GREENWAYS

The Legible Build

A database-first compiler for software that can be reasoned about

A technical and commercial paper on legible source, constrained generation, immediate execution, cross-language consistency, and the product opportunity created by a smaller reasoning surface.

Legible kernel	Typed PostgreSQL definitions express durable behaviour in a compact source that humans and agents can follow
Verified generation	Validation and intermediate representations produce SQL, RPC, schemas, events, and language bindings
Immediate proof	Behaviour is executed and tested before API, state, or user-interface assembly
Native edges	Web and mobile remain native while sharing identifiers, commands, errors, and required state transitions
Commercial thesis	Package the pipeline as an AI-native legible-build compiler, reasoning service, and contract registry

Productivity figures in this paper are modelled scenarios, not measured customer results. Repository counts are point-in-time observations and must be reproduced before external publication.

The thesis

AI makes code inexpensive to produce but makes it easier for a large system to become illegible. The primary advantage of the Greenways pipeline is that important behaviour remains possible to reason about without getting lost. Durable functionality is expressed once in a compact PostgreSQL source language, tested immediately, and expanded through deterministic generation into readable SQL and the contracts required by JavaScript, Dart, realtime runtimes, and application state. Speed, consistency, and maintainability follow from legibility: fewer authoritative decisions, visible derivation, and a bounded reasoning path for each change.

Contents

1 The executive case

Software delivery has acquired an unusual imbalance. Code can now be produced faster than a team can establish whether it belongs in the system. An AI agent can create a migration, service, endpoint, client hook, state store, and mobile model in minutes. Each file may compile. Each may even look competent. The expensive question is whether all of them describe the same behaviour, preserve existing invariants, and remain understandable after the next twenty changes.

The Greenways build pipeline is designed around a different economic unit. It treats *legible, verified behaviour* rather than written code as the output of development. Durable behaviour begins in typed PostgreSQL definitions that are compact enough to read as a coherent domain program. Those definitions are validated against the repository grammar, emitted as readable PostgreSQL, executed directly, and used to generate downstream RPC metadata, view and command descriptions, cross-language bindings, and application-model inputs. Native web and mobile interfaces consume those contracts instead of recreating them.

The economic proposition

Reduce the cost of a feature by keeping its reasoning path small: fewer independently authored representations, readable emitted targets, immediate executable proof, and downstream consistency derived by generation rather than reconstructed from developer memory.

This proposition matters most when five conditions occur together:

- the product contains substantial transactional or permission-sensitive behaviour;
- functionality must remain consistent across more than one client or runtime;
- the system is expected to change for years rather than survive only as a prototype;
- AI agents contribute a significant share of implementation work;
- the cost of inconsistent behaviour is higher than the cost of additional build discipline.

Statstrade satisfies all five. Its product surface includes identity, access, campaigns, tasks, points, assets, workflows, billing, markets, integrations, administration, realtime synchronisation, web applications, and mobile applications. A conventional stack can implement each of these. The difficulty is preventing each layer and platform from developing a slightly different interpretation of the product.

The pipeline does not make complexity disappear. It concentrates complexity into a smaller semantic kernel, grammar, generator, and runtime, while requiring the generated targets to remain inspectable. This creates a high-leverage maintenance core: mistakes in the core can propagate widely, but improvements also apply widely. The correct comparison is therefore not “complex framework versus simple application”. It is “reviewable central complexity versus distributed product entropy”.

1.1 What this paper claims

The paper makes four bounded claims.

1. **A smaller authoritative surface makes the system more legible.** A developer can reason from a domain definition through readable emitted code and tests without reconstructing behaviour from seven or eight independent representations.
2. **Database-first execution reduces time to trustworthy behaviour.** Transactions, authorization, idempotency, state transitions, and failure semantics can be exercised before the application is assembled.
3. **Generation is most valuable when it carries behaviourally meaningful contracts.** Generating table types is useful; generating commands, typed reads, events, errors, and portable state contracts is a more substantial reduction in repeated work.
4. **The pipeline can be productised.** The same mechanisms can become a compiler, contract registry, verified-build service, AI authoring boundary, and set of cross-language runtime packs for other PostgreSQL-backed products.

The paper does not claim that every project should adopt this approach, that database functions should own presentation concerns, or that a measured 20-fold productivity improvement has already occurred. The high-end multiplier is a scenario produced by explicit assumptions. It must be tested against completed slices, comparable implementations, defect histories, and maintenance outcomes.

1.2 The category opportunity

Existing products solve parts of the problem. Prisma generates a typed database client. tRPC propagates TypeScript server types to TypeScript clients. Supabase generates API types by introspecting PostgreSQL. PostgREST exposes database functions as RPC. Buf governs cross-language schemas and breaking changes. Encore derives infrastructure and service topology from application source. These are strong adjacent categories, not evidence that the problem is already solved.

The open category is a *legible AI-native application compiler*: a system in which durable PostgreSQL behaviour, permissions, generated contracts, events, runtime topology, cross-language clients, and verification evidence are produced as one governed and explainable build graph. Productisation should begin with this narrow advantage rather than presenting Greenways as a replacement for every application framework.

2 Why standard builds drift

A standard full-stack application divides responsibility among layers for good reasons. PostgreSQL stores durable data. A server written in TypeScript, Go, Java, or another language implements application behaviour. An API exposes that behaviour. Web and mobile clients implement their own state, caching, validation, and user experience. Each layer can be staffed, deployed, and scaled independently.

The architecture becomes expensive when one product concept is represented independently in too many places. A financially meaningful command may be expressed as:

- a table and migration;
- an ORM model;
- a service method and transaction;
- an API input schema;
- an API output schema;
- a TypeScript domain type;
- a web mutation and optimistic-state rule;
- a Dart model and service;
- a mobile state transition;
- a realtime invalidation or cache-update event.

The issue is not merely duplicated text. Each representation answers behavioural questions: which actor may act, which fields are required, what constitutes the same request, what happens concurrently, what error is returned, which cache becomes stale, and what state the user sees while the operation is pending.

