKB/TEMPLATE.md:3). The README addresses a prover who has the goal and no other context, and it supports each claim with a declaration or a file that a reader can check. Its paragraphs state when the entry applies, what to do, what fails and why, and the finished proofs that used the method (VKB/TEMPLATE.md:24–36). An entry may merge or drop a part that would be empty (VKB/TEMPLATE.md:36), and it may record a failure that has no fix, because the record saves the next prover the same work (VKB/README.md:9).

The file `entry.json` names the modules to import, the declarations that the entry relies on, its worked examples, and the goal shapes that it addresses (VKB/README.md:7). A goal shape is the Lean text of a goal with the constants that identify it, and only search reads it (VKB/TEMPLATE.md:18). The listed modules must import every listed declaration and example, and the template limits them to modules that the first step of the full check builds. A module that reads a file during elaboration, such as a file theorem's module, never appears among them, and the field `unimported` names the theorems that the README cites from such modules (VKB/TEMPLATE.md:20).

The entry on instructions that the encoder accepts shows the form, and it applies when new code emits an instruction that existing code does not emit. The instruction type of Talos, the WebAssembly interpreter written in Lean in whose semantics the compiler's theorem holds [10], has more forms

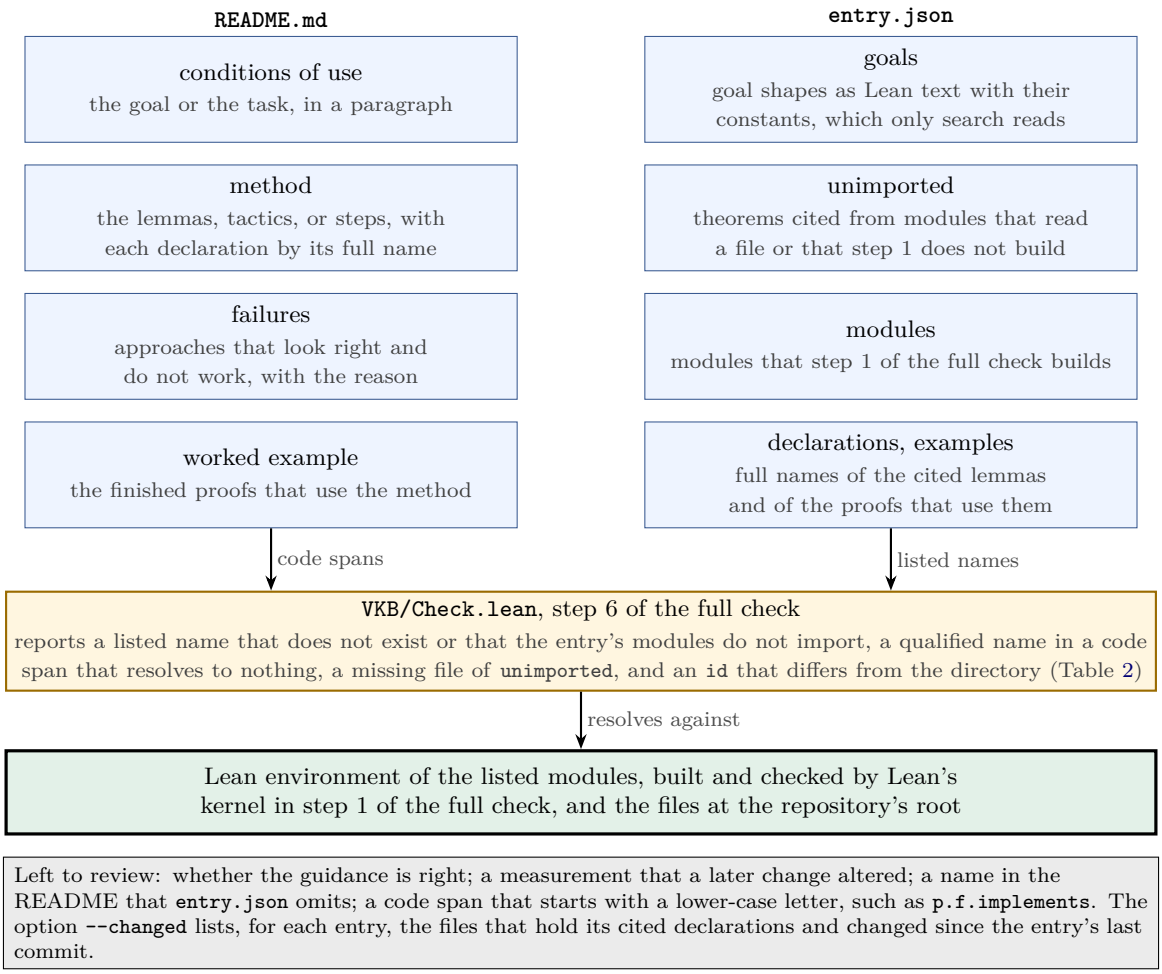

Figure 1: The two files of a public entry and the checker's tests. Blue boxes are the parts of an entry as the template gives them, with the README's four paragraphs on the left (VKB/TEMPLATE.md:24–36) and the fields of the metadata on the right (VKB/TEMPLATE.md:7–20). The orange box is the checker, a Lean program that runs outside the kernel, and the box with a heavy border holds the declarations that the kernel checked. The gray box lists what the check does not establish (VKB/README.md:143–149).

