Source: cppcc/docs/Three_Bets_Article_Public_Draft_0.1.md @ 68f96c8 — curated 2026-08-06; edits land in the source first.

Three Bets on the Future of Software

A position 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. Two bets, both unresolved

The current conversation about artificial intelligence has narrowed, more than is generally acknowledged, to two competing bets about how human-level capability gets built.

The first bet — held by the major labs and most of the venture capital flowing into AI — is that scaling neural networks produces general intelligence as an emergent property. Larger transformers, more training data, better feedback mechanisms, and the architectural questions take care of themselves. The empirical evidence for this bet is real: today's frontier models do things that surprised the field five years ago, and the scaling curves have not yet visibly saturated.

The second bet — held by a smaller community working in the lineage of cognitive science and symbolic AI — is that intelligence requires explicit architecture. Knowledge representation that can be inspected and modified. Reasoning mechanisms that compose. Memory systems that persist. Integration across multiple AI paradigms. OpenCog Hyperon is the most ambitious current instantiation of this second bet, with the Atomspace metagraph as integrative hub, MeTTa as the pattern- rewriting language, and PRIMUS as the cognitive architecture above both.

Both bets are unresolved on their own terms.

Scaling has produced remarkable capabilities and equally remarkable limitations — hallucination, weak multi-step reasoning, opacity, the inability to self-correct in ways the field documents extensively. The "comprehension without competence" phenomenon — models that can articulate

principles they cannot reliably execute — is a structural property of next-token prediction, not an incidental shortcoming larger models will outgrow. Whether further scaling resolves these limitations or whether they require architectural changes is the open question of the field.

Cognitive architectures have produced theoretical frameworks of considerable depth but, after decades of research, have not yet demonstrated capabilities competitive with what scaled neural networks deliver today. The cognitive-synergy hypothesis at the center of Hyperon — that combining multiple AI paradigms produces emergent capabilities none has alone — is conceptually attractive and empirically unverified at scale.

Reasonable people disagree about which bet will pay off, when, or whether either will. The disagreement is not about which side has better arguments — both sides have serious arguments. The disagreement is about which class of unresolved problem is more tractable.

There is a third question that neither bet addresses, and which may be more immediately consequential than the bet either is making.

II. The substrate question

That third question is about the substrate of software construction itself.

Most software, including most AI software, gets built through a workflow that has barely changed in three decades. Humans author code in text editors. Language tools provide syntactic assistance. Version control tracks revisions. The semantic structures that connect specifications to implementations to data models to APIs to test cases to documentation — these live primarily in the developers' heads and in scattered prose documentation that the toolchain cannot mechanically verify.

When AI assistance arrived, it slotted into this workflow at the code-generation end. Autocomplete, then chat, then agentic execution. Each layer made the human's typing faster, the human's debugging easier, the human's exploration of unfamiliar codebases more tractable. None of these tools changed the underlying substrate in which software lives. The semantic structures still live in human heads. The AI helps with the code-level surface that the heads project into text.

This arrangement has held even as AI capabilities have advanced dramatically because the substrate problem is invisible until you look for it. From inside the workflow, the friction looks like "the AI doesn't know my codebase well enough" or "I need better prompts" or "the context

window isn't large enough." These framings locate the problem at the AI-tool interface. The substrate framing locates the problem somewhere else entirely: at the absence of any mechanically verifiable representation of what the software is supposed to do, how its representations relate to one another, what transformations preserve meaning and which do not. The AI cannot help you with a structure that doesn't exist; it can only help with the code-level surface that projects from it.

A third bet — the bet this paper articulates — is that software construction is fundamentally the *transformation of semantic representations into one another*, and that the substrate supporting those transformations is what determines what AI assistance can actually do.

The claim is not metaphorical. It is a falsifiable hypothesis about the structure of programming work. If true, the implications are substantial: the bottleneck in software construction is not code generation (which contemporary LLMs do increasingly well), but the disciplined maintenance of semantic structures and their relationships (which LLMs do considerably less well, and which traditional toolchains were not designed to support).

