Elixir meets TPTP: Bringing Automated Reasoning to the BEAM Ecosystem

Johannes Schuster[1,*], David Fuenmayor[1] and Christoph Benzmüller[1,2]

[1]University of Bamberg, Kapuzinerstraße 16, 96047 Bamberg, Germany
[2]Freie Universität Berlin, Kaiserswerther Str. 16-18, 14195 Berlin, Germany

Abstract

Automated theorem provers (ATPs) are powerful tools for formal reasoning, yet integrating them into larger software systems remains cumbersome: existing interfaces are typically script-based, tightly coupled, or limited in working across multiple backends in a uniform way. We present AtpClient, an Elixir library that provides a unified, extensible interface to ATP services across four backends, SystemOnTPTP, StarExec, Isabelle, and locally installed provers, using TPTP standards. The library acts as a *truth-grounding* service for host applications: it exposes a consistent API that abstracts over differences in communication protocols, result formats, and termination semantics, and normalizes the verdict of each backend into a single result type. Built on the BEAM virtual machine, it benefits from robust process isolation, fault-tolerant result polling, and uniform cancellation, making it well-suited to multi-prover experimentation and portfolio solving. We describe the architecture, discuss key design decisions, and reflect on challenges encountered when bridging the gap between ATP services and a functional runtime. We further show how the library is consumed by two downstream tools built on it without backend-specific tinkering: a Livebook Smart Cell that turns it into an interactive TPTP editor, and a Model Context Protocol server that exposes the backends to LLM-based agents. AtpClient is open source and available on Hex, the Elixir package registry.

Keywords

Automated Theorem Proving, Elixir, Livebook, SystemOnTPTP, Isabelle/HOL, Model Context Protocol

1. Introduction

Automated theorem provers (ATPs) remain essential for guaranteed correctness in verification pipelines and have gained renewed relevance as a complement to neural reasoning systems. However, they often lack usability, requiring shell scripts, bespoke wrappers or one-off adapters for integration into larger systems, limiting their potential as plug-and-play solutions e.g. for formal verification in multi-agent systems or in logic education. Tools like SystemOnTPTP [1] and `sledgehammer` [2] in the interactive proof assistant Isabelle/HOL [3] address this by using a standardized language for communicating problems to a collection of provers. While both tools offer a broad portfolio of various ATPs and configuration options, they are ultimately designed for human experts. And even for advanced users, utilizing both systems for evaluating new provers against a ground truth of established results requires the effort of building dedicated evaluation environments for both backends. Furthermore, while virtually all ATPs accept TPTP syntax as input, their output format and termination status are generally not standardized.[1]

Our aim with AtpClient is therefore not to add yet another wrapper around a single service, but to make ATP access a first-class citizen of the BEAM: the virtual machine powering the Erlang, Elixir and Gleam programming languages (among many others), and a runtime that is increasingly used

Practical Aspects of Automated Reasoning (PAAR) 2026, as part of the Federated Logic Conference (FLoC) 2026, Lisbon, Portugal, July 25, 2026
*Corresponding author.
johannes.c.schuster@proton.me (J. Schuster); david.fuenmayor@uni-bamberg.de (D. Fuenmayor); christoph.benzmueller@uni-bamberg.de (C. Benzmüller)
https://jcschuster.github.io (J. Schuster); https://page.mi.fu-berlin.de/cbenzmueller/ (C. Benzmüller)
0009-0003-5339-7216 (J. Schuster); 0000-0002-0042-4538 (D. Fuenmayor); 0000-0002-3392-3039 (C. Benzmüller)

[1]Even though the TPTP World [4] promotes the SZS status ontology, this convention is not respected by all provers, e.g. the first-order theorem prover SPASS [5].

for agent infrastructure and LLM orchestration. We position the library as a *truth-grounding* service: host applications dispatch a problem to a backend and receive back a single, normalized verdict on whether the goal holds, without having to reconcile the differing protocols, output formats, and proving conventions of the underlying provers. The contribution is the combination of three things no existing tool provides together: uniform result semantics across four structurally different backends, fault-tolerant polling and cancellation built on lightweight processes, and an interactive front-end. We substantiate this with the library itself and two downstream consumers built on it without any backend-specific adapters, an interactive notebook editor and a Model Context Protocol (MCP) server for LLM agents. Concretely, our contributions include:

- A unified, backend-agnostic API[2] across SystemOnTPTP, StarExec, Isabelle, and locally installed provers, with shared configuration layering and uniform error semantics.
- A normalized result type that preserves the SZS verdict faithfully across backends, with prover-specific fallbacks and a structural classifier for Isabelle's `use_theories` output.
- A Livebook Smart Cell[3] that turns the library into an interactive multi-backend TPTP IDE, with debounced live linting, hover-card type information and one-click evaluation.
- A Model Context Protocol server[4] that exposes all four backends to LLM-based agents through a single tool surface, demonstrating that the architecture is agent-ready (section 9).

