WHITE PAPER

Thoughtware

A Category-defining Theory of Cognitive Software for the Intelligence Age, to build for reliability and scale.

Achin Kumar
Founder, Session Hero
Co-founder, Roro.

thoughtware.

Thoughtware: A Theory of Cognitive Software for the Intelligence Age

Defining the architecture, primitives and engineering principles of software built around judgment.

Achin Kumar
Founder, Session Hero · Co-founder, Roro
Roro Solutions

August 2026

WORKING PAPER · VERSION 1.0

CONTENTS

PART III · ENGINEERING

PART IV · CONSEQUENCES

FIGURES

ABSTRACT

A Theory of Cognitive Software for the Intelligence Age

Software has historically been built by translating human intent into explicit procedures. Its behaviour has largely depended on rules, algorithms, and deterministic logic specified in advance by human designers and engineers. Modern foundation models introduce a different capability: software can now perform forms of cognitive work at runtime, including interpretation, reasoning, evaluation, recommendation, and judgment. In this paper, judgment refers to the selection or construction of an appropriate conclusion or action where the correct result cannot be fully determined in advance by a fixed procedure over the available inputs.

This shift has produced a rapidly expanding implementation vocabulary. Terms such as large language model application, retrieval-augmented generation, agent, copilot, and agentic system describe important technologies, interaction patterns, and architectural mechanisms, but they do not adequately name the broader class of software that emerges when cognitive capability becomes a structural part of the system. This paper proposes Thoughtware as that category: cognitive software in which capabilities such as interpretation, reasoning, judgment, and evaluation are deliberately organised as first-class elements of the architecture.

The paper develops a conceptual framework for reasoning about such systems beyond specific models, vendors, and orchestration techniques. It examines how judgment can be bounded and located within software, how cognitive and deterministic responsibilities can coexist, how cognitive behaviour can be composed into larger systems, and how systems whose outputs cannot always be verified deterministically can instead be evaluated, governed, and improved.

The resulting theory provides a vocabulary, architectural model, set of primitives, and engineering principles for software whose behaviour depends materially on machine judgment.

PART I

The Shift

Why the arrival of runtime judgment is an architectural change rather than a feature, and why the software it produces still has no durable name.

SECTIONS 1 — 3

SECTION 1

From Computation to Cognition

Software has always been capable of producing outcomes that appear intelligent from the outside. A navigation system can find a route through a city, a banking platform can detect whether a transaction violates a rule, and an enterprise system can determine which workflow should follow a particular event. Yet in most traditional software, the intelligence behind those outcomes was supplied before the software ran. Human designers and engineers determined which conditions mattered, which rules applied, how exceptions should be handled, and how outputs should be produced. The software executed those decisions at runtime, but it did not usually make them in any meaningful sense.

This distinction is fundamental. In conventional software, much of the judgment is embedded during construction. A developer may encode a threshold, define a decision tree, construct a ranking formula, or specify a sequence of conditions. Even sophisticated systems often operate this way. Their behaviour may be complex, adaptive, or statistically informed, but the space of possible responses is substantially shaped in advance by the structure humans have given them. Runtime execution is therefore, to a large extent, the replaying of decisions already made during design and implementation.

Machine learning altered this model by allowing some behaviour to be learned from data rather than explicitly programmed. Predictive systems could estimate risk, classify images, rank recommendations, or detect anomalies without a developer writing every individual rule. This was an important shift, but the role of such models was typically narrow and predefined. A classifier could determine which category an input belonged to, for example, but the surrounding software still had to decide what that classification meant, what should happen next, and how the result should be combined with other information.

Foundation models extend this shift considerably. They make a broader range of cognitive capabilities available to software through a common computational substrate. A system can now be given an ambiguous request and infer what the user is trying to accomplish. It can compare conflicting evidence, interpret an unfamiliar situation, critique a proposed answer, recommend an action, distinguish relevant information from incidental detail, or weigh competing considerations before reaching a conclusion. These behaviours are not limited to one predefined prediction task. They allow software to participate in portions of the decision process that previously had to be encoded procedurally or delegated to a human.

