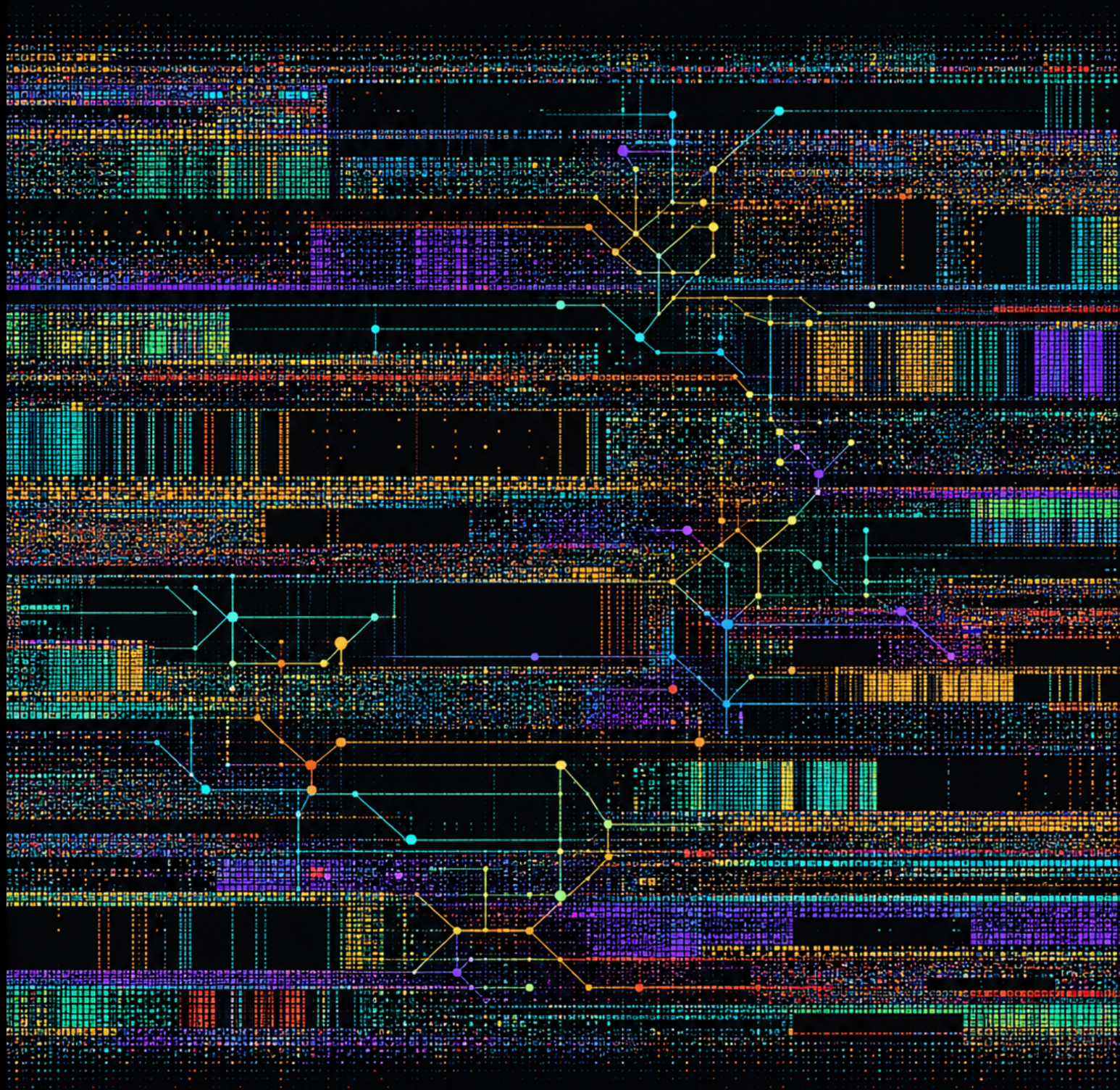


Neurolinguistic and Neurosymbolic Approaches to Memory Dump Analysis



Concept by Dmitry Vostokov
DumpAnalysis.org

Neurolinguistic and Neurosymbolic Approaches to Memory Dump Analysis

Representation access and functional dissociation in pattern-oriented diagnostics

Concept by Dmitry Vostokov

DumpAnalysis.org

Developed with AI assistance (GPT-6 Astra and Claude Fable 5.1)

Version 14 | 26 October 2026

Throughout this article, “we” and “our” refer to Dmitry Vostokov as the authorial plural. The acknowledgment of AI assistance does not attribute the cited prior work or historical recollections to the AI systems.

Abstract

A memory dump preserves an arrangement of software structures from which an analyst reconstructs aspects of computation. A structure may be present but inaccessible, accessible but incorrectly interpreted, or correctly interpreted but unable to participate in the required activity. Here we consider how distinctions investigated in neurolinguistics and cognitive neuropsychology can guide such reconstruction. We connect representation and access, functional dissociation, disconnection, and inappropriate persistence to concrete diagnostic questions. These investigative distinctions guide a proposed neurosymbolic specialization of the Analysis Pattern Network architecture built on our existing PrologWinDbg project. Its neural components recognize candidate patterns and select useful questions; its symbolic components represent evidence, evaluate pattern conditions, and preserve the dependencies of conclusions. Trace Ontology supplies a foundation for diagnostic meanings, and Trace World motivates relating heterogeneous evidence. The proposed design includes a debugger adapter, versioned semantic mappings, an evidence graph, coverage records, executable pattern contracts, an analysis controller, and an explanation service. Three worked scenarios illustrate the difference between a suggestive pattern, a supported structural finding, and a causal hypothesis. A novelty assessment locates the proposal relative to existing pattern-oriented diagnostics, model-based diagnosis, neural logic programming, and automated crash analysis. The published PrologWinDbg v0.2.0 extension and our book on Prolog for pattern-oriented diagnostics supply the implementation and diagnostic foundations. The additional functional profiles, pattern contracts, provenance services, and neural integration remain proposals. The repository records native x64 and ARM64 CDB validation, with interactive WinDbg checks still outstanding; no measured diagnostic improvements are claimed.

Keywords: memory dump analysis; neurolinguistics; cognitive neuropsychology; neurosymbolic AI; diagnostic analysis patterns; Analysis Pattern Network; Trace Ontology; Trace World; Prolog; WinDbg extension; evidence provenance; functional dissociation.

1 From memory contents to diagnostic meaning

A memory dump contains more than the bytes that an analyst happens to inspect. Those bytes participate in structures, and the structures participate in activities. A pointer may connect two objects. An index may provide access to a collection. A thread may own a resource needed by another thread. The diagnostic meaning of an individual value therefore depends on its relationships and on the operation under investigation.

For example, finding the text of a document in memory does not establish that an application can open that document. The text may belong to a live document object, an obsolete copy, a temporary buffer, or a freed allocation whose contents have not yet been overwritten. Even a valid object may be unavailable through the particular index used by the failed request. The question is not exhausted by locating a recognizable representation. We also need to reconstruct how the application was supposed to reach and use it.

Pattern-oriented software diagnostics provides a vocabulary for recurring structures and behaviors in software execution artifacts [1, 2]. Here we extend that perspective in two related directions. The first concerns the questions asked of the evidence. Neurolinguistic and cognitive-neuropsychological concepts suggest comparisons between preserved and impaired operations. The second concerns the machinery that evaluates those questions. Neurosymbolic methods provide ways to combine learned recognition with explicit constraints and inference.

The prefix *neuro* has a different reference in these terms. Neurolinguistics concerns the biological organization of language; neurosymbolic AI combines artificial neural networks with symbolic representations and reasoning. The two approaches can be used independently. An analyst can investigate representation and access without a neural model, and a neurosymbolic system can evaluate diagnostic patterns without adopting the analogies developed here.

The word *memory* requires care in this comparison. Computer memory is an addressable storage system; human memory comprises biological and cognitive processes. A dump is not a recording of a program's thoughts. The proposed transfer concerns methods of investigation and distinctions between mechanisms. It does not depend on assigning human disorders to software.

A dump also has an observation boundary. Its contents depend on the capture method, included address ranges, target state, and available metadata. Microsoft's user-mode dump documentation describes options that include different combinations of memory, handles, modules, and thread information [13]. A missing field may be unavailable to the analyst even when it existed in the target. Every diagnostic conclusion must remain within the scope of the evidence that supports it.

2 What the neurolinguistic perspective contributes

2.1 Representation and access

Neurolinguistics intersects with cognitive neuropsychology in investigating how impaired performance relates to underlying processing mechanisms. Representation versus access and functional dissociation also belong to cognitive neuropsychology more broadly; their use is not confined to language. Bates and colleagues studied associations between lesion locations and measures of language performance, including fluency and comprehension [5]. Plaut and Shallice investigated how damage to a connectionist model could produce characteristic combinations of reading errors [6]. These studies supply methodological precedents for relating patterns of performance to a structured account of processing. They do not establish a diagnostic method for software.

The software correspondence proposed here separates a representation from an operation that uses it. A record may remain intact while a lookup path is broken. Conversely, an index entry may resolve to an object whose contents no longer satisfy the application's requirements. This distinction suggests different next observations: inspect the access path in the first case and the object's state in the second.

Access is always access under some conditions. A request may use the wrong namespace, transaction, security context, generation, or encoding. Consequently, apparent access failure can result from a mismatch between the analyst's context and the application's context. The system design must retain these conditions rather than reduce every successful memory read to a claim that the application could use the object.

2.2 Functional dissociation

We use functional dissociation as an investigative comparison: one operation fails while a related operation succeeds. Suppose enumeration finds an object, but lookup by name does not. This profile directs attention to the structures used differently by the two operations. It does not immediately establish an index defect. Enumeration may include objects that are intentionally excluded from lookup, or the operations may have been observed at different times.

A stronger experiment compares two controlled perturbations. One perturbation disrupts lookup while preserving enumeration; another disrupts enumeration while preserving lookup. Such a double dissociation can help distinguish dependencies, provided the test conditions, inputs, and success criteria are comparable. It is evidence for a particular model of the software, not a proof that its components are wholly independent. Shared resources, thresholds, and different task difficulty remain alternative explanations.

The practical consequence is that a diagnostic report should record preserved functions as deliberately as failed functions. A failure description gives one part of a functional profile. Successful operations can eliminate explanations that would have prevented those operations as well.

2.3 Disconnection and distributed function

A software function may depend on several intact components and on the relationships between them. A producer can create a valid work item while the consumer never receives it because the item is linked into the wrong queue. A callback object can remain valid while its registration has been removed. These cases suggest examining relationships as possible sites of failure.

The proposed analogy is functional rather than anatomical. There is no general correspondence between a particular brain region and a software module. The location at which a symptom is observed may be downstream from the defective relationship. A waiting thread, for example, may be the visible consequence of a missing producer action elsewhere.

2.4 Persistence and experimental perturbation

Inappropriate persistence provides another useful question. Does a new request inherit an earlier request's identity, result, or execution context? The term perseveration can motivate this question, but a software diagnosis should name the established mechanism, such as stale session state or incorrect cache reuse. Repeated output may be legitimate retry behavior. A single dump generally cannot demonstrate a repeated temporal process without additional recorded history.

