A project report on proof data extraction, interaction, and evaluation in Isabelle

Jonathan Julián Huerta y Munive[1 [0000−0003−3279−3685]]

Aalborg University, Copenhagen, Denmark
jjhymuni@cs.aau.dk

Abstract. This project paper reports on an openly available and prototypical toolkit for Isabelle-based machine-learning research. The toolkit combines three components: extraction of stepwise proof traces from Isabelle theories, programmable interaction with Isabelle through Scala and Python REPLs, and a modular evaluation loop for training and testing proof-step predictors. Applied to the Archive of Formal Proofs (AFP), the extractor produces 241,126 JSON files from a January 2026 AFP snapshot. The extracted data is used to train and fine-tune language models, including T5 variants and a fine-tuned Gemma 3n model. The toolkit also serves to evaluate frontier-model backends via a REPL-based depth-first proof search. The experiments reproduce the expected advantage of domain-specific fine-tuning over training from scratch and show that it supports both local and API-based model backends. In proof completion on short AFP proofs, stronger frontier and fine-tuned models outperform the small T5 baselines, while Sledgehammer remains the strongest overall baseline. The contribution of the paper is primarily infrastructural: an open Isabelle toolkit for reproducible data extraction, model training, and multi-backend evaluation.

Keywords: Isabelle · interactive theorem proving · large language models · proof assistants · evaluation infrastructure

1 Introduction

Recent generative AI advances have shown that large language models (LLMs) can accelerate proof automation in formal verification. Systems such as Kimina-Prover [48] and Aristotle (Harmonic) [1] have achieved impressive results on competition-level mathematics benchmarks [53] by combining LLMs with symbolic proof search inside interactive theorem provers (ITPs). More broadly, LLM-assisted formal reasoning has enabled deep proof synthesis [21,12], the autoformalisation of undergraduate-level mathematics in Isabelle/HOL [11], and, more recently, semi-autonomous formalisation reaching research-level mathematics [24]. These advances reflect that coupling LLMs with ITPs is both feasible and effective as an approach to automated formal reasoning.

Open tooling has been key to this progress. For instance, LeanDojo [50] established a template of open data extraction, programmable read-eval-print loops (REPLs), and reproducible benchmarks that enabled a self-reinforcing

research ecosystem [29,30,19,51]. Analogous environments exist for Rocq [9,41], HOL4 [18,49], and HOL Light [8,7]. Besides PISA [25], which has not been maintained since 2023, no comparable infrastructure exists for Isabelle. Thus, developers have repurposed low-level internal tools for ITP-LLM integrations [4].

Two considerations make addressing the lack of open infrastructure for machine learning research in Isabelle timely. First, most high-performing systems are either commercial or require a frontier-model API subscription. Open-source infrastructure is necessary for independent scientific progress, reproducible claims about model capabilities, and the technological sovereignty of institutions that cannot route sensitive formalisation work through third-party services. Second, routing every proof step through remote APIs carries an ecological cost. Neurosymbolic workflows that combine local models with symbolic tools such as Sledgehammer reduce that cost relative to delegating the entire proof task to a frontier model. Open-source infrastructure, in contrast to the closed frontier models above, is thus natural for such workflows.

This report describes a prototypical toolkit that aims to become such an infrastructure for Isabelle by combining three tools in a single openly available code base: proof-data extraction, programmable interaction with Isabelle, and evaluation workflows for machine-learning (ML) models (Appendix A gives a schematic overview of the toolkit and these three contributions). Specifically, the toolkit provides a data-extraction algorithm (Section 2) that can traverse Isabelle sessions, detect the proofs in their `thy` files, and output a JSON representation of each proof that includes: all its proof steps; the variables, constants, and type-variables in that proof step; the user-seen proof state, and the proof methods and keywords available at that proof step. Where previous Isabelle-focused extraction efforts target theorem-level artefacts, here we provide stepwise interactive data consumable by next-step predictors and search loops. The resulting JSON format is designed as experimental infrastructure for ML evaluation.

By exposing Scala and Python REPLs for controlled experimentation within Isabelle (Section 3), the project aims to make these data usable in the same setting where the evaluation happens. This enables comparisons, benchmark construction, and the repurposing of ML models originally developed for other provers. To illustrate the toolkit's use, this article includes an end-to-end case study: training LLMs on the generated data, evaluating them with standard predictive metrics (Section 4), and placing them inside a simple REPL-based proof-search loop (Section 5). These experiments are intentionally modest; their role is to validate and exercise the infrastructure, and to summarise the initial empirical findings it makes possible. The emphasis is therefore on the integrated toolkit and these lessons, rather than on a single new theorem-proving algorithm. Related work appears in the next subsection, and limitations in Section 6. The immediate aim is to provide reusable infrastructure for future experiments and comparisons. The longer-term vision is to interface Isabelle with stronger ML models and, eventually, to support richer reinforcement-learning (RL) style environments [45,44]. In the spirit of open research, the source code for this work, model weights, and examples of the generated data are openly available [33,22].

1.1 Related work

Cross-prover ML environments. In HOL4, TacticToe [18] combines kNN-based tactic prediction with Monte Carlo tree search while TacticZero [49] alternatively employs RL. Tactician [9,41] integrates kNN, GNN, and LLM-based provers into the Rocq kernel, where online learners outperform offline models, including on premise selection [40]. HOList [8] provides an ML environment for HOL Light with GNN-based action ranking [37] and RL-trained provers [7].

Isabelle-specific prior work. Sledgehammer [10] remains Isabelle's primary automated-proof tool, combining MaSH-based premise selection (a kNN and naive Bayes ensemble) with external automated theorem provers (ATP) like E or Vampire. Subsequent work introduced regression-tree models for conjecturing and inductive proofs [36,34,35] and proof-method selection [42]. The Portal-to-Isabelle (PISA) project [25] was the only prior programmable Isabelle environment for ML research, yielding LLM-based tactic suggestion and autoformalisation [27], a Sledgehammer integration for LLMs (Thor [26]), and Magnushammer [32], a transformer-based premise selector that outperforms MaSH. PISA has not been maintained since early 2023, and subsequent work shifted focus to LLM proof-search efficiency [52] rather than infrastructure upkeep. Isabelle also ships Mirabelle, a batch tactic-benchmarking tool that runs as a build hook, but it is not designed for ML integration or dataset generation.

AutoCorrode. Amazon's AutoCorrode is a parallel effort to our toolkit comprising the Isabelle-assistant [4], an Isabelle-proxy [5] (jEdit-based), an MCP component [2] for enterprise LLM pipelines, and a command-line REPL [3]. Its repository appeared in March 2025 and was publicised in the Isabelle users mailing list in February 2026 [6]. Our toolkit's repository was created in November 2023 while the first snapshot of our project with trained models was published in a Zenodo repository in March 2025 [33,22]. AutoCorrode requires an Amazon account and targets engineering integration, while our prototype is fully open, supports multiple model backends (Ollama, HuggingFace, OpenAI, Gemini), and focuses on ML training pipelines and end-to-end evaluation.

Isabelle/server. Isabelle has a built-in `isabelle server` tool exposing a JSON-over-TCP interface for session management and theory loading. It differs from the prototype here in three respects: it is coupled to a Prover IDE (PIDE) session and does not expose a step-by-step REPL; it provides no proof-trace extraction; and it has no ML-backend integration. Following PISA [25], our prototype uses `scala-isabelle` [46], which is an unofficial library that operates headlessly wrapping lower-level constructs from Isabelle/ML[1]. This library should not be confused with Isabelle/Scala, the official Scala-based system-programming layer (PIDE, build, and server infrastructure) of Isabelle. The Isabelle developers have provided feedback on our work's implementation. The tooling is iterating toward tighter integration via the officially recommended channels. Thus, this report is best read as a snapshot of an evolving prototype.

[1] Henceforth, ML continues abbreviating "machine learning"; the sole exception is "Isabelle/ML", the augmented version of the Standard ML programming language in which Isabelle is implemented, which is unrelated to machine learning.

2 Data extraction

This section describes the data-extraction part of the toolkit: a collection of APIs, written in Isabelle/ML and Scala, that traverse an Isabelle session, identify all proofs in its `thy` files, and output a JSON representation of each proof step, including the user-seen proof state, the action taken, the available proof methods, and per-step typing information. The algorithm follows Isabelle's delegation of mathematical reasoning to the ML layer: extraction functions are written in Isabelle/ML and exposed to Scala via the `scala-isabelle` library [46].

Implementation Using Isabelle/ML's `thy`-file parser, the extraction generates a list of top-level *transitions*: user commands (e.g. `apply blast`) together with their resulting proof states. Our APIs augment each transition with the user-text that produced it, forming an *action* a_i[2].

Our implementation uses Isabelle's keyword classification to delimit proofs, definitions and other context blocks by identifying their starting and ending keywords in the `thy` file. The classification of commands in the data extraction presented here delegates to Isabelle/ML's own `Keyword.command_kind` function, which maps each command name to a category such as `prf_script`, `thy_goal`, or `qed`. Our APIs allow for the retrieval of these context blocks. In particular, they can produce a sequence of proofs $\langle p_j \rangle_{j \in J}$ from the sequence of the `thy`-file's actions $\langle a_i \rangle_{i \in I}$, where each proof is itself a sequence of actions.

To achieve this, our algorithm requires an initial `Theory` for each `thy` file. Our toolkit builds an import-dependency graph via `scala-isabelle`, which enables manipulating Scala representations of Isabelle/ML `Theory` values and resolves ROOT-file dependencies (even those outside the current directory). Each node in the graph supplies the initial top-level state for its corresponding `thy` file. In-memory manipulation of `Theory` constructs is specific to `scala-isabelle` and not the recommended Isabelle workflow, which processes theory sources and their edits. This dependency is slated for removal in future versions.

Finally, this work augments our above-defined actions a_i with the (previous) top-level state s_{i-1}, the proof's goal, the user-available methods at that point, the list of previous theorems used in the proof, and the constants, variables, and type variables in each proof step. Each step therefore carries partial type information: free variables and constants are listed with their types, and type variables with their sorts, but the string representing the user-seen goal is not accompanied by a fully typed internal AST. Pretty-printing this information into formatted strings yields the JSON object representation of each proof. This representation is user-syntax-oriented: terms, types, and hypotheses are serialised via Isabelle's pretty-printer, which is context-dependent and may produce different strings for the same internal object depending on notation imports. A more stable, canonical term representation is left as future work.

API example The data-extraction algorithm is accessible from the Scala interactive shell or via `sbt "run /read/dir/ /write/dir/"`. A `Writer` object

[2] Here, the term *action* follows RL terminology. A glossary of cross-disciplinary terms (e.g. *action*, *evaluation*, *tactic*, `scala-isabelle`, Isabelle/Scala) is in Appendix B.

wraps a `Minion` (an Isabelle process manager) and exposes `write_data` (single file) and `write_all` (full session) methods. A code example is in Appendix C.2.