The remainder of this paper will address some background information about the employed technologies (section 2), discuss the architecture of the AtpClient library (section 3), highlight the advantages of using the Elixir programming language for this process (section 4), explain the Livebook integration (section 5), promote different application scenarios (section 6), reflect on design decisions and challenges (section 7) and close with sections on related (section 8) and future work (section 9).

2. Background

Elixir Elixir is a functional, dynamically-typed language that runs on the BEAM virtual machine. BEAM provides lightweight preemptive processes (millions can coexist on one machine), supervision trees for fault tolerance, and pattern matching as a primary control-flow construct. These features will be relevant in section 4.

TPTP Syntax Layers and SZS Ontology The TPTP language [4] has long been the standard for encoding problems in first- and higher-order logics, and is widely supported across ATPs. It defines several syntax layers: most importantly `fof` (first-order form), `tff` (typed first-order form), which adds monomorphic or polymorphic types, and `thf` (typed higher-order form) for higher-order logics. Annotated TPTP formulae carry roles that either introduce new symbols (`type`, `definition`), axiomatize statements (`axiom`), or state a proof goal (`conjecture`).

The SZS ontology standardizes prover results and status messages. Systems following this ontology log status messages such as "`% SZS status Theorem`" or "`% SZS status Timeout`." Tools like `sledgehammer` then scan prover outputs for these messages and report solutions.

SystemOnTPTP, StarExec, Isabelle, and Local Backends The web interface of SystemOnTPTP [1] allows users to query various state-of-the-art ATPs with native integration of the TPTP problem library. It is also addressable programmatically,[5] accepting HTTP requests that mirror the functionality of the web interface. Programmatic access enables, for instance, listing all available systems or running selected systems on a problem, and forms the core of our SystemOnTPTP API.

[2]The main library: https://hexdocs.pm/atp_client/index.html
[3]Also available as Hex release: https://hexdocs.pm/kino_atp_client/index.html
[4]https://hexdocs.pm/atp_mcp/index.html
[5]via the URL https://tptp.org/cgi-bin/SystemOnTPTPFormReply

```elixir
problem = "thf(ext, conjecture, ![F:$i>$i, G:$i>$i]: ((![X:$i]: ((F @ X) = (G @ X))) => (F = G)))."
AtpClient.SystemOnTptp.query_system(problem, "Leo-III---1.7.20")
# => {:ok, :theorem}
AtpClient.SystemOnTptp.query_selected_systems(problem, ["cvc5---1.3.4", "Vampire---5.0.1"])
# => {:ok, [{"cvc5---1.3.4", {:ok, :theorem}}, {"Vampire---5.0.1", {:ok, :theorem}}]}
problem_isabelle = """
theory Example imports Main begin
    theorem "(! X. F X = G X) --> F = G" by auto
end
"""
AtpClient.Isabelle.query(problem_isabelle, "Example", my_server_info)
# => {:ok, {:ok, :theorem}}
```

Figure 1: High-level showcase of the API and the unified results as Elixir program.

StarExec [6] is at its heart a job scheduler for benchmarking provers, running as a web service. Its goals involve maintaining benchmark libraries, providing a unified infrastructure for ATP competitions, and allowing researchers to evaluate their own systems against benchmarks. Self-hosted instances can also be utilized for evaluating problems when trusted logic solvers are deployed [7].

The interactive theorem prover Isabelle/HOL [3] ships with a TCP server that exposes the full functionality of the proof assistant. This allows for calls to established tools such as `sledgehammer` [2], which queries a portfolio of trusted provers for proof search, or the model generator `nitpick` [8]. The Elixir library `isabelle_elixir`[6] additionally streamlines session management and the evaluation of Isabelle theories.

Finally, the most direct way to run a prover is to invoke a locally installed, TPTP-compliant binary such as E [9] or Vampire [10] directly, avoiding network round-trips entirely at the cost of managing the process, its resource limits, and its termination by hand.

3. Architecture of AtpClient

AtpClient is designed as an API for truth grounding across different application scenarios. As such it has three responsibilities: provide a unified API surface to the user, enable layered configuration supporting custom deployments, and handle polling and termination semantics across backends.