than leanexe's verified encoder accepts (VKB/entries/encoder-domain:3). Lean accepts a proof about code that uses such a form, and the failure appears later, when `verified_compile` stops with the message that the WebAssembly module does not encode (VKB/entries/encoder-domain:5). The entry tells the prover to check the encoder's domain, `Wasm.Encoding.InstructionForm`, before choosing instructions, and to build a missing form from accepted ones (VKB/entries/encoder-domain:7). Its worked example, `Verified.Wasi.fieldLoadCode`, loads a field of two bytes with two loads of one byte, and `Verified.Wasi.littleEndian_read16` proves the field's value.

No public entry contains Lean code of its own. An entry names existing declarations instead of restating them, and when several proofs repeat a lemma, it points to the copies (VKB/README.md:9). Replacing the copies with one shared lemma is a separate change to the code, after which the entry names the shared lemma and keeps one former copy as a worked example (VKB/README.md:123).

Table 2: Defects of an entry that the check reports, and defects that it misses, which the change that causes each one must correct (VKB/README.md:129, VKB/README.md:143–149).

Defect	Outcome	Source
A listed declaration or example that does not exist	Reported	VKB/Check.lean:123
A listed name that the entry's own modules do not import	Reported	VKB/Check.lean:126
A qualified name in a README's code span that is no declaration, module, or namespace and no file at the root, and that `unimported` does not list	Reported	VKB/Check.lean:137
A file of `unimported` that does not exist, or a name of it that an imported module declares	Reported	VKB/Check.lean:130, VKB/Check.lean:133
An `id` that differs from the entry's directory	Reported	VKB/Check.lean:53
A name in the README that `entry.json` omits	Not reported	VKB/README.md:145
A code span that starts with a lower-case letter, such as a partial name	Not reported	VKB/README.md:146
Guidance that a change to a declaration's statement, the compiler, or the library has made wrong	Not reported. The option `--changed` lists the changed files	VKB/README.md:147
A measurement that a later change altered	Not reported	VKB/README.md:148
A listed module that starts to read a file at elaboration	Not reported	VKB/README.md:149

3.2 The checker's tests and its report of changed files

The checker, the Lean program VKB/Check.lean, reads every entry, imports the modules that the entries list, and reports the problems in the upper part of Table 2 (VKB/Check.lean:113–138). It reads a code span of a README, a stretch of text between backticks, as a qualified name when the span consists of identifiers separated by dots and starts with an upper-case letter (VKB/Check.lean:28–38). A span such as `p.f.implements`, which starts with a lower-case letter, is an expression over local or generated names, and the check does not read it (VKB/TEMPLATE.md:38). The check stops with an error at a listed module that does not exist, an entry without a README, or metadata that does not parse (VKB/README.md:129). It runs as step 6 of leanexe's full check, after the first step has built the modules that the entries list (tools/full-check:24, tools/full-check:717).

With the option `--changed`, the checker prints a report of changed files and exits with status 0 (VKB/Check.lean:147–179). For each entry, the report considers the files that hold the entry's cited declarations and the files that its field `unimported` names. It lists those that differ in the working tree from the last commit that changed the entry (VKB/README.md:137). The report directs a reviewer to guidance that may be out of date, and the reviewer decides whether it is. A link from one entry to another names the other's README, so the repository's link check, step 11 of the full check, fails when a change renames or merges an entry (VKB/README.md:151, tools/full-check:34).

3.3 Entries outside the trusted base

No theorem depends on an entry. The verified compiler, the runtime, and the trusted statements import no part of the VKB, so an entry can help a proof or fail its own check and cannot change a theorem (VKB/README.md:3). Leanexe's trusted-base document lists the VKB among the generators that need no trust, whose output Lean's kernel checks (docs/trusted-base.md:15). Wrong guidance can cost a prover failed attempts, and a proof that follows it still has to pass the kernel's check.

A resolved name points to a declaration that Lean's kernel has checked. Each listed declaration lies in a module that the first step of the full check builds, and the kernel checks every declaration

before Lean adds it to the environment [13, §4.1]. The same step runs the axiom walk, a Lean program that follows the constants of every declaration of the six libraries that the step builds. The axiom walk fails when a declaration reaches an axiom other than `propext`, `Classical.choice`, and `Quot.sound` [13, §3]. The VKB check says nothing about whether the entry's prose describes the declaration correctly, or whether the declaration solves the problem that the entry names.