Controlled perturbation turns these analogies into a research method. In a test system we can remove an index entry, corrupt a field, alter a generation identifier, or suppress a notification, then compare the

resulting functional profiles and artifacts. The independent test harness records the intervention. The diagnostic system receives only the evidence made available for the experiment. This arrangement tests whether the proposed distinctions improve diagnosis rather than merely provide new names for known faults.

Table 1 summarizes the transfer. Every correspondence in the software column is a proposal for diagnostic use.

Investigative concept	Software question	Evidence needed
Representation and access	Does the required information exist in a valid form and remain accessible through the intended route?	Object identity, lifecycle, references, lookup context
Functional dissociation	Which related operations remain possible?	Comparable success and failure observations
Disconnection	Which required relationship is broken between otherwise usable components?	Queue, reference, registration, or ownership structure
Inappropriate persistence	Is earlier state being reused outside its valid context?	Request identity, generation, and temporal evidence
Controlled perturbation	Which controlled perturbation reproduces the observed profile?	Intervention record, reference execution, captured artifacts

3 Relationship to pattern-oriented diagnostics

Our earlier Analysis Pattern Network proposal already treats concrete analysis patterns as processing units that enrich software execution artifacts [3]. It permits machine-learning, rule-based, human-based, and transformation-based implementations. Later iterations operate on the enriched evidence and may use different techniques. The present architecture is a specialization of that proposal, with a more explicit account of evidence and inference. Its implementation foundation is our current PrologWinDbg project, whose experimental v0.2.0 prerelease was published on 20 October 2026 [21]. Earlier Prolog experiments and the 2011 wdes.dll proof of concept are discussed in Section 5.4 [24, 25].

This continuity matters. Combining machine learning and rules within a diagnostic workflow is not introduced here. The extension already contributes structured debugger evidence, snapshot scope, explicit incomplete results, and bounded queries [21]. The additional design work concerns the contracts between pattern implementations: what an observation means, how far a search was completed, which assumptions permit a conclusion, and how the conclusion changes when a dependency is withdrawn.

Our Trace Ontology pattern supplies a related semantic foundation [22]. It extends recognition of Basic Facts toward identifying domain entities, their kinds, relationships, and constraints. In the present proposal, we would apply that distinction to the interpretation of dump observations as well as trace messages. An address, a symbol, or a request identifier becomes diagnostically useful when its role and permitted relationships are understood. Applying the pattern to memory structures requires explicit mappings from debugger observations to application concepts; the pattern does not supply those mappings automatically.

Trace World describes how several trace sources and implementation discourses contribute to a shared diagnostic world [23]. We would extend this treatment to a case containing dumps, logs, and traces. Each artifact would contribute observations with its own capture conditions and provenance. A shared model would preserve uncertain identities and incompatible interpretations, rather than assume that collecting artifacts together establishes a consistent history. Section 8 develops the semantic responsibilities, and Section 18.2 considers the resulting research direction.

The conceptual vocabulary also connects to software narratology. The Software Narratology Square broadens the domain to software construction and post-construction narratives [4]. An incident description, a log message, an object graph, and an analyst's explanation occupy different evidential roles. A fluent explanation is a derived narrative. It should remain connected to the artifacts from which it was constructed.

For this proposal, we distinguish an evidence graph from a diagnostic narrative. The graph records observations and inference dependencies. The narrative presents a selected, ordered account for the analyst. A narrative can explain why a structure is suspicious without claiming to recover every event that produced it. This distinction prevents a plausible chronology from acquiring the status of recorded execution history.

Patterns also operate at different levels. An elementary pattern may establish that an address belongs to a captured allocation. A composite pattern may relate several allocations and ownership conditions. A problem hypothesis may connect the composite pattern to an observed application failure. These levels should be linked, but they should not be collapsed into one confidence score.

4 What makes the proposed system neurosymbolic

A neural component can learn similarities that are difficult to enumerate manually. It may retrieve relevant pattern descriptions, compare normalized stacks, recognize a recurring arrangement of object roles, or rank possible explanations of an incident description. These are useful capabilities even when the input is already structured.

A symbolic component represents entities and relationships explicitly. It can ask whether a wait dependency exists, whether an index entry satisfies an application invariant, or whether the premises of a pattern are available. Its conclusions are relative to the supplied rules and evidence. A valid derivation cannot correct a wrong type layout or an incorrectly asserted premise.

DeepProbLog integrates neural predicates with probabilistic logic programming [8]. Scallop combines deep learning with a relational, Datalog-based language and differentiable reasoning [9]. These systems demonstrate established approaches to neural and symbolic integration. They are possible foundations for later experiments, not mandatory dependencies of the first prototype.

We propose an initial modular design. A neural component produces typed candidates; a symbolic evaluator checks the relevant pattern contracts; the controller uses the resulting support, contradictions, and unknowns to choose the next observation. In this arrangement, neural output has an operational role in the analysis, and symbolic results constrain subsequent neural proposals. Merely asking an LLM to describe a debugger transcript would not implement this design.

The first version need not train the whole pipeline end to end. Separating extraction, candidate ranking, and rule evaluation makes failures easier to localize. A later experiment may learn parameters inside symbolic programs, but it must preserve the distinction between a learned assertion and a direct observation. A probability belongs to a specified model and calibration procedure. It is not a substitute for the address range, command result, or rule that supports a claim.

5 Prior work and boundaries

5.1 Model based and pattern based diagnosis

Reiter's theory of diagnosis relates observations that conflict with a system description to assumptions about abnormal components [7]. It provides a foundational precedent for retaining alternative diagnoses and using further observations to discriminate between them. The proposed system does not replace that reasoning with a new theory. It specializes diagnostic reasoning to incomplete software execution artifacts and to an existing pattern language.

Pattern-oriented software diagnostics contributes named structures, relationships, and workflows [1, 2]. The Analysis Pattern Network contributes iterative enrichment and heterogeneous pattern implementations [3]. The novelty discussion must therefore compare the present design with this earlier architecture, including its explicit provision for machine-learning and rule-based processing.

5.2 Automated analysis of crashes and memory

RETracer reconstructs program semantics from partial memory dumps using reverse execution for crash triage [16]. It establishes prior work on analyzing the relationship between a failure and preceding computation despite incomplete state. The present proposal does not claim that causal reconstruction from dumps is new.

AutoCrashFL uses an LLM agent with crash dumps and corresponding source code for fault localization in an industrial setting [10]. This is direct prior work against any broad claim to introduce LLM-assisted dump analysis. The distinction proposed here concerns explicit pattern contracts, evidence states, and a neurolinguistically motivated functional profile, rather than the presence of an LLM or debugger access.

MemHint combines LLM-generated memory-management summaries with symbolic checks and static analyzers to find memory leaks [11]. Its input and purpose differ from postmortem dump analysis, but it is close prior work on neural interpretation constrained by symbolic program reasoning. The October 2026 revision is the version referenced here.

Wu and colleagues introduce RAM-Weaver, a system that uses LLMs to support contextual queries about user activities in application memory dumps [17]. Its forensic objective differs from explaining a software failure, but it establishes another relevant application of language models to memory evidence. Li's accessible manuscript on automated triage evaluates log-based limitations and motivates combining logs, dumps, and metrics [18]. Its reported experiments should not be mistaken for an evaluation of the complete dump-analysis architecture proposed in this article.

5.3 Evidence quality as part of the problem

Ottmann and colleagues investigate quasi-instantaneous consistency in memory dumps acquired while a system is running [12]. Their work is relevant because a reconstructed inconsistency can originate in acquisition. Capture omissions and capture inconsistencies are different conditions: the first withholds evidence; the second may create an apparent combination of states that never coexisted.

These prior contributions leave useful engineering questions without establishing that this article is the first to ask them. The literature assessment is a focused comparison of the cited work as of 9 October 2026. It is not an exhaustive priority or patent search. Uncited work may overlap with individual elements or with their combination.

5.4 Earlier Prolog work and the existing extension project

Our use of Prolog for software diagnostics predates the current project. In a blog post dated 5 February 2011, we recalled considering a Prolog expert system for crash dump analysis five or six years earlier, placing those investigations approximately in 2005–2006 [24]. We had written a simple Prolog program to match product hotfixes and were contemplating a WinDbg extension after learning that SWI-Prolog provided an interpreter DLL that could be loaded into a Win32 application. The approximate date follows from that retrospective account.

Another post on 5 February 2011 documented Windows Debugging Expert System, a proof-of-concept extension supplied by Memory Dump Analysis Services [25]. We reported loading `wdes.dll` into WinDbg with x64 SWI-Prolog installed. The transcript shows the extension's help output and successful loading of SWI-Prolog through `!pload`. The post also states that further extension commands were awaited. It therefore documents an interpreter-loading prototype; it does not establish the diagnostic capabilities of the present PrologWinDbg implementation. These earlier experiments provide a historical foundation for the current work without implying uninterrupted development or continuity of the code.

Our published book, *Prolog for Pattern-Oriented Software Diagnostics: Logic Programming for Memory Dump, Trace, and Log Analysis* (OpenTask, 2026), develops these connections [20]. The publicly available Version 20 presents 192 analysis patterns with Prolog recognizers, transcripts, and limits of formalization. It connects artifact representation to pattern evaluation, explanation by meta-interpretation, and integration with diagnostic tools. Section 31.6 presents an embedded debugger-extension design; Section 31.7 documents PrologWinDbg as built; Section 31.9 develops Prolog verification of LLM-proposed hypotheses. The book supplies direct foundations within our own diagnostic program. Verification of a neural proposal by Prolog is already part of that published foundation.

The implemented project is published as PrologWinDbg in our DumpAnalysis GitHub repository [21]. Its experimental v0.2.0 prerelease, published on 20 October 2026, provides a native C++ extension, `prolog.dll`, connected through local pipes to a persistent SWI-Prolog worker, for native x64 and ARM64 targets. The release supplies separate Windows x64 and ARM64 archives containing the DLL, Prolog support files, examples, and documentation, together with SHA-256 checksums. SWI-Prolog is installed separately. The repository also supplies source, a Windows test target, build and smoke-test scripts, and validation records. Section 31.7 of the published book documents this process boundary alongside the embedded alternative in Section 31.6. We retain the existing worker architecture in the present design.

The initial project targeted native x64 user-mode dumps and live debugging; v0.2.0 adds native ARM64 support. The debugger, extension DLL, and native target must match architectures. The implemented providers expose threads, stack frames and stack pointers, symbols, loaded modules, and memory, with explicit coverage and read outcomes. General register predicates, typed fields, synchronization-specific providers, and automatic execution actions require further work. Section 11 separates these implementation boundaries from the article's proposed additions.

The extension makes Prolog a practical bridge between diagnostic vocabulary and debugger observations. The neurolinguistic perspective determines which functional distinctions are worth testing. Prolog expresses their conditions and dependencies. A neural component can propose candidates or useful observations. This division of responsibility does not make the extension itself a neural model, and the presence of the extension alone does not establish the proposed neurosymbolic workflow.