TypeScript improves local reasoning but cannot make these distributed answers identical. Its type annotations are erased and do not change runtime behaviour [1]. Prisma derives a TypeScript database client from a schema [2]; tRPC allows a TypeScript client to infer a TypeScript server router [3]. These mechanisms remove valuable duplication inside their supported boundary. They do not automatically prove that database invariants, non-TypeScript clients, realtime behaviour, or independently authored state machines agree.

2.1 AI increases the production of plausible alternatives

Human teams create drift gradually. AI can create it quickly. An agent working from partial context often finds several plausible local patterns and chooses one. If the repository already contains historical implementations, compatibility layers, generated artifacts, and experimental code, the agent can produce code that is internally reasonable but architecturally wrong.

Common failure modes include:

- adding a parallel API instead of extending the canonical one;
- creating a new local type that resembles but does not derive from the durable contract;
- editing a generated file because it is easier to find than its source;
- using conventional native syntax that the repository emitter does not support;
- widening types or returning generic JSON when exact reconciliation is difficult;
- implementing authorization in a route while leaving another route or client able to bypass it;
- inventing a cache update or error mapping that differs across platforms;
- copying an obsolete pattern because it has more textual examples than the newer abstraction.

The later cost is nonlinear. Once a parallel abstraction exists, subsequent agents use it as evidence. The repository begins to teach its own drift. More generated code does not solve this if the generator covers only types while behaviour continues to be authored independently.

AI productivity research demonstrates why task shape matters. GitHub reported a 55% speed improvement in a controlled JavaScript HTTP-server task [9]. METR later reported a 19% slowdown for experienced developers using early-2025 AI tools on real tasks in repositories they already knew [10]. The studies differ in tools, dates, participants, and methods, so their headline numbers should not be compared as if they were one experiment. Together they support a practical observation: AI performs best when the task is bounded, feedback is mechanical, and the relevant context is small.

2.2 Active code is not the same as repository size

A large generated repository can have a smaller maintenance surface than a smaller handwritten repository. The relevant measure is the code that developers must understand and modify independently when behaviour changes.

This distinction creates the central design objective: keep the application large in capability but small in the number of independently

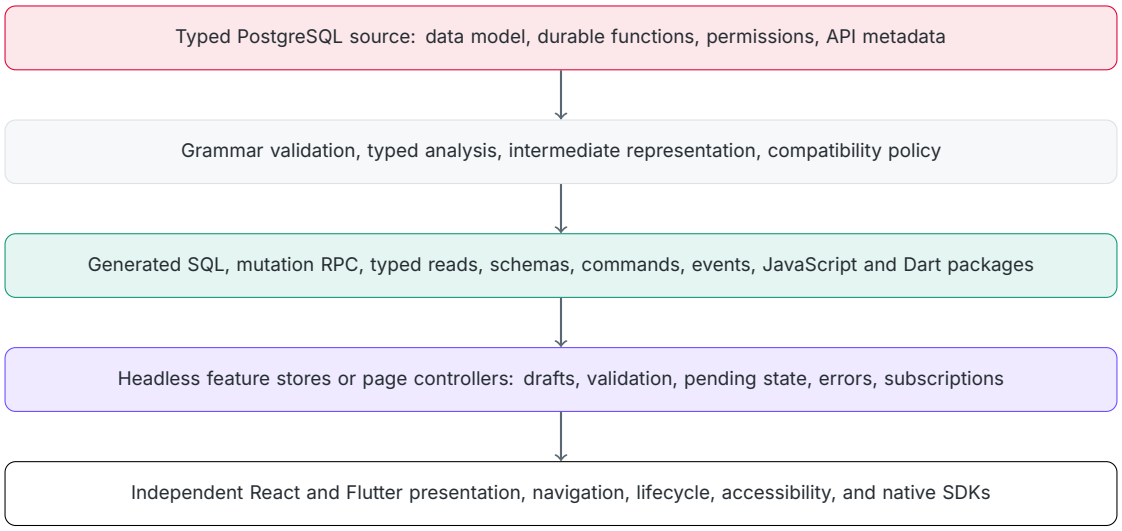
Table 1: Repository volume and active reasoning surface

Measure	Conventional interpretation	Source-owned interpretation
Checked-in lines	More lines usually imply more code to maintain.	Generated artifacts may be large but remain outputs of a smaller source.
Files touched	Every edited representation may carry a new decision.	Regenerated files record consequences of a smaller authoritative change.
Type count	More types may appear to increase safety.	Safety depends on whether types derive from one behaviourally authoritative contract.
Passing build	Layers compile against their local expectations.	Deterministic regeneration and executable contract tests also check derivation.
Maintenance unit	Route, service, client, and state implementation.	Domain behaviour, generator, portable state, and genuinely native presentation.

authored semantic decisions.

3 The Greenways pipeline

The Greenways pipeline begins with PostgreSQL because durable behaviour belongs at the boundary that owns data, transactions, constraints, and permissions. It does not begin with raw SQL. Repository-specific Clojure forms describe types, selectors, returns, and functions within a supported grammar. The source is analysed before it is emitted.



3.1 PostgreSQL as semantic kernel

The database owns behaviour that must remain true regardless of client:

- authorization and least-privilege access;
- transactional state transitions;
- uniqueness, idempotency, and revision conflicts;
- ledger conservation and durable audit records;
- typed reads and bounded mutation outputs;
- concurrency behaviour and rollback;
- domain error codes and completed-change events.

This does not imply that PostgreSQL owns page layout, navigation, animation, device permissions, or framework lifecycle. The database is the semantic kernel, not the entire application.

3.2 Why Clojure changes database-first development

Many application developers avoid PL/pgSQL. A conventional team consequently moves behaviour into a familiar service language even when the database is the stronger enforcement boundary. The Greenways DSL makes database functions available through structural Clojure forms, reusable macros, namespace organization, typed metadata, and immediate emission.

The important advantage is not that Clojure syntax is intrinsically faster to type. Clojure code is data. A source form can be analysed,

validated, transformed, used to infer contracts, and emitted to more than one target. A hand-written SQL function is executable, but it is a less convenient source for a compiler that must also produce RPC metadata, event descriptors, result schemas, and language bindings.

