Source: cppcc/docs/Two_Domains_Article_Public_Draft_0.1.md @ 68f96c8 — curated 2026-08-06; edits land in the source first.

Two Domains: Cross-Language Translation and Knowledge Representation

A companion paper on substrate-grounded AI-assisted software construction.

Draft 0.1 — June 2026

A publication from Bet Three LLC (B3). The technical substrate referenced in this paper is protected by US Patent 8,464,232. Companion technical material is available under commercial terms.

I. Where this paper sits

The companion paper *Three Bets on the Future of Software* argued that a third bet sits beneath the two dominant AI bets — a bet about the substrate on which AI-assisted software construction operates. The substrate makes semantic representations and their transformations into one another first-class artifacts. The substrate has been exercised on a project family of approximately twenty-six sibling repositories built over roughly twelve weeks using a verbal-specification-driven methodology. The argument is made in full there.

This paper does not restate that argument. It takes the substrate as given and considers two specific problem domains where the substrate changes the shape of what is possible. The domains are chosen because they sit at the boundary of what current tooling addresses well, because they are practically requested, and because they illustrate concretely what "the substrate makes the layer above tractable" looks like in worked form.

The two domains are:

1. **Cross-language code translation** — taking software written in one general-purpose language and producing equivalent software in another. The practical instance considered is Python to JavaScript or TypeScript, because both languages are sufficiently widespread that

the demand is concrete and sufficiently different in runtime semantics that the problem is genuinely hard.

2. **Knowledge representation and extraction** — building ontologies that conform to a formal standard such as OWL2, and then automating the process by which raw data is converted into axioms expressed in that ontology. The standard itself is straightforward to implement with the substrate; what is not well-defined in current practice is the automation of extraction, particularly extraction that maintains the ontology as it grows.

Both domains are presented at the architectural level. Implementation specifics, benchmark figures, and source-level material are reserved for technical conversations under commercial terms.

II. Two problems that look unrelated and are not

The two domains appear to live in different parts of software work. One is about source code; the other is about structured knowledge. They share a problem shape that current tooling addresses poorly, and they share an approach the substrate enables. Stating the shared shape first makes the rest of the paper easier to read.

The shared shape is the following. In both domains, there is a human-readable representation (Python source, raw text or tabular data) and a structured target representation (JavaScript source, OWL2 axioms). A direct transformation from one to the other is hard because the representations live in semantic worlds that do not align exactly. Python's class system and JavaScript's prototype system are not isomorphic. A sentence about a person and a typed relation between two ontology individuals are not isomorphic. Bridging the gap requires that something — a human, a tool, or an AI — supply the missing semantic structure.

When the bridge is supplied informally — through prose conventions, heuristic translation rules, hand-written mapping code — the result is fragile. The bridge cannot be inspected mechanically. Conflicts between intended meaning and produced output cannot be surfaced before they ship. The bridge erodes as the codebase or the ontology evolves, and the erosion is invisible until something visible breaks.

When the bridge is supplied formally — as a specification the toolchain can verify — the problem changes shape. The bridge becomes an artifact. The bridge can be tested. The bridge can be versioned. Most importantly, the bridge can be reasoned about by AI assistance in a way that ad-

hoc bridges cannot: an AI working against a formal specification has something to verify its output against, where an AI working against a prose convention has only its own probabilistic guess about whether it got the convention right.

The substrate's contribution to both domains is the same: it makes the bridge a first-class object. Specifically, the substrate provides a specification language in which transformations between representations are themselves specifications, an oracle (round-trip closure) that validates whether a transformation preserves what it is supposed to preserve, and a set of cooperating runtimes that allow the bridge specifications to be exercised against the data the bridge connects.

This is the move that makes the two domains tractable. The rest of the paper unpacks what it looks like in each.

III. Cross-language code translation

The practical question is: how does Python code get converted into JavaScript or TypeScript code, in a way that works?

The honest first answer is that arbitrary Python source converted to arbitrary JavaScript source is not solvable today by anyone. Python has a class system with multiple inheritance, an exception model that propagates through generator frames, a runtime type system with descriptors and metaclasses, and a standard library whose semantics are bound up in ways that simply do not have JavaScript counterparts. A converter that handles all of Python is not what we are talking about, and pretending otherwise wastes the reader's attention.

The substrate-grounded approach does something different. It treats the conversion problem not as a translation problem on source text, but as a representation problem on the data model the source is operating against.

The reframe is the following. A large fraction of the Python code that one would actually want to convert to JavaScript is code that operates against structured data — schema validation, serialization, transformation, aggregation, dispatch. The code is not arbitrary Python; it is Python *organized around a data model*. The data model is the load-bearing artifact. The Python is one rendering of operations on that model; the desired JavaScript is another rendering.