This third bet is independent of the first two. It works today on top of current LLMs as the language interface. It would work tomorrow on top of Hyperon-style cognitive architecture if cognitive synergy turns out to be real. The substrate is independent of which AGI bet pays off; it would integrate with whichever does, and remains useful in the meantime. A bet that does not depend on its competitors failing has different epistemic properties than a bet that does.

This paper describes that third bet — what it claims, what has been demonstrated, and what it implies for how AI assistance to software development might evolve in the next several years.

III. What the substrate is

The substrate referenced throughout this paper is a small set of cooperating artifacts that together provide a mechanically verifiable foundation for semantic representations. Five pieces, named here at the architectural level rather than at the artifact level — implementation specifics are reserved for technical conversations under commercial terms.

A self-defining specification language. Structured representations are specified in a language defined by its own productions. A single file describes the grammar of the language using the language's own syntax, which means the language is self-defining in a strict sense. Any specification written in the language is immediately consumable by the entire toolchain. There is no separate metaschema, no out-of-band declaration mechanism; the system bottoms out in itself.

A compiler-compiler. Given a specification in the language above, the compiler produces implementation artifacts: parser/serializer code, keyword tables, and binary-format encoders. The compiler is itself bootstrapped through the specification language — the meta- grammar defines the language the compiler reads, and the compiler compiles the meta-grammar to produce its own parser. The self-validation property follows directly: any grammar's parser must round-trip the grammar's source back to itself, and the meta-grammar's parser must round-trip the meta-grammar's source. If either fails, the toolchain has lost internal consistency in a way that is mechanically detectable.

A binary canonical form. A standardized binary serialization of any specification-conformant content that round-trips byte-identically to its textual source. The binary form is not just a serialization format; it is the canonical representation that all four runtimes — C++, Python, JavaScript, Rust — consume through a unified API. The same binary file is readable by all four. The "every wire crossing is structured-binary" property of substrate-built systems follows from this: any transport between any two substrate-aware components can carry binary structured content that any other substrate-aware component can decode.

Round-trip closure as the structural-correctness oracle. A transformation `F` from textual form `X` to binary form `Y`, paired with its inverse `G` from `Y` back to `X'`, is correct on input `X` if `X' = X` under whatever equality relation applies — exact, canonical, or semantic. Usually exact, for the substrate's wire formats. The oracle does not require the test author to know what the output should be in absolute terms. It requires only that the pair of transformations be consistent. This makes round-trip testing applicable to the validation of large amounts of generated code, including code generated by AI assistants from verbal specifications. When the oracle is the contract, the generation rate can run at the rate the AI can produce code rather than the rate the human can review code.

Isomorphic mappings between representations. The same conceptual structure may live as a grammar, an SQL schema, an object-oriented API, a wire protocol, a documentation fragment, and a test harness. The substrate makes these mappings first-class: each mapping is itself a specification, declared and tracked, with its own conformance criteria. When a representation changes, the mappings indicate which dependent representations need updating. The cross-representation consistency that traditional codebases maintain through prose convention and discipline becomes mechanically tractable.

These five pieces together constitute the substrate. None individually is revolutionary; each has antecedents in compiler construction, serialization theory, or model-driven engineering. What is

novel is the combination, and specifically the combination's relationship to AI-assisted software construction: the substrate is what makes AI assistance work at the *specification* layer rather than only at the code-generation layer.

The underlying technology is protected by US Patent 8,464,232.

IV. What has been demonstrated

The substrate's claims would be aspirational if they had not been exercised on real software construction at meaningful scale. The empirical record matters precisely because the third-bet thesis is empirical: either substrate-grounded AI-assisted software construction works or it doesn't. Theorizing about it is cheap; building things with it is the test.

The work to date forms a project family of approximately twenty-six sibling repositories under a single development tree, totalling roughly eighty thousand lines of code across multiple languages, produced over approximately twelve weeks of focused work using a methodology in which verbal specifications are the authoritative source and code is generated through interaction with an AI assistant and validated through round-trip oracles. The substrate itself was the foundation work prior to this twelve-week record; the demonstration tracks built on top of the substrate.