The constraint is deliberate. A developer cannot assume that arbitrary PL/pgSQL syntax is supported. When a required form is missing, the grammar and emitter must be extended with tests. This turns a one-off workaround into reusable system capability.

3.3 Generation beyond table types

Schema-derived types are already useful and commercially established. Supabase generates API types by introspecting database schemas [4]. Prisma generates a typed query client from its schema [2]. Greenways extends the generated boundary from data shape toward application behaviour:

- reads are represented as typed selectors and single-row returns;
- mutations originate as durable functions and become authenticated RPC commands;
- inputs and outputs become cross-language schemas;
- entity mutation metadata becomes event and invalidation information;
- commands, views, identifiers, and errors become XTalk descriptors;
- portable state consumes those descriptors without importing React or Flutter;
- JavaScript uses a shared-worker-owned node while Dart embeds a standalone node.

The native clients may compose different screens. Consistency is required in identifiers, payloads, commands, errors, events, and state transitions, not in pixels.

3.4 Legibility is compression without opacity

Small source is useful only when it remains understandable. A minified program is small but illegible. A proprietary generator can remove handwritten code while making the result impossible to diagnose. The Greenways objective is *semantic compression without opacity*:

- the source names domain types, reads, commands, and errors directly;
- the valid forms are intentionally limited and documented by neighbouring tests;
- the emitted PostgreSQL is readable target code rather than an inaccessible runtime plan;
- every generated artifact can be traced to a source namespace and build stage;
- a functional slice can be reviewed as a bounded packet: source, emitted SQL, direct tests, generated contract, and state tests;
- native presentation remains outside the compiler where platform-specific reasoning is genuinely required.

This provides reasoning locality. A reviewer should not need the entire repository in working memory to answer what a command does, who may call it, what it returns, which events follow, and which clients consume it. An AI agent benefits from the same boundary because the authoritative context can fit inside a smaller prompt and be checked against executable evidence.

The legibility test

For any durable command, a new maintainer should be able to locate its source, read its invariants, inspect the emitted PostgreSQL, run its focused tests, identify its generated consumers, and explain its failure semantics without searching for parallel implementations across the product.

3.5 A narrow source, a wide artifact graph

At the point-in-time measurement used for this working paper, the repository contains approximately 18,235 lines of GWDB domain source, 3,806 lines of Greenways build code, and 2,476 lines of GWLink source. Relevant Foundation XTalk source families add approximately 25,243 lines. These sources support roughly 109,135 lines of tracked SQL artifacts and 40,187 lines of generated JavaScript and Dart libraries.

The ratio is not a productivity benchmark. SQL monoliths and deployment slices duplicate material; generic runtime code serves more than one application; line density differs by language; and native UI remains outside the count. The observation nevertheless demonstrates artifact amplification: a materially smaller source and generator surface controls a much larger executable and cross-language output surface.

Reproduction boundary

Counts were taken from tracked files in the V2 repository and pinned Foundation checkout on 20 July 2026 using directory-level line totals. Before external publication, replace these point estimates with a checked-in measurement script that classifies source, generated, test, vendored, duplicated, and native-presentation files and records the exact V2 and Foundation revisions.

4 Quality begins before application assembly

The pipeline's most important acceleration is not generation. It is the ability to test meaningful functionality immediately after the database source emits.

In a conventional application, proving one workflow may require a migration, database client, service, endpoint, authentication middleware, serializer, frontend query, state store, and test fixture. The first end-to-end proof arrives after several layers have been assembled. A failure can belong to any of them.

In the Greenways sequence, the difficult part is executable earlier:

typed database function → emitted SQL → direct behavioural test → generated contract → headless state test → native UI

4.1 What direct database tests establish

Database execution can establish properties that local client types cannot:

- an unauthorized actor cannot invoke the operation through any client;
- a failed mutation rolls back all related changes;
- the same idempotency key cannot charge or award twice;
- concurrent operations preserve a ledger or uniqueness invariant;
- revision checks reject stale administrative changes;
- a durable state transition rejects invalid predecessor states;
- the returned row or object matches the declared function contract;
- errors are stable enough for generated clients to interpret.

This changes the meaning of a vertical slice. The UI no longer serves as the first available test harness for domain behaviour. It becomes a presentation and interaction layer over behaviour already demonstrated independently.

4.2 Quality as a chain of evidence

The pipeline can produce a stronger evidence chain than "the code compiled":

1. **Grammar evidence:** the source uses supported forms rather than plausible native syntax.
2. **Emission evidence:** the target PostgreSQL is inspected and parses as intended.
3. **Behaviour evidence:** direct tests exercise authorization, transactions, results, and failures.
4. **Generation evidence:** RPC, views, schemas, events, and language packages are current.
5. **Determinism evidence:** a second generation produces no additional changes.
6. **State evidence:** feature stores or page controllers pass without a UI runtime.
7. **Topology evidence:** the browser shared worker and standalone Dart node honour the same logical contract.
8. **Presentation evidence:** React and Flutter are tested independently for their native responsibilities.

Each step narrows the location of a defect. A database failure is not confused with a React lifecycle problem. A generated-schema mismatch is not hidden behind a mobile serializer. A platform presentation can evolve without changing the durable meaning of a command.

4.3 Why this is particularly suitable for AI

An AI agent performs better when the valid vocabulary is narrow and failure is immediate. The repository can provide:

- canonical neighbouring functions rather than a broad language manual;
- an explicit grammar and naming convention;
- a generator that rejects unsupported or ambiguous forms;
- focused tests that run without assembling the whole application;
- emitted source that a reviewer or another agent can inspect;
- a clean-generation check that catches source/artifact drift;
- architecture rules that prevent generated or experimental code becoming precedent.

This converts free-form application generation into constrained domain implementation. The agent is still capable of making an incorrect business decision. It has fewer opportunities to create ten mutually inconsistent technical interpretations of that decision.

The quality distinction

Most AI coding tools optimize the rate at which candidate code appears. A verified build pipeline optimizes the proportion of candidate code that survives semantic execution, cross-language consistency, review, and future change.