Proof data example An example of a generated JSON object is provided in Appendix C.3. Each generated file includes the proof's steps, available Isar keywords, apply-style keywords, methods (`methods`), and lemmas the proof depends on (`deps`). Additionally, each proof step includes the user `action`, the user state before said action, the proof-obligation as an Isabelle/ML `term`, extra user-added hypothesis `hyps`, previously `proven` terms within the proof, `variables`, `constants`, and `type variables` in that step. More examples appear in the accompanying Zenodo artefact [33].

Related Isabelle infrastructure Unlike `isabelle dump`, which exports static binary proof terms, the extraction here captures the live interactive trace, making it the natural source for next-step predictors (see Appendix C.1 for a detailed comparison). Partly inspired from our implementation, Isabelle developers have added the `record_theories` build option, which instructs `isabelle build` to record intermediate theory commands and results, enabling post-hoc retrieval via `Thy_Info.get_theory_segments`. This offers an alternative for extraction-only use cases, though it cannot support the interactive evaluation in Section 5.

3 Data interaction

Most popular machine learning (ML) libraries are often Python frontends for efficient implementations of training algorithms and ML models. To transparently connect such models with the APIs from Section 2, this work uses the Py4j library [15] to connect Python with the Java Virtual Machine. This library enables Python programs to access and manipulate objects defined in Scala. Thus, following the same approach as in Section 2, one can define Isabelle/ML constructs that can be accessed by Scala and Python, enabling the interaction between Isabelle and the ML models. Specifically, the solution described here comprises read-eval-print loops (REPLs) in Scala and Python.

Implementation The REPL's state is implemented in Isabelle/ML. It is a stack of records of an integer, an action, a top-level state and an error. Each function at this level ensures that the initial record is always in the stack. Two kinds of functions are provided: the expected operations such as `init`, `read`, `eval`, `print`, `reset`, `undo`, and information-retrieval functions such as `last_proof_of`, `is_at_proof`, or `last_error`. The `read` function receives a string and a REPL state, converts the input string into actions, and generates the accompanying top-level states and errors. The `eval` function adds these generated data to the stack. Crucially, `read` has a timeout of 10 seconds for each action. If no toplevel state is returned after 10 seconds, `read` reports a timeout error and uses the original toplevel state as the REPL's latest toplevel state. This bound discourages models from producing actions that take too long, favouring more efficient and explainable proofs. The integer on each record in the stack models the number of user inputs and is relevant in, for example, the `undo` function. A user may input the Isabelle text representing a lemma with its proof to `read`. While such

behaviour becomes many Isabelle transitions, for the REPL, it is only one user input. Thus, `undo` will remove all records numbered with the latest user input and go back to the state before the input lemma.

Usage Our REPL initialisation requires providing, at most, a "logic" and a "`thy` file" to the REPL constructor. The "logic" is really the name of an Isabelle session. The "`thy` file" is the name of a file in the scope of the input logic. If neither a logic nor a `thy` file is provided, the REPL defaults to importing `Main` under the name `Scratch.thy`, mirroring Isabelle's behaviour when opened without a project. As before, the imports dependency graph allows the correct retrieval of the `Theory` that the REPL needs to start. The Scala REPL exposes both the core mutation operations—`apply`, `reset`, `undo`, and bulk undo via `undoN`—and the information-retrieval methods from the ML level (`last_action`, `last_error`, `proof_so_far`, `without_subgoals`). The Python level lifts all Scala methods to Python and adds `switch_to` for changing the active theory without restarting the Isabelle process, reducing overhead when evaluating multiple theorems from the same session. Both levels ensure that communication with Isabelle is mostly done via simple types such as booleans, strings, and integers.

Beyond the basic proof-step and backtracking operations, the REPL supports Sledgehammer integration via `call_hammer`: it invokes `get_suggestions` in `hammer.ML` with configurable provers, time limit, and suggestion count, returning a ranked list of `(prover, proof, time)` triples. This allows proof-search loops to delegate hard goals to Sledgehammer and incorporate its output as a candidate tactic step. Navigation is supported by `go_to` and `go_to_end_of`, which initialise the REPL at an arbitrary position within a `thy` file, enabling evaluation to begin mid-file without replaying the entire prefix. The `proof_so_far` operation returns the accumulated proof text for the current goal, and `save_last_proof` writes a completed proof to disk as a `thy` file.

API Example The Python REPL exposes a single `apply(cmd)` method: it sends the command string to Isabelle and returns the rendered proof state. Sequential calls build up a proof step by step; `reset()` and `undo()` allow backtracking. An annotated interactive-session example is given in Appendix D.1.

The REPL implementation only requires 232 extra lines of Isabelle/ML, 174 lines of Scala, and 249 lines of Python code, which exhibits the generality and extensibility of the toolkit. Upcoming sections showcase its usage as an evaluation tool for ML models. The current integration is built on the `scala-isabelle` library and Py4J, which provide a synchronous call–response interface on top of Isabelle. A tighter integration via Isabelle's native PIDE protocol is architecturally feasible and would be the cleaner long-term design; it is acknowledged here as a direction of ongoing work.

4 Evaluation of the toolkit as training infrastructure

Here, we showcase the toolkit's support for LLM training on Isabelle data. It supplies the extracted data together with training and evaluation loops that consume it: custom Hugging Face [23] loops for text-to-text transfer transformer (T5)

models [39,13] and adapted Unsloth loops for an instruction-tuned Gemma 3n model. The experiments serve two purposes: first, to validate that the extracted data is suitable for ML training; second, to reproduce well-established NLP findings (the advantage of fine-tuning over training from scratch) in the Isabelle domain, providing a replicable baseline for future comparisons. The toolkit's support for locally deployed models is of particular interest: training and evaluating small models entirely on the generated data, without relying on frontier APIs, demonstrates that independent scientific progress is possible with modest computational resources. Concretely, the questions addressed here are:

1. Does increasing the model size improve its performance on Isabelle data?
2. Does supplying more information about a proof state improve the models' performance?
3. Do LLMs fine-tuned on Isabelle data perform better than those only trained from scratch on it?

Data preprocessing This work provides a Python package (`proofs`) of methods to manipulate the generated JSONs from Section 2 and preprocess the data. Methods like `proofs.orig_objective_of` or `proofs.user_proof_up_to` retrieve the required information about the proof. Two versions of the data are then employed. For a given proof state, the simplest version, `s` (for state), includes the user proof up to that point and the user-seen state, both separated by a `USER_STATE` separator. The more complex data version, `spk` (for state, premises, keywords), adds the name and content of the theorems used in the proof as well as the name of the proof methods available at that point. The models then must predict the next Isabelle action (defined in Section 2). Thus, the `spk` format simplistically models a premise and method selection setup. For the T5 models, if the input is smaller than the model's input size $n = 512$ (which usually happens in the `s` data format), a pad token is added to fill the n-sized (vector) model input. If the tokenized input is larger, the extra tokens become an extra data point with padding if necessary. To avoid changing the data format too much in the `spk` format, the experiments also consider augmenting some models' input size to $n = 1024$ in the `spk` setup. These sizes follow the token-length distributions: in the `s` format the tokenised inputs reach up to 19,521 tokens with a median of 119, motivating $n = 512$, whereas the richer `spk` format reaches up to 389,680 tokens with a median of 2,674. Finally, the proofs in each `thy` file are split sequentially in file order: the first 64% are reserved for training, the next 16% for validation, and the last 20% for testing. A recommendation for fine-tuning T5 models is to add a prefix indicating the task at hand. In the T5 experiments, the string "isabelle next step: " is prefixed to all inputs to fine-tuned models; the non-T5 backends use a different instruction prompt (reproduced in Appendix E.2). The number of samples per split and data format is given in Table 4 (Appendix E.1).

Overall, the data generation files comprise 1,488 lines of Isabelle/ML code, 290 lines of Scala, and 172 lines of Python. The T5 models in this section were trained and evaluated on an initial 17 December 2024 extraction of the AFP (187,210 proofs; an evaluation test split of approximately 43,000 proofs). The

extraction algorithm was subsequently improved, yielding the 30 November 2025 extraction (222,466 proofs, no split), whose test split (51,158 proofs) is characterised in Figure 1: it shows the number of steps (transitions) that the developers required to complete the proofs. The proof-step distribution is stable across these extractions. The mode is 1 step (30.46 % of proofs require a single tactic) and the median is 2. Proofs with five or fewer steps account for approximately 61 % of the test set. A subsequent extraction over the 15 January 2026 AFP snapshot produced 241,126 proofs and is the dataset used for the evaluation of frontier models in Section 5. The automated proving evaluation restricts attention to proofs with five or fewer steps for cost reasons; the remaining 39 % of longer proofs are left for future work.

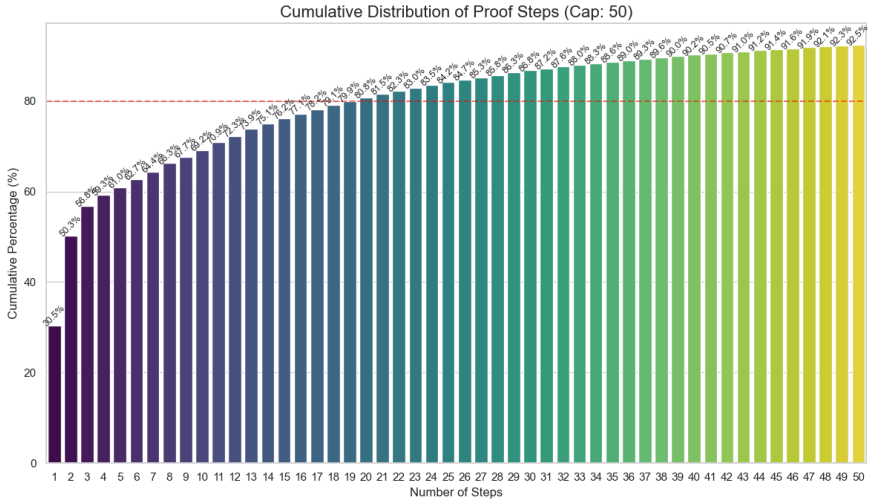


Fig. 1: Cumulative proof-step coverage for the 30 November 2025 AFP test split (51,158 proofs); dashed line at 80 %. The step-count bar chart is in Appendix F.1.

Machines specifications Training and evaluation happen on a system equipped with 8 NVIDIA Tesla V100-SXM2-32GB-LS GPUs, 503 GB of RAM and 80 CPUs. Despite these resources, data management for training is optimised so that observed RAM usage peaks at 35 GB. Moreover, a maximum of 14 processes are observed to be active during simultaneous model training and evaluation, while fully utilising the GPUs. Training loops typically consume approximately 23 GB of system RAM, separate from the GPUs' on-board memory, which is fully utilised. Partial evaluations are also possible on an M2 Pro Mac (12-core Apple Silicon, 16 GB RAM): remote-API experiments (OpenAI, Google Gemini) run there because all generation happens server-side, as does the Gemma 4 E4B Ollama run which fits in 16 GB at 4-bit quantisation. Fine-tuned models larger than a few billion parameters require a dedicated GPU.