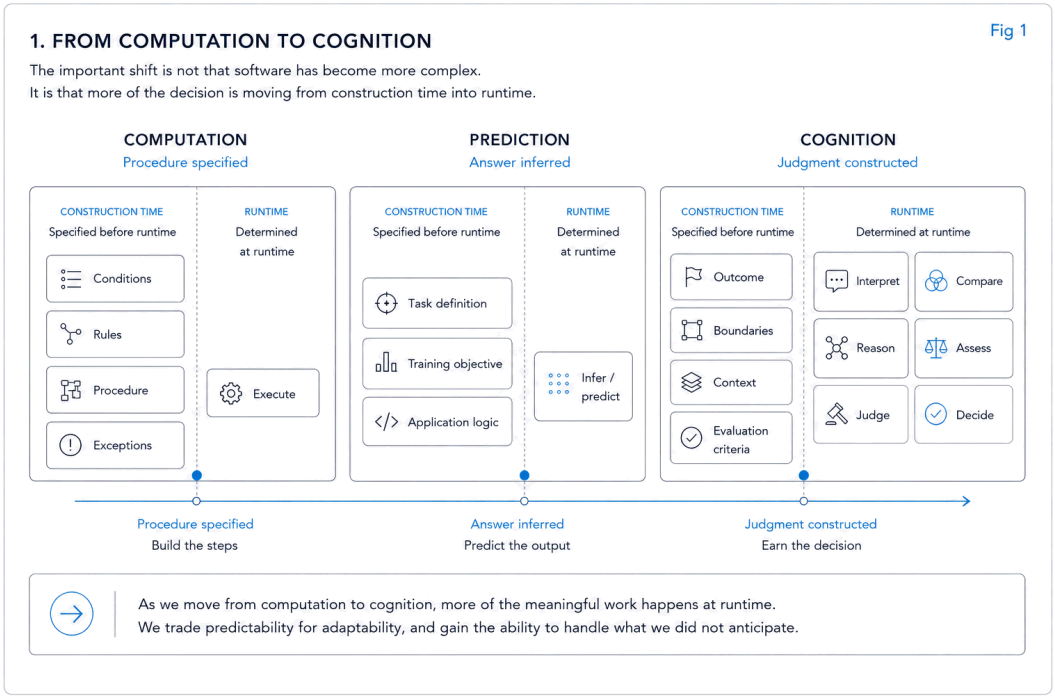
The important claim is not that software now thinks as humans do. That is a much larger philosophical and scientific question, and it is unnecessary to resolve it in order to understand the architectural change taking place. The more practical observation is that software can now perform classes of cognitive work that were previously difficult to express as conventional program logic. What matters for software design is the capability itself: interpretation, reasoning, evaluation, and judgment can increasingly occur inside the running system.

This changes the location of decision-making. Consider a customer complaint. Traditional software might search for particular categories, ask the user to choose from a predefined list, and then route the case according to a set of rules. A cognitive system can instead receive the complaint in natural language, determine what happened, assess its seriousness, compare it with policy and previous cases, and recommend an appropriate response. The developer no longer needs to enumerate every possible wording, circumstance, and combination of factors in advance. Part of the decision is deferred until the system encounters the actual situation.

The concept of judgment is therefore central to this paper. Here, judgment is used operationally rather than philosophically. It refers to the selection or construction of an appropriate conclusion or action where the correct result cannot be completely determined in advance by a fixed procedure over the available inputs. Some judgments are simple and low consequence. Others involve ambiguity, incomplete information, conflicting objectives, or meaningful consequences. What they share is that specifying the desired outcome is easier than exhaustively specifying the procedure that will always produce it.

This gives software a new kind of implementation material. Computation allows us to express procedures. Storage allows us to preserve state. Networks allow

systems to exchange information. Foundation models increasingly allow us to delegate bounded forms of judgment. The significance of the Intelligence Age is therefore not merely that models have become more capable, nor that software can generate text, images, or code. It is that judgment itself can now participate in the construction of software.