4.4 Where quality can still fail

Generation is not evidence that the generated behaviour is good. A defect in a grammar, inference rule, authorization policy, or runtime can propagate across every consumer. This concentrated risk requires stronger controls:

- small and versioned intermediate representations;
- golden and semantic emitter tests;
- compatibility checks against previously published contracts;
- explicit source ownership for every generated artifact;
- differential or property tests for important financial functions;
- provenance linking artifacts to exact source and dependency revisions;
- staged rollout for generator and runtime changes.

The result is a smaller but more consequential core. That is preferable only when the core is treated as product infrastructure rather than ordinary application glue.

5 A model for the delivery advantage

The pipeline's advantage should be expressed as a model that can be disproved. "Twenty times faster" is too broad. A useful unit is quality-adjusted functionality:

$$Q = \frac{\text{accepted functionality that remains consistent after change}}{\text{authoring + review + rework + maintenance effort}} \tag{1}$$

The comparative multiplier can be decomposed as:

$$M = G_{implementation} \times G_{coherence} \times G_{lifecycle} \tag{2}$$

The factors are separated for reasoning, but they are not perfectly independent. Any empirical study must avoid double-counting the same saved work in more than one factor.

5.1 Consistency relationships

Let n be the number of independently authored semantic representations of one feature. If every representation may disagree directly with every other representation, the maximum number of consistency relationships is:

$$E_{pairwise} = \binom{n}{2} = \frac{n(n-1)}{2} \tag{3}$$

Eight representations produce:

$$E_{standard} = \frac{8 \times 7}{2} = 28 \tag{4}$$

Two authoritative representations produce one relationship. This gives a 28-fold reduction in the broad consistency graph.

A disciplined conventional architecture resembles a star: every consumer follows one API contract. Its minimum coordination graph contains $n - 1$ relationships. Eight representations therefore produce seven contract relationships rather than twenty-eight. The defensible structural range is consequently about 7 to 28 times fewer relationships, depending on how effectively the standard project maintains a single centre.

This is not a delivery-speed result. It is the mathematical reason that a 20-fold consistency-maintenance effect is possible.

5.2 First-pass coherence

Let p be the probability that one effectively independent semantic surface is correct after an implementation pass, and let n be the correlation-adjusted number of surfaces. A deliberately simple approximation is:

$$P_{coherent} = p^n \tag{5}$$

For an AI-drift scenario with eight effective conventional surfaces and 85% per-surface correctness:

$$P_{standard} = 0.85^8 = 0.2725 \tag{6}$$

For two Greenways surfaces with 95% per-surface correctness after grammar and test constraints:

$$P_{Greenways} = 0.95^2 = 0.9025 \tag{7}$$

The modelled first-pass coherence ratio is:

$$G_{coherence} = \frac{0.9025}{0.2725} = 3.31 \tag{8}$$

This does not mean that only 27% of conventional pull requests merge. It means that the remaining work requires correction, reconciliation, broader testing, or later defect repair before it reaches the same modelled semantic coherence.

5.3 The 20-fold scenario

The high-drift scenario assumes:

- 2.2-fold less independently authored implementation and glue;
- 3.31-fold higher first-pass semantic coherence;
- 2.8-fold less lifecycle reconciliation and drift repair.

$$M = 2.2 \times 3.31 \times 2.8 = 20.4 \tag{9}$$

Table 2: Sensitivity of the quality-adjusted multiplier

Scenario	Standard coherence	Greenways coherence	Lifecycle factor	Total
Conservative, disciplined	77%	94%	1.3	2.4×
Expected large project	48%	92%	1.8	6.9×
Strong AI drift	27%	90%	2.8	20.4×

The model says that 20-fold improvement should not be expected in a small, disciplined, web-only TypeScript project. It becomes plausible when many semantic surfaces, multiple languages, rapid AI authorship, high change frequency, and expensive inconsistency occur together.

5.4 Context-token compression

The delivery multiplier should be kept separate from a second, potentially larger effect: the reduction in tokens required for an AI agent to reason about one change.

Let:

- B be the token count of the broad repository context an unconstrained agent might search, retrieve, or repeatedly inspect;
- R be the token count of the smallest complete reasoning packet selected from the compiler graph.

The context-compression ratio is:

$$K_{context} = \frac{B}{R} \tag{10}$$

A five-million-token application context reduced to a fifty-thousand-token reasoning packet gives:

$$K_{context} = \frac{5,000,000}{50,000} = 100 \tag{11}$$

This makes a 100-fold token reduction plausible, but only when the baseline is broad repository context. It is not a claim that every existing developer task currently consumes the whole repository. Strong human navigation and retrieval tools already select smaller contexts.

The point-in-time V2 measurement found approximately 37.9 MB of tracked text across common source, generated code, documentation, configuration, and tests. Using a rough four-bytes-per-token heuristic gives about 9.5 million tokens. The maintained Greenways database/build/link surface measured approximately 0.91 MB, or roughly 228,000 tokens. A minimal Contact slice packet containing its type, helper, user command, RPC-facing definition, and focused tests measured only 6,085 bytes, or roughly 1,500 tokens before dependencies, emitted SQL, and explanatory evidence were added.

Those figures imply three different comparisons:

Table 3: Illustrative context-compression boundaries

Baseline	Comparison	Interpretation
All tracked repository text	9.5M tokens versus a 10–50k packet	Approximately 190–950×; relevant when an agent otherwise searches generated, historical, documentation, and application surfaces broadly.
Maintained core source	228k tokens versus a 10–50k packet	Approximately 5–23×; relevant after the agent already knows the canonical architecture.
Minimal source/test slice	About 1.5k tokens before dependencies	Not yet a complete reasoning packet; demonstrates how small the source owner can be, not a usable standalone context ratio.

The product objective should therefore be explicit: automatically produce the smallest *complete* context, not merely the smallest context. Removing a required permission helper or event consumer makes the prompt cheaper and the answer worse. The compiler graph is valuable because it can select dependencies by semantic ownership rather than textual similarity.

Before publication, exact tokenizer counts should replace the bytes-per-token heuristic, generated and duplicated files should be classified separately, and several real slice packets should be measured. The 100-fold number should be presented as a target range demonstrated on specified repository baselines.