Models For training, this work uses several text-to-text transfer transformer (T5) models [39,13]. The T5 models have an encoder-decoder architecture, very similar to that of the original transformer [47] with minor changes [39]. For computational and time constraints, this work focuses on the `small` (77M parameters) and `base` (248M parameters) versions of the `google/flan-t5` family of models, leaving the `large` (783M), `xl` (2.85B), `xxl` (11.3B) variations out of the scope of the small-scale experiments here. Beyond T5, the toolkit's training infrastructure also supports the fine-tuning of instruction-tuned models via Unsloth [20]: specifically, Gemma 3n E2B [43] (2B effective parameters, 4-bit quantised) was fine-tuned using quantised Low-Rank Adaptation (QLoRA) [17] adapters (0.19% of 5.45B parameters trained) on the `state_prems` data format (`sp`), demonstrating that the extracted data can also serve as fine-tuning supervision for recent architectures. In addition, several frontier models are evaluated in Section 5 without training or further fine-tuning: GPT-5.3-chat (OpenAI), Gemini 3 Flash Preview and Gemini 2.5 Flash Lite (Google), and Gemma 4 E4B (local via Ollama). These are included to place the T5 and Gemma 3n results in context. All T5 models come from the Hugging Face library [23]. A rigorous contamination check — verifying whether AFP-specific identifiers or Isabelle syntax appear in the models' pre-training corpora — was not performed and is a limitation of this evaluation. In particular, the training corpora of the Gemma and proprietary backends are undisclosed, so contamination cannot be ruled out for them. As a qualitative indicator only: the reported datasets for the fine-tuned T5 models are `openai/gsm8k` [14] (grade-school math word problems), `deepmind/code_contests` [31] (competitive programming in Python, Java, C, C++), and `cimec/lambada` [38] (English-language fiction) — none of which are expected to contain Isabelle content. It is likely, that frontier models such as GPT-5.3 or Gemini 3 have seen AFP proofs during pre-training.

Training setup This work uses Hugging Face's `accelerate` library for distributed training and evaluation. It usually employs 4 processes and 4 GPUs for the `small` models and 8 processes and 8 GPUs for the `base` models. Due to GPU memory constraints, the `small` models can use batch sizes of 8, while `base` models are limited to batch sizes of 4. The training uses an `AdamW` optimiser with a constant learning rate of 0.00001 and a weight decay of 0.01. All training loops measure the standard categorical cross-entropy loss (over the token vocabulary). Since the experiments involve training the T5 models from scratch with Isabelle data, the original T5 tokenizer (SentencePiece) [28] is trained on all generated JSON files, producing a vocabulary size of 121,719 tokens. This work's T5 models use the newly trained tokenizer while the fine-tuned models use Hugging Face's pre-trained tokenizer for T5. Because the two tokenizers differ, the same proofs yield different sample counts: the generic Hugging Face tokenizer splits Isabelle text into more tokens than the domain-trained one, so more inputs exceed the model's maximum length and are divided into additional overflow samples (cf. Table 4). The number of underlying proof steps is unchanged. The Gemma 3n E2B fine-tuning used a different stack: Unsloth's `SFTTrainer` with QLoRA (rank 16, alpha 16, targeting all linear layers), an `AdamW_8bit` optimiser,

a linear learning-rate schedule with peak 2×10^{-4}, and a 1-device batch size of 1 with 8 gradient-accumulation steps (effective batch 16).

Training In a first run of the experiment, all T5 models are trained for at most four epochs (one epoch = one pass through each AFP entry's training samples), except for the `spk` longer-context configurations whose larger inputs made each epoch substantially more expensive. A second final replication was then run with a smaller, format-dependent number of epochs. The epoch counts are reported in Table 5 (Appendix E.3). Since the recorded step counts depend on sequence length, batch size, and distributed batching, the replication is interpreted by epoch rather than by step count. Loss and accuracy curves for the representative first-run `t5_small_finetune_s` model are given in Appendix E.4. In the replication, its validation perplexity decreased from 4.04 to 3.30 in four epochs. Finally, Gemma 3n E2B was fine-tuned for approximately 231 hours (75,320 steps, ≈2 effective epochs over the `sp` training split; see Table 6). Following the default Unsloth configuration, and to save training time, no separate validation run was performed for this fine-tuning. The Gemma 3n training loss curve shows rapid convergence from ∼1.0 to ∼0.4 in the first 20% of steps, stabilising thereafter and a final train perplexity of 1.5.

Evaluation Table 5 (Appendix E.3) shows the final available training metrics for all replicated T5 models, and Table 7 shows validation perplexity by epoch for equal-data-exposure comparison. These results are not presented as a definitive model ranking; the experimental conditions (data formats, training budgets, epoch counts) are not fully controlled across runs and the sample sizes are insufficient for strong comparative conclusions. Rather, the purpose is to exercise the toolkit across several configurations and record the qualitative patterns that emerge. With that caveat, the results for `t5_small_s` versus `t5_base_s` (both using the "state" format, question **1**) suggest that larger models modestly reduce loss and improve accuracy, though the difference is small compared to the effect of fine-tuning. Interestingly, by comparing `t5_small_s` and `t5_small_spk` one may suspect that increasing the amount of data per example makes the models perform worse after seeing approximately the same amount of training examples (question **2**). Overall, training the models from scratch produces higher losses and less accuracy in the same amount of epochs than fine-tuning them. This adds evidence in support of answering affirmatively to question **3** echoing lessons from the ML literature where the models have more difficulty predicting patterns in the data when they are not pre-trained on a diverse dataset [39]. Notably, `t5_small_spk_trim` has the worst validation loss and accuracy, which suggests that overflowing tokens as done in that case makes training ineffective as opposed to providing the proof context in a single example (`t5_small_spk`). The Gemma 3n E2B fine-tuning achieved a final training loss of 0.404 without a validation pass; its proof-completion performance is reported in Section 5. The low training loss (comparable to `t5_small_finetune_s`) despite the absence of validation monitoring suggests that the toolkit's data format is compatible with instruction-tuned models and that QLoRA fine-tuning on Isabelle data is feasible with modest compute.

Table 1 shows the models' performance on the test set according to three metrics. The first column corresponds to the ratio of examples where the models' (best-first) predictions exactly coincided with the user actions. The second column shows the ratio of examples where the first generated token matched the first token of the ground-truth user action. Since the first token of an Isabelle proof step typically corresponds to a command keyword (e.g. `apply`, `have`, `by`), this metric measures the model's awareness of the current proof stage rather than complete action correctness. Lastly, the third column indicates the ratio of examples where the models' predictions are entirely incorrect. Note that the three columns are not mutually exclusive: a prediction that is entirely wrong (third column) trivially fails the first two, but a prediction that matches the first token (second column) may still fail the exact-match criterion (first column).

Model	**Test Exact Acc.**	**Test 1st-Tok Acc.**	**Test All Wrong**
t5_small_s (4ep)	0.137	0.316	0.303
t5_small_finetune_s (4ep)	0.150	0.302	0.231
t5_small_spk (2ep)	0.042	0.254	0.553
t5_small_spk_trim (4ep)	0.022	0.194	0.371
t5_base_s (4ep)	0.170	0.307	0.299

Table 1: Testing model Performance

Consistently with the training results, the fine-tuned version of the small model achieves the most robust set of values. However, the difference between this version and the `t5_small_s` and `t5_base_s` models trained from scratch is not as pronounced as in the training and validation sets. In particular, the larger model performs slightly better on predicting exact user actions or first tokens, while the simplest `small` model is the best one on the first token prediction metric. These results slightly reinforce the previous evidence towards answering questions **1** and **3**. That is, using larger models and fine-tuning, slightly improves model performance. However, it does not decisively lead to pronounced differences in the models' generalisation capabilities. With respect to supplying more data (question **2**), both the `spk` and `spk_trim` variations consistently showed lower exact accuracy and higher ratios of wrong predictions on the test set. Overall, the values in Table 1 show that the toolkit can support fine-tuning and that domain-specific fine-tuning consistently outperforms training from scratch in this small-scale exercise, echoing the familiar lesson from the natural language processing (NLP) literature. These results serve as a sanity check that the data extraction and training pipelines work as intended, rather than as a definitive comparison between model configurations. The Gemma 3n E2B fine-tuned model was not evaluated on these token-level metrics: token-level evaluation served only as an exploratory diagnostic for the T5 models and, since it added little predictive value for proof completion—the primary objective (see next section)—the evaluation budget was spent on proof completion instead. Its proof-completion performance is reported in Section 5.

5 Evaluation of the toolkit for automated proving

While Section 4 tested the toolkit's capabilities as training infrastructure, this section tests it as an evaluation setting: specifically, it evaluates the performance of a next-step prediction strategy, implemented as a depth-first search loop over the REPL, across multiple model backends. The strategy is compared against Sledgehammer as an ATP reference, and the results are accompanied by a qualitative analysis of the produced proofs.

Strategy abstraction Our APIs enable the definition of interchangeable modules representing proof-search strategies in its subdirectories. The current prototype has two: `dfs_strategy` (the depth-first next-step prediction loop evaluated in this section) and `hammer_strategy` (a wrapper around Sledgehammer). Both share a common interface defined in our `shared.py` file. This design makes it straightforward to add alternative strategies, for example a *Draft, Sketch, Prove* [27] style approach that first generates a high-level proof sketch and then fills in tactic steps, or a best-first search variant. The present evaluation focuses on a next-step prediction `dfs_strategy` across five model backends.

Implementation Starting from a configuration file (config.json), the loop loads one of the models from Section 4, traverses the proofs on a split of the data set and makes the model predict the next steps based on the replies from Isabelle. The REPL loads the appropriate Isabelle context before the lemma-to-prove via its `go_to` method. In a depth-first search setting, when a proof is started, the evaluation loop retrieves the top-k model predictions using beam search (with $k = 5$ in all T5 runs), sends each candidate to Isabelle in order of decreasing beam score, and accepts the first one that does not produce an error. If the incoming top-level state is accompanied by an error, the loop counts this as a *failed step*, backtracks the REPL to a state before the error, and tries the next prediction. The models sometimes recommend using Isabelle's `by` command due to its frequency in the data. Since the `by` command should only be used to conclude proofs, and thus, hides the errors produced by the method employed, the loop replaces these `by` commands with `apply` commands. If a no-subgoals state is detected after the application of this replacement, the loop backtracks and applies the original model's produced `by`-prediction, and counts this successful *`by`-application*. If the REPL detects that the proof is *finished*, the loop retrieves it from the REPL and saves it in a `thy` file. If the `apply`-version of the model prediction does not produce an error and does not finish the proof, the loop counts this as an *incorrect `by`-application*. Nevertheless, non-error-producing actions are recorded as *syntactically accepted* by Isabelle and the loop continues until a fixed action depth is reached. Note that a syntactically accepted action does not necessarily advance the proof semantically: the proof state may remain unchanged (e.g. after a redundant `using` clause) or may even become harder to close. The term "progress" in Table 2 refers to syntactic acceptance only. Henceforth, "proof depth" means the number of sequential actions applied from the start of the proof; it does not refer to nested Isar block structure. In all T5 experiments, the model generates 5 candidate predictions and the maximum depth

explored is 5 actions. Yet, for time, memory, and budget constraints the models did not tackle the same number of proofs.