6 Novelty and proposed contribution

The proposed contribution is a neurolinguistically informed specialization of Analysis Pattern Networks to be implemented by extending the existing PrologWinDbg project. Prolog-based hotfix matching and the proposal to use Prolog within WinDbg already appear in our earlier work [24, 25]. Trace Ontology and Trace World are existing foundations for semantic representation and integration across trace sources [22, 23]. The contribution proposed here is architectural and methodological. The published extension supplies the native bridge, separate Prolog worker, snapshot scope, coverage reporting, and query bounds; the published book develops pattern recognizers and verification of neural hypotheses [20, 21]. These are existing foundations. The incremental contribution concerns how functional profiles, application-specific contracts, versioned semantic mappings, persistent evidence dependencies, and neural feedback are integrated and evaluated.

First, we propose using a functional profile as a first-class diagnostic object. It records operations reported as failed, operations observed as successful, the conditions of those observations, and the structures that each operation requires. Representation failure, access failure, contextual mismatch, and output failure become competing interpretations of the same symptom. Their names alone are not novel diagnostic mechanisms; the contribution would be their systematic integration into pattern selection and testing.

Second, we propose extending the project's existing coverage and incomplete-read semantics into application-specific contracts for patterns that depend on absence. The design uses a scoped record of completed inspection, compatible decoding, and capture consistency. A missing index entry can support a finding only within that scope. The proposed engineering contribution is the contract connecting a negative application-level observation to its completed inspection procedure. PrologWinDbg v0.2.0 already distinguishes full, partial, and unavailable reads and explicitly warns against inferring absence from a failed Prolog query [21]. Neither that distinction nor coverage reporting is introduced here.

Third, we propose representing neural candidates and symbolic conclusions in one provenance structure while preserving their different origins. A pattern instance records which candidates prompted it, which observations supported it, which rules were used, and which assumptions remain unresolved. A dependency change can invalidate a conclusion without deleting the history of the analysis.

Fourth, we propose using the same representation for incident analysis and controlled perturbation studies. A benchmark would distinguish faults that destroy a representation from faults that disconnect access, change context, or preserve obsolete state. This would make it possible to measure whether the conceptual framework produces better questions and fewer unsupported conclusions.

Table 2 separates established foundations from the contribution proposed here.

Element	Established foundation	Proposed extension
Mixed diagnostic implementations	Analysis Pattern Networks [3]	Typed contracts and explicit evidence dependencies
Neural and logical integration	DeepProbLog, Scallop, and the Prolog verifier design [8, 9, 20]	Functional profiles and feedback within the diagnostic contract workflow
Debugger and Prolog integration	2011 wdes.dll loading prototype [25]; PrologWinDbg v0.2.0 [21]	Application-specific providers and persistent derivation records

Element	Established foundation	Proposed extension
Reasoning about abnormal behavior	Model-based diagnosis [7]	Functional profiles linked to memory structures
Automated crash investigation	RETracer and AutoCrashFL [16, 10]	Pattern findings separated from causal hypotheses
Neural guidance for memory defects	MemHint [11]	Postmortem evidence rather than source analysis alone
Analysis of incomplete captures	Dump research and extension coverage semantics [12, 13, 21]	Application-level absence and consistency conditions carried into derivations
Diagnostic meaning	Trace Ontology [22]	Versioned mappings from dump observations to application concepts and contracts
Shared diagnostic world	Trace World [23]	Identity, provenance, and consistency conditions across dumps, logs, and traces

We do not claim novelty for neural networks, Prolog, debugger integration, process separation, evidence graphs, ontologies, shared diagnostic worlds, fault injection, or the general combination of learned and symbolic reasoning. The project chronology establishes continuity within our own work; it does not establish priority over unrelated systems. A defensible claim would concern the specified integration, supported by implementation artifacts and evidence that its distinctive components matter. If removing the functional profile or coverage contracts produces no measurable difference, that result would weaken the corresponding contribution.

7 Proposed architecture

7.1 Scope and requirements

The implementation starts from PrologWinDbg v0.2.0, whose target scope is native Windows x64 and ARM64 user-mode dumps and stopped live processes, with a matching debugger and extension DLL [21]. We would evaluate the new diagnostic contracts on reproducible dumps first, then on controlled live-debugging stops. A small set of application structures with known layouts and explicitly modeled synchronization objects would be added to the existing thread, stack, module, symbol, and memory evidence. Kernel dumps, WOW64, ARM32, ARM64EC, managed runtimes, and Linux core files remain later target or adapter work. Cross-platform portability would apply to the evidence schema and selected patterns, not to every extraction procedure.

The system must preserve the original artifact, identify the target build, record the decoding environment, and distinguish successful inspection from partial inspection. It must allow analysis without a neural service and must return unresolved findings when required evidence is unavailable. It must also retain multiple competing hypotheses and expose the dependency chain for each reported finding.

Source code is optional for the basic structural patterns. It becomes important for application invariants, control-flow checks, or source-level localization. An assertion about application behavior must carry a

reference to the relevant specification, reviewed rule, or build-matched source interpretation. If that information is missing, the assertion remains an assumption.

7.2 Components and data flow

Figure 1 shows the proposed logical architecture built around the existing extension and Prolog worker. The artifact store holds immutable inputs. The debugger adapter extracts observations into an evidence service. The neural service and symbolic evaluator consume scoped views of that evidence. The analysis controller chooses work and records its results. The explanation service produces reports from the evaluated findings.

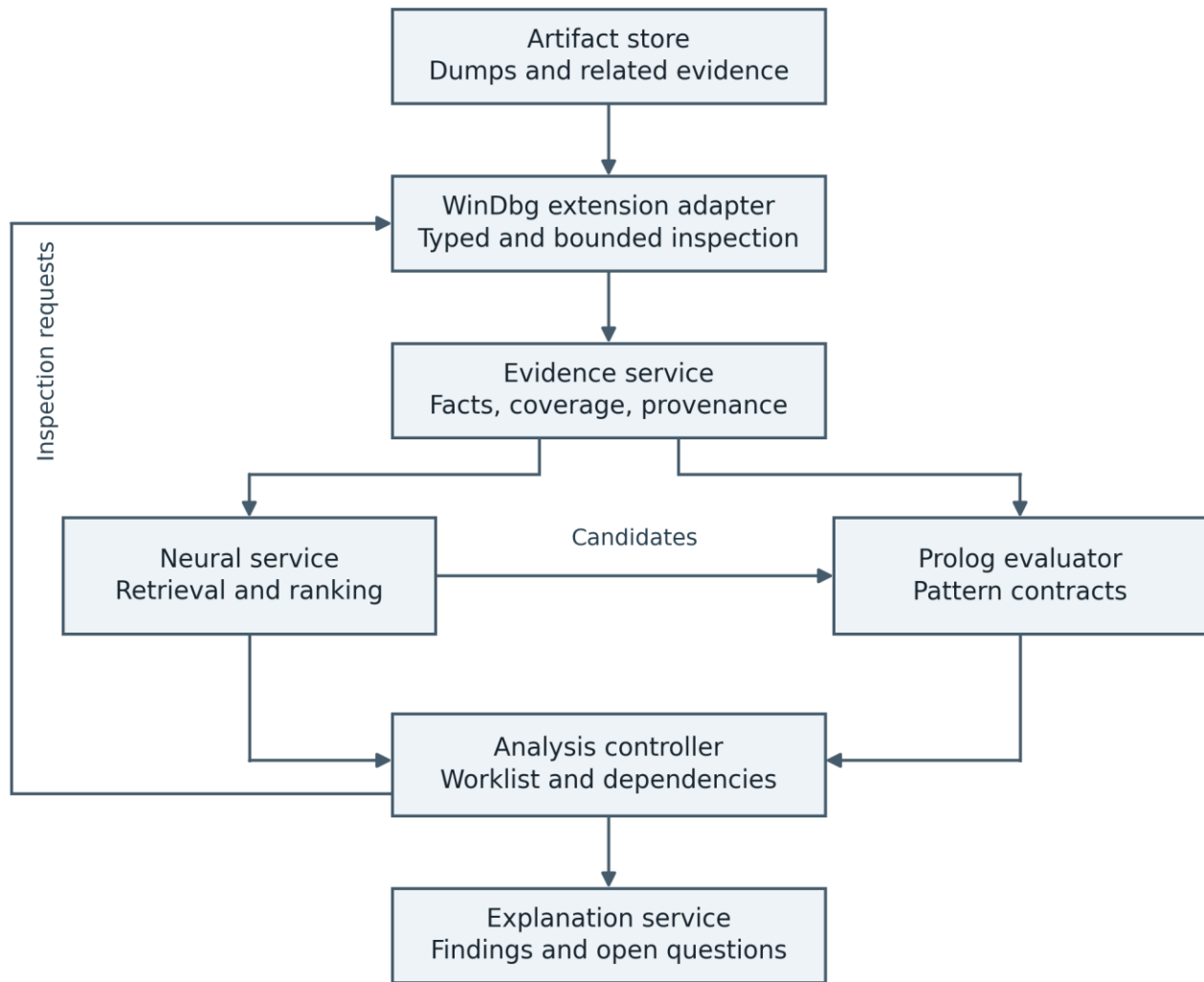


Figure 1. Proposed logical architecture extending PrologWinDbg. Arrows show evidence, candidates, results, and inspection requests. The controller also schedules neural and symbolic work; those control paths are omitted for clarity. Neural candidates enter the evaluator through a typed interface.

The pattern registry contains executable contracts, documentation, applicability metadata, and test fixtures. A proposed ontology registry defines the application concepts and relations used by those contracts. Reviewed mapping specifications connect each supported debugger or trace representation to that vocabulary. The case store contains observations, candidate hypotheses, coverage records, derivations, and the run manifest. Ontology, mapping, and rule versions are recorded with each analysis. Changing one creates a new analysis revision and reevaluates affected findings; it does not silently change the meaning of an existing result.

We would retain the project's WinDbg-hosted C++ bridge and persistent local Prolog worker, then add a controller, persistent case storage, and an optional neural worker. These are logical responsibilities; the first controller can be implemented as reviewed Prolog modules in the existing worker. A later standalone DbgEng host can reuse the adapter and contracts for batch analysis. The evidence graph does not require a graph database: indexed entity and relation tables can implement its first persistent form. The deployment relationship is shown in Figure 2.

7.3 The boundary of trust

The debugger adapter is the only component allowed to introduce observations attributed directly to target bytes or debugger state. A model can propose an interpretation of those bytes, but the proposal receives a different origin label. The proposed service must enforce this distinction even though the existing interactive extension intentionally permits trusted users to execute ordinary Prolog programs. Human annotations are also distinct from direct observations. They can supply important context without pretending to have been extracted from the dump.

The adapter itself is fallible. A wrong layout, failed unwind, or unsupported synchronization type can produce misleading output. Consequently, a direct observation is a claim with provenance and a decoding context, not an infallible statement. The system must be able to challenge extraction as well as application behavior.