5.5 A second model: cost over repeated change

Let:

- F be the number of functional slices;
- T be the average number of material contract changes per slice;
- a be authoritative implementation cost;
- c be the average cost of reconciling one independent consumer;
- n be the number of independently maintained consumers.

A simplified conventional lifecycle cost is:

$$C_{standard} = F\left(a + Tnc\right) \tag{12}$$

If generation reduces independently reconciled consumers from n to g, where g is close to one or two:

$$C_{generated} = F\left(a_g + Tgc + C_{platform}\right) + C_{core} \tag{13}$$

C_{core} is the fixed cost of grammar, generator, runtime, and compatibility infrastructure. The standard approach wins when F and T are small. The generated approach wins when enough slices and changes amortize the core.

This describes the expected project curve:

Product stage	Expected economics
One to five slices	Foundation cost dominates; a standard application can reach the first screen faster.
Five to fifteen slices	Reuse approaches break-even; generator defects and missing forms still consume meaningful effort.
Fifteen to forty slices	Cross-language consistency and repeated domain patterns create a strong cumulative advantage.
More than forty slices	Avoided translation, drift, and maintenance can dominate the original implementation savings.

5.6 What must be measured

The model parameters should be replaced with observed values:

- elapsed and active engineering time per accepted slice;
- source-authored versus regenerated files and lines;
- number of semantic corrections found at each gate;

- escaped defects by source layer and platform;
- time required to make one contract change across all consumers;
- review comments attributable to invented or stale abstractions;
- deterministic-generation failures and generator regressions;
- maintenance time after three, six, and twelve months.

Until those measurements exist, the conservative range belongs in external claims and the 20-fold result belongs in a clearly labelled scenario.

6 Comparison with adjacent architectures

The pipeline should be compared with strong alternatives, not with deliberately poor software. Four archetypes reveal where its claimed advantage begins and ends.

6.1 TypeScript monorepo: Prisma and tRPC

A modern TypeScript monorepo using PostgreSQL, Prisma, tRPC, Zod, React, and a shared package graph provides excellent web development ergonomics. Prisma generates query types from a schema [2]; tRPC carries router types into TypeScript clients [3]. When the server and all important clients share one TypeScript compilation boundary, the effective consistency surface is substantially smaller than traditional REST with handwritten DTOs.

Its limitations appear when durable behaviour is distributed between application services and the database, when runtime validation differs from compile-time types, or when a non-TypeScript client must reproduce the contract. Adding Flutter, native workers, external consumers, or database-enforced financial invariants weakens the single-language advantage.

Modelled comparison

Expected Greenways advantage: 2–4× for a web-only product; 4–8× when equivalent Flutter behaviour and long-lived transactional consistency are required. These are scenario ranges, not measurements of any named project.

6.2 Multi-client application server: Mattermost-like architecture

Mattermost documents access clients, an application-server layer, REST/JSON APIs, and backend infrastructure [6]. This is a proven way to build a large product. It also illustrates the coordination cost of a server language, PostgreSQL, web clients, mobile clients, realtime systems, caching, and release compatibility.

A mature organization manages these boundaries with specialist teams, compatibility policies, integration tests, release trains, observability, and years of accumulated knowledge. The architecture is not low quality. Its consistency is organizationally expensive.

Modelled comparison

Expected Greenways advantage: 3–6× for routine durable functionality, 6–12× for consistent web/mobile delivery, and potentially 12–25× under repeated AI-authored change. The higher range represents avoided coordination and drift, not raw server coding speed.

6.3 Database-generated APIs: Supabase and PostgREST

Supabase derives API types from PostgreSQL schemas [4]. PostgREST exposes tables, views, and accessible database functions, including functions under an RPC prefix [5]. These systems already validate the core premise that PostgreSQL can own more of an application's interface.

Greenways adds a source-language and build-graph layer above that capability. The intended differentiators are typed durable behaviour, mutation-specific exposure policy, typed read models, event inference, cross-language command and view descriptors, headless state, browser worker topology, standalone Dart topology, and deterministic multi-artifact generation.

Modelled comparison

Expected Greenways advantage: 1–2× at the database/API boundary and 2–5× for the full cross-language application contract. A team needing only generated CRUD should prefer the simpler tool.

6.4 Schema registries: Protobuf and Buf

Buf demonstrates the commercial value of treating schemas as governed products. Its tools generate SDKs and mechanically detect breaking changes during development, review, and registry publication [7]. Greenways can borrow this product pattern while operating at a different semantic level.

Protobuf defines messages and services. A Greenways contract registry would also understand database functions, permissions, transaction-facing commands, typed reads, events, errors, runtime capabilities, and generated state descriptors. The registry opportunity is not to compete with Protobuf at general transport schemas. It is to govern a PostgreSQL-backed application contract from durable behaviour through client consumption.

6.5 Application and infrastructure compilers: Encore

Encore parses application source to derive service and infrastructure topology and presents itself as infrastructure for humans and agents [8]. This validates an adjacent product thesis: constrained source plus whole-application analysis can make AI development safer and operations more automatic.

Greenways is differentiated by beginning at typed PostgreSQL behaviour and extending toward cross-language clients and state. Encore is differentiated by backend service and cloud infrastructure automation. The products could overlap, compete, or integrate depending on how far each expands.

6.6 The comparison table

Table 4: Architectural ownership comparison

Capability	TypeScript monorepo	Multi-client server	Database-generated API	Greenways pipeline
Durable behaviour	App service plus database	Application server	Database function or app code	Typed database function
Web types	Inferred or generated	Handwritten/generated from API	Generated from database	Generated contract
Non-TypeScript client	Separate contract path	Separate SDK/model	Client-library dependent	Generated Dart contract
State transitions	Web application state	Per-client state	Application-owned	Portable headless state where appropriate
Realtime/cache	Application integration	Server and per-client integration	Provider primitives	Shared logical model and runtime descriptors
Immediate domain testing	Service/database test	Server integration test	Database function test	Emitted database behaviour test
Breaking-change control	Typecheck and CI	API compatibility process	Schema diff and CI	Planned semantic contract registry
Best fit	TypeScript web products	Large staffed platforms	CRUD/data-centric APIs	Domain-heavy, AI-built, multi-runtime products

The comparison suggests a narrow initial market. Greenways should not sell "types for PostgreSQL". That category is crowded and insufficiently differentiated. It should sell verified behavioural propagation for complex PostgreSQL-backed products built by humans and AI across more than one runtime.