To exercise `dfs_strategy` across the full range of supported backends, more recent evaluations were run on the January 2026 AFP test split (approximately 20% of proofs, filtered to at most 5 steps) using the prototype's orchestrator infrastructure (supporting OpenAI, Gemini, Ollama, HuggingFace, and Unsloth backends). Each run used the `spk` format unless noted, chosen because it gave a modest proof-completion improvement over `s` (e.g. from 7.13% to 8.81% for `t5_small`) and supplies the richest context, even though it did not improve the token-level metrics of Section 4. Experimental conditions differ per run and are documented in the `config.json` files. Table 2 consolidates the results for all backends alongside a Sledgehammer baseline run via `hammer_strategy` on the same filtered split. GPT-5.3-chat was run on 300 proofs (3 candidates/step); Gemini 3 Flash Preview on 1,833, Gemini 2.5 Flash Lite on 2,807, and Gemma 4 E4B on 1,330 (all 5 candidates/step). The Gemma 3n E2B fine-tuned model used the `sp` format with depth 10 on 252 proofs. Because the backends were run on different hardware, on different AFP snapshots and entry subsets, and with different candidate budgets, Table 2 should be read as a demonstration that the toolkit supports diverse backends rather than as a controlled ranking. At the lemma level, only 26–30% of the frontier-model proofs overlap with the T5 test set. As a partial control, Appendix F.2 restricts the three frontier backends with nested entry sets to the proofs they all attempted and recomputes their completion rates on that common set. A fully controlled benchmark—same lemma set, snapshot, and candidate budget—is left as future work.

The three best-performing model backends (GPT-5.3-chat, Gemini 3 Flash Preview, Gemma 3n fine-tuned) all exceed the T5-base fine-tune `spk` baseline, demonstrating that the toolkit produces training data usable by both small fine-tuned models and large off-the-shelf ones. The Sledgehammer baseline (46.26%) was run via `hammer_strategy` with `hammer_params = []`. Thus, it passes an empty override list to `Sledgehammer_Commands.default_params` and thereby uses Isabelle 2025-2's default Sledgehammer parameters with no modifications—equivalent to invoking it in the PIDE without options.

Across all backends, the majority of completed proofs are single-step `by simp` or `by auto` invocations. Representative complex completions from the LLM backends — including locale discharging, rule instantiation, and structured Isar proofs — are reproduced in Appendix G.1. A detailed analysis of T5 prediction patterns is given in Appendix G.2.

The frontier models and fine-tuned Gemma 3n in Table 2 complete a larger fraction of proofs on the same $\leq$5-step slice, confirming that the REPL loop scales to diverse backends and that model capability matters more than search depth in this regime. Many of the completed proofs were found in less than 10 seconds, but `t5_small_s`' average time for processing proofs is 48.92 seconds barely using 1 GPU for the model generation and less than 12 GB of RAM. This suggests that spending time exploring the proof tree is not conducive to proof

Model	Backend	Proofs	Predictions	Progress	by cmds	Correct by	Finished	Finished %
small_s	HugFace GPU	43,538	7,426,213	61.58%	187,683	42.02%	3,103	7.13%
small_finetune_s	HugFace GPU	42,408	3,508,777	37.79%	162,986	8.40%	4,784	11.28%
small_spk	HugFace GPU	43,218	2,939,773	48.82%	55,195	14.76%	3,808	8.81%
base_s	HugFace GPU	43,001	5,955,871	59.45%	101,268	25.36%	4,090	9.51%
base_finetune_spk	HugFace GPU	38,532	2,315,016	52.57%	118,959	36.70%	6,438	16.71%
GPT-5.3-chat	OpenAI API	300	1,334	37.03%	164	29.27%	104	34.67%
Gemini 3 Flash Preview	Gemini API	1,833	1,351	51.74%	580	68.79%	430	23.46%
Gemma 3n E2B fine-tuned	Unsloth (GPU)	252	4,697	31.17%	351	15.38%	59	23.41%
Gemini 2.5 Flash Lite	Gemini API	2,807	18,264	19.49%	286	10.14%	55	1.96%
Gemma 4 E4B	Ollama (local)	1,330	6,191	4.36%	0	—	14	1.05%
Sledgehammer	Isabelle (ATP)	34,223	32,931	50.18%	—	—	15,832	46.26%

Table 2: REPL-based proof completion. T5 models (first group) used the 17 December 2024 AFP split (Section 4); LLM backends and Sledgehammer (second and third groups) used the January 2026 AFP test split (≤5 steps). **Proofs** = proofs attempted per run; **Predictions** = total tactic steps generated (or hammer invocations for Sledgehammer); **Progress** = fraction of Predictions syntactically accepted by Isabelle (for Sledgehammer: fraction of hammer calls returning "Used:"); **by cmds** = total `by`-predictions converted to `apply` for safety-checking; **Correct by** = fraction of those that closed the goal; **Finished** = proofs fully closed; **Finished %** = Finished relative to Proofs.

completion for the `t5_small_s` model. Yet, more work is required to conclusively analyse these preliminary qualitative results.

The **by cmds** and **Correct by** columns of Table 2 reveal a qualitative split in proof-closing mechanisms. Among the T5 models, `by`-conversions account for a small fraction of all predictions, with notable differences in the Correct by rate across configurations (detailed breakdown in Appendix G.2). By contrast, the by-conversion mechanism is the dominant completion path for the LLM backends: Gemini 3 Flash Preview closed 399 of its 430 proofs (92.8%) via a by-conversion, Gemma 3n E2B fine-tuned 54 of 59 (91.5%), and GPT-5.3-chat 48 of 104 (46.2%). These models generate `by simp` or `by auto` as their idiomatic single-step proof-closing form, which the loop converts to `apply` for safety-checking before restoring the `by` form on success. The exception is Gemma 4 E4B via Ollama, which never generated a `by`-prefixed prediction, consistent with that model's very high syntactic-rejection rate (5,921 failed predictions versus 270 progressing).

Concerning model size, improving the performance of the models (question **1**), Table 2 shows consistent results with Section 4. Namely, model size seems to improve performance but the increased percentage of finished proofs is not as pronounced as that of fine-tuning. Yet, contrastingly with Section 4, the values in Table 2 support the notion that increasing the amount of data seen per sample helps models perform better (question **2**). Regarding question **3**, the comparison between Gemma 3n E2B fine-tuned (23.41%) and Gemma 4 E4B off-the-shelf (1.05%)—both evaluated on the same $\leq$5-step slice—provides the most direct test within the new backend runs: the older, fine-tuned model completes more than 22 times as many proofs, strongly reinforcing the fine-tuning advantage found in Section 4. However, the two models differ in size and architecture, so the comparison conflates fine-tuning with model capability; a controlled study fine-tuning Gemma 4 E4B on the same data is left as future work.

A breakdown of proof methods used in completed proofs across all backends is given in Appendix G.2, where `simp` and `auto` dominate across all models, while fine-tuned and frontier models show a slightly broader method repertoire. Per-proof wall-clock times are reported in Appendix F.3.

This section evidences that the toolkit's previously described data extraction and data interaction of Sections 2 and 3 respectively are robust tools for training and evaluating ML models. The toolkit is openly available and an artefact snapshot is provided [33].

6 Limitations

The foundation for the toolkit's limitations is a single architectural choice: the current version interacts with Isabelle through the unofficial `scala-isabelle` library rather than through Isabelle/Scala and Isabelle's official PIDE document model. We group the limitations accordingly, from this overall-approach issue down to its concrete symptoms below.

Ecosystem integration. The `scala-isabelle` library interacts with Isabelle via headless command-line processes and exposes in-memory `Theory` ob-

jects, whereas Isabelle natively treats proof development as the processing of theory sources and their edits through a PIDE document model. This affects the toolkit's contributions as follows. As seen in Section 2, the data extraction reimplements source handling. For instance, proof and context-block boundaries are recovered with a keyword heuristic and ROOT files with a regular expression, instead of using Isabelle's official outer-syntax parsing and session infrastructure. More native routes could use Isabelle's own build-time theory recording (`record_theories`, `Thy_Info.get_theory_segments` or consume PIDE document snapshots as Isabelle/dump does. The data interaction of Section 3 is a bespoke state-stack REPL over `scala-isabelle` rather than a client of the PIDE protocol: this suffices for the next-step-prediction experiments reported here, but it is not in itself a prelude to deeper integration, for which targeting PIDE directly is the cleaner long-term design. A later, not-yet-public version of the toolkit removes the `scala-isabelle` dependency and the keyword-based delimitation as first steps in this direction [22].

Data representation. All data transfers between Isabelle and the ML layer pass through Isabelle's pretty-printer into formatted strings, trading robustness and efficiency for accessibility and portability to ML tooling. Since pretty-printing is context-dependent, the same internal term may serialise to different strings, so the format is not a canonical representation. Isabelle's structured export together with a canonical term encoding could be more robust, however, optimised token-level encoding may also alter ML generalisation [41] and thus the effects of this design choice is left for future work.

Usability and adoption. Ease of use is a primary determinant of whether a tool is adopted [16]. The prototype currently raises this barrier in avoidable ways that are all symptoms of the same non-native integration: installation paths are fixed by editing a Scala source file (`directories.scala`) rather than passed as command-line options; the Python layer requires a manually launched Py4J gateway with a port-registry file; and the build depends on `sbt` and an off-tree library. Native Isabelle/Scala tooling would subsume most of these.

Evaluation scope. The empirical study is deliberately small. No controlled ablation over training choices (dataset size, tokenisation, or fine-tuning strategy) was performed, so the model results are a proof-of-concept rather than causal claims. The multi-backend results of Section 5 reflect the available compute budget rather than a uniform benchmark. Extraction is also not loss-free: on the January 2026 AFP run, 1,317 proofs exceeded the per-proof serialisation budget and 633 entries with non-typical structure defeated the keyword-delimiting approach of Section 2.

7 Conclusion and future work

The work here contributes openly available infrastructure for Isabelle-based ML research: a data-extraction algorithm that produced 241,126 proof JSONs from the AFP; Scala and Python REPLs for interactive proof-state manipulation, and a modular evaluation loop tested across T5, fine-tuned Gemma, and fron-

tier API models. The T5 experiments confirm that fine-tuning on domain-specific data substantially outperforms training from scratch; the multi-backend evaluation demonstrates the toolkit's extensibility. Source code, data samples, and model weights are available at [33]. Future directions include stronger baselines, reinforcement-learning integration, and more controlled ablations. The present work is a first project report: a reusable prototype and an invitation for Isabelle and ML researchers to build on it collaboratively.