The substrate itself — the compiler, the four runtimes that implement the unified binary API, the keyword definition emitters across five target languages — is the foundation. Building this took longer than the demonstration tracks above it. It validates that the substrate is internally consistent: every grammar processed by the compiler produces a parser that round-trips the grammar's own source; every binary representation produced by any runtime is consumable by every other runtime; the meta-grammar self-round-trips. These are not properties that an inconsistent system could have.

On top of the substrate, two demonstration tracks were built against contemporary AI-agent frameworks. The first substitutes structured- binary wire formats for the JSON serialization that LangGraph uses internally for tools, memory, telemetry, agent-to-agent messaging, and LLM calls. The second does the same against the Claude Agent SDK's interface points. Both tracks consume their host framework without modifying it; they operate at the framework's published extension points. Both produce demonstration agents whose every wire crossing carries

structured-binary content, where every artifact is independently decodable through the substrate's API in external consumers.

The economic observation from this work is that adding the substrate to a second AI framework cost approximately 40% of what adding it to the first framework cost, with the difference being inheritance from the substrate that the first framework's work had already established. This is the substitution-library claim made empirical: the marginal cost of supporting an additional framework decreases as the substrate accumulates.

The methodological observation is more surprising. Each demonstration milestone — eight for the first framework, five for the second — required between three days and two weeks of focused work to land in working form. The original effort estimates, written before any milestone was executed and using conventional single-engineer scoping, suggested twelve to twenty-seven weeks per framework. The realized effort came in roughly four-to-nine times under conventional estimates. This is not because the work was easier than expected; it is because the throughput model is different from conventional engineering. Verbal specifications drive AI-assisted code generation, round-trip oracles validate the results, and the human-AI loop runs at a rate that conventional estimation does not account for.

A separate measurement track explored the substrate's behavior in distributed-system contexts, specifically integrating with the Hyperon ecosystem's Distributed Atomspace through a Rust consumer reading MeTTa-encoded binary payloads through the substrate's API. Benchmark data showed that for clusters where producer/consumer fan-out reaches five or more downstream readers, binary transport with substrate-grounded structure dominates text re-parsing for any consumer operating at C++ or Rust speeds. Per-atom costs in Rust come in roughly four to twelve times faster than the same operations in Python, because the substrate eliminates the FFI boundary that dominates Python's atom-insertion cost. Memory footprint is roughly fifty times smaller. The substrate enables deployment topologies that the text-format path cannot reach at acceptable cost.

These measurements support a narrower and sharper version of the third-bet thesis. The substrate does not make any general software construction problem trivial. It makes specific classes of work substantially more tractable, particularly: cross-language wire formats requiring byte-stable structured representation; multi- framework substitution work where the substrate amortizes across framework instances; distributed-system topologies where the producer-consumer cost asymmetry favors structured-binary transport; and AI-assisted code generation where round-trip oracles validate the generated artifacts structurally rather than behaviorally.

What the substrate does not do is solve any of the hard AGI problems the first two bets engage with. It does not propose a path to general intelligence. It does not claim that the mechanical verification it provides is in any sense a model of cognition. The bet is narrower and, for that reason, more tractable: build the substrate on which AI-assisted software construction can operate at specification-level rather than code-level, validate it on real construction work, and let it integrate with whichever AGI bet matures.

V. The Hyperon connection

The work most directly relevant to the broader AI conversation is the substrate's integration with the Hyperon ecosystem. This is worth treating specifically because it illustrates the third bet's relationship to the cognitive-architecture bet in concrete terms.

Hyperon's bet is that cognition is the dynamics of a self-modifying metagraph (the Atomspace) under pattern-rewriting (MeTTa) governed by a cognitive architecture (PRIMUS). MeTTa is the language in which programs are written and in which the Atomspace's content is expressed; programs are themselves graph structures other programs can rewrite. The architecture is, by current standards, ambitious and pre-alpha.