8 Evidence model and observation semantics

8.1 Entities and relationships

The initial entity types are Artifact, Capture, Process, Thread, Module, Frame, Allocation, Object, Resource, Operation, Observation, CoverageRecord, PatternInstance, Hypothesis, and Derivation. Relations include references, owns, waitsFor, contains, belongsTo, observedIn, supports, contradicts, dependsOn, and supersedes. These names describe the proposed schema, not a requirement to use a particular database product.

An address alone is not an object identifier. An object key includes the artifact, process address space, address, and any available allocation generation or lifetime evidence. The same virtual address in another capture may describe a different object. Cross-capture identity is an explicit hypothesis supported by application identifiers and other evidence. It must not arise automatically from matching addresses.

We would retain addresses in a representation that preserves the target's full width. When serialized as JSON, hexadecimal strings avoid accidental conversion through a floating-point number in a consumer. Symbol names supplement module identity and relative offsets. Normalized features used for learning remain linked to the original addresses so the analyst can return to the evidence.

8.2 Trace Ontology and diagnostic meaning

We would operationalize Trace Ontology through three distinct responsibilities. The ontology specifies the intended kinds of entities and relations. The evidence graph records claims about particular entities, including their sources, observation conditions, and status. Executable pattern contracts evaluate those claims under explicit application assumptions. A neural model may propose a role or relation, but the proposal remains separate from an accepted mapping or an observed fact. Table 3 distinguishes these responsibilities in the proposed system.

Component	Responsibility	Cancellation example
Diagnostic ontology	Defines entity kinds, relations, roles, and lifecycle meanings	Request, cancellation request, acknowledgment, completion, and worker activity
Evidence graph	Records case claims, provenance, scope, and uncertainty	A message reports an acknowledgment; a dump contains a reference to the request
Prolog pattern contract	Checks applicability, premises, and permitted conclusions	Test whether observed work violates the request lifecycle contract

Mappings need more than matching names. A trace request identifier is not a debugger thread identifier, and a pointer to an object does not by itself establish ownership or active processing. The adapter would preserve the raw observation while a reviewed mapping states its proposed diagnostic meaning and the conditions under which it applies. Source format, application build, ontology version, and mapping version become part of that interpretation's provenance. If two sources use the same word for different states, both meanings must remain available until a justified correspondence is established.

For example, a log message containing cancelled may report that cancellation was requested, that the request was acknowledged, or that termination completed. We would model these as distinct events or states. A dump may show a worker retaining the request object, but retention alone does not show that work continues: the reference may support cleanup, auditing, or deferred destruction. A Prolog contract could support a lifecycle violation only if the evidence identifies the same request generation, establishes the relevant temporal order, demonstrates the activity forbidden after cancellation completion, and supplies the application rule that forbids it. If completion, identity, activity, or timing remains unknown, the result remains unresolved. The ontology makes the distinctions expressible; it does not manufacture the missing premises.

This also makes the neurolinguistic connection more precise. Recognizing a diagnostic label, retrieving the corresponding entity, assigning its role, and relating it to the correct context are different operations in the analyzer. A functional profile can record which operation failed. These are proposed distinctions for evaluating diagnostic interpretation, not evidence that software exhibits a human language disorder. The same test cases should compare whether the distinctions improve investigation over a simpler vocabulary.

8.3 Observation records

Table 4 gives the proposed observation contract beyond the extension's current snapshot-scoped facts and result terms. Fields that are unknown remain explicitly unknown; they are not silently populated by a model.

Field group	Required contents
Identity	Case identifier, observation identifier, artifact digest
Target context	Capture identifier, process address space, architecture, byte order
Proposition	Typed predicate, arguments, value, explicit positive or negative assertion
Origin	Extracted, human supplied, neural proposed, or symbolically derived

Field group	Required contents
Location	Address range, register, stack frame, log record, or source location
Extraction	Adapter version, operation, parameters, result status, inspected byte count
Decoding	Module build identity, symbol identity where available, layout version
Semantics	Ontology version, mapping version, source role, and applicability conditions
Time and scope	Capture time or interval, request context, scope identifier
Dependencies	Premise identifiers, rule identifier, assumption identifiers
Quality	Coverage and consistency references, warnings, calibration metadata if applicable

A failed read needs its own result status. The distinction between not captured, unreadable, unsupported, timed out, and malformed is diagnostically useful. A short read cannot become a successful read by padding the remainder with zeros. A traversal that reaches a limit returns a partial result and records the limit.

8.4 Unknown and conflicting evidence

The proposed claim evaluator retains support and contradiction separately. This is a finding-level service above the extension's current read and query outcomes. It presents four evidence states: supported, refuted, unresolved, and conflicted. Supported means that the applicable derivations support the claim and no accepted counterevidence currently defeats it. Refuted means that accepted evidence contradicts it. Unresolved means that the relevant decision conditions have not been established. Conflicted means that accepted evidence supports incompatible results that require investigation.

This is a design convention, not a claim to introduce a new logic. In particular, ordinary Prolog negation as failure must not determine whether target data is absent. A missing predicate in the fact store can mean that no extractor has inspected the relevant region. The engine may use closed-world reasoning inside a completed, explicitly bounded enumeration, but that closure must not extend to the whole dump or the whole application.

A PrologWinDbg protocol status such as false, timeout, or answer_limit describes query execution [21]. It must not be mapped mechanically to a refuted or supported diagnosis. The controller must interpret the applicable pattern contract and retained evidence. Candidate lifecycle is a separate dimension. A candidate can be queued, evaluated, awaiting evidence, or archived while its evidence state is unresolved. Keeping these dimensions separate prevents a completed analysis task from being displayed as a confirmed diagnosis.

8.5 Coverage and consistency

A coverage record identifies the exact predicate domain inspected. For an index, it might describe the root object, bucket array, overflow chains, key-comparison semantics, and the capture revision. It records whether all required ranges were available and whether traversal completed without a decoding or resource-limit error. A byte-complete capture is not automatically a logically complete index inspection.

Consistency is recorded separately. A paused process may have been captured during a legitimate multi-step update. A capture of a running system may combine observations from different moments [12]. The evaluator therefore needs an invariant appropriate to that program point, or evidence that the structure was at a stable boundary. If these conditions are unknown, absence may be observed locally without supporting a global inconsistency finding.

9 Executable diagnostic pattern contracts

The existing extension can load and evaluate Prolog rules; the proposed contract layer adds an explicit diagnostic interface to selected reviewed rules [21]. Each executable pattern needs an identifier and version, an applicability predicate, required observations, permitted assumptions, supporting conditions, refuting conditions, and a result vocabulary. It also declares the inspections it may request and the evidence it promises to preserve. Documentation should distinguish established catalog patterns from new experimental specializations.

For example, an experimental Index Access Inconsistency contract would apply only to an identified index implementation and a compatible application lifecycle rule. It requires a valid live object, the relevant lookup key and namespace, a stable invariant requiring index membership, and a completed lookup inspection. It refutes its missing-entry explanation when the required entry is present in the relevant scope. It remains unresolved when the index is only partially captured.

The following Prolog-style fragment illustrates the symbolic interface. The predicates beginning with `accepted` are supplied by a provenance-aware evidence service. They succeed only when the referenced evidence is accepted for the current revision and has no unresolved conflict. This is an interface specification, not a complete implementation of that service.

```
index_access_inconsistency(Capture, Object, Key, Scope) :-  
    accepted_live_object(Capture, Object),  
    accepted_object_key(Capture, Object, Key, Scope),  
    accepted_membership_requirement(Capture, Object, Scope),  
    accepted_stable_boundary(Capture, Scope),  
    accepted_complete_lookup(Capture, Key, Scope, Evidence),  
    accepted_lookup_result(Evidence, absent).
```

No negative fact is inferred merely because an `index_entry` predicate fails. The lookup extractor must explicitly return `absent` after a completed inspection under the specified key semantics. If it returns `partial`, `unsupported`, or `conflicted`, the rule does not establish the pattern. The controller still records the candidate and its missing premises; rule failure alone must not be presented as refutation.

A real implementation would emit a derivation record containing the identifiers of all accepted premises and the pattern version. It would also record which premise prevented a result. This provides a useful explanation of non-diagnosis: the system can say that it could not establish index completeness, rather than merely report that no pattern matched.

Pattern composition requires similar discipline. One pattern's output may satisfy another pattern's input, but the dependency remains visible. Two patterns that consume the same suspect pointer are not independent confirmations. When the underlying pointer interpretation is invalidated, both results are invalidated or reevaluated.

10 Analysis control and hypothesis refinement

The controller begins with target validation and a small structural inventory. It records capture coverage, architecture, module identities, thread contexts, and extraction failures. These facts define which patterns can run. An incident description contributes a functional profile, but its claims retain their human-supplied origin until corroborated.

The next phase generates candidates from three sources: deterministic pattern prerequisites, retrieval of similar reviewed cases, and neural interpretation of the current evidence view. This combination avoids making the neural model the sole gatekeeper. A simple structural pattern should still run when an unfamiliar incident description confuses the model.

For each candidate, the evaluator returns supporting premises, counterevidence, missing evidence, and applicability failures. The controller then chooses a bounded next inspection. Initially, a transparent priority policy is sufficient: prefer an inexpensive observation that distinguishes several remaining candidates, respect dependency order, and avoid repeating an unchanged operation. A learned policy would be evaluated later against this baseline.

The proposed control loop is summarized below. Every step appends a record to the analysis history.

- validate the target and establish an evidence revision
- collect a bounded structural inventory
- propose candidates and retain their origins
- evaluate applicable contracts against accepted evidence
- select a useful permitted inspection for unresolved candidates
- record its observations, coverage, and failures
- invalidate or reevaluate affected derivations
- stop at a fixed point, a budget limit, or an analyst stop request
- report findings, alternatives, unknowns, and next evidence needs

A fixed point means that the configured worklist has no new actionable item under the current evidence and policy. It does not mean that the incident has been completely explained. A budget stop must preserve pending work. Repeated candidates are deduplicated by their typed arguments and evidence revision so that the system cannot consume its budget by rephrasing the same hypothesis.

Neural suggestions cannot promote themselves into facts through repetition. The controller stores their evidence dependencies and excludes a generated report from becoming independent support for its own premises. Similarly, an analyst's acceptance of a report is a review event; it does not transform every inference in that report into a direct observation.

11 Debugger integration and system design

11.1 Existing bridge and worker architecture

Microsoft's debugger engine supports debugger extensions and standalone applications [14]. The Debugger Data Model provides additional object-model facilities that may support later typed providers [15]. The existing project already supplies the initial bridge: native prolog.dll uses C++ and DbgEng, while a persistent SWI-Prolog process communicates with it over local pipes [21]. We would retain that boundary as the implementation foundation.

In PrologWinDbg v0.2.0, each !prolog.query collects a fresh structural snapshot. The C++ bridge performs capture and services lazy memory and symbol-name requests on the debugger command thread. The separate Prolog worker enumerates answers. The worker persists across queries, but debugger facts and

read caches do not: they are cleared after each query. The snapshot identifier is an observation identifier local to the extension instance, not a retained copy of target memory.

Figure 2 distinguishes that existing process boundary from the proposed analysis services. The controller can initially run as reviewed modules in the Prolog worker. Its persistent case store must copy the required observations before the current snapshot is cleared. The optional neural service returns typed candidates to the controller, which dispatches approved pattern evaluation and inspection requests.