7 Productising the legible build

The build process can become a product if it is packaged around an outcome customers already feel: their AI-assisted codebase is growing faster than they can understand it. The product promise should not begin with Clojure, XTalk, or a claim to replace application frameworks. It should begin with a simpler statement:

> **Working product promise**
>
> Build complex PostgreSQL-backed applications with AI without losing the ability to explain, test, and change what the system does.

Productisation should expose the reasoning path that makes the internal pipeline valuable. An opaque hosted generator would discard the central advantage.

7.1 Product layer one: the local compiler

The narrowest product is a local developer tool that accepts typed PostgreSQL domain source and produces verified artifacts.

Core capabilities would include:

- typed schemas, selectors, returns, and durable mutation functions;
- grammar validation and actionable diagnostics;
- readable PostgreSQL emission;
- generated RPC wrappers, input/output schemas, events, and errors;

- JavaScript, Dart, and optional additional language bindings;
- deterministic generation and stale-artifact detection;
- focused local PostgreSQL execution and test scaffolds;
- source maps from every artifact back to its semantic definition.

The local compiler establishes developer trust. It must work offline, produce ordinary inspectable files, and avoid requiring customers to send proprietary domain source to a hosted service.

7.2 Product layer two: the reasoning workspace

The most differentiated product may be a reasoning workspace rather than a generator dashboard. For a selected command or slice, it would assemble a bounded *reasoning packet*:

- the authoritative source definitions and their dependencies;
- emitted SQL with source correspondence;
- permissions and actor paths;
- inputs, outputs, errors, events, and state transitions;
- direct test results and uncovered branches;
- affected generated consumers and compatibility changes;
- exact source, generator, runtime, and artifact revisions;
- a semantic diff explaining what behaviour changed.

This is valuable to human reviewers and AI agents. Instead of retrieving large quantities of text by similarity, an agent receives the exact semantic slice selected by the compiler graph. Generated files can be marked read-only to the agent, while source-owned files and valid extension points remain writable.

The workspace should report its context-compression result: repository tokens considered, authoritative tokens selected, evidence tokens added, excluded generated or historical surfaces, and the reason every included dependency is required. A modelled 100-fold reduction becomes commercially meaningful when the customer can inspect the packet and reproduce the selection rather than trusting a hidden retrieval score.

Potential interfaces include a CLI, IDE panel, pull-request report, web console, and MCP server. The MCP interface should answer questions such as “which source owns this RPC?”, “what will regenerate if this field changes?”, “which actors can call this command?”, and “show the smallest complete context required to modify plan redemption.”

7.3 Product layer three: the contract registry

A hosted or self-managed registry would store versioned semantic contracts and generated artifacts. Buf demonstrates that organizations pay for schema governance, breaking-change enforcement, and generated SDK distribution [7]. A Greenways registry would apply this model to database-owned application behaviour.

Registry capabilities could include:

- immutable versions of typed database and application contracts;
- semantic compatibility policies for functions, reads, errors, and events;
- generated SDK publication for supported languages;
- provenance and cryptographic digests for source and generated artifacts;
- organization-wide grammar and security policies;
- consumer graphs showing which applications depend on each contract;
- deprecation windows and migration guidance;
- audit evidence for sensitive or regulated changes.

The registry creates network effects inside each customer organization: every additional service or client increases the value of one governed source.

7.4 Product layer four: verified build cloud

The managed service would execute builds in isolated environments and return evidence rather than merely a green checkmark.

A verified build could:

1. validate source and policy;
2. emit PostgreSQL and all selected language targets;
3. create an ephemeral PostgreSQL environment;

4. run direct behavioural, permission, and migration tests;
5. regenerate a second time and require no diff;
6. run client package analysis and compatibility fixtures;
7. produce a signed manifest containing exact inputs, outputs, digests, and results;
8. optionally create a preview application environment.

The commercial unit is a verified contract build, not generic CI minutes. The service can charge according to contract modules, target languages, build frequency, retained provenance, policy enforcement, and private execution requirements.

7.5 Product layer five: runtime and domain packs

Customers should not need the complete Statstrade runtime. Package reusable capability in layers:

- **Runtime packs:** PostgREST/Supabase RPC, realtime events, browser workers, standalone mobile nodes, caching, and synchronisation.
- **State packs:** forms, collections, CRUD, sessions, workflows, feedback, and administrative controllers.
- **Domain packs:** audited ledgers, revisioned configuration, entitlements, workflows, messaging, billing boundaries, or marketplace primitives.
- **Policy packs:** multi-tenant access, privileged administration, PII redaction, audit requirements, and compatibility profiles.

Domain packs should remain source-visible and overridable through declared extension points. Selling opaque templates would recreate the reasoning problem the product is intended to solve.

7.6 Product layer six: migration and assessment

An initial services-led offer can identify where a customer's existing system has become illegible. The assessment would map one representative workflow across schema, services, API types, clients, state, and tests; calculate its active reasoning surface; and rebuild the slice through the pipeline.

The deliverable is measurable:

- before-and-after authoritative surface count;
- before-and-after consistency relationships;
- time to locate and explain the workflow;
- time to introduce one controlled contract change;
- direct test coverage and generated consumer coverage;
- migration plan for adjacent slices.

This creates design partners, produces comparative evidence, and reveals which parts of the internal pipeline are genuinely reusable.

8 Commercial model and route to market

The product should begin where the pain and willingness to pay are strongest. A small web application with one TypeScript client can already achieve excellent productivity with Prisma, tRPC, or Supabase types. Greenways should not ask that team to adopt a new semantic compiler without a correspondingly difficult problem.