MeTTa has a grammar. The grammar is small — S-expression syntax with permissive symbol naming, a few prefix conventions for variables and space references, and one comment syntax. The substrate treated MeTTa as a target language: a specification in the substrate's language describing MeTTa's surface syntax was authored, the compiler produced the parser-serializer pair, and the round-trip oracle was exercised against existing MeTTa test corpora. The result is a four-runtime read path for MeTTa source — binary representations of MeTTa programs, consumable in C++, Python, JavaScript, and Rust through the same API any other substrate-built parser uses.

This produced an observation that matters for the broader AI conversation: a Rust consumer reading MeTTa-as-binary and populating a Rust atomspace operates without the FFI cost that dominates the equivalent Python path. The per-atom cost goes from the thousand- nanosecond range (Python crossing the language boundary to insert atoms) to the hundred-nanosecond range (Rust native operation). Memory footprint drops by an order of magnitude. The deployment topology where one producer feeds many cluster nodes — the natural shape for any Atomspace-based system at scale — benefits substantially.

The point of this observation is not the magnitude of any particular speedup. It is that the substrate's value to Hyperon does not depend on Hyperon's AGI bet paying off. Whether or not cognitive synergy turns out to be real, whether or not PRIMUS produces human-level intelligence, the Atomspace as a distributed metagraph needs a wire format for moving structured content between producer and consumer nodes. The substrate provides one, with round-trip closure as the oracle and four runtime consumers natively supported. The substrate is useful at the engineering layer whether or not Hyperon's broader claims mature.

If Hyperon does mature — if cognitive synergy turns out to deliver the capabilities its proponents argue for — the substrate is positioned to integrate as the foundation layer beneath it. The Atomspace's content, the MeTTa programs that rewrite it, the wire formats that move metagraph fragments between cluster nodes: all have natural representations in the substrate's specification language with corresponding binary forms. The integration path is engineering work, not architectural rethinking.

If Hyperon does not mature — if the cognitive-architecture bet turns out to require breakthroughs that don't arrive — the substrate remains useful for what it does: substrate-grounded wire formats, AI-assisted construction of structured-data systems, the substitution- library pattern against whichever frameworks turn out to be load-bearing in whatever paradigm dominates. The substrate is agnostic about which AGI bet wins because it sits beneath the question.

VI. The architectural implication

The third bet, fully articulated, implies a class of AI-assisted software construction system that does not yet exist commercially but is buildable with current technology.

Such a system would maintain a collection of formal specifications, each layered: a natural-language description for human consumption, a refined data model in the substrate's specification language, formal constraints, and a set of isomorphic mappings to other specifications and to implementation representations. The collection would evolve coherently — adding a specification preserves the consistency of existing mappings; refining a data model propagates changes through dependent specifications; conflicts between specifications surface as structural inconsistencies the system can identify.

The user-facing layer would accept high-level goals, constraints, and feedback. An LLM provides the language interface throughout — translating between natural language and the formal

specifications, proposing grammar designs when the system encounters domains it has not seen before, generating prose documentation from the formal layer, and orchestrating dialogue with the user when specifications need refinement.

The execution layer performs what is here termed Adaptive Programming over the specifications. For each specification in the collection, the AI generates the implementation artifacts the specification implies — APIs, schemas, test cases, documentation, transformations between representations — using the round-trip oracle to validate each artifact structurally. The workflow discipline of plan-then- implement, already demonstrated on the substrate's own construction, applies recursively: the AI proposes a plan, the human reviews and refines, the execution proceeds, the result is validated, the specification collection is updated.

What such a system would look like to a user is closer to what people describe as "human-like thinking" than to what they associate with AI assistants today. It would maintain internal models. It would decompose problems into structured representations. It would reason about relationships between those representations. It would autonomously drive solutions toward completion. The behaviour would resemble goal-directed problem solving rather than prompt-response generation.

Critically, this behaviour does not require solving any of the AGI problems Hyperon engages with. It does not require consciousness, embodied cognition, or any deep theoretical breakthrough. It requires the substrate (which exists), the LLM language interface (which exists), and the orchestration layer that connects them (which is the engineering work the third bet implies).