That change creates a new architectural problem. Once judgment exists inside a system, it cannot be treated as an undifferentiated layer of "AI." We need to decide which judgments should be delegated, where each judgment belongs, what information it may use, how its responsibility is bounded, how it interacts with deterministic software, whether it can be reused, and how its quality can be evaluated. Conventional software architecture was not designed around these questions because conventional software did not routinely contain this kind of runtime cognitive responsibility.

A new theory becomes necessary not because software has ceased to compute, but because computation is no longer the only kind of work being organised within it.

SECTION 2

The Missing Category

The emergence of cognitive capability in software has produced an equally rapid expansion of vocabulary. We speak of LLM applications, AI applications, copilots, chatbots, retrieval-augmented generation, agents, agentic systems, workflows, multimodal systems, and increasingly complex combinations of these ideas. Each term is useful within its own scope. Some describe a model or technical mechanism. Others describe an interaction pattern, an orchestration strategy, or a degree of autonomy. Together, they give us a language for discussing how contemporary AI systems are implemented.

> **What they do not yet give us is a durable name for the class of software being created.**

This distinction matters because implementation vocabulary and category vocabulary serve different purposes. HTTP is fundamental to the web, but we do not define a web application as an HTTP application. A product that relies extensively on a relational database is not normally understood as a SQL application. Cloud systems may depend on containers, orchestration platforms, distributed storage, and service meshes, yet “container software” would tell us very little about the nature of the system itself. These technologies remain important, but they sit beneath a more stable description of what has been built.

Emerging technologies tend to make their infrastructure unusually visible. When a capability is new, both builders and users pay close attention to its mechanisms. The mechanism becomes part of the product vocabulary because it is novel, differentiating, and still poorly abstracted. Over time, successful infrastructure becomes less conspicuous. It does not become less important. It becomes

sufficiently normal that it can disappear beneath the conceptual surface of the product.

Much of the current language of AI reflects this transitional moment. “RAG application” identifies a retrieval technique. “Agent” identifies a form of goal-directed orchestration. “Copilot” usually describes a relationship between a user and an assisting system. “Chatbot” describes an interface pattern. “Multimodal” describes the kinds of information a system can process. Even “LLM application” defines software by the model family currently responsible for some of its capability. None of these terms necessarily survives a change in the mechanism beneath it.

A useful category should survive such changes. It should continue to describe the system if one model is replaced by another, if retrieval techniques evolve, if prompts become automatically generated, if an agent framework disappears, or if the product stops presenting itself through a conversational interface. It should describe what has become structurally different about the software rather than which generation of AI infrastructure happens to implement it.

This paper proposes Thoughtware as that category.

> **Thoughtware is software in which cognitive capabilities such as interpretation, reasoning, judgment, evaluation, and adaptation are organised as first-class components of the system.**

The emphasis is on organisation. Thoughtware is not simply software that happens to call a model. It is software whose architecture assigns meaningful responsibilities to cognitive capability. Some part of the system is expected not merely to execute a predetermined procedure, but to interpret a situation, form a judgment, evaluate possibilities, or adapt its behaviour within defined boundaries. These responsibilities become part of how the software is decomposed, specified, tested, governed, and improved.