Result Normalization We provide a unified result type `atp_result` across all backends, defined as a tuple of two fields: an error status (`:ok` or `:error`), following Elixir conventions, and the inferred problem status. In the success case, the second element is the SZS verdict reported by the prover, downcased to an Elixir atom, e.g. `:theorem`, `:unsatisfiable`, `:counter_satisfiable`, `:satisfiable`, `:gave_up`, `:timeout`, `:resource_out`, or `:memory_out`. Rather than collapsing these into a handful of coarse categories, we preserve the SZS verdict faithfully, so that callers do not lose the distinction between, for instance, a `Theorem` (the conjecture follows from the axioms) and an `Unsatisfiable` (the clause set has no model) result. In the error case, the second element represents the reason the call could not produce any SZS verdict (e.g. `{:unrecognized_output, _}`, `{:prover_not_found, _}`, or an HTTP error). A high-level demonstration of this API is shown in figure 1.

Across all backends, `:theorem` reflects the prover's or tool's claim of success, not kernel verification. This keeps the result type uniform: "a prover reported `Theorem`" is the strongest claim available for SystemOnTPTP, StarExec, and local provers, and we treat Isabelle's `sledgehammer` output the same way: status `:theorem` may be reported without the proof having been double-checked. For stronger

[6] https://hexdocs.pm/isabelle_elixir/index.html

guarantees within Isabelle, callers shall invoke methods that actually submit the proof object to the kernel. The atoms `:counter_satisfiable` and `:satisfiable` are kept distinct, mirroring the SZS ontology: for a problem with a conjecture, a model of the negated goal witnesses `CounterSatisfiable`, whereas `:satisfiable` arises for instance on the Isabelle backend, where `nitpick[satisfy]` models the formula itself as a positive existence claim rather than refuting a conjecture.

To extract this result from TPTP-style provers, we define a mapping of the SZS ontology to our result type, and use prover-specific fallbacks for systems that use other formats. For the standardized cases this mapping is direct, taking each SZS status to the correspondingly named atom. Following the approach used in `sledgehammer`'s source code, we also maintain a table of 24 prover-specific output patterns covering 7 provers, namely Alt-Ergo [11], E [9], iProver [12], LEO-II [13], SPASS [5], Vampire [10], and Waldmeister [14]. These systems either do not use the SZS ontology at all, or fall back to custom messages for some error and termination cases. Unlike `sledgehammer`, which interprets these messages against its negated-conjecture pipeline, we map each pattern to the most faithful SZS atom the prover would itself have emitted: SPASS's "Completion found", for instance, signals a saturated, contradiction-free clause set and is recorded as `:satisfiable` rather than a generic failure.

For the Isabelle backend, we are especially interested in results from `sledgehammer`, `nitpick`, and internal proof procedures like `auto`. Beyond hand-written theory text, the backend also accepts a TPTP problem string directly: annotated THF formulae (`thf(name, type|axiom|conjecture, …)`) are translated into Isabelle `type` and `lemma` declarations via `isabelle_elixir`'s THF notation bridge, with a configurable proof method (`by auto` by default) appended to each generated goal. This is a notation-level translation of annotated formulae, not a parser for the full TPTP file grammar. The API for Isabelle evaluates one proof problem per call, which may itself contain multiple formulae, while supporting multiple tool calls within it. This allows, for example, a combination of `nitpick` for checking counter-satisfiability, `sledgehammer` for calling a portfolio of provers, and `nitpick[satisfy]` to check satisfiability. Concrete design decisions for the Isabelle backend will be discussed in section 7. Result extraction matches on the same tool outputs that the Isabelle GUI would display to the user. When several tool messages are present, a refuting verdict (a Nitpick or Quickcheck counterexample, mapped to `:counter_satisfiable`) takes precedence over a proof, which in turn precedes a satisfiability witness and other status messages.

Result interpretation can be disabled by passing `raw: true` to the function calls. This returns the raw output of the queried systems, e.g. including inference steps.

Configuration and Termination Semantics We opt for layered and dynamic configuration of the different backends, as it promotes flexibility and a clear structure for custom backends. Users can either specify their targeted infrastructure in a configuration file, or pass this information in the API calls. The SystemOnTPTP integration works out of the box, while Isabelle and StarExec require configuration (URLs, authentication, etc.). Configuring Isabelle requires some care, since the Isabelle server expects concrete `.thy` files rather than raw strings. A directory shared by client and server must therefore be specified. This typically includes the path from the client process to this directory (`local_dir`) and the path from the server side (`isabelle_dir`), which becomes the `master_dir` of the Isabelle server.

