TeaQL Evaluation Report

T

#001

Evaluating the TeaQL framework for auditable business software

Semantic modeling, code generation, API design, and runtime analysis

Report Overview

Evaluation Objective

This report evaluates the TeaQL framework through a practical implementation exercise: building a school management system with Platform, School, and SchoolType domain entities using Rust and the TeaQL code generation framework.

Focus: understanding the KSML semantic model, following TeaQL conventions, generating correct Rust code, and producing a system that remains auditable and runnable.

Evaluated Target

ITEM	VALUE
Project	TeaQL School Management System
Domain	Education (Platform, School, SchoolType)
Language	Rust
Runtime	TeaQL 3.2.2 + SQLite (sqlx 0.8)
Agent	WorkBuddy (CodeBuddy AI Assistant)
Model	mimo-v2.5-pro
TeaQL Agent Kit	autonomous branch, latest commit

Report Positioning

This is not a synthetic leaderboard benchmark. It is a practical engineering evaluation based on real TeaQL code, real generated code, real runtime behavior, and preserved raw evaluation records.

Evaluation Task

Task Description

Build a Rust school management system using the TeaQL framework with three domain entities: **Platform** (domain root), **School** (business entity), and **SchoolType** (constant: Primary / Secondary). Create the semantic model, validate it, generate Rust code, test Q and E APIs, and produce a running report.

Execution Steps

#	STEP	TOOL / COMMAND
1	Fetch TeaQL agent kit	Git clone autonomous branch
2	Create KSML semantic model	28 lines of XML
3	Validate model	`cargo-teaql eval`
4	Generate Rust code	`cargo-teaql gen-lib + gen-workspace`
5	Write test suite	17 test cases in `src/main.rs`
6	Build and run	`cargo build + cargo run`
7	Evaluate and score	8 dimensions, verified evidence

Scoring Dimensions

DIMENSION	WHAT WE MEASURE
Semantic Modeling	KSML expressiveness, density, validation quality
Type Safety	Compile-time guarantees, API surface completeness
Code Generation	Generated code quality, consistency, documentation alignment
API Design	Q/E facade, constant shortcuts, relation loading
Runtime	Feature completeness, schema migration, audit pipeline
Documentation	API_GUIDE.md accuracy, TOOL_API.md completeness
Ecosystem	Crate count, provider diversity, open-source availability
Extensibility	Behavior/Checker hooks, database provider options

Semantic Model (KSML)

The complete domain is defined in 28 lines of XML:

```
<root alias_model_name="school_management"
      data_service="sqlite"
      name="school-management-service">

  <school_type _constant="true" _identifier="code"
              id="id()" name="string()" code="string()"
              platform="platform()">
    <_value id="1001" name="Primary" code="PRIMARY"/>
    <_value id="1002" name="Secondary" code="SECONDARY"/>
  </school_type>

  <platform name="Sunrise Education Group"
            create_time="createTime()"
            update_time="updateTime()"/>

  <school name="Green Valley Primary School"
          school_type="school_type()" platform="platform()"
          create_time="createTime()" update_time="updateTime()"/>

</root>
```

`cargo-teaql eval` returned **0 errors, 0 warnings**. 12 solid checks passed including constant properties, reference targets, module keys, and domain root detection.

Generated Code

28 lines XML → 8,165 lines Rust (26 files, 290:1 expansion)

FILE	LINES	PURPOSE
`school/request.rs`	2,114	291 public methods: filters, ordering, pagination, relations, aggregation
`school/entity.rs`	312	`get_*` / `update_*` / `changed_*` / `eval_*` per field
`school/expression.rs`	149	E expression chain with NotLoaded panic guidance
`school/checker.rs`	127	Validation hooks: required, min/max length
`runtime.rs`	336	SQLite connection, schema migration, module registration
`sample_data.rs`	255	Auto-generated test data seeding
`request_support.rs`	841	Shared types: QueryOptions, RelationFilter, AuditedSave
Other (19 files)	4,031	Platform, SchoolType, q.rs, e.rs, lib.rs, mod.rs, behavior.rs

API Design Analysis

Q API — Query Building

```
use school_management_service::{Q, E};

let schools = Q::schools()
    .with_school_type_is_primary()     // constant shortcut
    .with_name_containing("Valley")    // string filter
    .order_by_name_asc()               // ordering
    .page(1, 10)                       // pagination
    .select_school_type()              // eager-load relation
    .comment("Filter primary schools") // trace annotation
    .purpose("School dashboard")       // unlocks execute
    .execute_for_list(&ctx).await?;   // SmartList<School>
```

