Elixir meets TPTP

Bringing Automated Reasoning to the BEAM Ecosystem

Paper

Johannes Schuster, David Fuenmayor, Christoph Benzmüller

July 25, 2026

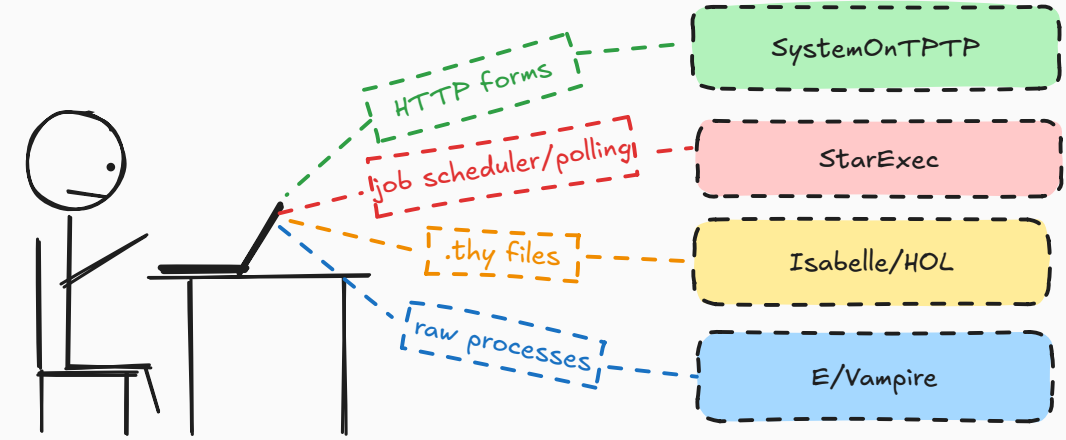
University of Bamberg, Chair for AI Systems Engineering

PAAR'26 @ FLoC, Lisbon

Slides

Demo Files

Integration of ATPs today



script-based, tightly coupled, no uniform way to work across backends

What we are claiming

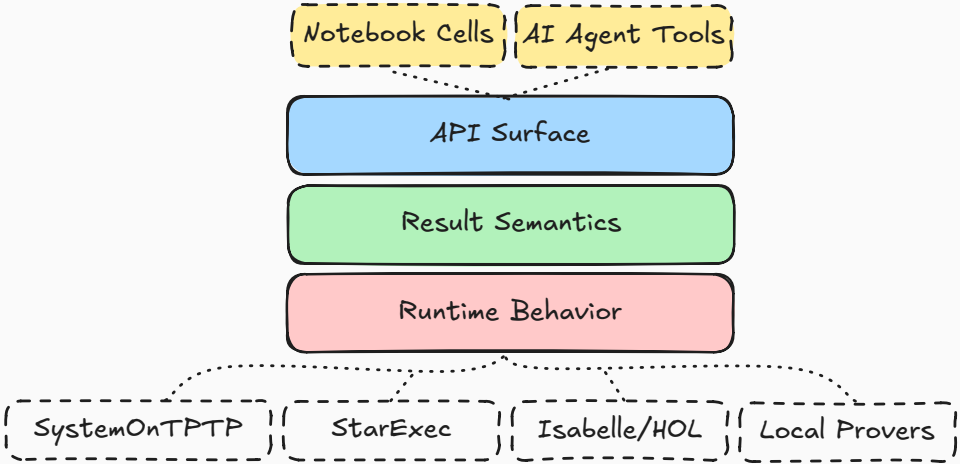
Not a wrapper around one service. But:

One uniform way to use four structurally different backends.

The caller states a problem and a backend, and gets back a verdict in the SZS vocabulary. Transport, polling, and output conventions stay on our side of the interface.

The caller supplies *no* output parser, *no* cancellation mechanism, *no* error handling — you'll see this hold live in both demos.

Our contribution



Backend-specific mechanisms

	SystemOnTPTP	StarExec	Isabelle/HOL	Local
Transport	synchronous HTTP	job scheduler	TCP server	OS binary
Sync mechanism	request/response	poll for completion	poll via session	wait on process
Cancellation	drop request	explicit job-deletion	session link release	kill PID
Quirk	form-encoded	multipart + redirect cookies	needs shared `.thy` on disk	manually killing processes

Four columns, no two alike — the caller should see *none* of it, and one runtime absorbs all four.

Elixir, Erlang, and the BEAM

The language. Functional and gradually-typed, released 2012. Immutable data, process-oriented (actor model), pattern matching. Compiles to BEAM bytecode.

The runtime. The BEAM — Erlang's VM, built at Ericsson from 1992 for telephony. Concurrency, distribution and fault tolerance are runtime primitives, not libraries.

The ecosystem. Phoenix (web), Nx (numerical computing), Livebook (notebooks).
#3 most admired language, Phoenix: #1 web framework (Stack Overflow 2025).

The runtime is the orchestration layer

Preemptive lightweight processes	$\Longrightarrow$	a whole portfolio runs at once — each prover its own process
Supervision & “let it crash”	$\Longrightarrow$	a dying backend is contained; the host keeps proving
Process links	$\Longrightarrow$	“stop everything I started” is free — death propagates, no cancellation tokens
Pattern matching as control flow	$\Longrightarrow$	a dozen SZS result shapes branch readably

We didn’t build an orchestration layer — the runtime is one.