3.4 Rules for the text of an entry

The template's rules for the text require each claim of an entry to cite something that a reader or the check can find. An entry cites tracked files, declarations by their full names, and documents by their section headings (VKB/TEMPLATE.md:42). It gives no line numbers, no commit hashes, and no counts of files, theorems, declarations, or copies. The check resolves a full name, while a line number or a count becomes wrong at the next edit of the file it describes, and the check cannot detect that change. An entry refers to another entry by a link to its README, whose text names the other entry's subject, so that the link check fails when a change renames or merges the other entry (VKB/TEMPLATE.md:44).

The template also sets the form of measurements, recorded failures, names of targets, and proposals. A measurement appears only when a tracked file records it with its configuration, or as the ratio that a fix gave (VKB/TEMPLATE.md:42). A failure that only a journal records appears as its cause, its symptom, and its remedy, with the remedy's declaration or file. An entry names targets by the codes of leanexe's index (VKB/TEMPLATE.md:44). A proposal in a journal or a report has the content of the two files, so that a reviewer can admit it without rewriting it.

4 Review of entries and removal of project details

4.1 Admission, revision, and retirement in the public VKB

Every proof task in leanexe ends with its lessons in the form of the template (VKB/README.md:121). The task's agent writes an entry, or a change to an entry, in its own branch when the lesson names existing declarations and applies to other programs. It records a proposal in its journal or report when the lesson is uncertain or comes from one use.

The reviewer of the merge admits an entry when its guidance applies beyond the task, its declarations exist, and no entry already covers it. Otherwise the reviewer merges the lesson into an existing entry or declines it with a reason. The checker must pass after every admitted change (AGENTS.md:55).

Revision follows changes to the code. A change that renames, removes, or restates a declaration that an entry names updates those entries in the same change (VKB/README.md:123). A proving run that finds an entry's guidance wrong or incomplete revises the entry, and so does a change after which a measurement that the entry cites no longer holds. A separate change retires an entry that is wrong beyond repair, whose code is gone, or whose lesson another entry holds. The goal shapes of a retired entry move to the entry that absorbs it (VKB/README.md:125).

An entry does not state how often its lesson recurs, so the agent's report gives that evidence to the reviewer (VKB/README.md:121). Leanexe's instructions keep checked material that is narrow or specific to one program as worked examples, and generality and recurring evidence decide its promotion (AGENTS.md:51). They remove an example only when it is invalid, stale, unsafe to disclose to a measured proof task, or a duplicate without a distinct lesson.

Table 3: The review's decisions on a candidate in the internal VKB, from its guidelines.

Decision	Condition	Result
Admit	The candidate records a method, pattern, pitfall, or workaround that applies beyond its project, its declarations exist, and no entry of either VKB covers it	A new internal entry in the form of leanexe's template
Merge	An internal entry covers part of the lesson	A revision of that entry
Upstream	The candidate reports a defect or a missing general theorem in leanexe, or a gap in leanexe's documents or VKB	A proposal for leanexe (Section 5.2)
Decline	Leanexe's VKB, manual, or demonstrations cover the lesson, or the lesson applies only to its project	A reason in the review's record

4.2 Decisions on candidates in the internal VKB

exaspec's sessions write their lessons as candidates in the form of leanexe's template, and the internal VKB keeps each candidate as its session wrote it. A candidate may become an entry or a proposal only when it comes from work that its account shared, because exaspec's terms allow only shared work to enter the knowledge base [8]. Until the service has accounts, every session's work belongs to exaspec. An entry carries no session, account, or project, and only the review's private record links a candidate to its result.

A review decides each candidate in one of the four ways of Table 3. When the review can separate a candidate's method from its report of a leanexe defect, it yields both an entry and a proposal. The internal guidelines keep methods, patterns, and workarounds in the internal VKB until an entry moves under the rule of Section 5.3. They send each defect of leanexe upstream, because its fix helps every user of leanexe.

4.3 Rules that remove a project's details

An internal entry states its lesson for a class of programs and describes the class in the terms of Lean, Mathlib, and leanexe: types, dialect forms, operations, back ends, and goal shapes. A class such as a function over binary64 words that returns a record of interval words names no project. Six rules apply to every part of an entry and of a proposal, from its directory's name and title to its code, examples, worked case, and row in the index.

1. The entry names no application, domain, or purpose of the program. Ordinary technical words stay, unless together they describe the project.
2. A formula appears as general mathematics, without the name of the program's algorithm.
3. The entry uses no name from the project, whether of a repository, module, file, declaration, structure, field, or constant. Examples use neutral names, and names that leanexe, Lean, or Mathlib declare are exempt.
4. The entry gives no number from the project, such as a size, a count, or a time. A measurement that is itself the lesson appears as a rate in general quantities, to one significant figure, with the statement that it comes from one use.
5. The entry copies no code from the project, except a lemma or tactic that mentions no declaration of the project, and the reviewer writes each example fresh.
6. The worked case takes one or two sentences, which give the class of program and the steps of the entry that the work used, in the entry's final paragraph.