Calls to SystemOnTPTP are synchronous HTTP requests and do not require special handling. Both the Isabelle and StarExec backends, however, need status polling to make the API synchronous, each with its own quirks. Getting results from Isabelle is entirely handled by the `isabelle_elixir` Elixir library. For StarExec, a completion predicate can be configured by the user. The BEAM facilitates enforcing timeouts with `System.monotonic_time/1`, which provides a monotonic clock for computing deadlines and is unaffected by wall-clock adjustments.

Local Provers, Timeouts, and Cancellation The local backend invokes a configured prover binary via `Port.open/2` and normalizes its output through the same SZS classifier; a missing binary surfaces as a typed `{:error, {:prover_not_found, name}}` rather than an exception. Termination is enforced at two layers: a prover-side CPU limit, passed as an argument so the prover can emit a clean `SZS status`

`Timeout` of its own, and an independent BEAM-side wall-clock timeout that kills a wedged process; since a bare `Port` does not propagate a kill to its operating-system child, the wall-clock path terminates the OS process by PID. Both paths fold into the same `{:ok, :timeout}` result, so callers never branch on which side noticed the deadline. More generally, AtpClient uses death-as-cancellation as a uniform contract. If the process that issued a query dies, all the resources held on its behalf are released. These include in-flight requests, polling loops, OS children and Isabelle session links. This release is achieved through BEAM process linking rather than an explicit cancellation token. If a backend has options other than waiting for the next poll, it utilises them. The StarExec backend, for example, issues an explicit job-deletion request when its caller is cancelled so that abandoned jobs do not remain in the queue.

4. Advantages of Elixir

The choice of Elixir for AtpClient is motivated by three properties of the BEAM runtime that map naturally onto the requirements of multi-backend ATP integration: lightweight concurrency, fault isolation, and pattern matching as a primary control-flow construct.

A major strength of the BEAM ecosystem is its native support for concurrency. Querying multiple provers on a single problem is the canonical form of ATP usage in benchmarking and proof exploration. On the BEAM, each prover query naturally lives in its own preemptively scheduled process, so a portfolio call reduces to a single combinator.

```
problem = "thf(ext, conjecture, ![F:$i>$i, G:$i>$i]: ((![X:$i]: ((F @ X) = (G @ X))) => (F = G)))."
AtpClient.SystemOnTptp.list_provers()
|> Task.async_stream(
    fn sys -> {sys, AtpClient.SystemOnTptp.query_system(problem, sys)} end,
    max_concurrency: 16, timeout: 30_000, on_timeout: :kill_task)
|> Enum.find(&match?({:ok, {_sys, {:ok, :theorem}}}, &1))
```

A straggling prover is killed at the deadline without affecting its peers, and the result of the first successful proof can be returned while the others are still running.

Another feature promoting Elixir as a base is robustness against errors. External provers may segfault, return malformed output, hang on ill-formed input, or simply disappear after network issues. The BEAM's "let it crash" philosophy, combined with supervision trees, lets AtpClient treat each backend interaction as a process that may fail without taking the host application with it. A long-lived Isabelle session that dies during a benchmark run is restarted by its supervisor, and a single failed HTTP call to SystemOnTPTP propagates as an `{:error, reason}` value rather than as an exception that the caller must remember to catch.

5. Livebook Integration

Livebook is an Elixir-native, reactive notebook environment. Beyond ordinary code and Markdown cells, it lets library authors define *Smart Cells*: live Elixir processes, sharing the notebook's BEAM node, that render a custom user interface and generate the Elixir source they represent. Unlike a notebook widget, a Smart Cell is a first-class UI surface with its own serialized attributes and a `to_source/1` function. The adoption cost is that Livebook is part of the Elixir ecosystem: running it means installing Elixir, though it is available as one-click install.[7] We use these facilities to ship two Smart Cells in the open-source library KinoAtpClient.