Example: portfolio solving as a single combinator

```elixir
problem = "fof(b, axiom, a(a(a(b,X),Y),Z) = a(X,a(Y,Z))).
fof(w, axiom, a(a(w,X),Y) = a(a(X,Y),Y)).
fof(strongFixpointBW, conjecture, ? [F] : ! [X] : a(F,X) = a(X,a(F,X)))."
AtpClient.SystemOnTptp.list_provers()
|> Task.async_stream(
    fn sys -> {sys, AtpClient.SystemOnTptp.query_system(problem, sys)} end,
    timeout: 30_000, on_timeout: :kill_task,
    ordered: false)
|> Enum.find(&match?({:ok, {_sys, {:ok, :theorem}}}, &1))
# => {:ok, {"Zipperpin---2.1", {:ok, :theorem}}} (Non-deterministic)
```

- `Task.async_stream`: each query is its own preemptively-scheduled process
- `on_timeout: :kill_task`: all unfinished queries are killed at its deadline (the timeout per task); the others are unaffected
- `ordered: false` + `Enum.find`: the first success wins as it arrives; halting the stream shuts the remaining tasks down

One behavior, two surfaces

Every backend implements the same abstract contract — the **`AtpClient.Backend`** behavior, a single `query/2` callback:

```elixir
@callback query(problem :: String.t(), opts :: keyword()) ::
            {:ok, szs_status()} | {:error, term()}
```

So a *consumer* of the library never sees *StarExec* or *Isabelle* specifics — it talks only to `query/2`. Two consumers utilize exactly that:

KinoAtpClient — a Livebook Smart Cell

a TPTP IDE in a notebook: live linting, hover docs, one-click solve, backend dropdown.

AtpMcp — an MCP server

all four backends as one `query_backend` tool for LLM agents.

Demo 1 — Livebook

KinoAtpClient: one notebook cell, all four backends

Demo 2 — Claude Code + AtpMcp

The argument: *all fish live in water; Nemo is a clownfish;*

therefore Nemo lives in water.

Expected: *not* a theorem.

No premise links “clownfish” to “fish” — a faithful formalization should expose the gap.

Related work — and what we adapted

	What it does well	Where we sit
sledgehammer	prover-specific output tables	we **borrowed** them — and generalized the strategy beyond Isabelle to **arbitrary hosts**
Why3	powerful goal transformations	we don't touch the goal; **logistics only**, different host language
OnlineATPs / atp / tptp (Haskell)	CLI and parse-level building blocks	a real TPTP parser like `tptp` would **strengthen** our linter's structural tier
TPTP Editor (VS Code)	closest editor tool	two-tier linting (structural + TPTP4X); lives in a **notebook sharing runtime state** with neighbouring cells

First-class ATP on the BEAM

Recap: uniform result type across four backends · fault-tolerant concurrency · thin notebook and agent surfaces

Future work:

- **more backends** (incl. non-TPTP bridges) and per-call metadata (runtime, proof statistics, etc.)
- a BEAM-native **agent framework** where human and LLM agents collaborate on a shared, SZS-grounded proof state
- a Livebook-based next-generation **ITP system**: TPTP-based or with a new specification language

Our releases

AtpClient — *use your favorite theorem prover*
Hex: `atp-client.hexdocs.pm`
GitHub: `github.com/jcschuster/AtpClient`

KinoAtpClient — *notebook front-end with syntax highlighting*
Hex: `kino-atp-client.hexdocs.pm`
GitHub: `github.com/jcschuster/KinoAtpClient`

AtpMcp — *ATP tools for LLM agents*
Hex: `atp-mcp.hexdocs.pm`
GitHub: `github.com/jcschuster/AtpMcp`

Backup: a unified result type

```
problem = "thf(funExt, conjecture,
    ![F:$i>$i, G:$i>$i]: ( ( ![X:$i]: ( (F @ X) = (G @ X) ) ) => (F = G) )
)."

AtpClient.SystemOnTptp.query_system(problem, "Leo-III---1.8.0")
# => {:ok, :theorem}

AtpClient.SystemOnTptp.query_selected_systems(problem,
    ["Zipperpin---2.1", "E---3.3"])
# => {:ok, [{"Zipperpin---2.1", {:ok, :theorem}},
#           {"E---3.3", {:ok, :theorem}}]}

AtpClient.Isabelle.query(problem, proof_method: "by auto")
# => {:ok, :theorem}    # same problem, same verdict, different backends
```

Elixir in ten seconds: `:theorem` is an *atom* (a symbolic constant); `{:ok, :theorem}` is a *tuple*; callers *pattern-match* on these shapes.

Backup: one caveat before you trust the verdict

Our atoms are SZS statuses lowercased — `:theorem`, `:counter_satisfiable`, `:timeout`, `:gave_up`, and so on.

> `:theorem` is the tool's *claim* of success — nothing more. It is kernel-verified only when the proof method itself runs through the kernel (e.g. `by auto`); Isabelle's sledgehammer verdicts are not re-checked.

How to proceed with non-TPTP backends and provers that don't emit SZS tags?

Backup: normalization policy

Classification layers:

1. Explicit `% SZS status <S>` line
2. Permissive SZS fallback (any CamelCase status name)
3. Prover-specific patterns — 24 patterns / 7 provers, following sledgehammer's known-failures tables (Alt-Ergo, E, iProver, LEO-II, SPASS, Vampire, Waldmeister)
4. Otherwise: `{:error, {:unrecognized_output, raw}}` (no guessing). `raw: true` bypasses classification entirely, on every backend.

Isabelle's tool output can also be directly mapped to SZS.