Before admission, the reviewer checks an entry against its project's material. The reviewer reads the candidate's record of its source and the requests that the record lists. The reviewer searches the

entry for the names of the project's files and declarations, for the domain words of the requests, and for each number in them. The reviewer then judges whether the entry would read the same if any of several unrelated projects had produced it. An entry that fails the search or the judgment goes back for revision, and the guidelines required the same revision of the entries admitted before these rules existed.

The privacy rules remove words that a later agent could use to search for an entry. An internal analysis of 24 November 2026 found that the revision for privacy removed nearly all the domain words that the first entries held at admission. A search by the words of a later request in the same field then found no internal entry. The rules keep the Lean terms by which a search can find a method: the goal shapes, the constants, and the cited declarations.

4.4 Records of what each review checked

Each internal entry and proposal ends with a paragraph that states what the review checked, and an entry's final paragraph also holds its worked case. The paragraph names the declarations found in leanexe's source with the commit read, the Lean code that the review built, and the observations that come from one session only. The review runs leanexe's checker at the leanexe commit that exaspec's developments require, on a copy of the internal entries, and the paragraph lists every name that the check did not resolve.

The internal VKB has no build of its own. A review has built a block of Lean code in an internal entry only if the entry's final paragraph says so. Some final paragraphs name code from a session that the review did not build.

5 The two VKBs and the paths between them

5.1 The public VKB and the internal VKB

Leanexe's public VKB began with the commit "Start the verification knowledge base" of 30 October 2026 [18]. Its entries come from leanexe's own proof tasks, by agents and people, under the rules of Section 4.1. The internal VKB began on 23 November 2026 and holds entries from exaspec's developments, which the exaspec examples page describes [6]. It uses leanexe's template, and its entries also hold blocks of Lean code that the review wrote with its own names. exaspec's VKB page shows the size of each VKB's entries after every update [9].

exaspec's terms state that exaspec "generalizes each addition to its knowledge base and removes names and other details specific to your project" [8]. The VKB page states the plan to move knowledge from the internal VKB to the public one as exaspec analyzes and refines it [9]. Figure 2 shows the paths by which lessons reach each VKB and pass from the internal VKB to leanexe.

5.2 Proposals that become leanexe issues and fixes that edit public entries

A proposal follows the rules of Section 4.3, states what fails, gives a minimal reproduction written fresh in leanexe's terms, and proposes a change. It names no user, account, session, or project of exaspec. exaspec's staff reproduce it at a stated leanexe commit and Lean version and file it as an issue on leanexe. A proposal that does not reproduce stays unfiled. The issues "Add an equivalence lemma for loops of Boolean conjunctions" and "Invariant lemmas for LeanExe.repeatWhile in LeanExe.Dialect" are two such issues [14, 15].

A fix in leanexe can add a lesson to a public entry. The commit "Add loop_and_iff for loops of Boolean conjunctions" fixed the issue on loops of Boolean conjunctions and edited the public entry on