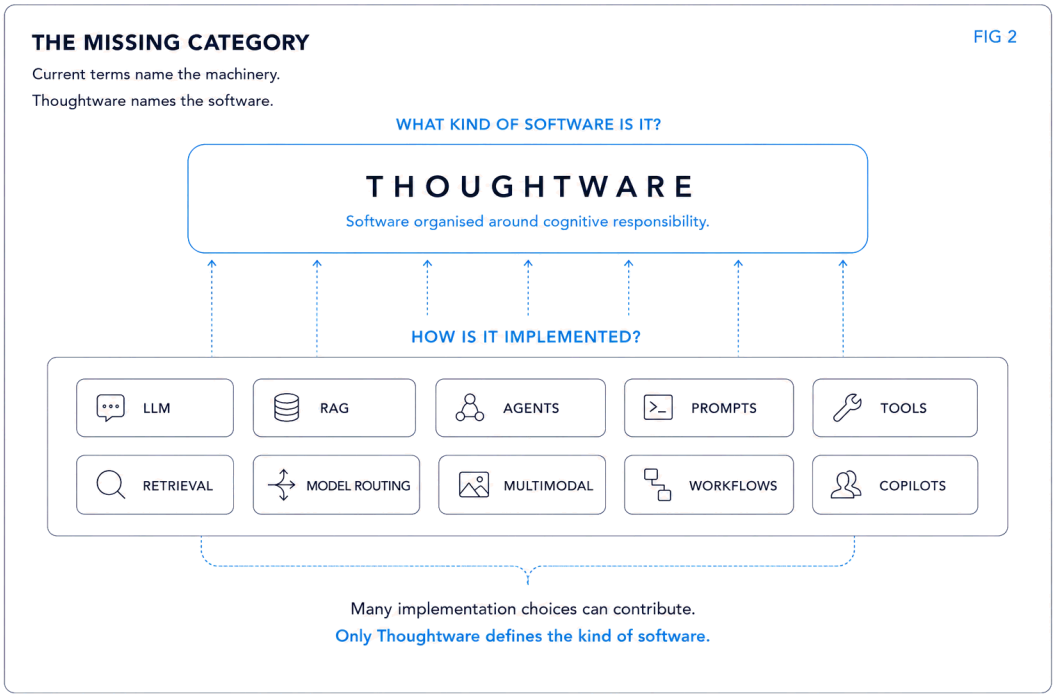
This boundary is important because the presence of an LLM alone is not enough. A deterministic product that generates a friendly greeting with a language model has not necessarily become Thoughtware. Neither has a static summarisation button, a thin API wrapper around a model, or a chatbot added to the edge of an

otherwise unchanged application. Such features may use sophisticated models, but the cognitive capability remains incidental to the architecture. The underlying system could often remove that capability without changing how its primary responsibilities are organised.

Thoughtware begins to emerge when cognition becomes structurally consequential. A practical qualification test is therefore useful. A system increasingly belongs to this category when meaningful judgment is delegated to it, when cognitive responsibilities are deliberately located within the architecture, when the inputs and outputs of those responsibilities can be identified, when their behaviour is evaluated rather than merely assumed to be satisfactory, and when those capabilities can evolve without requiring the surrounding product to be conventionally rewritten each time.

These criteria are intentionally architectural rather than technological. They do not require a particular model, framework, interface, or deployment pattern. A Thoughtware system might contain language models, vision models, specialist models, deterministic algorithms, conventional services, human approvals, and tools that have no cognitive behaviour at all. The defining question is not which technologies are present. It is whether cognitive responsibility has become an organised part of the software.

This leads to the central proposition of the category: Thoughtware is an architectural category, not a model category. Models provide capability, but they do not by themselves determine the structure of the resulting system. The architectural task begins when that capability must be divided into responsibilities, combined with deterministic mechanisms, constrained by authority, observed, evaluated, reused, and improved.



Naming the category therefore does more than provide a convenient label. It changes the level at which the problem can be discussed. Instead of asking only which model, retrieval method, prompting technique, or agent framework should be used, we can ask what kinds of cognitive responsibility the software should contain and how those responsibilities should be organised.

Having named the category, the next question is more fundamental: what, exactly, is the new material being organised inside it?

SECTION 3

Judgment as a Software Material

Software architecture is, in part, the discipline of deciding how different kinds of capability should be organised. Computation is placed into functions and services. State is given persistence and lifecycle. Storage is separated according to access patterns and durability requirements. Messaging connects components that should not depend on synchronous execution. Permissions define who or what may act. Transactions establish boundaries within which changes must remain consistent. Interfaces determine how capabilities are exposed to people and other systems. Each of these is more than a technology. It is a material from which software systems are composed.

The emerging material of cognitive software is not “AI” in the abstract. AI is too broad a description to provide useful architectural guidance. The more consequential capability is the ability to delegate bounded cognitive responsibilities to the running system. Software can increasingly be asked not only to execute a procedure, but to determine an appropriate conclusion or action when that procedure cannot be exhaustively specified in advance. In this sense, judgment becomes something architecture must learn to place, constrain, combine, and evaluate.