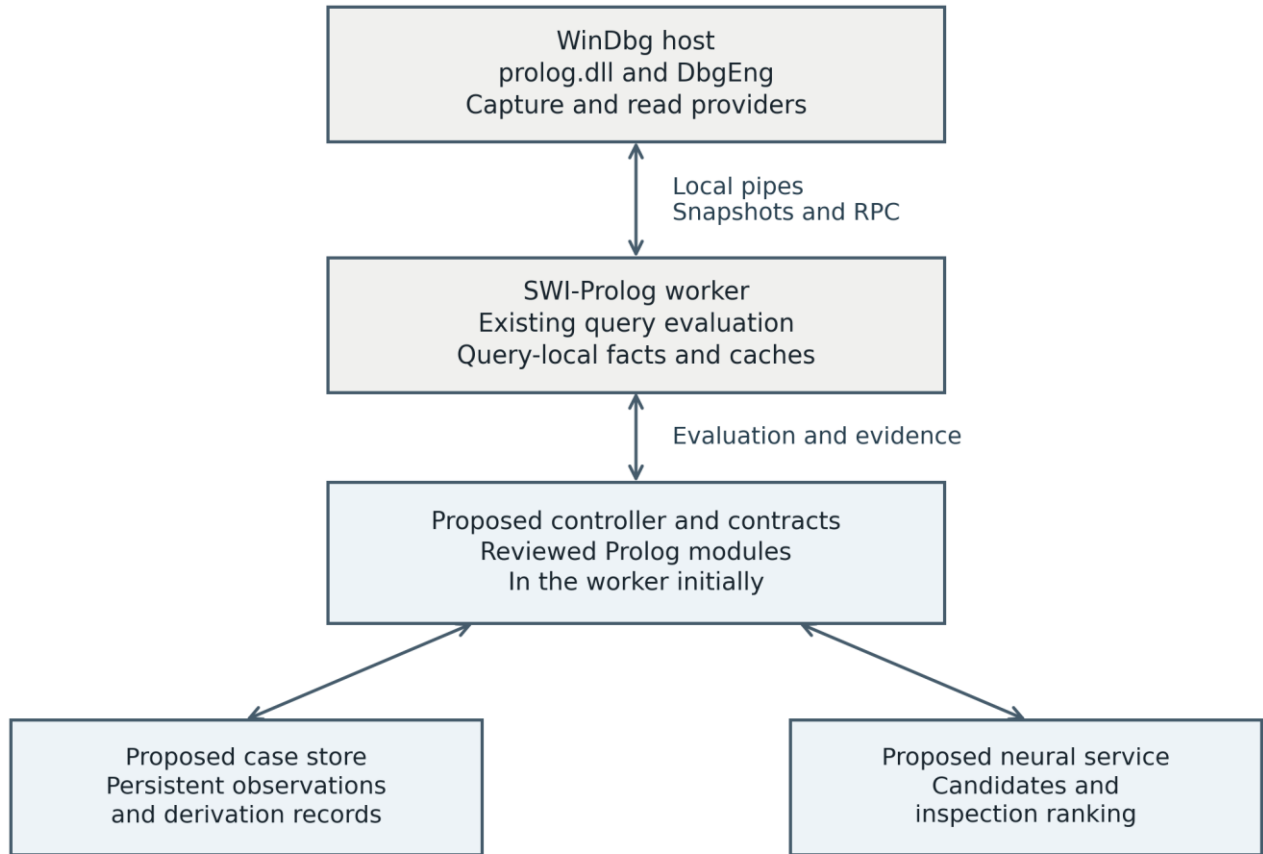


Figure 2. Deployment built on the existing local pipe bridge. The upper two boxes represent the PrologWinDbg v0.2.0 host and worker. Lower boxes represent proposed services. DbgEng calls remain in the WinDbg host; the Prolog worker does not directly own debugger interfaces. Connecting a neural service, persistent evidence store, and contract controller requires additional implementation.

Process separation permits worker recovery but does not make a synchronous DbgEng call interruptible. PrologWinDbg v0.2.0 checks time and interruption between debugger calls and while waiting on the worker. Symbol loading or a remote read can still block inside the engine. Ordinary Prolog time and inference limits preserve the worker; interrupted IPC or transport failure can require restarting it and consulting rules again [21]. These are operational conditions to preserve in the extended system.

11.2 Typed inspection interface

Table 5 describes proposed logical operations across the adapter, controller, and explanation service. These names specify the new interface; they are not commands already exported by PrologWinDbg v0.2.0. ReadRegion and EnumerateThreads can build on existing providers. DecodeObject, InspectIndex, and InspectWaits require additional application or resource semantics. Each proposed response includes the evidence revision, provenance, completion status, and any coverage limitation.

Operation	Main input	Main result
OpenArtifact	Local artifact reference and expected digest	Target identity and supported capabilities
ReadRegion	Address space, address, requested length	Bytes, actual ranges, read status
EnumerateThreads	Capture and bounded filter	Thread identities, contexts, completion record
DecodeObject	Address, approved layout, build identity	Typed fields and per-field read provenance
InspectIndex	Root, key, namespace, implementation contract	Present, absent, partial, or unsupported result
InspectWaits	Thread set and approved resource models	Wait relations and semantic qualifications
EvaluatePattern	Pattern version and typed arguments	Finding state, derivation, missing premises
ExplainFinding	Finding and evidence revision	Evidence-backed explanation and alternatives

An adapter capability response must distinguish implemented operations from operations applicable to the current target. A native object decoder does not imply support for every C++ library container. Missing private symbols may permit raw reads while preventing typed decoding. Such conditions belong in capability and applicability results.

11.3 Storage and reproducibility

The artifact store is immutable and addressed by content digest. The proposed case store holds extracted observations and revisions. PrologWinDbg v0.2.0 retains user rules but discards debugger facts and caches after each query; persistent evidence storage is therefore an addition, not a description of the current worker. The pattern, ontology, mapping, and model registries use immutable versions. A run manifest records artifact digests, adapter and debugger versions, symbols and layouts, ontology and mapping versions, rule versions, model identifiers, prompts or prompt digests, decoding parameters, resource limits, and human annotations.

Caching uses these identities. A decoded object cache keyed only by address would be unsafe across captures or symbol changes. A pattern result cache must include the relevant evidence revision and contract version. Reanalysis can reuse unchanged observations while producing a new derivation graph.

When the model cannot be reproduced exactly, the original candidate payload remains available for deterministic symbolic reevaluation.

The graph should be materialized on demand. A large heap need not be converted into one enormous language-model prompt. Start from relevant threads, allocations, and incident entities; follow typed relations with explicit bounds; expand when a candidate requires more evidence. Traversal limits become coverage facts rather than invisible implementation details.

11.4 Prolog and constraint solving

The project's existing SWI-Prolog worker is the initial symbolic execution environment. It receives structured facts and supports lazy read and lookup requests through the pipe protocol [21]. Our published book compares the embedded design with PrologWinDbg as built and develops a verifier for LLM proposals [20]. SWI-Prolog's C interface supports embedding [19], but the current project already selects a separate process. The proposed pattern evaluator extends that worker with explicit contracts, derivation records, and missing-premise results.

The rule subset should be controlled: finite domains, bounded traversals, and explicit handling of recursion and negation. Termination is not guaranteed merely because a query is expressed in Prolog. Timeouts return an incomplete reasoning result. A later SMT component could check selected arithmetic or path constraints, but a satisfiable encoding would establish consistency with that encoding, not the historical truth of the proposed execution.

11.5 Delivered capabilities and proposed additions

Table 6 maps PrologWinDbg v0.2.0 to this article. The GitHub repository and release documentation establish its implementation scope. The validation record distinguishes completed native CDB tests from the remaining interactive WinDbg checks [21].

Capability	PrologWinDbg v0.2.0 foundation	Work proposed here
Debugger access	Native x64 and ARM64 user-mode dump and stopped-live providers	Complete remaining deployment checks and add application providers
Structured evidence	Threads, stack prefixes, symbols, loaded modules, and memory	Typed objects, index traversal, resource and lifecycle semantics
Incomplete evidence	Coverage facts and full, partial, or unavailable reads	Absence contracts and four-state finding evaluation
Query lifecycle	Fresh snapshot per query and no cross-query read cache	Persist observations with stable case identities and dependency records
Prolog evaluation	Trusted rule loading, bounded answers, time and inference limits	Reviewed pattern contracts and explanations tied to premises
Neural participation	No neural component in the package	Candidate ranking, case retrieval, and inspection feedback
Execution control	Read-only debugger providers; normal WinDbg commands control execution	Keep diagnostic queries separate from any future action mechanism

The repository's validation record dated 20 October 2026 reports 99 checks on each native Windows architecture: 22 C++ protocol checks, 36 architecture-pairing checks, 16 Prolog rule tests, and 25 worker integration tests. Native x64 and ARM64 MSVC builds also passed CDB smoke tests for full user-mode dumps and stopped live processes. These exercised stack and memory queries, selected-thread and frame-scope restoration, timeout recovery, worker reset, and DLL unload/reload. Worker integration tests use a simulated debugger peer; the CDB tests use the real extension and supplied diagnostic process. The record identifies the tested source commit and retains logs [21]. These are reported project results, not new measurements of the proposed neurosymbolic system.

Interactive WinDbg Ctrl+Break, custom exception/register-context overrides, and additional application dump formats remain manual validation tasks. General register predicates, typed fields, exception providers, kernel-specific data, WOW64, ARM32, ARM64EC, cross-architecture contexts, managed objects, and rule-triggered execution actions remain outside v0.2.0's scope. A planned provider must not appear as implemented in an applicability response. The completed native CDB tests provide an integration baseline; the remaining checks still limit broader claims.

11.6 From an existing query to a diagnostic contract

After loading the matching release DLL into WinDbg or CDB and configuring the installed SWI-Prolog runtime, the analyst can use the following command from the PrologWinDbg v0.2.0 interface. It requests captured stack frames for the current query snapshot [21].

```
!prolog.query wd:current_snapshot(S), wd:stack_frame(S,T,Depth,IP).
```

The analyst can obtain the corresponding stack coverage through `wd:coverage/3`. A returned frame is positive evidence. A function missing from a captured stack prefix does not establish that it was absent from the complete stack. The package's memory interface likewise returns `ok(Bytes)`, `partial(Bytes,RequestedLength,Reason)`, or `unavailable(Reason)`, and its sample signature rule can return an inconclusive result.

For the document scenario, we would add reviewed predicates that decode the document and index, establish the applicable namespace and lifecycle invariant, and record whether lookup inspection completed. The existing `wd:memory/4` and `wd:lookup/3` providers supply access to bytes and symbol addresses. They do not by themselves establish object liveness or index membership. Those claims require the new application contract and its evidence records.

A neural component would submit a pattern identifier, bindings, and evidence references to that contract layer. It would not send arbitrary source text to `!prolog.query` or `!prolog.consult`. The controller validates the candidate, evaluates a reviewed rule, and returns support, counterevidence, or specific missing premises. An unresolved index inspection can then request another permitted read. This closes the proposed neural-symbolic feedback loop while preserving the extension's existing debugger boundary.