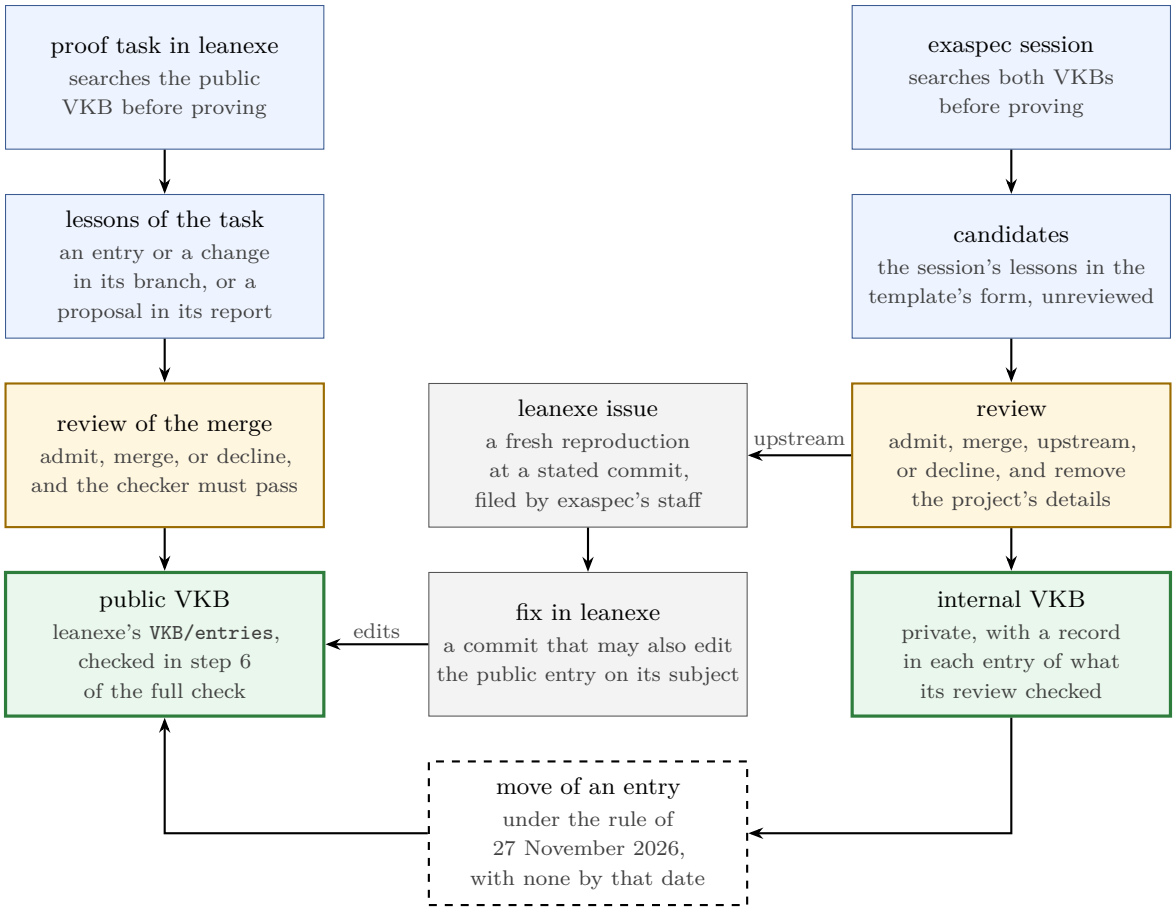

Figure 2: The paths of lessons into the two VKBs and from the internal VKB to leanexe. Blue boxes are proof work and its unreviewed lessons, orange boxes are reviews, and green boxes with heavy borders are the two VKBs. Gray boxes are leanexe's issues and fixes, and the dashed box is a move under the rule of Section 5.3, which no entry has made. The public path follows `VKB/README.md`:121–125, and the internal path follows the internal VKB's guidelines.

loops and builds in the same change [17]. That entry now names `LeanExe.Dialect.loop_and_iff` and the copy of its forward direction in a demonstration (`VKB/entries/loop-combinators`:9). Commits of leanexe's main branch before the pinned commit fixed every issue filed from a proposal on 24 November 2026, and most of those commits also edited the public entry on the issue's subject. Some later issues ask for a change to a public entry, such as "The room of the first store in the entry on specification call theorems" [16].

The path through issues and fixes brings a fix in leanexe and its lesson to the public VKB. A method, pattern, or workaround that addresses no defect of leanexe stays in the internal VKB, as the internal guidelines require. Methods of the internal VKB reach the public VKB only by the move of Section 5.3.

5.3 The rule for moving an entry to the public VKB

A rule adopted on 27 November 2026 in the internal VKB's guidelines sets the conditions under which an internal entry moves to leanexe's VKB. The rule carries out the plan that exaspec's VKB page states [9], with conditions on privacy, on exaspec's terms, and on approval. An entry moves

only when all five conditions hold.

1. Generality: a review has recorded that the entry's method applies beyond the projects it came from.
2. Material: a second review at the time of the move confirms that the entry meets every rule of Section 4.3, and the move removes the worked case, which describes private work.
3. Terms: exaspec's terms permit the move for the sources of the entry. The terms permit entries of the knowledge base and leanexe reports from shared work. The rule asks exaspec's owner to confirm, before the first move, that they permit publishing an entry in leanexe.
4. Form: the moved entry follows leanexe's template where it can. Code that the public VKB does not hold goes into leanexe as a separate change, or the entry states the method in prose and cites what exists. Leanexe's checker passes with the moved entry.
5. Approval: exaspec's owner approves each batch of moves, from a list that gives each entry, its sources, and the second review's result.

After a move, the internal VKB keeps a short entry that links the public one and records the move's leanexe commit, so that the two texts do not diverge. An entry that does not meet the conditions stays in the internal VKB. At this report's date, no entry has moved, and the internal VKB records no confirmation of the condition on the terms.

6 Use of entries by agents and the evidence of their effect

6.1 Search before a proof and lessons after it

Leanexe's instructions tell an agent to search the VKB's entries for the declarations and goal shapes of a proof task before the task (`AGENTS.md:55`). The skill `verified-request`, a file of instructions for turning a request in English into a verified program, states the search (`.claude/skills/verified-request/SKILL.md:176–179`). Before each proof, the agent searches the `entry.json` files for the constants of the goal, such as `LeanExe.repeatWhile`, and reads the README of every entry that names them. The skill also names the entries that cover the common shapes of its proofs, and it ends each stage of proof with the lessons that apply beyond the program (`.claude/skills/verified-request/SKILL.md:204–209`). exaspec's development sessions consult both VKBs before they write proofs [9].

6.2 Recorded use

The records of entries in use that this report found come from exaspec's own developments over five days. The first sessions began with an empty internal VKB, and the later ones among them read the unreviewed candidates of earlier sessions within three minutes of starting. Their journals credit candidates that the review then declined or sent upstream, so agents used lessons that no review had admitted. The internal analysis of 24 November 2026 records these observations, and none of the sessions it covered opened an admitted entry.