The distinction can be expressed simply. A deterministic component is given an input and a procedure: given X, execute specified procedure Y. If the specification is complete and the implementation is correct, the expected behaviour follows from that procedure. A cognitive component receives something different: an input together with a specification of what a good result should achieve, and it must determine the result. Its contract is closer to: given X and a specification of what good judgment means, determine Y.

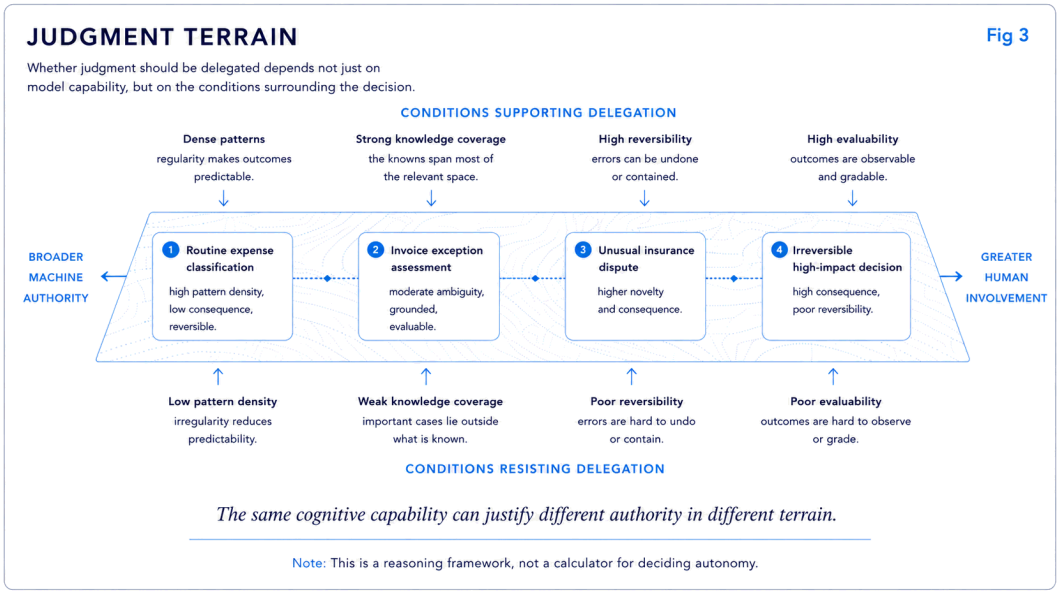
This does not create a clean boundary between “non-intelligent” and “intelligent” software. Cognitive responsibility exists across a spectrum. Consider a sequence of common business decisions. Calculating VAT is predominantly computational: the relevant values and rules can usually be specified precisely. Assigning an expense to a category may involve classification, especially when descriptions are inconsistent. Determining what a customer complaint is actually about requires interpretation. Proposing an appropriate remediation introduces recommendation and the balancing of several considerations. Deciding whether the remediation can be executed automatically or requires human escalation introduces consequential judgment involving policy, uncertainty, authority, and risk.

The important change across this sequence is not simply an increase in complexity. It is a gradual reduction in our ability to prescribe the complete path from input to correct output. Where the desired result can be reliably expressed as a procedure, deterministic implementation remains preferable. Where the problem requires interpretation of incomplete, variable, or contextual information, the specification increasingly describes the quality and boundaries of the decision rather than every operation required to produce it.

This distinction also prevents an unhelpful architectural tendency: treating every opportunity for model use as an opportunity for cognitive delegation. Judgment is not valuable merely because it is possible. A language model can be asked to perform arithmetic, generate identifiers, decide access permissions, or reproduce rules that conventional software can execute exactly. Doing so may add variability where none is needed. Cognitive capability becomes useful when the problem itself contains meaningful uncertainty, interpretation, or contextual choice. Thoughtware therefore depends as much on knowing where not to use judgment as on knowing where to introduce it.