8.1 Initial customer profile

The best early customer is likely to have:

- PostgreSQL as the durable system of record;
- five to fifty software engineers or an AI-heavy development model producing equivalent change volume;
- both web and non-TypeScript clients, or several independently deployed consumers;
- permission-sensitive, financial, workflow, marketplace, or operational behaviour;
- visible drift among schema, API, client types, state, and documentation;
- long review cycles because senior engineers must reconstruct context;
- a willingness to adopt source generation but not a proprietary runtime lock-in.

Candidate sectors include fintech infrastructure, marketplaces, logistics, health administration, community platforms, enterprise operations, and AI-native software companies. Regulated customers may value provenance and executable policy evidence, but the product should not claim compliance certification until its controls have been independently assessed.

8.2 Packaging

Table 5: Illustrative product ladder

Offer	Buyer outcome	Commercial model
Compiler community edition	Local source validation, readable SQL, core generation, and one language target.	Open source or source-available adoption layer.
Team workspace	Reasoning packets, semantic diffs, shared policies, CI reports, and retained build evidence.	Per-team subscription plus verified-build usage.
Contract registry	Versioned modules, generated SDKs, compatibility enforcement, provenance, and consumer graphs.	Organization subscription based on modules, users, and retention.
Enterprise platform	Private runners, SSO, policy administration, audit export, support, and self-managed registry.	Annual platform agreement.
Runtime/domain packs	Supported cross-language runtimes and pre-verified domain capabilities.	Annual pack subscription or commercial licence.
Assessment and migration	Before-and-after reasoning-surface analysis and one migrated proof slice.	Fixed-scope professional service leading into platform adoption.

Pricing should be anchored to avoided review, rework, and platform duplication rather than seats alone. A customer that removes one week of senior reconciliation from every material cross-platform change can justify a much higher price than a customer using the compiler only to generate DTOs.

Any initial price bands should be treated as interview hypotheses. A possible discovery range is:

- team workspace: hundreds to low thousands of Australian dollars per month;
- private registry and verified builds: low tens of thousands per year;
- enterprise platform and supported runtime packs: mid five to low six figures per year;
- assessment and proof-slice migration: a fixed engagement priced against senior engineering time and risk.

These bands require validation against procurement expectations, deployment model, support burden, and the maturity of the compiler.

8.3 Open core and trust

A compiler that controls application behaviour faces a trust barrier. Customers must be able to inspect generated code, reproduce builds, and retain their application if the vendor disappears. This favours an open-core or source-available strategy:

- keep the source language, local compiler, artifact formats, and basic runtimes available;
- charge for organization governance, hosted verification, private execution, registry, analytics, enterprise policy, and supported packs;
- publish compatibility specifications so customers are not trapped by an undocumented intermediate representation;
- make every hosted build reproducible with exact compiler and dependency versions.

The moat should not be hidden output. It should be the accumulated grammar, emitter correctness, semantic compatibility engine, cross-language runtimes, test corpus, domain packs, provenance network, and quality of the reasoning experience.

8.4 Go-to-market sequence

1. **Prove internally.** Complete several representative Statstrade slices and record source size, elapsed time, failures caught, generated targets, and maintenance changes.
2. **Publish the method.** Release this paper, a reproducible benchmark, and one end-to-end example whose source, SQL, JavaScript, Dart, tests, and semantic diff can be inspected.
3. **Run design-partner assessments.** Select teams with real cross-platform or AI-drift pain and migrate one bounded workflow.
4. **Package the reasoning workspace.** Productise source ownership, semantic slicing, artifact provenance, and generated-impact explanation before building a broad cloud platform.
5. **Add registry governance.** Introduce compatibility policies, retained versions, SDK distribution, and organization controls after several teams share contracts.

6. **Expand supported targets carefully.** Add languages and runtimes only when fixtures prove semantic equivalence; do not turn an accurate narrow compiler into a broad approximate one.

8.5 Defensibility

The source syntax alone is not defensible. A competitor can create another DSL. Defensibility accumulates in:

- a large corpus of emitted PostgreSQL and semantic tests;
- precise inference from durable behaviour into commands, events, and errors;
- cross-language runtime equivalence;
- compatibility history and migration intelligence;
- organization-specific policy graphs;
- benchmark evidence showing reduced reasoning and review cost;
- domain packs proven under real operational workloads;
- trust created by readable artifacts and reproducible builds.

The product becomes more valuable as it remembers how a customer's semantic system evolved. That history should remain exportable and auditable rather than becoming a lock-in mechanism.

9 Validation, risks, and roadmap

The paper should be published as a thesis with a measurement programme, not as a retrospective claiming results that have not been observed. The strongest next step is to make legibility measurable.

9.1 The legibility benchmark

Select three representative slices:

- a permission-sensitive administrative workflow;
- an idempotent ledger or entitlement operation;
- a realtime cross-platform workflow with drafts and failure recovery.

Implement or reconstruct each slice in three architectures:

1. a strong TypeScript monorepo using Prisma, runtime schemas, and tRPC;
2. a database-generated API using PostgreSQL functions and PostgREST/Supabase types;
3. the Greenways typed PostgreSQL, XTalk, and headless-state pipeline.

Where practical, add a non-TypeScript mobile client to each. Use equivalent acceptance tests and security requirements.

Measure:

- authoritative files and semantic decisions;
- total files generated or changed;
- time to first executable domain proof;
- time to first complete web and mobile behaviour;
- review time and number of clarification cycles;
- defects injected into hidden edge cases;
- context required for a new developer or agent to explain the slice;
- time to apply a later breaking and non-breaking change;
- difference between generated and deployed contracts;
- escaped inconsistencies after a fixed maintenance period.

9.2 Reasoning-surface metrics

Line count is insufficient. The benchmark should introduce metrics specifically for legibility:

The time-to-explanation test is especially important. If the pipeline produces correct code that only its original author can understand, it has failed its central claim.