Before each of a later series of exaspec sessions, the person who ran them wrote down the entries that the session should use. The records compare those predictions with the entries that the sessions' journals credit, for three sessions. The journals credited fewer than a third of the predicted entries, and most of the entries that they credited had appeared among the predictions. In two of the three sessions, the journal records that an entry changed the program or its theorem statements.

One comparison of cost exists, without a control. Three sessions added a C form to a development, and the two later ones, whose journals credit a candidate on native forms, cost about half as much

as the first. The requests differed in scope, so the comparison shows a direction and no size.

6.3 Measurements that the record lacks

exaspec's records do not compare cost, time, or failed builds with entries and without them for the same request. They do not measure the spread between repeated runs of one request, whether the search finds the entries that apply, or whether declined candidates would have helped. The sessions of the internal analysis read leanexe's VKB, and the records measure no effect of its entries. Leanexe's instructions name the dimensions on which to judge a proof iteration: "proof-generation time, successful VKB retrieval, agent revisions, proof structure and size, shared abstraction use, and applicability beyond the measured program" (`AGENTS.md`:47). No record that this report found gives these dimensions for an entry.

exaspec's records of use are weak in three ways. In two of the three sessions with predictions, the agent printed every entry's metadata at the start, so the opening of an entry does not show that the agent chose it. A credit is the agent's own statement in its journal. One person wrote the predictions, ran the sessions, and started the reviews.

The internal analysis designed five experiments that would measure an entry's effect.

1. A noise floor: one fixed request run several times, to measure the spread that any comparison must exceed.
2. Probes: small goals of each entry's shape, which a fresh agent attempts with access to the entry and without it.
3. Held-out requests: requests that produced no entry, run with the entries and without them.
4. Recall of the search: the entries that an agent's search opens, against those that a person marks as relevant.
5. An audit of declines: the tasks of declined candidates, run as probes with the public VKB alone and with the candidate.

The internal VKB records no result of any of them. LeanDojo and LEGO-Prover measured the effect of their libraries by removing them and holding the prover fixed (Section 7), and the VKBs lack that comparison.

7 Related work

Table 4 compares the VKBs with libraries of formal proofs, with the libraries of two machine provers, and with a catalog of specification patterns. The VKBs differ from each of them in what an entry holds: a method in prose, with cited names that a checker resolves. The last column gives each source's measurement of use or effect, the measurement that Section 6 finds missing for the VKBs.

7.1 Libraries of formal proofs

Mathlib is Lean's community library of formalized mathematics, whose definitions and theorems Lean's kernel checks [22]. Its maintainers approve pull requests in their areas of expertise, and the command `#lint` checks, among other things, whether a file meets Mathlib's documentation requirements [22, §6.2, §7.1]. Mathlib's documentation guide requires a docstring on every definition and major theorem [19]. The Mathlib paper measures the library's size and its commits over time, and it reports no measure of how often a lemma is used or found [22, §9].

Prose that spans several declarations appears in Mathlib's module docstrings and in library notes [3, 19]. A library note is a declaration of Batteries' type `LibraryNote` whose docstring holds

Table 4: Comparison of the VKBs with libraries and catalogs. Each row follows the system's primary source.

System	Content	Check of the content	Measured use or effect
Public VKB	Prose entries on proof methods and failures, with metadata that names declarations	Cited names resolve in the environment that the full check builds. Review judges the prose	None recorded
Internal VKB	Entries of the same form, with Lean code that the review writes	The review builds code at a stated leanexe commit and runs leanexe's checker on a copy	Journal credits against predictions for three sessions, with no control
Mathlib [22]	Definitions and theorems, with docstrings and library notes	Lean's kernel checks the proofs, maintainers approve pull requests, and `#lint` checks documentation	Size and commits over time
AFP [1, 4]	Refereed Isabelle developments	Isabelle builds each entry in the current release	Imports between articles: 106 of 215 isolated in 2015
LeanDojo, ReProver [24]	Theorems, proofs, and premise annotations extracted from Mathlib	Lean checks each generated proof	Pass@1 51.2% with retrieval and 47.6% without, random split
LEGO-Prover [23]	Lemmas generated while proving, with their proofs	Isabelle verifies each lemma before it enters the library	55.3% with the library and 50.4% without, miniF2F validation, 100 attempts
Specification patterns [5]	Patterns of temporal properties with mappings into formalisms	No mechanical check reported	Survey: 92% of collected specifications matched a pattern

the note, and other docstrings cite it as "See note [...]" [3]. Mathlib's docstring role for library notes warns when no note with the cited label is in scope [20], a check of the same kind as the VKB checker's resolution of names. A Mathlib lemma is a proof that later proofs apply, while a VKB entry is guidance that a prover reads. The VKB's entries cite Mathlib's lemmas where a proof uses them.

