SQL Debug Harness (VS Code Extension) | Technical Design Document

Field	Value
Project	vscode-sql-debug-harness
Extension ID	`DeeprajAdhikary.sql-debug-harness`
Package / CLI	`sql-debug-harness`
Version	0.0.1-beta.1
Engine	In-process TypeScript (`src/engine/`)
License	MIT
Last updated	2026-07-19

---

1. Purpose and scope

1.1 Problem statement

Developers who edit T-SQL stored procedures need a safe way to **analyze** procedure structure and **generate debug harness scripts** without manually commenting out DML, wrapping everything in `BEGIN…ROLLBACK`, or risking writes against shared/production-adjacent databases.

1.2 Solution

SQL Debug Harness ships as a **standalone VS Code extension** with an in-process TypeScript engine. It:

- Parses / scans T-SQL stored procedures (hybrid AST + text scan)
- Rewrites `INSERT` / `UPDATE` / `DELETE` into `SELECT` previews
- Neutralizes TCL (`BEGIN TRAN` / `COMMIT` / `ROLLBACK` / `SAVE TRAN`)
- Stubs `EXEC` calls with `PRINT` diagnostics
- Injects variable traces (`PRINT` or `RAISERROR`)
- Surfaces unsupported constructs (dynamic SQL, cursors, `WHILE` , `MERGE` , `OUTPUT`) as loud warnings

No Python, pip, or external runtime is required beyond VS Code's Node.js host.

1.3 Out of scope (v1)

Item	Rationale
Live debugging / breakpoints	Static rewrite tool only
Live DB connection / execute mode	Credential + rollback edge cases deferred to v2
Multi-dialect (Postgres, Oracle, MySQL)	T-SQL only for v1
Guaranteed rewrite inside cursors / `WHILE`	Detect + warn
Dynamic SQL rewrite	Detect + warn

---

2. Architecture

2.1 High-level flow

```
SQL text
   │
   ▼
prepare (strip GO, deploy preamble, CREATE PROC → DECLARE)
   │
   ├──► node-sql-parser (optional AST; often fails on real T-SQL)
```

```
    │
    └──► text scan (DML / TRY-CATCH / unsupported)
              │
              ├── analyze → Summary / Warnings / Identified (webview)
              │
              └── transform → DML previews, EXEC stubs, TCL neutralize, traces
                                → debug script document
```

2.2 Module layout

Path	Role
`src/extension.ts`	Activation, commands, generate/analyze orchestration
`src/engine/`	Pure TS engine (no `vscode` imports) — testable + CLI
`src/engine/prepare.ts`	GO strip, deploy preamble, `CREATE PROC` → `DECLARE`
`src/engine/scan.ts`	Comment-aware DML / TRY-CATCH line scan
`src/engine/dmlPreview.ts`	DML → SELECT preview builders
`src/engine/execPreview.ts`	EXEC → PRINT stubs
`src/engine/transform.ts`	Orchestration + TCL neutralize + traces
`src/engine/inventory.ts`	Analyze report
`src/engine/unsupported.ts`	Dynamic SQL / cursor / WHILE / MERGE / OUTPUT flags
`src/engine/parser.ts`	Best-effort `node-sql-parser` TransactSQL wrapper
`src/harnessWorkbench.ts`	Workbench webview: source + debug + analysis + active log + per-artifact save
`src/harnessSidebar.ts`	Activity-bar tree view (welcome content host)
`src/configureSettings.ts`	Interactive settings QuickPick
`src/cli.ts`	`npx sql-debug-harness` entry

2.3 Why hybrid (not AST-only)

`node-sql-parser`'s TransactSQL build often fails on enterprise stored procedures (deploy preambles, `TRY/CATCH`, multi-batch scripts). Text/regex scanning remains authoritative for DML detection and rewrites; AST is opportunistic. Correctness over coverage — unsupported constructs are warned, not silently left as live DML.

2.4 Why no Python backend

Shelling out to a PyPI package created adoption friction (PATH, `pip install`, corporate lockdown, dual language maintenance). The engine now ships inside the VSIX via **esbuild** bundling (`node-sql-parser` included). Install = Marketplace / VSIX only.

2.5 Engine API

```
generate(sql, { traceStyle?, stubDml?, addBlockMarkers?, stripComments?, onProgress?, onLog?
  → { sql, stats, parseErrors, stepLog }

analyze(sql, { onLog? })
  → AnalyzeReport { title, isParsable, summary, warnings, identified, stepLog, plainText }
```

---

3. Extension UX

Command	Behavior
`spDebug.generate`	In-process `generate()` → **Workbench** with debug script + log
`spDebug.analyze`	In-process `analyze()` → **Workbench** with analysis sections + log
`spDebug.openWorkbench`	Open workbench for current file/selection

`spDebug.openInWorkbench`	Load a chosen `.sql` file into the workbench
`spDebug.configure` / `openSettings`	Settings UI

The workbench supports **individual saves** for analysis report, debug `.sql`, and step log.

Settings retained: `spDebug.traceStyle`, `spDebug.logToOutput`, `spDebug.saveLogFile`, `spDebug.quietWhenLogging`.

Removed: `spDebug.pythonPath`, `spDebug.pipPackage`, `spDebug.autoInstallBackend`, `spDebug.verifySetup`.

---

4. CLI

```
sql-debug-harness generate -i <file.sql> [-o <out.sql>] [--trace-style print|raiserror]
sql-debug-harness analyze  -i <file.sql>
sql-debug-harness version
```

Shares the same `src/engine/` module as the extension. Exit code **2** from `generate` when warnings are present.

---

5. Build and packaging

Script	Purpose
`npm run compile`	esbuild bundle → `out/extension.js`, `out/cli.js`
`npm test`	Jest suite against `samples/fixtures/`
`npm run package`	Produce `dist/sql-debug-harness.vsix`
`bin.sql-debug-harness`	Optional `npx` CLI sharing the same engine

`vscode` is marked external; all other runtime deps are bundled.

---

6. Testing

Regression fixtures under `samples/fixtures/` cover MVP1 cases: simple DML, `UPDATE…JOIN`, TCL, `TRY/CATCH`, temp tables / table variables, CTE/`MERGE`, dynamic SQL/cursors, pure `SELECT`, `OUTPUT`, malformed input, and the `my_proc` sample. Jest tests live in `src/engine/__tests__/`.

---

7. Security and privacy

- No telemetry.
- No network calls at runtime for generate/analyze.
- MIT license.