The first, *ATP Solver*, is a TPTP code editor with a backend mode switch that drives all four AtpClient backends from a single problem editor; switching the backend swaps both the per-call controls (a solver

[7] https://livebook.dev

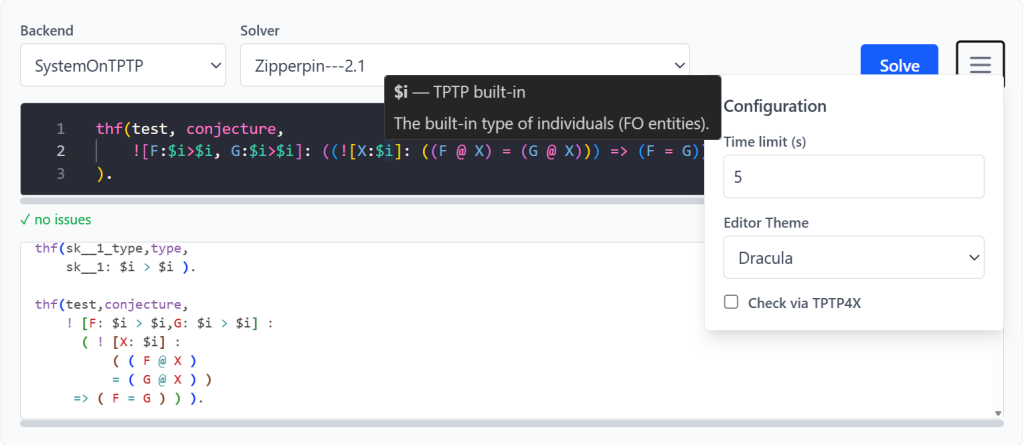

Figure 2: Screenshot of the ATP Solver Smart Cell inside a Livebook (Windows 11, Brave). A backend mode switch drives all four backends from a single TPTP problem editor; shown here is SystemOnTPTP. It provides quick evaluation with the selected prover, syntax highlighting for the TPTP language, debounced hybrid syntax checking (locally and via TPTP4X), tooltips on hover, colored brackets and configuration options.

picker for SystemOnTPTP, a proof method for Isabelle) and the result rendering. The second, *ATP Backend Configuration*, is a schema-driven form built automatically by walking the backends exposed by the library and rendering each backend's declared configuration fields, grouped into connection, defaults, and advanced sections, with a Verify button that probes reachability before a run. For one-off TPTP editing and prover invocation, the cells are usable as a black-box GUI without writing any Elixir; only composing them into a larger automated workflow requires Elixir code. A screenshot of the ATP Solver Smart Cell is provided in figure 2.

The Smart Cell uses the Monaco editor[8] to implement syntax highlighting for the full TPTP language, driven by a tokenizer matching the keywords and symbols of the TPTP grammar.[9] It provides live linting via a two-tier `AtpClient.Lint` pipeline, combining a cheap local structural checker with a request to TPTP4X[10] (the traditional TPTP syntax checker and pretty-printer from the TPTP World) for harder cases, invoked only after a brief period of inactivity to avoid overwhelming the backend. Monaco markers surface issues inline (e.g. underlining an unknown formula role), hovering over a built-in symbol shows an explanatory card that is especially valuable for novices, and user-defined symbols collected during linting join the built-ins in a completion menu. The output panel shows the syntax-highlighted system output for solvers from SystemOnTPTP and a mapped result (such as "Theorem" or "Counter-satisfiable") for the other backends, which extends to a per-lemma table for Isabelle.

Because each Smart Cell is a live process on the notebook's BEAM node rather than an opaque kernel, hover data, lint results, and solver output flow over the same event channel that powers the rest of the notebook, and the cell can share runtime state with surrounding cells, which is harder to achieve with Jupyter's `ipywidgets`. The local Isabelle backend benefits in particular: when the `isabelle` executable is on the PATH, the Smart Cell spawns a local server, opens a session, and reuses it, so Isabelle works with zero configuration from inside the notebook.

6. Application Scenarios

We envision three broad classes of use: teaching, research experimentation, and embedding ATP services in larger Elixir applications.

In a teaching context, the ATP Solver Smart Cell turns a Livebook into a zero-install TPTP IDE, giving

[8] https://microsoft.github.io/monaco-editor/
[9] The full BNF for the TPTP language is defined under https://tptp.org/UserDocs/TPTPLanguage/SyntaxBNF.html
[10] https://github.com/TPTPWorld/TPTP4X

students syntax highlighting, live linting, hover documentation for built-ins, and one-click access to a portfolio of state-of-the-art provers, all from inside a browser. Because a notebook can interleave explanatory Markdown, worked examples, and exercises and is shared as a single `.livemd` file, the same document serves as both lecture notes and a working environment.