The suitability of a judgment for delegation also varies with the terrain in which that judgment occurs. Several factors are especially important. Pattern density describes how strongly the situation resembles cases for which useful patterns are available. Knowledge coverage concerns whether the system has access to the information necessary to reach a sound conclusion. Novelty captures how far a situation departs from familiar cases. Consequence describes the cost of a poor decision. Reversibility asks whether an incorrect action can be easily undone. Evaluability concerns how readily the quality of a decision can be assessed.

Together, these factors form what this paper calls the Judgment Terrain. They do not produce a universal formula for automation, but they provide a more useful way to reason about delegation than a binary question of whether a model is capable of performing a task. A highly familiar, well-grounded, reversible, and easily evaluated judgment may be suitable for substantial machine authority even if some uncertainty remains. A novel, poorly grounded, irreversible judgment with severe consequences may require human involvement despite impressive model performance. The same underlying cognitive capability can therefore justify very different levels of autonomy depending on the conditions surrounding its use.



This is why capability and authority must remain separate architectural concerns. A system may be capable of recommending a refund without being authorised to issue it. It may be capable of interpreting a medical document without being permitted to make a consequential decision from that interpretation. It may assess whether evidence supports an insurance claim while remaining required to escalate cases above a defined threshold. Delegating judgment does not imply delegating unlimited action.

Treating judgment as a software material therefore changes the central design question. The question is no longer simply whether AI should be used. It is which

judgments should be delegated, where those judgments should reside, what information they should be allowed to use, how much authority they should possess, and how their quality will be established.

Once judgment becomes an architectural material, it requires something software architecture has always demanded of important responsibilities: locality. A judgment needs a place where its purpose, inputs, outputs, boundaries, and quality can be made explicit. That need leads to the smallest structural primitive of Thoughtware: the Cognitive Unit.

PART II

The Architecture

The structural levels of Thoughtware: where a judgment lives, what owns a goal, and how experience becomes reusable capability.

SECTIONS 4 — 7

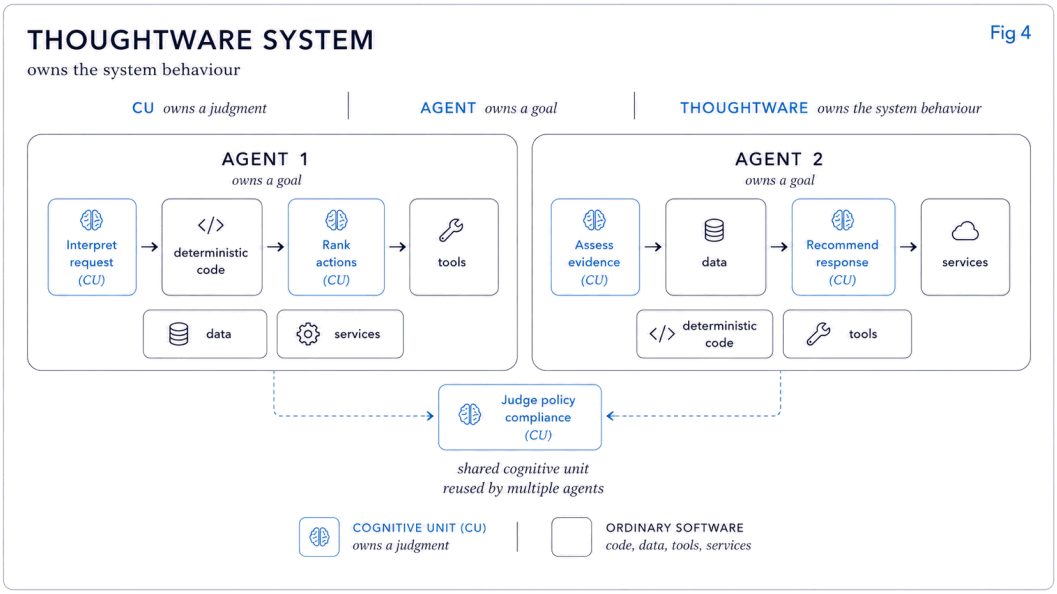
SECTION 4

The Architecture of Thoughtware

If judgment is to become a first-class material of software, it cannot remain concentrated inside a single undifferentiated layer labelled “AI.” It must be decomposed, assigned, composed, and governed in much the same way that software architecture already decomposes computation, data, interfaces, and services. Thoughtware therefore requires structural levels that make cognitive responsibility explicit.