Once the problem is reframed this way, the substrate handles it naturally. The substrate's specification language can express the data model. The substrate's compiler-compiler produces

parsers, serializers, schema enforcers, and accessor APIs in multiple target languages from the same specification. Python and JavaScript are both targets. So is TypeScript, with type information preserved through the compilation. So is Rust. So is C++. The data model becomes the source of truth; the language-specific renderings are generated artifacts.

For code that is *already* organized this way, no conversion is required: both renderings exist simultaneously and are kept consistent by the substrate's machinery. The two renderings cannot drift apart, because both are generated from the same specification. This is the case that requires no novel work — the substrate already produces it.

For code that is *not yet* organized this way, the path is to lift the existing Python to the data-model layer. This is where AI assistance enters. An AI that has the substrate's specification language as a target — rather than the JavaScript source as a target — has a much better-defined job. Its job is to recover the data model that the Python source is implicitly operating against, express that model in the specification language, and verify the recovery by round-tripping selected operations through the substrate and comparing to the original Python's behavior on the same inputs. When the recovery is complete, the JavaScript rendering is produced by the substrate, not by the AI; the AI's work is done at the specification level, where its output can be checked.

This is a substantially more tractable problem than direct Python-to-JavaScript transpilation, for several reasons. The specification language is small and uniform, where Python is large and full of corners. The AI's output target is a structure the substrate can validate, where direct transpilation gives no target the AI can self-verify against. The behavioral check — does the generated code do the same thing as the original — runs against the data-model contract, where direct transpilation has to certify behavioral equivalence at the source-text level. And the work amortizes: once a domain has been lifted to the data-model layer, adding a new target language is a substrate compilation step, not a new translation project.

The honest scoping is that this approach works for the data-model- shaped fraction of real Python codebases, not for the script-shaped fraction. A Python utility that orchestrates shell commands, threads a custom debugger, or rests on a Python-specific library that has no analog in the target language is not what the substrate is for. But the data-model-shaped fraction is a large fraction of the code people actually want to move between languages. Schema-driven web APIs, data validation libraries, structured serialization, business- rule engines, configuration systems, ontology processing — these are not arbitrary Python. These are Python organized around structures the substrate can represent and emit in any of its target languages.

For this fraction, the conversion question becomes the lift question. Lift the implicit data model to an explicit specification. The substrate handles the rest.

The Python-to-JavaScript instance is not architecturally special. The same approach applies to Java-to-Go, C++-to-Rust, TypeScript- to-Python, or any other pair where the source code is operating against structured data and the target language has a comparable runtime model. The substrate's contribution is that the specification layer is shared across all targets, and lifting work done for one source-target pair amortizes across the entire matrix.

IV. Knowledge representation and extraction

The second question concerns formal ontologies in the OWL2 family and the automation of knowledge extraction from raw data against those ontologies.

OWL2 itself is a standardized specification. Building a substrate implementation of OWL2 is straightforward: the language has a grammar, the substrate's compiler-compiler produces parsers and serializers for grammars, and the resulting toolchain handles OWL2 documents with the same machinery it handles any other substrate-implemented language. This is not where the difficulty sits.

The difficulty sits in the next phase, which current practice addresses only partially. Given an OWL2 ontology and a body of raw data — text, tables, observational records, conversational transcripts, sensor traces — how does one automate the extraction of axioms that populate the ontology from the data, in a way that maintains the ontology as new data arrives?

The current state of the art for knowledge extraction is a mix of named-entity recognition, relation extraction, schema-aware language models, and pipelines that combine these with hand-crafted post- processing. The pipelines work for cases that match their training, fail in characteristic ways when the data is unusual, and require human attention every time the ontology evolves. The pipelines do not reason about the ontology; they pattern-match against it. When the data implies that the ontology needs to grow — a new relationship type, a refined class hierarchy, a concept the ontology did not anticipate — the pipeline produces inconsistencies the pipeline cannot resolve. A human decides what to do.

The substrate's contribution to this problem is best stated by a metaphor a practitioner offered in conversation: it is the difference between looking *at* a three-dimensional model from inside three dimensions and looking at it from a fourth dimension. The fourth dimension is where decisions

about the model itself become tractable, because both the model and the data that the model represents are visible simultaneously.

The substrate operates from that fourth dimension. To make this concrete: the substrate's specification language can express the ontology, the data-to-ontology mapping, and the ontology's own schema as first-class artifacts. All three are inspectable by the toolchain. All three can be reasoned about by AI assistance with a formal target to verify against. The ontology is not just a thing the extraction pipeline produces; it is a specification the pipeline runs against, the pipeline's behavior is itself a specification, and inconsistencies surface as structural conflicts rather than as silent quality degradation.