12 Neural components and learning strategy

No neural component is included in the delivered extension. The first addition would retrieve and rank patterns from an approved catalog. Its inputs would include the incident description, normalized structural summaries, and the functional profile. Its output would be a typed list of pattern identifiers, bindings, supporting evidence references, and suggested inspections. An unknown pattern identifier or invented observation identifier is rejected by schema validation.

A second component could compare cases using learned representations of stacks, object roles, and relation neighborhoods. Address normalization must remove incidental layout differences while preserving identity relationships. Renaming every pointer to an unrelated token, for example, would destroy the information that two fields reference the same object. Module versions and application families should remain available for applicability checks even if some names are masked during learning.

A third component could rank the next useful observation. Training examples would consist of analysis states, candidate sets, selected inspections, and their observed contribution to resolution. Analyst command frequency alone is an inadequate target: a frequently issued command may be convenient rather than diagnostically discriminating. The evaluation must compare a learned policy with a transparent rule-based scheduler.

The neural worker returns a ranking score with its meaning declared. The interface does not label every score as a probability. If probabilistic outputs are used, calibration is evaluated on held-out cases and under distribution shift. Correlated evidence must not be multiplied as though it consisted of independent observations. DeepProbLog-style integration would therefore require an explicit probabilistic model as well as a neural classifier [8].

Models may also propose new rules for human review. Such proposals enter a development workspace with examples, counterexamples, and intended scope. They are not installed automatically in the active registry. A rule that fits one incident may simply encode that incident's identifiers. Generalization requires independent cases, semantic review, and tests that alter irrelevant names and addresses.

13 Worked diagnostic scenarios

The following scenarios are constructed examples of the proposed reasoning process above the existing extension. They are not experimental results. In each case, the diagnosis depends on evidence and application semantics that the example explicitly supplies.

13.1 A document exists but cannot be found

Suppose a service reports that document D17 cannot be found by name. Enumeration from the same namespace and request epoch succeeds. The dump contains a live Document object with identity D17 and a valid payload. The application's reviewed invariant states that every published document in that namespace must have a corresponding entry in a particular name index.

In the proposed extension of the project, the neural ranker proposes an access-path inconsistency. The evaluator requests the object's lifecycle state, namespace identity, index root, and lookup result. The adapter establishes that the object is published, the capture is at a stable boundary for the invariant, and the lookup inspection completed across all relevant buckets and overflow chains. The result is explicitly absent.

These premises support Index Access Inconsistency for the specified capture and scope. They do not show which instruction removed or failed to insert the entry. The report can therefore distinguish a supported structural finding from competing causal hypotheses: incorrect removal or incomplete insertion. A build-matched trace or a controlled reproduction can discriminate between these hypotheses.

Now remove one required memory page from the same capture. The object may still be valid, but the lookup inspection becomes partial. The result must change to unresolved. Likewise, if the capture occurred during a legitimate publication update, the stable-boundary premise fails. The proposed system should resist the temptation to retain the original conclusion simply because the visible object and error message are unchanged.

Figure 3 shows how the relevant evidence supports a finding while leaving the initiating event open.

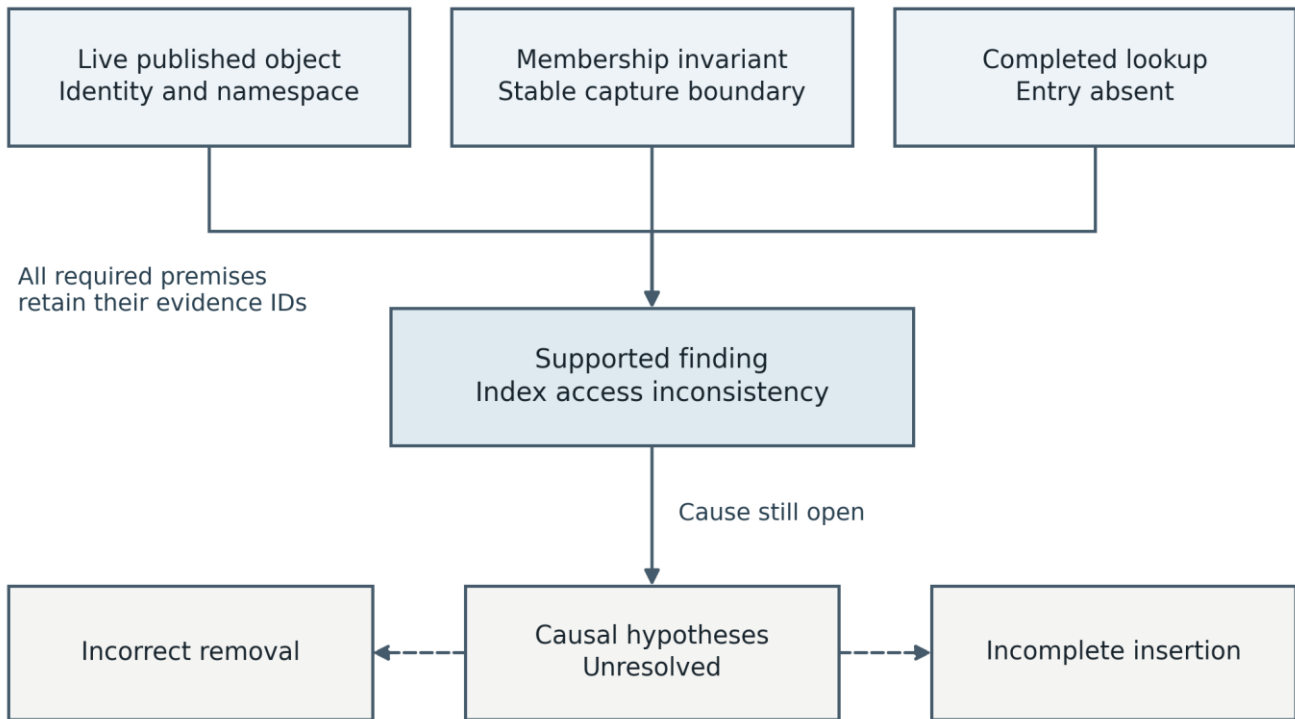


Figure 3. Evidence dependencies for the document example. A structural finding requires all stated premises. The possible initiating events remain hypotheses until additional evidence distinguishes them.

13.2 Two threads appear to wait on each other

Suppose thread T1 owns resource R1 and waits for R2, while T2 owns R2 and waits for R1. The extracted relations form a cycle. A neural component may recognize a familiar deadlock-like arrangement, but the symbolic evaluator first asks what owns and waits mean for these resources.

For ordinary single-instance exclusive locks, a closed wait cycle can support a deadlock finding when the involved threads are blocked, cannot release their held locks while blocked, and have no modeled timeout, cancellation, external release, or alternative acquisition path that breaks the cycle. The capture must support these conditions for a coherent state. If resource semantics are unknown, the finding is a wait cycle with unresolved deadlock applicability.

This distinction is necessary for events, semaphores, reader-writer locks, and application-specific synchronization. A resource may have several providers, an apparent owner field may have another meaning, or a timeout may allow progress. Even a correctly identified deadlock describes a state; the causal question about the lock-ordering decisions that produced it requires a separate argument.

The neurolinguistic contribution in this example is the attention to distributed function and disconnection. Individual threads and objects can be structurally valid while the configuration prevents progress. The symbolic contribution is the explicit resource model that determines whether the observed relations establish the proposed pattern.

13.3 A new request receives an old result

Suppose a service responds to request Q2 with a result tagged for Q1. Logs establish that Q1 and Q2 are distinct requests, while a dump associates Q2's active worker with a context object whose generation and response identifier still refer to Q1. The request contract requires those fields to be reset before the worker processes Q2.

The preserved payload might be completely valid for Q1. The defect candidate concerns context and lifecycle rather than payload corruption. The evaluator needs evidence that the worker is actually processing Q2, that the identifiers have not legitimately been reused, and that the contract applies to this execution phase. An error in logging or correlation remains an alternative until these conditions are checked.

Here the ontology distinguishes a logical request, a worker, a response, and an execution context. The evidence graph would retain separate source identifiers for the log request and the decoded context object, together with the evidence supporting their correspondence. The contract then checks that the response and active request refer to the same permitted generation. A thread carrying a request identifier in a log and a pointer found in a dump are not joined merely because their textual values resemble one another. This is the proposed application of Trace Ontology and Trace World to the scenario [22, 23].

If the conditions hold, the system can support a stale-context finding. A sequence of such events may justify discussing inappropriate persistence as an investigative analogy. One retained response buffer alone does not establish that the service repeatedly reused it. The temporal claim belongs to the trace evidence, whereas the dump supports the structural association at capture.

The three examples illustrate different relationships between structure and function. A representation can remain valid while access fails; components can remain valid while their coordination prevents progress; and a valid representation can be used in an invalid context. The architecture keeps these alternatives available instead of forcing every symptom into generic memory corruption.

14 Failure modes and operational boundaries

The diagnostic system can have defects of its own. A symbol mismatch can make ordinary bytes resemble an impossible object. An incorrect unwinder can produce a plausible stack. An index walker can stop early and report a missing entry. The first response to conflicting findings should therefore include checks on extraction and decoding, not only new hypotheses about the target.

An unsupported conclusion can also arise through information flow. A model may invent an address, treat an example from retrieved documentation as current evidence, or mistake text inside the dump for an instruction. The proposed boundary is simple: artifact contents and retrieved cases are data.

PrologWinDbg v0.2.0 accepts ground, whitelisted snapshot facts, but its query text and consulted files are intentionally executable trusted Prolog [21]. The new neural interface must therefore enforce a reviewed pattern vocabulary before dispatch. Artifact text must not install rules, change extraction policy, or authorize debugger operations. Typed outputs, provenance checks, and a fixed operation vocabulary enforce this boundary.

Memory artifacts can contain credentials, personal data, and proprietary implementation details. The default deployment should analyze them locally. If a remote neural service is used, the controller sends only the selected, policy-approved evidence view and records that view. Redaction is itself a transformation with a provenance record; it may remove evidence required by a pattern and must therefore affect coverage or applicability.

The initial diagnostic evaluation uses read-only postmortem analysis. The existing extension also targets stopped live processes, with a fresh snapshot for each query and checks for target changes [21]. It exposes no target-writing or execution-control predicates; normal WinDbg commands control execution. This already separates debugger actions from Prolog backtracking. Any future action mechanism must preserve that separation. Controlled fault injection belongs to a separate test harness, and observations from separate stops must not be combined as one instantaneous state. The proposed persistent case store must retain each observation epoch after the query-local facts are cleared.

A separate worker provides runtime recovery, not a security sandbox. Trusted Prolog rules can perform ordinary host-side operations, and rerunning a query can repeat their side effects [21]. The neural service must use the restricted contract interface described above. Resource control also affects epistemology. A solver timeout, traversal limit, unavailable model, or exhausted analysis budget is not a negative diagnostic result. Each must produce a specific incomplete status. The report should distinguish a pattern that was refuted from one that could not be evaluated.