The Archive of Formal Proofs (AFP) is a collection of Isabelle developments, organized like a scientific journal, whose referees review its submissions [1]. A submission should run in the current Isabelle release and produce a PDF document, and the archive's maintainers update every entry to Isabelle's development version [2]. Blanchette et al. measured reuse in the AFP through the graph of imports between articles [4, §3]. In the snapshot of 16 April 2015, 106 of 215 articles neither imported another article nor had an importer.

An AFP article is checked code, so its reuse appears in the imports of other articles. Nothing imports a VKB entry, so its use appears only in what agents read and what their journals credit. An import graph cannot measure that use, and a VKB needs records of retrieval and experiments of the kind that Section 6 lists.

7.2 Libraries of machine provers

LeanDojo extracts from Lean and Mathlib the theorems, their proofs, and the premises that each tactic uses, where a premise is an existing lemma or definition that a tactic takes as an argument [24, §3, §4]. Its prover ReProver retrieves premises from those accessible to the current theorem, by the similarity of embeddings of the proof state and the premise [24, §5]. It concatenates the retrieved premises with the state as the input of its tactic generator. On the random split of its

benchmark, ReProver proved 51.2% of the test theorems with one attempt each, and the same prover without retrieval proved 47.6% [24, Table 2]. On the split whose test proofs use premises that training never used, the two figures were 26.3% and 23.2%.

LEGO-Prover grows a library of lemmas while it proves competition problems in Isabelle, and a lemma enters the library only after Isabelle verifies its proof [23, §3]. The prover retrieves lemmas from the library by the nearest neighbors of their embeddings [23, §3.1]. A component that the authors call the evolver writes further lemmas from the library's contents.

On the validation half of miniF2F, a benchmark of competition problems, LEGO-Prover solved 55.3% of the problems in 100 attempts with the library and 50.4% without it [23, Table 2]. The evolver consumes about as many tokens as the prover. To match the cost, the authors ran the prover without the library for 189 attempts, and it reached 53.2% [23, App. A.3]. By manual inspection, retrieved lemmas aided 24% of the solved validation problems [23, §4.3.2].

The papers on LeanDojo and LEGO-Prover both hold the prover fixed, remove the library, and count the problems solved on a fixed benchmark. Neither reports repeated runs or their spread. Each library holds code that a prover checks, and its prover retrieves from it with a trained retriever or an embedding model. A VKB entry holds a method in prose that no prover checks, and an agent finds it by searching text for constants and goal shapes.

7.3 Catalogs of patterns

Dwyer, Avrunin, and Corbett define a property specification pattern as a generalized description of a commonly occurring requirement on the permissible sequences of states or events of a finite-state model [5, §2]. Each pattern gives its name, its intent, mappings into common specification formalisms, examples of known uses, and relationships to other patterns. The authors state that this format is a variant of the pattern format of Gamma et al. [5, §2].

Patterns enter the catalog because they appear frequently as property specifications. Of the 555 specifications that the authors collected, 92% matched one of their patterns [5, §3]. The paper reports no mechanical check of the catalog's mappings. It states that a definitive answer on whether patterns yield more correct specifications would require experiments with human subjects.

A VKB entry has parts that correspond to a pattern's: its conditions of use to the intent, its worked examples to the known uses, and its links to other entries to the relationships. In place of mappings into formalisms, an entry gives a method and the approaches that fail. Its cited declarations add a check that the catalog lacks: names that a checker resolves against a library whose declarations Lean's kernel checked. The survey measures how often collected requirements match a pattern, and the VKBs have no corresponding measurement of how often proof tasks meet an entry's goal shape.

8 Proofs, tests, and assumptions

Table 5 separates the statements that Lean's kernel checks, the statements that checks and records support, and the statements that the VKBs assume without support. The VKBs hold no theorem, so the proved items concern the declarations that entries cite. The author read the VKB's files at the pinned commit and did not run the checker or the full check.

Table 5: Proved, tested, and assumed items of the two VKBs.

Kind	Item	Source
Proved	The declarations that public entries cite are definitions and theorems of leanexe that Lean's kernel checked in the build of step 1	`VKB/TEMPLATE.md:18`, [13, §4.1]
	No theorem concerns an entry's guidance or its effect on a proof	`VKB/README.md:3`
Tested	Every listed name and every qualified code span of the public entries resolves, in step 6 of the full check	`tools/full-check:717`
	The report of changed files lists the files of cited declarations that changed since each entry's last commit	`VKB/README.md:137`
	The link check fails when a change renames or merges a linked entry	`VKB/README.md:151`
	Internal reviews build the code they admit at a stated leanexe commit and run leanexe's checker on a copy of the entries	Internal guidelines
	Reviewers search each internal entry for the project's names, domain words, and numbers	Internal guidelines
	A record of predicted use against journal credits, for three sessions	Internal records
Assumed	Agents find the entries that apply to their task, and no measurement gives the recall of the search	Section 6
	An entry's guidance is right at the leanexe commit that a project uses	`VKB/README.md:147`
	A journal's credit of an entry reflects use	Section 6
	The rules of Section 4.3, applied by the reviewer's judgment, remove what identifies a project	Internal guidelines

9 Limitations

A check that an entry's names resolve says nothing about whether its guidance is right or whether it shortens a proof. The check misses the defects in the lower part of Table 2, and the report of changed files directs a reviewer to guidance without deciding whether it is out of date. The internal VKB has no build of its own and no report of entries whose cited files changed. A workaround stays in an entry until a review revises it after the fix that removes its cause.