Table 6: Candidate legibility metrics

Metric	Definition
Authoritative surface count	Files or definitions in which behaviour may be changed independently without regeneration.
Semantic decision count	Distinct choices about authorization, validation, state transition, errors, events, and data shape.
Trace distance	Number of source and build steps from a durable function to one consuming UI action.
Reasoning packet size	Tokens, lines, or graph nodes required to explain and safely modify one slice.
Context compression	Ratio between the declared baseline repository context and the smallest complete compiler-selected reasoning packet.
Parallel implementation count	Number of independently authored implementations of the same rule.
Source-to-artifact coverage	Percentage of shipped behaviour with a machine-readable source owner and provenance path.
Executable explanation rate	Percentage of documented invariants directly exercised by focused tests.
Change fan-out	Number of handwritten edits required for one contract change across all supported consumers.
Time to explanation	Time for an unfamiliar engineer to produce an accurate account of behaviour and impact.

9.3 Technical risks

- **Centralized blast radius.** A generator defect can affect every consumer. Mitigation: versioned IR, semantic tests, staged release, and compatibility fixtures.
- **DSL ceiling.** Unsupported PostgreSQL behaviour may tempt raw escapes or application-layer duplication. Mitigation: publish the supported grammar, extension process, and target conformance tests.
- **Opaque magic.** Aggressive inference can make behaviour surprising. Mitigation: readable outputs, source maps, explain commands, and explicit override points.
- **Database overload.** Not every computation belongs in PostgreSQL. Mitigation: define boundaries for durable transactions, asynchronous workers, external integrations, and CPU-heavy tasks.
- **False cross-platform parity.** Generated types may compile while runtime semantics differ. Mitigation: shared fixtures plus real JavaScript and Dart execution.
- **Framework resistance.** Teams may reject Clojure or a new build system. Mitigation: generated ordinary artifacts, gradual adoption, strong editor tooling, and a narrow proof slice.
- **Commercial overreach.** Building a hosted platform before proving external demand can consume the project. Mitigation: productise the reasoning workspace and assessment first.

9.4 Near-term repository roadmap

Before an external product claim, Greenways should:

1. complete deterministic SQL, RPC, XTalk, JavaScript, and Dart generation gates;
2. close the remaining full SQL build and cross-language acceptance gaps;
3. deliver three to five nontrivial functional slices through the same pipeline;
4. create a checked-in measurement command that classifies source and generated artifacts;
5. add semantic contract snapshots and breaking-change comparison;
6. generate a per-slice provenance manifest linking source, emitted SQL, tests, contracts, and consumers;
7. run an unfamiliar-developer and AI-agent time-to-explanation study;
8. publish one reference slice with all intermediate and generated outputs visible.

9.5 Decision gates for productisation

Gate	Evidence required	Decision
Internal repeatability	Three representative slices, deterministic builds, and measured maintenance changes.	Continue from project tooling to reusable compiler package.
External legibility	Two design partners can explain and modify a proof slice faster than their baseline.	Invest in reasoning workspace and migration tooling.
Governance demand	Multiple teams need retained contracts, compatibility policy, and generated SDK distribution.	Build hosted or private registry.
Verified-build value	Customers pay for executable evidence, provenance, and private runners.	Expand managed verification infrastructure.
Pack repeatability	The same domain capability is adopted by several customers without forking the core.	Commercialize supported domain packs.

10 Conclusion

The build pipeline's largest advantage is not that it can produce more code. AI already produces more code than many teams can comfortably review. The advantage is that it can produce *legible code*: behaviour with a compact source, readable target, immediate executable proof, visible derivation, and bounded impact.

That legibility changes delivery economics. It reduces the number of authoritative representations, shrinks the consistency graph, improves first-pass coherence, and makes later changes easier to reason about. Under conservative assumptions the benefit may be two to four times. In a large, cross-platform, AI-drifted system, the quality-adjusted effect can plausibly approach twenty times. The number must be proven; the structural mechanism can already be described.

The product opportunity is therefore not another code generator. It is a legible-build platform: compiler, reasoning workspace, semantic contract registry, verified-build service, and set of transparent cross-language runtime and domain packs. Its promise is simple enough to test:

The promise

As the product grows, the amount of shipped functionality may become large. The amount of context required to understand and safely change one piece of that functionality should remain small.

References

[1] Microsoft, *TypeScript Handbook: The Basics – Erased Types*. Type annotations are removed during compilation and do not change runtime behaviour. https://www.typescriptlang.org/docs/handbook/2/basic-types.html.

[2] Prisma, *Generating Prisma Client*. Prisma Client is generated from a project schema and provides database query types for TypeScript. https://docs.prisma.io/docs/orm/v6/prisma-client/setup-and-configuration/generating-prisma-client.

[3] tRPC, *End-to-end typesafe APIs*. Server router types are inferred by TypeScript clients without a separate code-generation step. https://trpc.io/.

[4] Supabase, *Generating TypeScript Types*. Supabase introspects database schemas to generate API type definitions. https://supabase.com/docs/guides/api/rest/generating-types.

[5] PostgREST, *Functions as RPC*. Accessible PostgreSQL functions are exposed beneath the REST RPC prefix. https://docs.postgrest.org/en/latest/references/api/functions.html.

[6] Mattermost, *Application Architecture*. The documented platform separates access clients, application-server services, APIs, and backend infrastructure. https://docs.mattermost.com/deployment-guide/application-architecture.html.

[7] Buf, *Detecting Breaking Changes*. Schema compatibility checks are enforced during development, review, and registry publication. https://buf.build/docs/breaking/.

[8] Encore, *Backend Infrastructure for Humans and Agents*. Encore derives an application and infrastructure graph from TypeScript or Go source and provides generated operational tooling. https://encore.dev/.

[9] GitHub, *Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*, 2022. The controlled JavaScript HTTP-server task reported a 55% speed improvement for the Copilot group. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

[10] METR, *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*, 2025. The randomized study reported a 19% slowdown for experienced developers completing real tasks in familiar repositories. https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study-paper.pdf.

[11] Google, *Why Google Stores Billions of Lines of Code in a Single Repository*. The paper describes shared infrastructure and large-scale change practices used to maintain a very large codebase. https://research.google/pubs/why-google-stores-billions-of-lines-of-code-in-a-single-repository/.

[12] Greenways, *xt.substrate Functional Slice Architecture* and *xt.ui Application Boundary*, repository architecture guides 0020 and 0030, 2026.