15 Evaluation plan

15.1 Questions and baselines

The main research question is whether the proposed integration improves supported diagnosis under incomplete evidence. We would use the existing extension and reviewed rules as the starting symbolic baseline, with equalized providers and pattern coverage. We would compare a symbolic-only pattern engine, a neural analyst with the same permitted inspection tools, a modular neural-plus-symbolic pipeline, and the complete design with functional profiles and coverage contracts. A conventional debugger workflow provides a practical reference for analyst effort.

All systems should receive equivalent artifact access, source availability, and inspection budgets. A neural baseline deprived of tools would make an unfair comparison with the full system. Conversely, a symbolic baseline with substantially fewer patterns would confound architecture with knowledge coverage. Pattern and tool inventories must therefore be reported alongside results.

15.2 Cases and ground truth

The first dataset would use small applications with explicit invariants and controlled defects. It should include destroyed representations, disconnected access paths, context mismatches, genuine wait cycles, non-deadlocking cycles, stale state, and multiple simultaneous faults. Each fault-bearing case needs a corresponding healthy or benign lookalike case. Partial captures, wrong symbols, and interrupted traversals should be generated separately from application faults.

The experiment harness records the injected fault, execution conditions, independent behavioral checks, and intended capture point. That record supplies ground truth and is hidden from the diagnostic systems. Injection identifiers must be removed from filenames and incident descriptions. A fault injected into an unexecuted path does not establish that the observed failure was caused by that fault.

Real incidents can form a second dataset when causes are supported by fixes, reproductions, or independent analysis. Split cases by incident family, application, version, and time where possible. Dumps from the same failure must not be distributed across training and test sets as independent examples. Retrievable case repositories require the same separation.

15.3 Measures and ablations

Table 7 defines the principal measures. Results should include sample counts, uncertainty intervals where appropriate, and separate outcomes for complete, partial, and inconsistent evidence.

Measure	What is counted
Supported finding precision	Reported structural findings that satisfy independently checked conditions
Root cause localization	Correct top-ranked and ranked-list locations on cases with established causes
Unsupported conclusion rate	Definitive claims lacking required evidence or relying on invalid assumptions
Abstention behavior	Unresolved outputs on insufficient evidence, plus avoidable abstentions on sufficient evidence
Explanation validity	Claims whose cited premises and rule applications survive independent checking
Inspection efficiency	Time, bytes read, operations, model cost, and analyst effort per resolved case
Robustness	Change under address relocation, symbol loss, capture omission, and software version shifts

Coverage and error should be reported together. A system can obtain high precision by refusing almost every case. Similarly, a long explanation can cite many observations without establishing the conclusion. Explanation evaluation must check the derivation, including its assumptions and negative-evidence scope, rather than count citations.

The central ablations remove one distinctive element at a time: the functional profile, coverage preconditions, provenance-based invalidation, neural ranking, or feedback from symbolic evaluation. A useful functional-profile ablation holds the rest of the evidence constant but removes information about preserved operations. A coverage ablation tests whether hidden missing pages cause false absence claims. These experiments directly examine the proposed contribution. A semantic-mapping study would hold the observations fixed while varying identifier aliases and lifecycle terminology. It should test whether the analyzer preserves known equivalences, rejects unsupported identity joins, and reports incompatible meanings instead of forcing them into one state.

16 Implementation roadmap

The experimental PrologWinDbg v0.2.0 prerelease provides a working integration baseline with recorded native x64 and ARM64 CDB validation [21]. The roadmap starts by reproducing that baseline in the intended deployment and completing the remaining manual checks, then adds the diagnostic services. It retains the existing C++ bridge, local pipe protocol, and Prolog worker. Table 8 separates deployment checks from the proposed research increments.

Stage	Deliverable	Acceptance condition
1 Deployment validation	Reproduce native x64/ARM64 checks and complete manual tests	Recorded CDB baseline reproduced; interactive interruption, custom contexts, and application captures checked
2 Persistent evidence	Case store, observation identifiers, manifests, and dependency records	Query-local observations remain traceable after facts and caches are cleared
3 Functional contract	Document/index providers, functional profile, and healthy counterexamples	Explicit support, refutation, and unresolved results match controlled fixtures
4 Neural assistance	Pattern retrieval, typed candidates, and inspection feedback	Neural failure leaves deterministic checks usable and cannot introduce direct observations
5 Pattern composition	Wait and context contracts with invalidation	Premise changes reevaluate affected findings and preserve conflicts
6 Comparative study	Held-out benchmark and ablations	Diagnostic results are reproducible and compared with equally equipped baselines
7 Broader targets	Additional target families, runtimes, and batch hosts	Capability and evidence semantics remain explicit across targets

A small document service with an explicit publication invariant can provide the first additional diagnostic slice. Healthy captures and deliberately omitted index entries would exercise the representation-versus-access distinction. Further fixtures would remove an essential memory page or capture a permitted intermediate update, checking that insufficient or inapplicable evidence produces an unresolved result. The report should preserve the exact observations needed to reopen the finding in the debugger.

Before extending the scope, the prototype should pass several concrete checks. Removing an essential index page must prevent an absence-based conclusion. Changing an irrelevant address layout must not change the semantic finding. Supplying a wrong type layout must trigger a decoding or applicability problem. A model suggestion that names nonexistent evidence must be rejected. Contradictory observations must be retained for investigation rather than resolved by arbitrary arrival order.

These are proposed acceptance criteria for the added diagnostic capabilities. The 99 checks per native architecture and CDB dump/live smoke tests recorded for PrologWinDbg v0.2.0 cover the extension's bridge, rules, and worker behavior; they do not validate these new contracts or establish improved diagnostic accuracy. The prototype should publish its case fixtures, pattern versions, and evaluation scripts so that later revisions can be compared without relying on a narrative account of improvement.

17 Limitations

The neurolinguistic vocabulary is useful only when it changes diagnostic practice in a testable way. Some proposed distinctions are already familiar under other names in debugging, program analysis, and database consistency checking. Their value here would come from systematic organization and operational integration. An experiment may show that ordinary terminology works equally well.

The proposed evidence model also has costs. Application invariants require maintenance. Capture consistency can be difficult to establish. Provenance graphs can grow large, and apparently independent observations may share hidden dependencies. Unknown results may remain common for small dumps, stripped binaries, or unsupported runtimes. These limitations should be measured rather than concealed by more confident language generation.

18 Further directions

The implementation roadmap establishes the first diagnostic system. Further work should investigate which parts of its representation and reasoning can become reusable across incidents, applications, and diagnostic tools. We would organize this work around questions that the initial prototype cannot yet answer: whether functional profiles improve pattern discovery, whether additional artifacts discriminate causes, and whether learned inspection remains useful when evidence is incomplete. The following directions extend the proposed architecture; they are not capabilities claimed for the current experimental prerelease of PrologWinDbg or results already demonstrated by this article.

18.1 A catalog of functional failure profiles

We would develop a catalog in which representation, access, coordination, context, and output are explicit dimensions of a diagnostic profile. Each entry would identify the operation under investigation, preserved operations, failed operations, and the evidence required to distinguish their possible mechanisms. A document that can be enumerated but cannot be opened would be compared with a document whose bytes can be read but whose format cannot be interpreted. Similar error messages would not make these profiles equivalent. The catalog would connect the functional description to existing analysis patterns and application contracts through explicit ontology terms. A profile would identify whether interpretation failed at vocabulary recognition, entity identification, role assignment, or contextual association, as well as which target operation failed.

A useful study would hold the underlying defect constant while varying its visible symptoms, then hold the symptom constant while varying the defect. We would compare profile-guided analysis with ordinary pattern retrieval under the same evidence budget. The relevant outcome is whether the profile selects a more discriminating inspection or prevents a false conclusion. New terminology alone would not establish a contribution. Profiles that repeatedly require the same missing evidence could also identify where a new debugger provider or application instrumentation would be useful.

18.2 Reasoning across captures and traces

Trace World provides a foundation for this direction because it treats several trace discourses as contributions to a shared diagnostic model [23]. We would extend that model to include dumps and debugger stops through the mappings described in Section 8.2. A persistent case store would relate the observations without treating them as one instantaneous state. Each capture would remain an observation epoch. Object correspondence would require a stable application identifier, an allocation generation, or another justified mapping. A reused address would be a candidate correspondence, not

proof of continuity. Clock disagreement, missing intervals, and capture uncertainty would remain attached to the temporal relations.

This would extend the stale-context scenario from a structural finding to an investigation of change. A trace might record publication, cancellation, or reuse, while successive captures show which references survived. The system would test whether these observations distinguish competing histories and retain alternatives where they do not. We would evaluate it on controlled executions with missing events and deliberately ambiguous identities. A useful result would state which causal hypotheses the combined evidence excludes and which additional observation would separate those that remain. We would also vary source vocabularies and mapping versions, including cases where cancellation acknowledgment and cancellation completion share a label. Evaluation would check whether the shared model preserves that ambiguity and whether corrections invalidate the affected conclusions without erasing the original evidence.

18.3 Learning which evidence to inspect

The next inspection should be judged by what it can resolve. We would extend the controller to compare the possible outcomes of permitted reads and traversals against the unresolved premises of competing patterns. An index-page read may distinguish a missing entry from incomplete capture; reading another unrelated stack may add little. Inspection cost would include debugger time, bytes transferred, symbolic evaluation, and model use. Where outcome probabilities are not calibrated, we would use an explicit ranking or a bounded comparison of possible outcomes rather than present an information-gain estimate as a measured quantity.

A learned policy could then be compared with the deterministic scheduler under identical budgets. Evaluation should include cases in which the best action is to stop, request a different artifact, or ask the analyst for an application invariant. Later experiments could use differentiable pattern evaluation to learn how noisy indicators contribute to candidate ranking, building on the approaches discussed earlier [8, 9]. Such scores would remain distinct from the supported, refuted, unresolved, and conflicted states produced by the evidence contracts.

18.4 Checkable rules and counterfactual models

The published book's Prolog verifier and the extension's bounded query interface provide foundations for further work on checkable diagnostic rules [20, 21]. We would investigate restricted rule forms whose required observations, negation scope, and possible outcomes can be examined before deployment. A rule package would include its assumptions, positive and negative fixtures, and cases that must remain unresolved. Generated rules would enter this package as proposals for review. A successful example would establish behavior on that example; it would not authorize a general claim about all dumps.

Selected contracts could also be exported to a constraint solver or proof checker. The immediate question would be whether a conclusion follows from the stated encoding and premises. This would not independently establish that a decoder read the target correctly. Counterfactual experiments could vary one modeled relation, such as index membership or resource ownership, and ask whether the finding still follows. Those model changes would remain separate from target-memory writes. Controlled executions would be needed to test whether the same intervention changes the application's behavior and supports a causal explanation.

18.5 Semantic transfer across target environments

The existing native x64 and ARM64 support offers a practical starting point for studying which diagnostic relations survive a change of architecture [21]. We would use equivalent application fixtures on both targets and compare normalized findings, evidence dependencies, and unresolved conditions. Equality would be required only where the application contract and observation coverage are equivalent. Differences in unwind results or captured data would be reported as differences in evidence, rather than normalized away to obtain matching diagnoses.