In practice this enables three tiers of extraction work that current pipelines collapse into one undifferentiated activity.

The first tier is extraction of axioms that are consistent with the current ontology. An AI processes raw data, proposes triples or higher-arity assertions, and the substrate validates that each proposal is well-formed against the current ontology schema. Valid proposals are added; ill-formed proposals are rejected. This is the case current pipelines handle reasonably well; the substrate's contribution is making the validation mechanical rather than statistical.

The second tier is extraction that the current ontology cannot accept as-is. The AI produces a proposed triple; the substrate flags it as conflicting with an existing axiom or as referring to a class or property the ontology has not declared. This is where current pipelines lose information: they either drop the conflicting extraction, force-fit it into an inappropriate slot, or surface it to a human as an unstructured error. The substrate treats the conflict as a structured object. A triage step — which can itself be AI-assisted — looks at the conflict and determines whether the extracted triple is wrong (the AI misread the data), the existing axiom is wrong (a previous extraction was incorrect and got accepted), or the ontology is incomplete (the data is asserting something genuine that the ontology has not yet articulated). The triage produces a decision; the decision is a structured edit to one of the three specifications (the extraction mapping, the existing axioms, the ontology schema); the substrate validates the edit; the system moves on.

The third tier is ontology evolution. When the second tier repeatedly identifies the same missing concept — when the data keeps asserting things the ontology cannot accommodate — the appropriate response is to grow the ontology. This is the operation that current pipelines cannot reach automatically, because it requires reasoning about the ontology's own structure. The substrate makes this tractable because the ontology's structure is itself a specification, expressed in the same language as everything else. An AI can propose a refactoring of the ontology — a new

class, a refined hierarchy, a renamed property — and the substrate can validate that the refactored ontology accepts the previously conflicting extractions, that it remains consistent with the extractions that were already accepted, and that the new schema continues to round-trip cleanly through the toolchain. The refactoring becomes a checked edit rather than an unstructured restructuring. The human supervises; the substrate makes the supervision tractable by reducing the verification surface.

The "fourth dimension" framing is not metaphorical in the loose sense. The substrate operates one specification layer above the ontology and the extraction mapping. From that layer, both are visible, both are inspectable, both are editable, and the consistency between them is mechanically checkable. That layer does not exist in current ontology toolchains — they have the ontology, they have the data, and they have whatever ad-hoc tools connect the two. The substrate makes the connection a first-class artifact and gives the system a place to stand when the connection needs to change.

The practical value of this is straightforward. Knowledge extraction at scale, against ontologies that need to evolve as the data evolves, is a class of work that has resisted automation because the maintenance problem dominates the extraction problem. The substrate reframes maintenance as edits to specifications the toolchain understands, which makes maintenance susceptible to the same AI-assisted workflow that has been demonstrated on substrate construction itself. The class of work becomes tractable.

This is not a claim that the substrate solves knowledge extraction. It is a claim that the substrate moves the problem from "build a brittle pipeline and rebuild it every time the ontology changes" to "maintain a small set of cooperating specifications that an AI-human team can evolve coherently." The work the substrate enables in this domain is the kind of work knowledge engineering has wanted to do for decades and could not, because the verification surface for ontology evolution was too large to be practical. The substrate shrinks that surface.

V. What both domains share

The two domains have been presented separately, but the underlying move is the same in both.

In code translation, the bridge between Python and JavaScript that current tools cannot maintain becomes maintainable when the bridge is recast as a data-model specification and the substrate

handles the rendering into each target language. The AI's contribution shifts from generating target-language source to recovering the data model the source code is operating against.

In knowledge extraction, the bridge between raw data and OWL2 axioms that current pipelines cannot maintain becomes maintainable when the bridge is recast as a set of cooperating specifications (extraction mapping, ontology schema, axiom database) and the substrate gives the system a place to inspect and edit all three. The AI's contribution shifts from end-to-end pipeline construction to AI-assisted evolution of specifications the substrate validates.

In both, the common move is: identify the implicit semantic structure that the current workflow treats as background, lift it to an explicit specification the substrate can hold, and let the substrate carry the consistency burden the human or the AI was carrying badly.

This is the third-bet thesis in operational form. The substrate makes the layer above tractable. The layer above is where the interesting AI-assisted work lives, in both domains. The substrate is what allows that work to happen with mechanical verification rather than with hope.