Research experimentation is the second setting. Comparing solver performance across a set of problems would ordinarily mean an ad-hoc shell script juggling authentication, output parsing, and timeout bookkeeping; with AtpClient the same experiment is a few lines of Elixir.

```
for problem <- benchmarks, sys <- systems, into: %{} do
  {{problem.id, sys}, AtpClient.SystemOnTptp.query_system(problem.body, sys)}
end
```

Since the result type is uniform across backends, one experiment can send a problem to a prover on SystemOnTPTP, solve it as an Isabelle theory, and run it against a locally installed binary, all without post-processing the results. Moving a benchmark from the public SystemOnTPTP endpoint to a local E or Vampire prover instance, whether for faster turnaround or offline operation, is then a one-line change of backend rather than a rewrite of the harness.

The third setting is embedding. Elixir is a popular choice for distributed and web-facing systems, typically via the Phoenix framework,[11] and AtpClient is designed to live inside them: a Phoenix LiveView application could expose a TPTP editor to end users, dispatch problems to a prover portfolio in the background, and stream results back to the browser as they arrive, all without leaving the BEAM. The same holds for truth-grounding in multi-agent systems whose agents discharge proof obligations against an external solver. Such long-running, interactive workflows also benefit from the Isabelle session API, which exposes `open_session/1`, `prove_theory/4`, and `close_session/1` as separate operations, so that many theories can be checked against a single long-lived `HOL` session without repeatedly paying the startup cost, as when a `nitpick`/`sledgehammer`/`auto` cycle is repeated tens or hundreds of times while developing a formalization.

Embedding extends naturally to LLM-based agents in mathematical applications, which increasingly need to discharge proof obligations, model-finding queries, and counter-satisfiability checks against trusted external reasoners. Because every backend answers to a single `query/2` entry point, a thin server can offer "run this TPTP problem on backend *X*" as a single tool and leave the choice of backend to the agent. We realize this concretely with a Model Context Protocol server, described in section 9.

7. Design Decisions and Challenges

Integrating diverse ATP systems into a unified library requires bridging several environmental and architectural gaps. For instance, the Isabelle server cannot accept raw theory text; theories must exist as `.thy` files on a filesystem visible to both client and server. When the two run on the same machine this is transparent, but in containerized or remote setups the BEAM node and the Isabelle process often see the shared directory under different paths. Rather than encode container-vs-host conventions into the library, we expose two configuration keys, `local_dir` and `isabelle_dir` (the latter passed verbatim as the server's `master_dir`). On a "Cannot load theory file" error, we enrich the message with both paths side by side so the mismatch is immediately visible.

Once a theory is loaded, we face the challenge of aligning Isabelle's rich proof semantics with our standard backend interface. To resolve this, the Isabelle module is designed with two distinct tiers of interaction. Through the unified `AtpClient.Backend` behaviour (`query/2`), an Isabelle call aggregates the outcome of multiple goals into a single `atp_result`. For instance, when a TPTP problem is 'isabellized' into multiple lemmas, the backend applies a weakest-link aggregation, returning `{:ok, :theorem}` only if every generated lemma is successfully discharged. This preserves structural

[11] https://www.phoenixframework.org/

uniformity, allowing host applications to seamlessly swap Isabelle for remote SystemOnTPTP queries or local binaries. However, rather than limiting users to this simplified view, the API natively exposes its full capabilities through granular functions like `query_lemmas` and `query_tptp`. These functions return detailed, per-lemma results—pairing each lemma's name with its classified verdict—and support efficient stateful execution by chaining multiple proofs within a persistent, re-usable Isabelle `Session`. This dual-layered design positions `AtpClient` as a robust, interchangeable truth-grounding service without sacrificing the fine-grained control required for advanced Isabelle automation.

This need to balance a unified interface with practical usability also shaped our approach to input validation. Live syntax checking on every keystroke is incompatible with a round-trip to TPTP4X, so we split the linter into two tiers. A pure-Elixir structural checker runs in microseconds to catch common errors (unmatched brackets, unknown role, missing terminator), while an authoritative TPTP4X pass runs only when the local pass is clean. This hybrid approach keeps editor integrations responsive while still providing full syntactic analysis.

A similar pragmatic compromise was necessary for processing prover outputs. Although the SZS ontology provides a principled target for normalization, several mainstream provers either ignore it for some termination paths (E, Vampire) or never emit it at all (SPASS). Following the precedent set by `sledgehammer`, we maintain a small table of prover-specific output patterns alongside the standard SZS table, accepting that this will require updates as new provers are added. Callers that need to bypass our classification and handle non-compliant output themselves can skip this normalization entirely by passing `raw: true`.