References

1. Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., Michelsen, M., Patterson, R., Rodriguez, E., Scharff, L., Shanker, V., Sicca, V., Sowrirajan, H., Swope, A., Tamas, M., Tenev, V., Thomm, J., Williams, H., Wu, L.: Aristotle: IMO-level automated theorem proving (2025), https://arxiv.org/abs/2510.01346
2. Amazon Web Services: AutoCorrode: `iq` (MCP-based Isabelle interface). https://github.com/awslabs/AutoCorrode/tree/main/iq (2025)
3. Amazon Web Services: AutoCorrode: `ir` (Isabelle REPL). https://github.com/awslabs/AutoCorrode/tree/main/ir (2025)
4. Amazon Web Services: AutoCorrode: `isabelle-assistant`. https://github.com/awslabs/AutoCorrode/tree/main/isabelle-assistant (2025)
5. Amazon Web Services: AutoCorrode: `isabelle-proxy`. https://github.com/awslabs/AutoCorrode/tree/main/ip (2025)
6. Amazon Web Services: AutoCorrode announcement on the Isabelle users mailing list. Isabelle Users Mailing List, https://lists.cam.ac.uk/sympa/arc/cl-isabelle-users/2026-02/msg00027.html (2026), 2026-02-16
7. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C.: Learning to reason in large theories without imitation. CoRR **abs/1905.10501** (2019), https://arxiv.org/abs/1905.10501
8. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C., Wilcox, S.: HOList: An environment for machine learning of higher order logic theorem proving. In: ICML 2019. PMLR, vol. 97, pp. 454–463. PMLR (2019), http://proceedings.mlr.press/v97/bansal19a.html
9. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician - A seamless, interactive tactic learner and prover for Coq. In: Benzmüller, C., Miller, B.R. (eds.) CICM 2020. LNCS, vol. 12236, pp. 271–277. Springer (2020). https://doi.org/10.1007/978-3-030-53518-6_17
10. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. JFR **9**(1) (2016)
11. Bryant, D., Huerta y Munive, J.J., Kaliszyk, C., Urban, J.: Munkres' general topology autoformalized in Isabelle/HOL (2026), https://arxiv.org/abs/2604.07455
12. Chen, L., Gu, J., Huang, L., Huang, W., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Ma, K., Ren, C., Shen, J., Shi, W., Sun, T., Sun, H., Wang, J., Wang, S., Wang, Z., Wei, C., Wei, S., Wu, Y., Wu, Y., Xia, Y., Xin, H., Yang, F., Ying, H., Yuan, H., Yuan, Z., Zhan, T., Zhang, C., Zhang, Y., Zhang, G., Zhao, T., Zhao, J., Zhou, Y., Zhu, T.H.: Seed-Prover: Deep and broad reasoning for automated theorem proving (2025), https://arxiv.org/abs/2507.23726

13. Chung, H.W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S.S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E.H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q.V., Wei, J.: Scaling instruction-finetuned language models (2022). https://doi.org/10.48550/ARXIV.2210.11416
14. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., Schulman, J.: Training verifiers to solve math word problems (2021), https://arxiv.org/abs/2110.14168
15. Dagenais, B.: Py4j. https://github.com/py4j/py4j (2024)
16. Davis, F.D.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly **13**(3), 319–340 (1989), http://www.jstor.org/stable/249008
17. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: QLORA: Efficient finetuning of quantized LLMs. In: Advances in Neural Information Processing Systems (NeurIPS 2023). NIPS '23, Curran Associates Inc., Red Hook, NY, USA (2023)
18. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: TacticToe: Learning to prove with tactics. JAR **65**(2), 257–286 (2021). https://doi.org/10.1007/S10817-020-09580-X
19. George, R.J., Huang, S., Song, P., Anandkumar, A.: LeanProgress: Guiding search for neural theorem proving via proof progress prediction (2026), https://arxiv.org/abs/2502.17925
20. Han, D., Han, M.: unsloth. https://github.com/unslothai/unsloth (2023)
21. Hu, J., Zhang, J., Zhao, Y., Ringer, T.: HybridProver: Augmenting theorem proving with LLM-driven proof synthesis and refinement (2025), https://arxiv.org/abs/2505.15740
22. Huerta y Munive, J.J.: DeepIsaHOL (Nov 2023), https://github.com/yonoteam/DeepIsaHOL
23. Hugging Face: Hugging face – the ai community building the future., https://huggingface.co/, online; accessed 23-March-2025
24. Ilin, V.: Semi-autonomous formalization of the Vlasov-Maxwell-Landau equilibrium (2026), https://arxiv.org/abs/2603.15929
25. Jiang, A.Q., Li, W., Han, J.M., Wu, Y.: LISA: Language models of ISAbelle proofs. 6th Conference on Artificial Intelligence and Theorem Proving (2021)
26. Jiang, A.Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygózdz, T., Milos, P., Wu, Y., Jamnik, M.: Thor: Wielding hammers to integrate language models and automated theorem provers. In: NeurIPS 2022 (2022), http://papers.nips.cc/paper_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html
27. Jiang, A.Q., Welleck, S., Zhou, J.P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., Wu, Y.: Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In: ICLR 2023. OpenReview.net (2023), https://openreview.net/pdf?id=SMa9EAovKMC
28. Kudo, T., Richardson, J.: SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing (2018), https://arxiv.org/abs/1808.06226
29. Kumarappan, A., Tiwari, M., Song, P., George, R.J., Xiao, C., Anandkumar, A.: LeanAgent: Lifelong learning for formal theorem proving (2025), https://arxiv.org/abs/2410.06209

30. Lamont, S., Walder, C., Dezfouli, A., Montague, P., Norrish, M.: 3D-Prover: Diversity driven theorem proving with determinantal point processes (2025), https://arxiv.org/abs/2410.11133
31. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D.J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., Vinyals, O.: Competition-level code generation with AlphaCode. Science **378**(6624), 1092–1097 (Dec 2022). https://doi.org/10.1126/science.abq1158
32. Mikula, M., Antoniak, S., Tworkowski, S., Jiang, A.Q., Zhou, J.P., Szegedy, C., Kucinski, L., Milos, P., Wu, Y.: Magnushammer: A transformer-based approach to premise selection. CoRR **abs/2303.04488** (2023). https://doi.org/10.48550/ARXIV.2303.04488
33. Huerta y Munive, J.J.: Snapshot of contributions from the DeepIsaHOL project (Mar 2025). https://doi.org/10.5281/zenodo.15080049
34. Nagashima, Y.: SeLFiE: Modular semantic reasoning for induction in Isabelle/HOL. CoRR **abs/2010.10296** (2020), https://arxiv.org/abs/2010.10296
35. Nagashima, Y.: Faster smarter proof by induction in Isabelle/HOL. In: Zhou, Z. (ed.) IJCAI 2021. pp. 1981–1988. ijcai.org (2021). https://doi.org/10.24963/IJCAI.2021/273
36. Nagashima, Y., He, Y.: PaMpeR: proof method recommendation system for Isabelle/HOL. In: Huchard, M., Kästner, C., Fraser, G. (eds.) ASE 2018. pp. 362–372. ACM (2018). https://doi.org/10.1145/3238147.3238210
37. Paliwal, A., Loos, S.M., Rabe, M.N., Bansal, K., Szegedy, C.: Graph representations for higher-order logic and theorem proving. CoRR **abs/1905.10006** (2019), http://arxiv.org/abs/1905.10006
38. Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N.Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., Fernandez, R.: The LAMBADA dataset: Word prediction requiring a broad discourse context. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1525–1534. Association for Computational Linguistics, Berlin, Germany (August 2016), http://www.aclweb.org/anthology/P16-1144
39. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research **21**(140), 1–67 (2020), http://jmlr.org/papers/v21/20-074.html
40. Reichel, T., Henderson, R.W., Touchet, A., Gardner, A., Ringer, T.: Proof repair infrastructure for supervised models: Building a large proof repair dataset. In: Naumowicz, A., Thiemann, R. (eds.) ITP 2023. LIPIcs, vol. 268, pp. 26:1–26:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ITP.2023.26
41. Rute, J., Olsák, M., Blaauwbroek, L., Massolo, F.I.S., Piepenbrock, J., Pestun, V.: Graph2Tac: Learning hierarchical representations of math concepts in theorem proving. CoRR **abs/2401.02949** (2024). https://doi.org/10.48550/ARXIV.2401.02949
42. Song, P., Yang, K., Anandkumar, A.: Towards large language models as copilots for theorem proving in Lean. arXiv preprint arXiv: Arxiv-2404.12534 (2024)
43. Team, G.: Gemma 3 (2025), https://goo.gle/Gemma3Report

44. Towers, M., Kwiatkowski, A., Terry, J., Balis, J.U., Cola, G.D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Tan, H., Younis, O.G.: Gymnasium: A standard interface for reinforcement learning environments (2024), https://arxiv.org/abs/2407.17032
45. Towers, M., Terry, J.K., Kwiatkowski, A., Balis, J.U., Cola, G.d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Shen, A.T.J., Younis, O.G.: Gymnasium (Mar 2023). https://doi.org/10.5281/zenodo.8127026
46. Unruh, D.: scala-isabelle. https://github.com/dominique-unruh/scala-isabelle (2024)
47. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need (2023), https://arxiv.org/abs/1706.03762
48. Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M.D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., Li, J.: Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning (2025), https://arxiv.org/abs/2504.11354
49. Wu, M., Norrish, M., Walder, C., Dezfouli, A.: TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) NeurIPS 2021. pp. 9330–9342 (2021), https://proceedings.neurips.cc/paper/2021/hash/4dea382d82666332fb564f2e711cbc71-Abstract.html
50. Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., Anandkumar, A.: LeanDojo: Theorem proving with retrieval-augmented language models. In: Neural Information Processing Systems (NeurIPS) (2023)
51. Zhang, Y., Lee, J.D., Liu, F.: Statistical learning theory in Lean 4: Empirical processes from scratch (2026), https://arxiv.org/abs/2602.02285
52. Zhao, X., Li, W., Kong, L.: Subgoal-based demonstration learning for formal theorem proving. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net (2024), https://openreview.net/forum?id=pSnhA7Em1P
53. Zheng, K., Han, J.M., Polu, S.: MiniF2F: a cross-system benchmark for formal olympiad-level mathematics. arXiv preprint arXiv:2109.00110 (2021)

A Toolkit architecture

Figure 2 summarises the toolkit and its three contributions. The architecture is a three-layer, linearly ordered stack: an Isabelle/ML layer that retrieves data and drives the prover, a Scala layer that hosts a headless Isabelle process through `scala-isabelle`, and a Python layer that exposes Isabelle's constructs through Py4J. Two Scala files act as the inter-layer intermediaries: `minion.scala` bridges Isabelle/ML and Scala, and `py4j_gateway.scala` bridges Scala and Python. Cutting across the three layers are the three contributions—(a) dataset extraction, (b) the REPL, and (c) the evaluation loops—each realised by the source files named in the corresponding cell. The toolkit connects two endpoints: the ML models (top) and the Isabelle prover (bottom).