Further adapters could address other dump formats, operating systems, or managed runtimes. The reusable unit would be a pattern contract over defined relations, accompanied by explicit adapter capabilities. A runtime with moving objects or a debugger with different thread identifiers would require its own identity mapping. We would measure how much of a contract transfers unchanged, which providers require replacement, and where the original semantics cease to apply. This would test portability of diagnostic knowledge independently of portability of the WinDbg extension.

18.6 Diagnostics of the analyzer and analyst collaboration

The same representation-and-access questions can be applied to the analyzer. An observation may exist in the case store but be omitted from retrieval; a correct rule may be inapplicable because its required provider was not selected; an explanation may retain a conclusion after its premise was invalidated. We would inject such defects into the analysis pipeline and compare the failure profiles of extraction, neural ranking, symbolic evaluation, and reporting. A second model could critique a proposed explanation, but agreement between models would remain a review signal rather than independent evidence of the target's state.

The analyst interface should expose the premise whose resolution would change the diagnosis and permit that premise to be reopened in the debugger. Human corrections would be recorded with their origin and would trigger reevaluation of dependent findings. Evaluation could measure the time needed to identify a misleading explanation, recover the relevant observation, and correct the case. A reproducible collection of these failures, alongside successful analyses and unresolved cases, would support both training and regression checks. The longer-term objective is an analyzer whose conclusions and limitations can themselves be diagnosed.

19 Conclusion

A memory dump can preserve information that the application could no longer reach, combine structures that could no longer cooperate, or retain a valid result beyond the context in which it was valid. These situations invite a diagnostic vocabulary that separates representation, access, context, and function.

Here we have proposed connecting that vocabulary to a neurosymbolic specialization of Analysis Pattern Networks built on the published PrologWinDbg project and the diagnostic foundations developed in our Prolog book [20, 21]. Trace Ontology supplies a semantic foundation, while the proposed extension of Trace World relates evidence from dumps and traces. Neural components suggest patterns and useful observations. Symbolic components evaluate explicit conditions. Coverage records restrict conclusions about absence. Provenance connects each finding to its evidence and assumptions. The resulting explanation can state both what the capture supports and what remains to be established.

The next step is to reproduce the existing extension's validation in the intended deployment, complete the remaining manual checks, and add a small number of carefully specified functional contracts with controlled examples. Its success should be judged by the quality of its supported findings, its handling of uncertainty, and its ability to guide further investigation. The proposed architecture makes these questions concrete enough to test.

Appendix A Example finding record

The following illustrative record shows a supported structural finding whose cause remains unresolved. Identifiers refer to the constructed document scenario. A production record would also include the artifact digest, full premise records, decoding context, and the run manifest.

```
{
  "schema_version": "1.0",
  "case_id": "example-document-17",
  "evidence_revision": 4,
  "pattern_id": "experimental.index-access-inconsistency",
  "pattern_version": "0.1",
  "ontology_version": "document-domain-0.1",
  "mapping_version": "document-adapter-0.1",
  "bindings": {
    "capture": "capture-1",
    "object": "document-17",
    "key": "D17",
    "scope": "namespace-A"
  },
  "evidence_state": "supported",
  "premises": ["obs-11", "obs-12", "obs-18", "obs-21"],
  "coverage_record": "coverage-7",
  "invariant": "published-document-membership-v1",
  "stable_boundary_evidence": "obs-22",
  "finding": "Required name-index membership is absent",
  "causal_status": "unresolved",
  "next_question": "Which update changed index membership?"
}
```

The controller must verify that all identifiers resolve within the declared evidence revision. Display text is generated from the structured result; editing that text does not change the underlying finding. A later revision can supersede the finding while preserving this record for audit and comparison.

References

The sources below establish the cited foundations and related work. The functional contracts, persistent provenance services, neural integration, scenarios, and comparative evaluation in this article are proposals. The existing project baseline and its validation limits are described in Section 11 and reference [21]. Sources [1–19] were checked for the earlier article on 9 October 2026; sources [20–25] were consulted on 21 October 2026. References [15], [17], [20], and [21] were rechecked on 22 October 2026. The published book in reference [20] and the release and implementation documentation in reference [21] were checked again on 26 October 2026; the book reference now identifies the public Version 20. Reference [17] includes the authors' university record. Reference [18] is an accessible manuscript; its reported scope is distinguished from its title in Section 5.

- [1] Vostokov, D. Theoretical Software Diagnostics: Collected Articles. Fourth Edition. OpenTask, 2024. ISBN 978-1-912636-91-4. Author's reference copy, revision 4.01. Contents: <https://www.dumpanalysis.org/TSD/Theoretical-Software-Diagnostics-Fourth-Edition-TOC.pdf>
- [2] Vostokov, D. Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection. Fifth Edition. OpenTask, 2023; author's reference copy, revision 5.06, August 2026. ISBN 978-1-912636-58-7.
- [3] Vostokov, D. General Architecture of Analysis Pattern Networks. Reprinted in Theoretical Software Diagnostics, Fourth Edition, pp. 343-347. <https://www.dumpanalysis.org/analysis-pattern-networks>
- [4] Vostokov, D. Software Narratology Square. Software Diagnostics Library, 11 March 2012. <https://www.dumpanalysis.org/blog/index.php/2012/03/11/software-narratology-square/>
- [5] Bates, E., Wilson, S. M., Saygin, A. P., Dick, F., Sereno, M. I., Knight, R. T., and Dronkers, N. F. Voxel-based lesion-symptom mapping. *Nature Neuroscience*, 6, 448-450, 2003. <https://doi.org/10.1038/nn1050>
- [6] Plaut, D. C., and Shallice, T. Deep dyslexia: A case study of connectionist neuropsychology. *Cognitive Neuropsychology*, 10(5), 377-500, 1993. <https://doi.org/10.1080/02643299308253469>
- [7] Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1), 57-95, 1987. [https://doi.org/10.1016/0004-3702\(87\)90062-2](https://doi.org/10.1016/0004-3702(87)90062-2)
- [8] Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., and De Raedt, L. DeepProbLog: Neural Probabilistic Logic Programming. *Advances in Neural Information Processing Systems* 31, 2018. <https://proceedings.neurips.cc/paper/2018/hash/dc5d637ed5e62c36ecb73b654b05ba2a-Abstract.html>
- [9] Li, Z., Huang, J., and Naik, M. Scallop: A Language for Neurosymbolic Programming. *Proceedings of the ACM on Programming Languages*, 7, PLDI, Article 166, 2023. <https://doi.org/10.1145/3591280>
- [10] Kang, S., Yun, S., Hong, J., Yoo, S., and An, G. Finding the Needle in the Crash Stack: Industrial-Scale Crash Root Cause Localization with AutoCrashFL. *arXiv:2510.22530v1*, 2025. <https://arxiv.org/abs/2510.22530v1>
- [11] Huang, H., Shi, J., Wang, B., Yang, Z., and Lo, D. Finding Memory Leaks in C/C++ Programs via Neuro-Symbolic Augmented Static Analysis. *arXiv:2603.27224v5*, 6 October 2026. <https://arxiv.org/abs/2603.27224v5>
- [12] Ottmann, J., Cengiz, Ü., Breiting, F., and Freiling, F. As if Time Had Stopped — Checking Memory Dumps for Quasi-Instantaneous Consistency. *arXiv:2307.12060*, 2023. <https://arxiv.org/abs/2307.12060>

- [13] Microsoft. User-mode Dump Files. Microsoft Learn. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/user-mode-dump-files>
- [14] Microsoft. Debugger Engine Overview. Microsoft Learn. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-engine-overview>
- [15] Microsoft. Debugger Data Model C++ Interfaces Overview. Microsoft Learn. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/data-model-cpp-overview>
- [16] Cui, W., Peinado, M., Cha, S. K., Fratantonio, Y., and Kemerlis, V. P. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. Proceedings of ICSE, 2016. <https://doi.org/10.1145/2884781.2884844>
- [17] Wu, M.-C., Chang, J.-A., and Wu, Y.-S. Poster: High-Fidelity and Contextual User Activity Memory Forensics. System: RAM-Weaver. Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, pp. 4818-4820, 2025. <https://doi.org/10.1145/3719027.3760737> Abstract and bibliographic record: <https://scholar.nycu.edu.tw/en/publications/poster-high-fidelity-and-contextual-user-activity-memory-forensic/> Accessed 22 October 2026.
- [18] Li, Y. AI-Augmented Automated Triage for System Error Reporting Using Log and Memory Dump Analysis. Accessible manuscript, June 2026. The manuscript byline identifies Yuxuan Li. https://www.researchgate.net/publication/406318975_AI-Augmented_Automated_Triage_for_System_Error_Reporting_Using_Log_and_Memory_Dump_Analysis
- [19] SWI-Prolog. Foreign Language Interface. SWI-Prolog Reference Manual. <https://www.swi-prolog.org/pldoc/man?section=foreign>
- [20] Vostokov, D. Prolog for Pattern-Oriented Software Diagnostics: Logic Programming for Memory Dump, Trace, and Log Analysis. OpenTask, 2026. ISBN 978-1-919135-06-9. Published PDF, Version 20, October 2026. Sections 31.6, A Debugger Extension: A Design; 31.7, PrologWinDbg: The Extension as Built; and 31.9, Prolog as a Verifier for LLM-Based Agents. Book page: <https://www.dumpanalysis.org/prolog-for-pattern-oriented-diagnostics>. [Public PDF \(Version 20\)](#). Accessed 26 October 2026.
- [21] Vostokov, D. PrologWinDbg: A WinDbg Extension with a Persistent SWI-Prolog Worker. GitHub repository, DumpAnalysis/PrologWinDbg. Experimental prerelease v0.2.0, 20 October 2026. <https://github.com/DumpAnalysis/PrologWinDbg>. Implementation scope and recorded validation: README.md, docs/API.md, docs/PROTOCOL.md, and docs/VALIDATION.md. Release: <https://github.com/DumpAnalysis/PrologWinDbg/releases/tag/v0.2.0>. Accessed 21 October 2026; rechecked 26 October 2026.
- [22] Vostokov, D. Trace Ontology. Trace Analysis Patterns (Part 263). Software Diagnostics Library, 6 July 2026. <https://www.dumpanalysis.org/blog/index.php/2026/07/06/trace-analysis-patterns-part-263/>
- [23] Vostokov, D. Trace World. Trace Analysis Patterns (Part 264). Software Diagnostics Library, 8 July 2026. <https://www.dumpanalysis.org/blog/index.php/2026/07/08/trace-analysis-patterns-part-264/>
- [24] Vostokov, D. Tesing [sic] Python WinDbg Extension. Software Diagnostics Library, 5 February 2011. <https://www.dumpanalysis.org/blog/index.php/2011/02/05/tesing-python-windbg-extension/>
- [25] Vostokov, D. Windows Debugging Expert System WinDbg Extension. Software Diagnostics Library, 5 February 2011. <https://www.dumpanalysis.org/blog/index.php/2011/02/05/windows-debugging-expert-system-windbg-extension/>