Finally, to ensure these abstractions reliably handle the complexities of remote execution, we validated the StarExec backend end-to-end against a self-hosted instance (deployed from a containerized StarExec image [7]). This environment allowed us to exercise the full lifecycle from a running client: form-based login, solver and benchmark upload, job creation, completion polling, and SZS-normalized result retrieval. By using the E prover on representative TPTP problems with expected statuses already known from the local backend, any divergence could be strictly attributed to transport rather than prover behavior. This validation surfaced interoperability quirks that documentation alone misses—such as form-authentication cookies being set only on the post-login redirect, multipart job submission requirements, and job identifiers arriving in redirect `Location` headers. Absorbing these details is precisely the value the library provides. To maintain regression coverage without running live submissions in CI, the smoke-test scripts are shipped with the repository and the captured job JSON and prover `stdout` are saved as offline fixtures.

8. Related Work

The most direct point of comparison is the source of SZS-handling code in Isabelle's `sledgehammer` [2], which we explicitly draw on for the prover-specific output mappings in section 3. Where `sledgehammer` is tightly integrated with Isabelle, AtpClient generalizes the same normalization strategy to arbitrary host applications. Why3 [15] provides a multi-prover backend with sophisticated transformations between proof obligations and prover-specific input formats. AtpClient is narrower in scope (no goal transformation), but supports a different class of host language and integrates an interactive notebook front-end.

In the functional-programming world, the closest tools are written in Haskell. OnlineATPs [16] is a command-line client for SystemOnTPTP that lets a remote ATP be used as if it were installed locally; it overlaps cleanly with our SystemOnTPTP coverage but targets a single backend and the command line rather than a unified, embeddable, multi-backend API. The `atp` library [17] lets users express first-order theorems in Haskell and discharge them with E or Vampire, and the `tptp` library [18] provides a parser and pretty-printer for the CNF, FOF, and TFF subsets of the TPTP language. These operate at the level of constructing and parsing TPTP rather than orchestrating heterogeneous prover services with uniform result and termination semantics, which is our focus; a full TPTP parser such as `tptp` would in fact be a natural component to strengthen the structural tier of our linter.

On the editor side, the closest existing tool is the TPTP Editor extension for VS Code [19], which provides syntax highlighting for `.p` and `.s` files, and a webview front-end to SystemOnTPTP. KinoAtpClient differs in two main respects. First, live linting goes beyond regex tokenization through a two-tier pipeline that combines a structural checker with TPTP4X. Second, the editor lives inside a Livebook notebook, where it shares runtime state with surrounding cells and enables programmatic experiment composition around the same problem.

9. Conclusion and Future Work

We have presented AtpClient, an Elixir library that unifies access to SystemOnTPTP, StarExec, Isabelle, and locally installed provers behind a single result type grounded in the SZS ontology, and KinoAtpClient, a pair of Livebook Smart Cells that turn it into an interactive multi-backend TPTP editor with live linting and one-click prover invocation. The combination shows that a functional, BEAM-based runtime is a good fit for multi-backend ATP integration: portfolio solving becomes a one-liner, fault isolation comes for free, and one notebook surface drives both student exercises and research experimentation.

AtpMcp, our second downstream consumer, makes the architecture's agent-readiness concrete: a Model Context Protocol (MCP) server [20] that exposes the backends as tools for LLM hosts. Using the current MCP revision over standard input/output, it offers a unified `query_backend` tool, routed through the same `query/2` entry point every backend implements, alongside backend discovery and verification, single-prover and portfolio SystemOnTPTP calls, and a linting tool. That this falls out of the architecture as a thin wrapper over the existing behaviour, with no backend-specific logic, is the point: making ATP access uniform at the library level makes it cheap to surface wherever it is needed. We also note one alignment with where the protocol is heading: the current MCP revision introduces an experimental *Tasks* primitive for call-now, fetch-later execution with polling and deferred result retrieval, which maps almost exactly onto the long-running, poll-for-completion character of ATP invocation already embodied in the StarExec and local backends.

Further future work falls along two lines. First, additional backends, including bridges to non-TPTP provers, and richer per-call result metadata such as runtime and proof statistics. Second, the present infrastructure paves the way for a next-generation interactive proof assistant on the BEAM: rather than wrapping Isabelle's proof-document model, we see two promising directions, a TPTP-native proof assistant that builds structured proof terms on top of the same backends AtpClient already supports, or a new specification language better suited to programmatic, multi-backend reasoning. In either direction, AtpClient's uniform result type, notebook integration, and agent-facing MCP server provide a foundation for tool use by LLM-based multi-agent systems.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Opus 4.7 and 4.8 in order to: Grammar and spelling check as well as to support code generation in the corresponding prototype and drafting of section content. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Sutcliffe, System description: SystemOnTPTP, in: D. McAllester (Ed.), Automated Deduction - CADE-17, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 406–410. doi:10.1007/10721959_31.