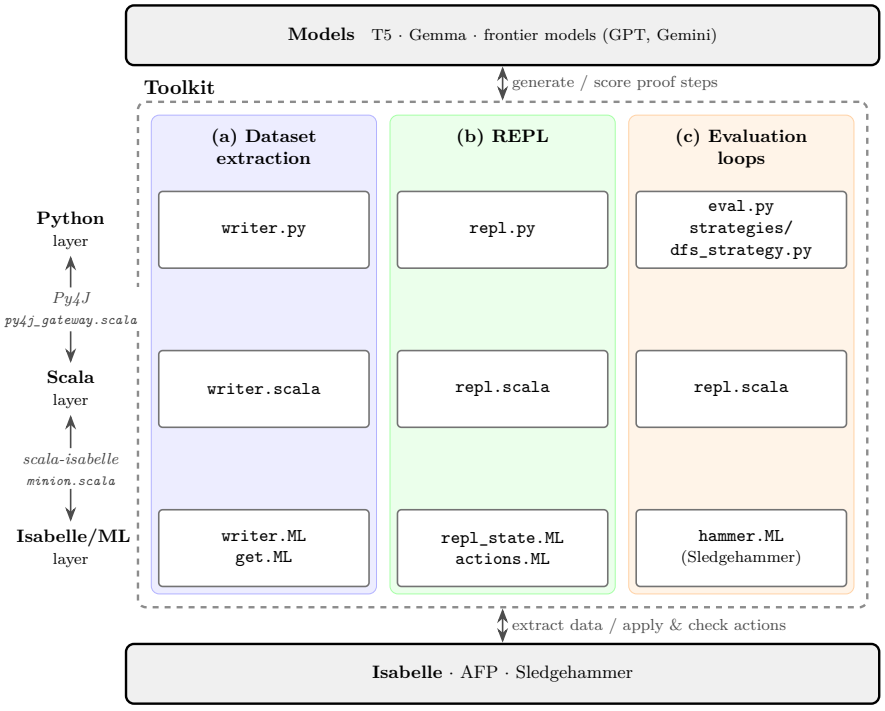


Fig. 2: Schematic of the toolkit. Rows are the three architectural layers (Python / Scala / Isabelle/ML) connected by the `py4j_gateway.scala` and `minion.scala` intermediaries; columns are the three contributions (dataset extraction, the REPL, the evaluation loops), each labelled with its key source files. The toolkit connects the ML models (top) to the Isabelle prover (bottom).

B Glossary

For readers from either the theorem-proving or the machine-learning community, we clarify a few cross-disciplinary terms used throughout.

Action. A single user command together with the user-text that produced it and the resulting top-level state (Section 2). It is the atomic unit of a proof in our data representation.

Evaluation. Used in the machine-learning sense—measuring a model's performance, e.g. next-step prediction accuracy or proof-completion rate—and, by extension, exercising the toolkit as training and proof-search infrastructure. It never refers to evaluating Isabelle/ML expressions.

Tactic. A proof-state-transforming step. In Isabelle such steps are invoked through *proof methods* (e.g. `simp`, `auto`, `blast`); following machine-learning usage we also call a single such step a *tactic*.

`scala-isabelle`. An unofficial Scala library [46] that drives an Isabelle process headlessly and exposes Isabelle/ML values to Scala. It is distinct from Isabelle/Scala and is a dependency the toolkit is moving away from.

Isabelle/Scala. The official Scala-based system-programming layer of Isabelle; not to be confused with `scala-isabelle`.

C Data extraction

The following subsections supplement Section 2: a feature comparison with `isabelle dump` (§C.1), an annotated Scala extraction example (§C.2), and an abbreviated proof JSON (§C.3).

C.1 Comparing `Isabelle/dump` with our extractor

Table 3: Outputs of `isabelle dump -A theory` and our extractor.

Exported item	`isabelle dump`	**our APIs**
All theory theorems with full proof terms	✓	—
All theory constant declarations (with types)	✓	—
Types, axioms, locales, class hierarchies	✓	—
Term encoding	Isabelle-specific pretty-printed strings	
Per-step action text	—	✓
User-visible proof state	—	✓
Goal term (pretty-printed)	—	✓
In-scope hypotheses	—	✓
Per-step free variables in goal (with types)	—	✓
Per-step constants in goal (with types)	—	✓
Available proof method names	✓ (global)	✓ (per-proof)
Apply / Isar keyword classifications	—	✓
Proof-used theorem names and statements	—	✓

C.2 Data extraction API example

The snippet below shows how to use the data-extraction algorithm from the Scala interactive shell. Alternatively, run `sbt "run /read/dir/ /write/dir/"` from the project root. Each `Writer` wraps a `Minion` (managing the Isabelle process) and exposes `write_data` (single file) and `write_all` (full session). Call `isabelle.destroy()` to release the process.

```
import isabelle_rl._
import de.unruh.isabelle.control.Isabelle

val logic = "LTL"                       // Isabelle session
    name

val writer = new Writer(
    "/path/to/a/read/directory/",        // directory with
        ROOT or .thy files
    "/path/to/a/writing/directory/",
    logic)

writer.write_data("your/file.thy")      // extract one file
writer.write_all()                      // extract whole
    session

val minion = writer.get_minion()
implicit val isabelle: Isabelle = minion.isabelle
isabelle.destroy()                      // release the
    Isabelle process
```

Listing 1.1: Data extraction via the Scala API

C.3 JSON proof data example

The listing below abbreviates a generated JSON object representing a proof from the LTL entry in the AFP.

```
{
  "proof": [
    {
      "action": "lemma semantics_ltlc_sugar [simp]:\n  \"\\<
          xi> \\<Turnstile>\\<^sub>c \\<phi> iff\\<^sub>c \\<
          psi> = (\\<xi> \\<Turnstile>\\<^sub>c \\<phi> \\<
          longleftrightarrow> \\<xi> \\<Turnstile>\\<^sub>c
          \\<psi>)\"\n  \"\\<xi> \\<Turnstile>\\<^sub>c F\\<^
          sub>c \\<phi> = \\<xi> \\<Turnstile>\\<^sub>c (true
          \\<^sub>c U\\<^sub>c \\<phi>)\"\n  \"\\<xi> \\<
          Turnstile>\\<^sub>c G\\<^sub>c \\<phi> = \\<xi> \\<
          Turnstile>\\<^sub>c (false\\<^sub>c R\\<^sub>c \\<
          phi>)\"",
      "user_state": "",
      "term": "\\<lbrakk>\\<xi> \\<Turnstile>\\<^sub>c \\<phi
          > iff\\<^sub>c \\<psi> = (\\<xi> \\<Turnstile>\\<^
          sub>c \\<phi> = \\<xi> \\<Turnstile>\\<^sub>c \\<
          psi>); ...\\<rbrakk> \\<Longrightarrow> ...",
      "hyps": [],
      "proven": [],
      "variables": {
        "Type0": "\\<phi>, \\<psi> :: 'a ltlc",
        "Type1": "\\<xi> :: nat \\<Rightarrow> 'a set"
      },
      "constants": {
        "Type0": "(\\<Longrightarrow>), (&&&) :: prop \\<
            Rightarrow> prop \\<Rightarrow> prop",
        "...": "..."
      },
      "type variables": {
        "Sort0": "'a :: type"
      }
    },
    { "...": "additional steps omitted" }
  ],
  "apply_kwrds": ["apply", "by", "done", "..."],
  "isar_kwrds": ["have", "show", "fix", "assume", "..."],
  "methods": ["Metis.metis", "Pure.unfold_locales", "..."],
  "deps": [{"name": "Pure.protectI", "term": "PROP ?A \\<
      Longrightarrow> PROP ?A"}, "..."]
}
```

D Data interaction

This appendix supplements Section 3. The subsection below demonstrates interactive theorem proving via the Python REPL, illustrating the read-eval-print loop interface on which the automated evaluation loop is built.

D.1 Python REPL API example

The script below proves $\forall x.\, P\, x \Rightarrow P\, c$ step by step using the Python REPL. Each `apply` call sends a command to Isabelle and returns the rendered proof state as a string; an empty string signals that the proof is closed.

```
import sys
sys.path.append('/path/to/this/project/src/main/python')
from repl import REPL

# Start Isabelle with HOL logic
repl = REPL("HOL")    # prints: "REPL and minion initialized."

# State the lemma goal
state = repl.apply(
    "lemma \"\\<forall>x. P x "
    + "\\<Longrightarrow> P c\""
)
# Returns:
#   'proof (prove)\ngoal (1 subgoal):\n'
#   ' 1. forall x. P x ==> P c'

# Apply the simplifier; Isabelle reduces the goal to empty
state = repl.apply("apply simp")
# Returns: 'proof (prove)\ngoal:\n No subgoals!'

# Close the proof; empty string confirms proof is complete
state = repl.apply("done")
# Returns: ''
```

Listing 1.2: Interacting with Isabelle via the Python REPL

E Training

This supplements Section 4 with subsections about the training, validation, test sample counts per data format (§E.1), the model prompts (§E.2), the final T5 replication metrics with first-run and replication epoch counts (§E.3), the Gemma 3n fine-tuning summary (Table 6), the replication validation-perplexity table used for equal-epoch comparison (Table 7), and the loss and accuracy curves for the representative first-run `t5_small_finetune_s` model (§E.4).

E.1 Data split counts

Format	Training	Validation	Testing
`s`	1,077,920	280,600	496,400
`finetune_s`	1,502,736	394,368	707,242
`spk`	4,322,842	1,081,944	1,983,106
`spk_trim`	7,694,240	1,924,416	2,358,016

Table 4: Number of samples per data split and data format.

E.2 Model input prompts

The fine-tuning and evaluation of T5 models (Section 4) used the prefix "`isabelle next step: `" prepended to every input. The frontier and instruction-tuned backends of Section 5 did not use this prefix; instead, each next-step query used the instruction prompt:

```
Recommend the next Isabelle proof step given the context below.
Enclose the suggestion in <SUGGESTION>-</SUGGESTION> tags.
Only return the suggestion, do not include any other text.
```

The instruction and response were additionally wrapped with Unsloth's recommended chat markers `<start_of_turn>user` and `<start_of_turn>model` for the Gemma 3n E2B supervised fine-tuning.

E.3 Training performance

Model	First run ep.	Replication ep.	Train Loss	Train Acc.	Valid Loss	Valid Acc.	Train Ppl.	Valid Ppl.
t5_small_s	4	2	3.02	0.445	3.13	0.436	20.58	22.91
t5_small_finetune_s	4	4	1.28	0.706	1.19	0.726	3.59	3.30
t5_small_spk	3	1	2.96	0.417	3.29	0.375	19.31	26.72
t5_small_spk_trim	4	2	2.71	0.438	3.21	0.371	15.08	24.81
t5_base_s	4	2	2.84	0.474	3.06	0.441	17.13	21.28
t5_base_finetune_spk	1	1	1.63	0.601	1.88	0.578	5.08	6.58

Table 5: T5 training performance in the final replication, with perplexity (exp(loss)) computed from the final available epoch loss.

Model	Format	Steps	Epochs	Runtime h	Train Loss	Train Ppl.	Valid Loss	Valid Ppl.
gemma_3n_e2b_ft_sp	state_prems	75,320	2.07	231.69	0.404	1.50	—	—

Table 6: Gemma 3n E2B supervised fine-tuning summary. Perplexity is exp(loss); validation data are absent because no separate validation pass was run.