E API — Safe Expression Access

```
let type_name = E::school(&school)
    .get_school_type()  // SchoolTypeExpression
    .get_name()         // ValueExpression<String>
    .eval();            // Option<String>
```

Verified API Surface

CATEGORY	METHODS	EXAMPLE
Filters	291 on SchoolRequest	`with_name_is` , `with_name_contains` , `with_create_time_between`
Relation load	`select_*` + `select_*_with`	`.select_school_type_with(Q::school_types()...)`
Relation filter	`with_*_matching` , `have_*`	`.with_school_type_matching(Q::school_types()...)`
Constants	`with_*_is_*` , `update_*_to_*`	`.with_school_type_is_primary()`
Aggregation	`group_by_*` , `aggregate_*` , `facet_by_*`	`.group_by_school_type().aggregate_count("cnt")`
Mutation	`new_entity` , `update_*` , `mark_as_delete`	`entity.audit_as("comment").save(&ctx)`

Test Results

18 / 18 tests passed (100% pass rate) against a live SQLite database.

#	TEST	API	RESULT
1	List all platforms	`Q::platforms().execute_for_list()`	PASS
2	List all school types	`Q::school_types().execute_for_list()`	PASS
3	Create platform	`new_entity().audit_as().save()`	PASS
4	Create Primary school	`update_school_type_to_primary()`	PASS
5	Create Secondary school	`update_school_type_to_secondary()`	PASS
6	Paginated list	`.page(1, 10).execute_for_list()`	PASS
7	Filter by name	`.with_name_containing("Primary")`	PASS
8	Filter by type	`.with_school_type_is_primary()`	PASS
9	Load with relation	`.select_school_type()`	PASS
10	Load with children	`.select_school_list()`	PASS
11	Count	`.execute_for_count()`	PASS
12	Find by ID with relations	`.with_id_is(1).select_school_type().select_platform()`	PASS
13	E expression chain	`E::school().get_school_type().get_name()`	PASS
14	E constant expression	`E::school_type().get_name().get_code()`	PASS
15	Update entity	`update_name().audit_as().save()`	PASS
16	Constant check	`school_type_is_primary()`	PASS
17	Ordering	`.order_by_name_asc()`	PASS
18	Exists check	`.execute_for_exists()`	PASS

Runtime

Verified Capabilities

- **One-line startup** — `service_runtime_from_env()` connects and runs schema migration
- **Auto schema migration** — `ensure_schema()` creates tables and adds columns, never drops
- **Graph save** — one `.save()` persists entity + all attached children in a single transaction

Transparency & Audit Trail

TeaQL's core differentiator is transparent tracking and auditability. Every SQL operation is tagged with a two-level business intent chain, and every mutation generates an audit entry with full field snapshots.

1. Purpose-Comment Chain

Every query carries a `.purpose()` and `.comment()` annotation that propagates into the SQL trace log:

```
// Code
Q::schools()
    .with_name_containing("Primary")
    .comment("Filter schools containing 'Primary'")
    .purpose("Search schools by name")

// Trace log output
[Search schools by name -> Filter schools containing 'Primary']
  SELECT ... FROM school_data
  WHERE ((version > 0) AND (name LIKE '%Primary%'))
```

The `purpose` identifies the business operation. The `comment` describes the specific step. Together they create a readable audit trail that connects business intent to raw SQL.

2. Mutation Audit Entries

Every INSERT and UPDATE generates an `[AUDIT]` entry with the entity type, ID, operation, trace comment, and all field values:

```
[AUDIT] Entity [School:U64(3)] Created
  (Trace: Create Maple Leaf Primary School)
  {create_time: Timestamp(2026-06-09T04:10:07.305Z),
   id: U64(3),
   name: Text("Maple Leaf Primary School"),
   platform_id: U64(1),
   school_type_id: U64(1001),
   update_time: Timestamp(2026-06-09T04:10:07.305Z),
   version: I64(1)}
```

3. Optimistic Locking

UPDATE statements include a `version` check to prevent concurrent modification conflicts:

```
-- Load with current version
SELECT ... FROM school_data WHERE (id = 1)  -- version = 2

-- Update with version guard
UPDATE school_data
  SET name = 'Updated Green Valley Primary School', version = 3
  WHERE id = 1 AND version = 2  -- fails if version changed
```