No controlled measurement shows how much entries shorten proofs, and the recorded use comes from exaspec's own developments over five days. The comparison of predictions with credits covers three sessions, and the person who wrote the predictions also ran the sessions. The privacy rules remove domain words that an agent could use in a search, and no measurement shows how well the remaining Lean terms serve the search.

This report describes the internal VKB from its guidelines and records, which are private, so a reader cannot check those statements. The figures of Section 6 come from exaspec's private records of its sessions. The internal VKB records no confirmation of the move rule's condition on the terms, and no entry has moved.

References

[1] AFP. About the Archive of Formal Proofs. Web page, 2026. URL `https://www.isa-afp.org/about/`. Read 27 November 2026.

[2] AFP. Submission guidelines. Archive of Formal Proofs web page, 2026. URL `https://www.isa-afp.org/submission/`. Read 27 November 2026.

[3] Batteries. `Batteries/Util/LibraryNote.lean`. Batteries, commit c579adab of 21 June, 2026. URL `https://github.com/leanprover-community/batteries/blob/c579adabb5358810627b03947f1b55f0c65cd6fe/Batteries/Util/LibraryNote.lean`.

[4] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the Archive of

Formal Proofs. In *Intelligent Computer Mathematics (CICM 2015)*, volume 9150 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015. https://doi.org/10.1007/978-3-319-20615-8_1.

[5] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 411–420, 1999. https://doi.org/10.1145/302405.302672.

[6] exaspec. exaspec examples. Web page, 2026. URL https://exaspec.com/examples. Read 27 November 2026.

[7] exaspec. exaspec. Web page, 2026. URL https://exaspec.com/. Read 27 November 2026.

[8] exaspec. Terms of use. exaspec web page, 2026. URL https://exaspec.com/terms. Read 27 November 2026.

[9] exaspec. Verification knowledge bases. exaspec web page, 2026. URL https://exaspec.com/vkbs. Read 27 November 2026.

[10] exaspec. Core leanexe: a verified compiler for a subset of Lean. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-core.pdf.

[11] exaspec. Leanexe's numerical library: proved accuracy for compiled floating-point code. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-numerics.pdf.

[12] exaspec. Leanexe's targets: composed theorems in stated semantics, with exact trusted bases. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-targets.pdf.

[13] exaspec. Leanexe's guarantees: kernel-checked theorems and their written assumptions. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-trust.pdf.

[14] Leanexe. Add an equivalence lemma for loops of Boolean conjunctions. Leanexe issue, opened 24 November, closed 26 November, 2026. URL https://github.com/jsmorph/leanexe/issues/1.

[15] Leanexe. Invariant lemmas for LeanExe.repeatWhile in LeanExe.Dialect. Leanexe issue, opened 26 November, open on 27 November, 2026. URL https://github.com/jsmorph/leanexe/issues/23.

[16] Leanexe. The room of the first store in the entry on specification call theorems. Leanexe issue, opened 27 November, open on 27 November, 2026. URL https://github.com/jsmorph/leanexe/issues/32.

[17] Leanexe. Add loop_and_iff for loops of Boolean conjunctions. Leanexe commit 78c3aaf8 of 26 November, 2026. URL https://github.com/jsmorph/leanexe/commit/78c3aaf84409d7a6ae2099b1a9e39c416f81e38b.

[18] Leanexe. Start the verification knowledge base. Leanexe commit 365fee9b of 30 October, 2026. URL https://github.com/jsmorph/leanexe/commit/365fee9b6d4d49fadb54113fc0825a8de1931b7c.

[19] Mathlib. Documentation style. Mathlib web page, 2026. URL https://mathlib.org/contribute/doc.html. Read 27 November 2026.

[20] Mathlib. `Mathlib/Util/VersoRoles.lean`. Mathlib, commit 2dc7c52a of 27 November, 2026. URL https://github.com/leanprover-community/mathlib4/blob/2dc7c52a612156bc7b283c0c429483a9529a5d29/Mathlib/Util/VersoRoles.lean.

[21] Morphism LLC. LeanExe. Git repository, commit 4556fa84 of 26 November, 2026. URL https://github.com/jsmorph/leanexe/tree/4556fa84205eee51b989e365a8519a33655833a2.

[22] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 367–381, 2020. https://doi.org/10.1145/3372885.3373824.

[23] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-Prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/85dca46374dc0f27b4bb5f265b3d17f0-Abstract-Conference.html.

[24] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, pages 21573–21612, 2023. https://doi.org/10.52202/075280-0944.