At the highest level, the architecture can be understood through three distinct constructs: Cognitive Unit, Agent, and Thoughtware System.

These should not be interpreted merely as increasingly large containers. Their importance lies in the different responsibilities they own. A Cognitive Unit owns a bounded judgment. An Agent owns progress toward a goal. A Thoughtware System owns the complete behaviour of the product or operational system in which cognition participates. The three levels may be implemented in many different ways, and their internal technologies will change over time, but the separation of responsibility remains useful because it gives cognitive capability a stable architectural form.



4.1 Cognitive Unit

The Cognitive Unit is the smallest architectural unit of Thoughtware. It is responsible for one coherent cognitive decision or judgment.

A Cognitive Unit might determine whether an invoice discrepancy is material, identify the underlying intent in a user's request, judge whether available evidence supports a claim, assess the relevance of a document, or rank several candidate actions according to a defined objective. Each of these tasks may require interpretation or reasoning, but each can still be named as a bounded responsibility.

This boundedness matters. If all cognitive work is delegated to a single general-purpose model call or broad agent, the architecture loses the ability to reason clearly about what is being decided. Failures become harder to locate, capabilities become difficult to reuse, and evaluation becomes attached to the entire system rather than to specific responsibilities. By contrast, when a judgment has a defined architectural home, its inputs, outputs, context, expected behaviour, and evaluation criteria can be made explicit.

The Cognitive Unit is therefore not defined by how many model calls it contains, which model it uses, or whether its internal implementation is simple or

sophisticated. Its defining property is that it owns a coherent cognitive responsibility. The implementation may evolve while the architectural responsibility remains stable.

4.2 Agent

A single judgment rarely accomplishes a meaningful goal by itself. Real work often requires a sequence of decisions, actions, observations, and adjustments. This is the responsibility of the Agent.

An Agent is a goal-oriented component that can determine how to proceed toward an outcome. It may invoke Cognitive Units, execute deterministic code, retrieve knowledge, use tools, consult memory, inspect intermediate results, and decide whether further work is necessary. Unlike a Cognitive Unit, which owns a decision, an Agent owns progress.

For example, an invoice-resolution Agent might retrieve an invoice and purchase order, invoke a Cognitive Unit to interpret the discrepancy, use deterministic code to calculate the numerical variance, consult another Cognitive Unit to judge whether the discrepancy is material, inspect company policy, recommend a resolution, and finally determine whether the case can be completed or must be escalated.

The important architectural distinction is that these individual judgments need not disappear inside the Agent. The Agent coordinates them. It decides what capabilities to invoke and when the goal has been sufficiently achieved, while individual cognitive responsibilities can remain separable and reusable.

This separation prevents the Agent from becoming an opaque concentration of intelligence. A large general-purpose Agent may be capable of performing many judgments internally, but architectural clarity improves when recurring and meaningful decisions are represented as explicit capabilities. The Agent can then focus on orchestration, state, sequencing, authority, and completion rather than absorbing every cognitive responsibility into its own reasoning process.

4.3 Thoughtware System

The Thoughtware System is the complete software system within which these capabilities operate. It may contain one Agent or many, numerous Cognitive Units, traditional services, databases, APIs, user interfaces, policy engines,

deterministic workflows, human approval points, tools, knowledge sources, and evaluation infrastructure.

This level is important because Thoughtware should not be mistaken for a collection of models or agents. A real product still has to authenticate users, persist data, enforce permissions, manage transactions, communicate with external systems, render interfaces, observe failures, comply with policy, and maintain operational integrity. Cognitive capability participates in this architecture, but it does not eliminate the rest of it.