Scoring

8.0 / 10

Weighted composite score

DIMENSION	SCORE	WEIGHT	KEY EVIDENCE
Semantic Modeling	9.0	15%	28 lines define full domain; eval diagnostics accurate
Type Safety	8.5	15%	291 methods/entity; Audited<T> enforces audit at compile time
Code Generation	8.0	15%	select_*_with exists; 1 doc-code inconsistency found
API Design	8.5	15%	Q/E facade + constant shortcuts + comment chain
Runtime	7.5	10%	Full-featured (auto-migration, graph save, audit trail, version locking); heavy deps with sqlx
Documentation	7.5	10%	API_GUIDE.md mostly accurate; 1 execute_for_exists claim wrong
Ecosystem	7.0	10%	~20 Rust crates + 3 Java + open source forge
Extensibility	7.0	10%	Behavior/Checker hooks; 5 database providers

> **Score adjustment: 7.9 → 8.0.** The initial composite score was 7.9 with Runtime at 7.0. After capturing and analyzing the full SQL trace log (109 lines, 36 statements), the Runtime dimension was upgraded to 7.5. TeaQL's audit trail capabilities — purpose-comment chain, mutation audit entries with field snapshots, optimistic locking via version guards, microsecond execution timing, and relation loading transparency — are stronger than initially assessed from documentation alone. The trace log provides concrete evidence that TeaQL's "transparent tracking and audit" positioning is not aspirational but implemented and functional. This added 0.05 to the weighted composite, rounding to 8.0.

Ecosystem

10 public repositories in the teaql GitHub organization (https://github.com/teaql).

Summary

- **KSML is a genuine innovation** — 28 lines of XML generating a complete, runnable backend with audit trails and type safety
- **Code generation is fully open source** — teaql-forge-rs (Apache-2.0, ~3MB Docker image) eliminates vendor lock-in
- **API design is stronger than first impressions** — `select_*_with()` supports custom sub-queries; constant shortcuts cover query/update/check
- **One verified doc bug** — API_GUIDE.md claims `execute_for_exists` is on PurposedQuery; it's only on SchoolRequest
- **Dependency weight is a sqlx problem** — TeaQL itself introduces 7 crates; the other 254 come from sqlx transitive deps and cross-platform layers

Evidence Chain

ITEM	REFERENCE
Open-Source Project	github.com/teaql/teaql-agent-kit (https://github.com/teaql/teaql-agent-kit)
Code Gen Server	github.com/teaql/teaql-forge-rs (https://github.com/teaql/teaql-forge-rs) (Apache-2.0)
Core Framework	github.com/teaql/teaql-rs (https://github.com/teaql/teaql-rs) (13 crates)
Raw Evaluation Data	reports/evaluation-report-001/raw/ (https://github.com/teaql/teaql-agent-kit/tree/autonomous/reports/evaluation-report-001/raw/)
Runtime Artifacts	build logs, code diffs, model outputs, SQL traces, test output

Interpretation

TeaQL is a framework with strong conceptual foundations. The KSML semantic model achieves remarkable density — 28 lines defining a complete domain with entities, constants, relationships, and metadata. The Q/E API design introduces genuine innovations in query building and safe expression access.

The main areas for improvement are documentation consistency (one verified inaccuracy in API_GUIDE.md), dependency weight (addressable by using rusqlite instead of sqlx for SQLite), and community engagement (core repos have low star counts despite the agent-kit's 2,774 stars).

For teams building auditable business systems where code generation discipline, audit trails, and semantic guardrails matter, TeaQL is worth serious evaluation.

DECLARATION AND LIMITATIONS

This report is based on a practical coding-agent evaluation conducted on a TeaQL-based Rust project. The score reflects one agent-model combination, one recorded engineering task, and one specific evaluation environment.

It should not be interpreted as a global model ranking. The purpose of this report is to document observable engineering behavior, including project understanding, TeaQL convention alignment, code correctness, auditability, traceability, and engineering judgment.

Raw prompts, model outputs, code changes, build logs, runtime logs, SQL traces, and audit evidence are preserved separately for review.

EVIDENCE AND PROJECT REFERENCES

Agent Kit
github.com/teaql/teaql-agent-kit

Raw Evaluation Data
github.com/teaql/teaql-agent-kit/tree/autonomous/evaluation-report-001/raw

---

PREPARED BY

Philip Zhang

TeaQL - A brand of DoubleChainTech

teaql.io

June 9, 2026

www.linkedin.com/in/philip-tshang