Model	Epoch 1	Epoch 2	Epoch 3	Epoch 4
t5_small_s	24.21	22.91	—	—
t5_small_finetune_s	4.04	3.63	3.43	3.30
t5_small_spk	26.72	—	—	—
t5_small_spk_trim	26.23	24.81	—	—
t5_base_s	26.91	21.28	—	—
t5_base_finetune_spk	6.58	—	—	—

Table 7: Validation perplexity (exp(loss), natural-log basis) by epoch in the final T5 replication. Dashes mark unavailable canonical epochs; comparison is by equal data exposure rather than recorded steps.

E.4 T5 training curves

The plots show the training dynamics for `t5_small_finetune_s` over four epochs. Curves for the other models are omitted because they are similar; final training and validation metrics are reported in Table 5 (§E.3).

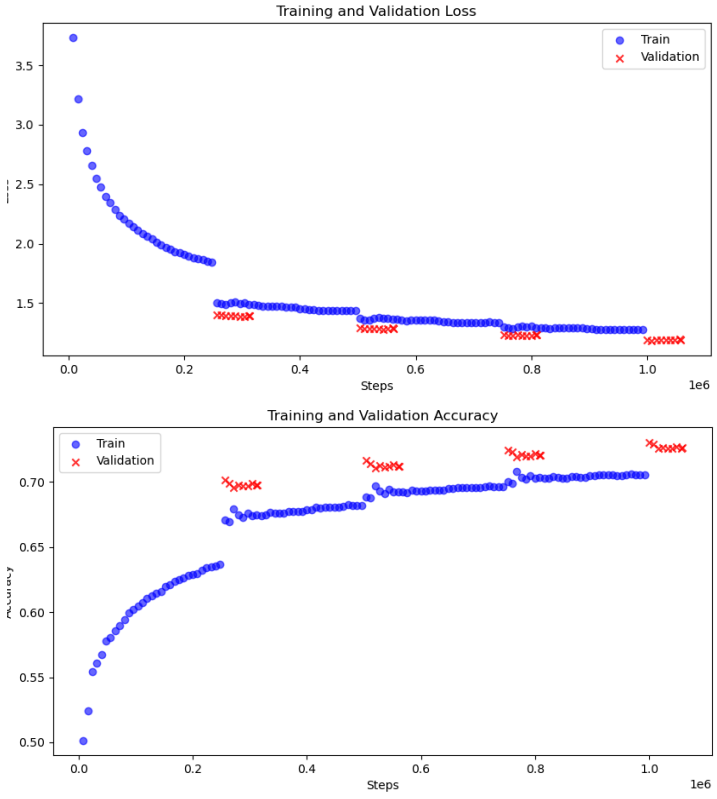


Fig. 3: Average train and validation loss (top) and accuracy (bottom) for `t5_small_finetune_s` over four epochs.

F Evaluation

This appendix supplements Section 4 and Section 5. The subsections provide the proof-step distribution of the full AFP snapshot used in evaluation (§F.1), a like-for-like comparison of the frontier backends on their common proof set (§F.2), and per-proof wall-clock times for all evaluated backends (§F.3).

F.1 AFP proof-step distribution

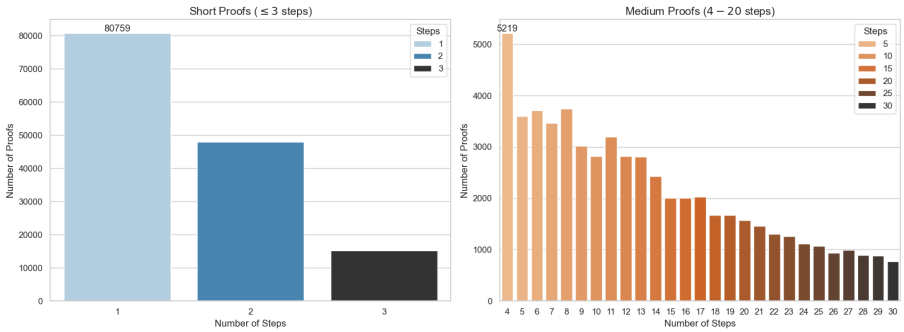


Fig. 4: Step-count distribution for the full 30 November 2025 AFP snapshot: 1–3 step proofs (left bars) and 4–30 step proofs (right bars).

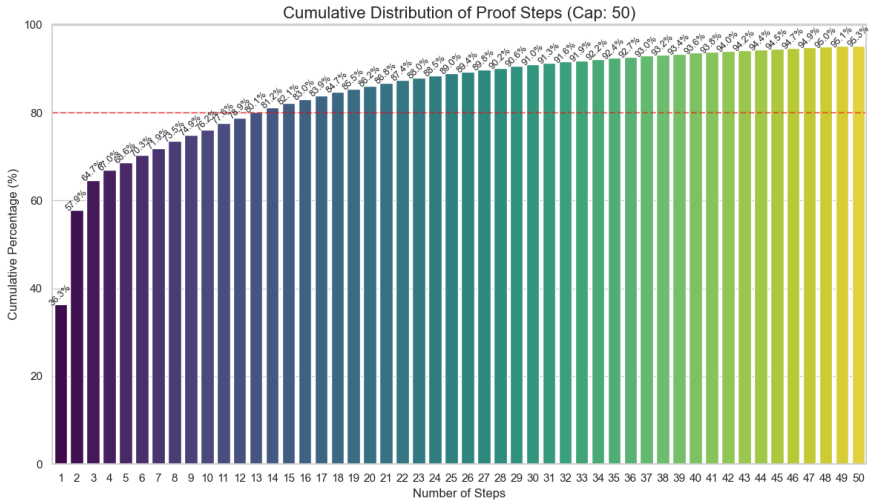


Fig. 5: Cumulative proof-step coverage for the full 30 November 2025 AFP snapshot (222,466 proofs; the test-split view is Figure 1).

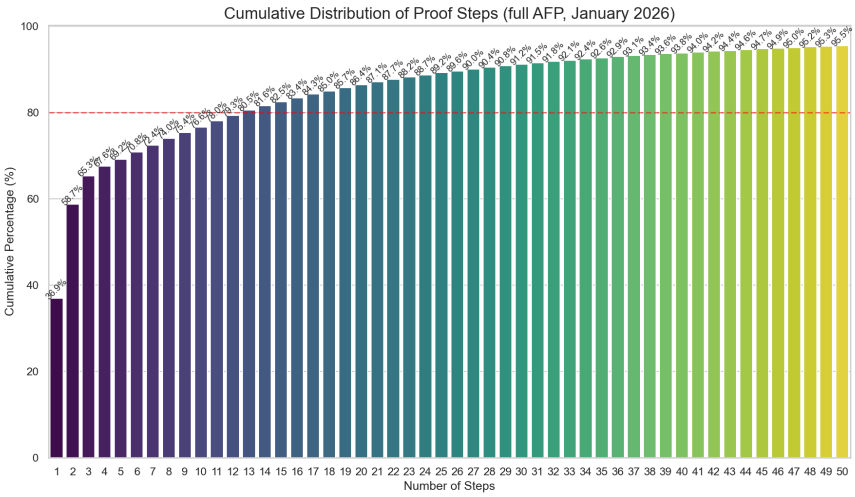


Fig. 6: Cumulative proof-step coverage on the 15 January 2026 AFP snapshot (241,126 proofs) used for proof-search evaluation.

F.2 Common-subset comparison of the frontier backends

Table 2 reports each backend on the proofs it actually attempted, which differ in number because of budget limits. To compare on an equal footing, we restrict the three backends whose AFP-entry sets are nested—Gemma 4 E4B (36 entries) $\subseteq$ Gemini 3 Flash Preview (46) $\subseteq$ Gemini 2.5 Flash Lite (84), all on the 15 January 2026 snapshot—to the 36 entries they share, and within those to the 1,336 proofs that all three attempted. GPT-5.3-chat shares only one entry with this group, and the fine-tuned Gemma 3n ran on a different snapshot and data format, so neither can join a fair common set; both are omitted. The metrics in Table 8 are recomputed from the run logs over exactly this shared set, so **Finished %** is directly comparable across the three models. The headline ordering of Table 2 is preserved and sharpened: it is not an artefact of the differing proof-set sizes.

Model	Proofs	Predictions	Progress	Finished	Finished %
Gemini 3 Flash Preview	1,336	1,351	51.74%	430	32.19%
Gemini 2.5 Flash Lite	1,336	8,950	20.32%	29	2.17%
Gemma 4 E4B	1,336	6,218	4.36%	14	1.05%

Table 8: Frontier backends restricted to the 1,336 proofs common to all three (the 36 shared AFP entries on the 15 January 2026 snapshot). Columns carry the same meaning as in Table 2, recomputed over the shared set; **Proofs** is therefore identical across rows, making **Finished %** a like-for-like comparison. On this common set Gemini 3 finishes 32.19% of proofs, against 2.17% and 1.05% for Gemini 2.5 and Gemma 4.

F.3 Evaluation wall-clock times

The wall-clock times per-proof are reported here. T5 model averages: 48.92 s (`t5_small_s`), 40.09 s (`t5_small_finetune_s`), 29.84 s (`t5_small_spk`), 50.33 s (`t5_base_s`). LLM backend mean / median per attempted proof: GPT-5.3-chat 35.03 s / 15.06 s, Gemini 3 Flash Preview 4.74 s / 5.01 s, Gemini 2.5 Flash Lite 3.99 s / 1.86 s, Gemma 4 E4B 8.07 s / 6.64 s. For Gemma 3n E2B fine-tuned, only per-completed-proof timing is available: mean 12.69 s, median 8.86 s. No correlation between time and completion rate is claimed; the variation reflects API latency, beam-search cost, DFS branching, and proof-set size differences. Table 9 complements these means and medians with the full five-number summary and standard deviation. All backends are right-skewed—the mean exceeds the median and the maxima are far above the third quartile—so a few slow proofs dominate the average while the typical proof is processed near the median.

Model	n	Min	Q1	Median	Q3	Max	Mean	STD
GPT-5.3-chat	300	5.46	10.71	15.06	39.78	612.49	35.03	49.81
Gemini 3 Flash Preview	1,833	0.15	0.35	5.01	6.35	62.51	4.74	5.30
Gemini 2.5 Flash Lite	2,807	0.67	1.28	1.86	3.26	137.29	3.99	8.86
Gemma 4 E4B	1,330	3.52	6.13	6.64	7.70	148.89	8.07	6.43
Gemma 3n E2B fine-tuned	59	2.33	6.99	8.86	14.59	70.69	12.69	11.37

Table 9: Per-proof wall-clock time distribution (seconds) for the proof-search backends. The first group is measured per *attempted* proof; Gemma 3n E2B fine-tuned (last row) logged only per-*completed*-proof times. **Q1**/**Q3** are the first and third quartiles and **STD** the sample standard deviation. The T5 runs recorded only an aggregate average (reported above) and so do not appear here.

G Qualitative analysis

This appendix supplements Section 5 with qualitative evidence. §G.1 presents three representative model-generated proofs with full theory context; §G.2 analyses the distribution of proof methods used by each backend.

G.1 Example model-generated proofs

The three proofs below are representative of the more complex completions produced by the LLM backends. Each listing shows the full `.thy` file as saved by the evaluation loop.