The substrate is what makes the orchestration layer tractable. An orchestration layer that tried to maintain "structured semantic representations" using ad-hoc data formats, with verification by testing alone, would be the kind of project that gets attempted periodically and abandoned because the cross-representation consistency problem dominates. With a substrate that handles the semantic representations mechanically — grammars, mappings, round- trip oracles — the orchestration layer becomes a question of workflow design and AI integration, not of solving the verification problem from scratch.

This is the work the third bet is positioned for. Whether it becomes the next phase of B3's own development, or whether other practitioners build comparable systems on substrates that emerge from independent work, the architectural shape is the same. The substrate makes the layer above tractable. The layer above is where AI-assisted software construction becomes specification-driven rather than code-driven.

VII. What's open

A paper that did not name what remains uncertain about its own claims would be doing the wrong job. Several things about the third bet are genuinely open.

The substrate has been exercised on a project family of substantial size, but the family is the product of one practitioner's work over roughly a year. Independent external validation — practitioners who have not been involved in the substrate's development reaching for it, finding it useful, building things with it — does not yet exist. The patterns and disciplines that the work surfaced may or may not transfer cleanly to other practitioners. The 4-9× throughput multiplier over conventional engineering estimates may or may not hold when other practitioners adopt the methodology.

The substrate's scaling properties to substantially larger projects than what has been built are inferable but not measured. The project family is large enough to be a serious test case but not large enough to test the limits. What happens at ten times the current scale, or with multi-practitioner concurrent development, or with substantially more diverse target domains, is open.

The orchestration layer above the substrate — the system described in §VI — is sketched, not built. The natural next phase of this work is the construction of that layer, and that work has not started. Until it has, the architectural claim is theoretical rather than demonstrated. The substrate is demonstrated; the layer above it is a forward bet.

The relationship to commercial deployment is also open. The substrate's proprietary status under US Patent 8,464,232 means it is not currently available as an open-source artifact. Whether, when, and under what terms this changes is a strategic question the project's principals will resolve through commercial conversations rather than through any predetermined disclosure schedule. Practitioners interested in using the substrate engage through commercial channels rather than through public package managers.

These open questions do not weaken the third bet. They locate honestly where the bet sits: substrate proven, integration with Hyperon-class systems demonstrated, methodology validated on real construction, throughput characteristics measured. Orchestration layer above the substrate is forward-pointing. External adoption and independent validation are dependent on commercial-channel maturation. Public availability is gated.

The third bet is not less serious than the other two for sitting where it does. It is less ambitious — by design — and at this stage of its development, less ambitious is what allows it to be empirically

grounded rather than aspirational.

VIII. Where this sits

The argument, restated:

The AI conversation has narrowed to two bets — scaling produces intelligence; cognitive architecture produces intelligence — both unresolved. A third question sits beneath both: what substrate supports AI-assisted software construction, and what does that substrate enable that current toolchains do not?

The third bet's claim is that this question has an answer. The answer is a substrate that makes semantic representations and their transformations first-class artifacts. The substrate operates on top of current LLMs, integrates with cognitive-architecture systems where they mature, and is independent of which AGI bet pays off. The substrate has been validated on a project family of approximately twenty-six sibling repositories built over twelve weeks using a methodology in which verbal specifications drive AI-assisted construction. The substrate's value extends to ecosystems beyond its own — the Hyperon integration is a worked example.

Whether the third bet's broader vision — AI orchestration over collections of specifications, isomorphic mappings, Adaptive Programming as the workflow discipline — gets built by the project that originated this substrate or by others who reach independently for comparable substrates, the architectural shape is the same. The substrate makes the layer above tractable. The layer above is where AI-assisted software construction becomes specification-driven rather than code-driven.

The point of this paper is not to argue that this is the only path forward, or even the best one. It is to note that the path exists, is buildable with current technology, and addresses a class of problem the two dominant AGI bets do not address. That seems worth saying clearly enough for the right readers to find it.

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 — is shared under appropriate terms once a conversation has scope.

Inquiries: [your preferred contact channel].

End of paper. ~3,500 words. Companion to the technical document family operated by B3. Readers reaching this paper through external channels should treat it as a position statement; the technical material is shared through commercial conversation.