VI. What is demonstrated, what is open

Both domains discussed in this paper are at different points along the path from substrate to demonstrated practice.

The code-translation approach is partially demonstrated. The substrate's ability to emit a single data model into multiple target languages, validated by round-trip closure in each language, is the foundation of the substrate itself and has been exercised across its construction. A small worked example exists in which a structured-data object — entities and relationships, expressed once in the substrate's specification language — is emitted as parser/serializer/accessor APIs in C++, Python, and JavaScript, with each language's emission validated against the same canonical binary form. Round-trip oracles pass in all three languages. This is the part of the code-translation argument that is empirical.

The lifting work — taking arbitrary Python source that is not already in the substrate and producing the corresponding data-model specification, AI-assisted, validated by round-tripping behavior through the specification — is sketched but not built. The substrate provides the target the AI works against; the workflow discipline for the lift has been demonstrated on substrate construction itself; combining the two into a production-grade lifting tool is forward work. Until

that tool exists, the code-translation argument applies to data-model-shaped code that has been intentionally organized this way, not to arbitrary existing Python.

The knowledge-representation approach is, at this paper's writing, the more aspirational of the two. A substrate implementation of OWL2's surface grammar is a tractable engineering exercise — every new target language for the substrate is one — but it has not been built as a worked example yet. The three-tier extraction loop described in §IV — accept-against-ontology, triage-on-conflict, ontology-evolution-on-pattern — is sketched at the architectural level. The components the substrate would contribute (specification of ontology schema, specification of extraction mapping, round-trip validation against both, AI-assisted edits to all three) have direct analogs in substrate construction work that has been done. But the integration, the user experience, and the production characteristics are forward work.

Honest scoping is the same in both domains: the substrate is what exists; the worked applications in each domain are in different stages of maturity. The architectural argument is not aspirational because the substrate is not aspirational; it is exercised across the project family it was used to build. What is aspirational is the specific worked applications described here. Their value is in articulating what the substrate makes possible in concrete domains, not in claiming the worked instance has already been built.

The reason for stating this clearly is the reason given in the companion paper. A paper that did not say what remains uncertain about its claims would be doing the wrong job. The reader who expects either domain to be a turnkey commercial product today is the wrong reader; the reader who expects a substrate that makes the domains *tractable* in a way they are not today is the right reader.

VII. Where this sits

Two domains have been considered. Code translation between general- purpose languages, illustrated by Python to JavaScript or TypeScript. Knowledge representation and extraction, illustrated by OWL2 ontologies that maintain themselves under data evolution. Both domains have the same underlying problem: the bridge between two semantic worlds is hard to maintain when the bridge is informal, and the maintenance burden swamps the apparent simplicity of the transformation.

The substrate's contribution in both is to make the bridge a first-class artifact. The bridge becomes a specification; the specification has a mechanical correctness oracle; the AI assistance that is already valuable for generating code becomes valuable for maintaining the specification, because the specification gives the AI a target to verify against. The bridge stops being something that erodes invisibly and becomes something that is checked every time it is exercised.

This is what the third-bet thesis offers in worked-domain form. The substrate does not solve translation. The substrate does not solve knowledge extraction. The substrate makes both addressable through a workflow discipline in which AI assistance operates at the specification layer rather than at the artifact layer. That discipline has been demonstrated on the substrate's own construction. Carrying it into the two domains discussed here is the forward work, in different stages of maturity for each.

For practitioners working on either problem and frustrated by the characteristic failure modes of current approaches — translation that erodes as the codebase evolves, extraction pipelines that cannot accommodate ontology growth — the substrate-grounded approach is worth understanding as an architectural alternative. For organizations facing such problems at scale, the relevant conversation is a commercial one.

The point of articulating these two domains in writing is to make visible to the right readers a class of approach that current tooling does not deliver and current research does not generally discuss in these terms. The architectural shape is unusual enough that practitioners working on the problems described above tend to arrive at the substrate's design space through different routes. Making the shape explicit, as worked examples in concrete domains, is a way of inviting that arrival to be a conversation rather than an independent reinvention.

Contact

The substrate referenced throughout this paper, the methodology that produced it, and the body of demonstrated work that exercises it are operated by Bet Three LLC under US Patent 8,464,232.

Engagement is through commercial channels. Companion technical material — substrate specification, reference implementation roadmap, methodology disciplines, integration case studies for the domains discussed in this paper — is shared under appropriate terms once a conversation has scope.

Inquiries: [your preferred contact channel].

End of paper. Companion to "Three Bets on the Future of Software" (Public Draft 0.1). Readers reaching this paper through external channels should treat it as a position statement; the technical material is shared through commercial conversation.