Listing 1. *GPT-5.3-chat*: In the lemma `gen_dfs_post_imp_eq` from the Isabelle session `CAVA_LTL_Modelchecker.NDFS_SI`. The model identifies set equality as requiring two subset inclusions and applies the two named DFS library lemmas.

```
theory NDFS_SI17
```

```
  imports CAVA_LTL_Modelchecker.NDFS_SI
begin

lemma gen_dfs_post_imp_eq:
  assumes "gen_dfs_pre E U {} V0 u0"
  assumes "gen_dfs_post E U {} V0 u0 V False"
  shows "V = V0 ∪ E^*``{u0}"
apply (rule Set.subset_antisym)
subgoal using assms by (rule NDFS_SI.gen_dfs_post_imp_reachable)
subgoal using assms by (rule NDFS_SI.gen_dfs_post_imp_complete)
done

end
```

Listing 2. *Gemini 3 Flash Preview*: `PE_is_not_definable` from `PCF.PCF`. The model instantiates the `not_definable` rule with witness `R="unlr PE.delta"` and chains three PCF-specific lemmas.

```
theory PCF66
  imports PCF.PCF
begin

theorem PE_is_not_definable:
  "¬(∃f. definable f ∧ appFLv f [PE_arg_rel] = PE_result_rel)"
apply (rule PCF.not_definable [where R="unlr PE.delta"])
apply (rule PCF.PCF_lr_PE_delta)
apply (simp add: PCF.lr_PE_arg_rel)
apply (rule PCF.lr_PE_result_rel)
done

end
```

Listing 3. *Gemma 3n E2B fine-tuned*: lemma `ordered_incidence_sysII` from `Fishers_Inequality.Incidence_Matrices`. The model requires `unfold_locales` to discharge an obligation. Removing that command stops the proof from being accepted.

```
theory Incidence_Matrices90
  imports Fishers_Inequality.Incidence_Matrices
begin

lemma ordered_incidence_sysII:
  assumes "finite_incidence_system 𝒱 ℬ"
  and     "set 𝒱s = 𝒱"
  and     "distinct 𝒱s"
  and     "mset ℬs = ℬ"
  shows   "ordered_incidence_system 𝒱s ℬs"
using assms
apply (unfold_locales,
```

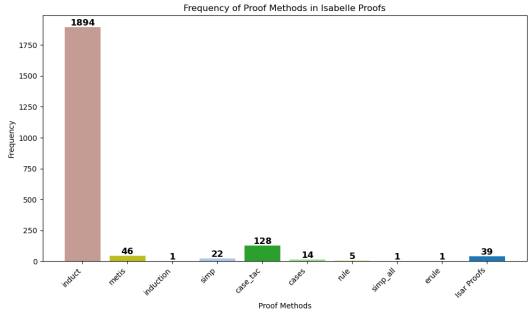
```
       simp add: finite_incidence_system_def
                 incidence_system.wellformed)
apply (rule incidence_system.wellformed)
apply (simp_all add: finite_incidence_system_def)
done

end
```

G.2 Proof-method analysis

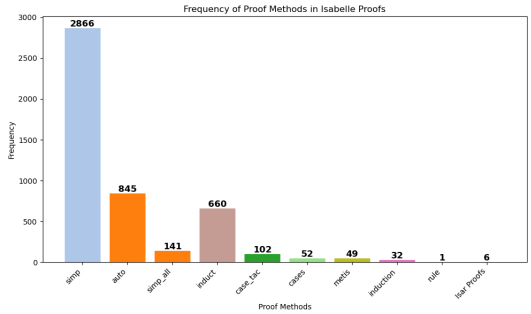
T5 prediction patterns. After `by`-conversion to `apply`, the simplest `t5_small_s` model suggests the highest percentage of predictions that make progress in the proof. However, despite progress in several proofs, its fraction of completed proofs is the smallest. This suggests that its model predictions are, although valid, unsound. Qualitatively exploring the loop-produced log files shows that many "successful" `t5_small_s`-actions were repeated applications of the `case_tac` method or, in proof scripting sections, repeated applications of the Isar command `fix`. The `case_tac` method receives an argument and splits the proof in two goals depending on the truth of the argument (e.g. `apply (case_tac "x = 0")`). The model's `case_tac`-actions are usually accompanied by equalities, augmenting the number of proof obligations but not necessarily advancing the proof. Similarly, the Isar command `fix` introduces a locally fixed variable into scope (e.g. `fix x`), which is useful when the current subgoal begins with a universal quantifier or when building a structured proof. Repeatedly applying `fix` without subsequently making a claim about the fixed variable does not advance the proof obligation. Despite these limitations, `t5_small_s` successfully completed approximately 6.82% of its attempted proofs. Many completions are simple uses of the simplifier (e.g. `by simp`) or simple induction proofs (e.g. `by (induct n, simp_all)`).

T5 by-conversion breakdown. Across the five T5 models trained on the AFP (`small_s`, `small_finetune_s`, `small_spk`, `base_s`, `base_finetune_spk`), `by`-conversions represent only a small fraction of all predictions: 2.53%, 4.65%, 1.88%, 1.70%, and 5.14% respectively. Curiously, `small_finetune_s` had the lowest percentage of progressing predictions while its fraction of `by`-conversions was the second largest, and it also has the lowest percentage of correctly suggested `by`-statements (8.40%). By contrast, `base_finetune_spk` has the largest fraction of `by`-conversions (5.14%) and a much higher correct-`by` rate (36.70%), suggesting that the extra size and data improve accuracy on this metric. Taken in isolation, some per-column percentages can be misleading: a high syntactic-acceptance rate does not imply a high proof-completion rate. Among these five models, `base_finetune_spk` achieves the highest proof-completion rate (16.71%), with `small_finetune_s` second (11.28%), consistent with the training results in Section 4 and with the general NLP finding that domain-specific fine-tuning is more effective than training from scratch. These results serve primarily to validate that the REPL loop operates correctly across different model configurations and should not be over-interpreted as a definitive ranking.

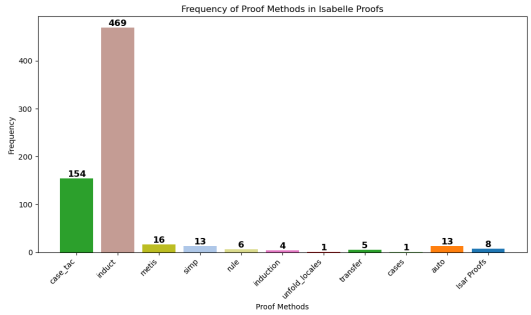
Proof-method distributions. Figures 7 and 9 show the distribution of the first proof method in completed proofs for T5 and LLM backends respectively. Both plots report the frequency of the first method appearing after an `apply` or `by` command in the saved proofs, and include Isar-scripted completions. `simp` and `auto` dominate across all backends. Among the T5 models, `small_finetune_s` relies heavily on the simplifier, while `small_s` makes comparatively more use of Isar proofs. `base_finetune_spk`, which completed the most T5 proofs (6,438), contributes the largest absolute sample and its distribution is shown alongside the other T5 models in the second figure of this group. Some completions employ less common methods: `unfold_locales` for locale expansion, `intro_classes` for Isabelle class operations, the `transfer` method, and the dot `.` method for closing `interpretation` statements. Among the LLM backends, Gemma 4 E4B (Ollama) is notable for its very small number of completed proofs (14), which are almost exclusively `simp` or `auto`; this contrasts with GPT-5.3-chat and Gemini 3 Flash Preview, which both show a broader method distribution including `rule`, `blast`, and locale-specific tactics.



(a) `t5_small_s`

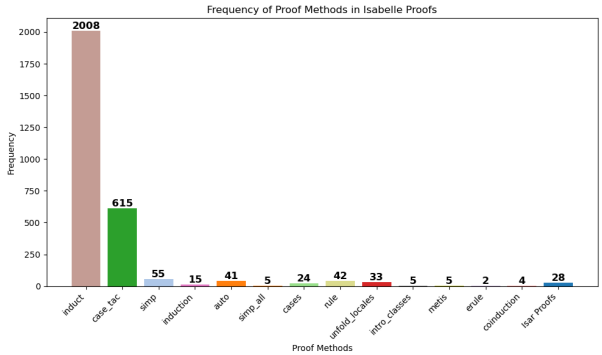


(b) `t5_small_finetune_s`

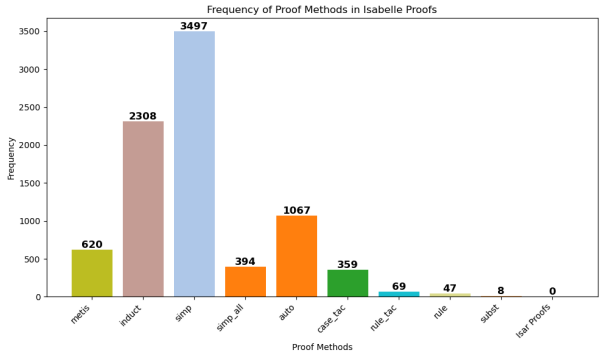


(c) `t5_small_spk`

Fig. 7: Distribution of the first proof method in completed proofs (T5 models).

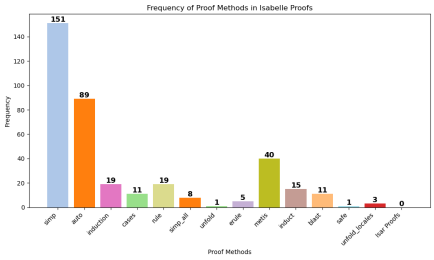


(a) `t5_base_s`. Methods rare at this scale are confirmed present in the raw log files; the dot `.` method is one such case.

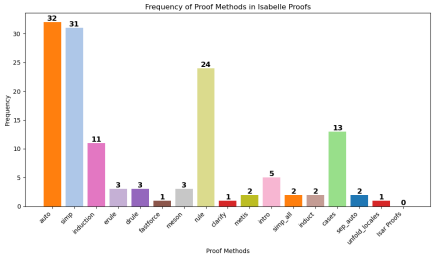


(b) `t5_base_finetune_spk`

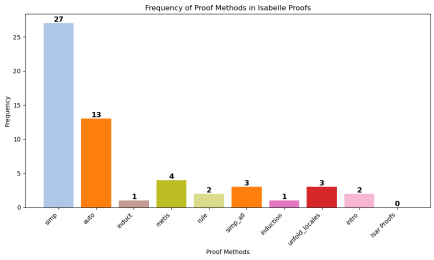
Fig. 8: Distribution of the first proof method in completed proofs (T5 models).



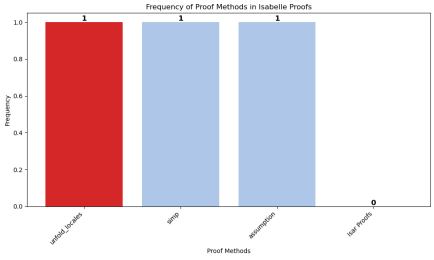
(a) Gemini 3 Flash Preview



(b) GPT-5.3-chat



(c) Gemma 3n E2B fine-tuned



(d) Gemma 4 E4B (Ollama)

Fig. 9: Distribution of the first proof method in completed proofs (LLM backends).