[2] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, J. Urban, Hammering towards QED, Journal of Formalized Reasoning 9 (2016) 101–148.

[3] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL - A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*, Springer, 2002. doi:10.1007/3-540-45949-9.

[4] G. Sutcliffe, Stepping Stones in the TPTP World, in: C. Benzmüller, M. Heule, R. Schmidt (Eds.), Proceedings of the 12th International Joint Conference on Automated Reasoning, volume 14739 of *LNCS*, 2024, pp. 30–50. doi:10.1007/978-3-031-63498-7_3.

[5] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS Version 3.5, in: R. A. Schmidt (Ed.), Automated Deduction – CADE-22, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 140–145. doi:10.1007/978-3-642-02959-2_10.

[6] A. Stump, G. Sutcliffe, C. Tinelli, StarExec: A cross-community infrastructure for logic solving, in: S. Demri, D. Kapur, C. Weidenbach (Eds.), Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings, LNCS, Springer, 2014, pp. 367–373. doi:10.1007/978-3-319-08587-6_28.

[7] D. Fuenmayor, J. McKeown, G. Sutcliffe, Towards StarExec in the cloud, in: K. Korovin, S. Schulz, M. Rawson (Eds.), Proceedings of the 14th and 15th International Workshops on the Implementation of Logics, volume 21 of *Kalpa Publications in Computing*, EasyChair, 2025, pp. 45–60. doi:10.29007/nsxs.

[8] J. C. Blanchette, T. Nipkow, Nitpick: A counterexample generator for higher-order logic based on a relational model finder, in: M. Kaufmann, L. C. Paulson (Eds.), ITP 2010, volume 6172 of *LNCS*, Springer, 2010, pp. 131–146.

[9] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, LNCS, Springer, 2019, pp. 495–507. doi:10.1007/978-3-030-29436-6_29.

[10] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Springer, 2013, pp. 1–35. doi:10.1007/978-3-642-39799-8_1.

[11] S. Conchon, A. Coquereau, M. Iguernlala, A. Mebsout, Alt-Ergo 2.2, in: SMT Workshop: International Workshop on Satisfiability Modulo Theories, Oxford, United Kingdom, 2018.

[12] K. Korovin, iProver - an instantiation-based theorem prover for first-order logic (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings, volume 5195 of *LNCS*, Springer, 2008, pp. 292–298. doi:10.1007/978-3-540-71070-7_24.

[13] C. Benzmüller, L. C. Paulson, F. Theiss, A. Fietzke, LEO-II - a cooperative automatic theorem prover for classical higher-order logic (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings, volume 5195 of *LNCS*, Springer, 2008, pp. 162–170. doi:10.1007/978-3-540-71070-7_14.

[14] T. Hillenbrand, A. Buch, R. Fettig, WALDMEISTER: High performance equational theorem proving, in: Automated Deduction — CADE-14: 14th International Conference on Automated Deduction Townsville, North Queensland, Australia, July 13–17, 1997 Proceedings, Springer, 1997, pp. 250–253. doi:10.1007/3-540-63104-6_26.

[15] J. Filliâtre, A. Paskevich, Why3 - where programs meet provers, in: M. Felleisen, P. Gardner (Eds.), Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, LNCS, Springer, 2013, pp. 125–128. doi:10.1007/978-3-642-37036-6_8.

[16] J. Prieto-Cubides, OnlineATPs: A command-line client for SystemOnTPTP, Haskell package and source repository, 2018. https://github.com/jonaprieto/online-atps.

[17] E. Kotelnikov, atp: A Haskell interface to automated theorem provers, Haskell package and source repository, 2019. https://github.com/aztek/atp.

[18] E. Kotelnikov, tptp: A parser and pretty-printer for the TPTP language, Haskell package and source repository, 2019. https://github.com/aztek/tptp.

[19] D. Li, G. Sutcliffe, TPTP editor: A VS Code extension for the TPTP language, 2025. https://marketplace.visualstudio.com/items?itemName=DE.tptpeditor.

[20] Anthropic and the Model Context Protocol community, Model context protocol specification, revision 2025-11-25, Specification, 2025. https://modelcontextprotocol.io/specification/2025-11-25.