An Agent is therefore not synonymous with Thoughtware. An Agent is one construct inside a Thoughtware architecture. A Thoughtware System may contain several specialised Agents, or in some cases no long-running Agent at all. It may rely heavily on Cognitive Units invoked through otherwise conventional application flows. What makes it Thoughtware is not the presence of a particular orchestration pattern, but the deliberate organisation of cognitive responsibility throughout the system.

This distinction is especially important while the industry is using “agentic” as a broad description for increasingly autonomous software. Autonomy is only one dimension of cognitive architecture. A system can exercise significant judgment without operating through an autonomous Agent, while an Agent may itself coordinate many deterministic operations. Treating the Agent as the entire category obscures these other structural possibilities.

Cognition and Determinism

Thoughtware also does not imply replacing ordinary software with model-driven behaviour. Deterministic mechanisms remain preferable wherever the required behaviour can be specified reliably.

> **Cognition belongs where judgment is required. Deterministic software belongs where procedure is sufficient.**

Calculations should normally remain calculations. Permissions should be enforced deterministically. Data persistence should not depend on interpretive judgment. Transactions require explicit guarantees. Validation rules, API calls, identifiers, policy constraints, and many forms of business logic are better

implemented using conventional software mechanisms when their expected behaviour can be stated precisely.

The architecture of Thoughtware therefore involves an allocation problem. Some responsibilities are computational. Others are cognitive. Many workflows combine both. A Cognitive Unit may decide what should happen, while deterministic code ensures that the resulting action is executed correctly. An Agent may determine which capability should be invoked, while conventional infrastructure controls whether that action is permitted. A Thoughtware System succeeds not by maximising the amount of cognition it contains, but by placing cognition where it creates value and preserving determinism where certainty is available.

The three levels provide a structural basis for making these decisions. Cognitive Units localise judgment. Agents organise progress toward goals. Thoughtware Systems combine these capabilities with the full machinery required to produce dependable software.

The resulting proposition is straightforward: Thoughtware does not replace software architecture. It extends software architecture with explicitly organised cognitive responsibility. The next question is therefore how the smallest of these responsibilities should be designed. That requires examining the Cognitive Unit in greater detail.

SECTION 5

The Cognitive Unit

Once judgment becomes part of software architecture, a practical question follows immediately: where should an individual judgment live? Traditional software has long answered similar questions for computation. Functions localise operations. Modules group related behaviour. Services create independent boundaries around capabilities. These constructs make software easier to reason about because they give responsibilities a place. Thoughtware needs an equivalent unit of cognitive locality.

This paper calls that unit the Cognitive Unit, or CU.

> **A Cognitive Unit is a sealed component that accepts defined inputs, performs one bounded cognitive responsibility, and returns an evaluable result. Its purpose is not to represent intelligence in general. It is to own one coherent judgment well enough that the rest of the system can rely on it as a capability.**

A CU might determine whether an invoice discrepancy is material, identify the intent behind a customer request, assess whether evidence supports a claim, classify the urgency of a case, judge whether a proposed answer is sufficiently grounded, or select the most appropriate option from a small set of candidates. What matters is that the responsibility can be named clearly enough that a caller understands what decision is being delegated.

This creates decision locality. Instead of allowing judgment to diffuse across a large prompt, workflow, or general-purpose agent, the architecture gives a

meaningful decision a defined home. That makes it possible to ask precise questions about the decision itself: What information does it require? What result should it produce? Under what conditions should it decline to decide? What constitutes good performance? What kinds of failure are acceptable? When should its behaviour be changed?

A strong Cognitive Unit therefore has several identifiable properties. It has a clear responsibility and defined inputs. Its output has a form that the caller can understand and use. The context available to the unit is bounded rather than implicitly inherited from an entire application state. Its dependencies, such as knowledge sources or tools, can be identified. Its quality can be measured through an evaluation suite. Its likely failure modes are understood well enough for callers to respond appropriately. Its behaviour can also be versioned, so that an improvement in judgment can be introduced, compared, and, where necessary, rolled back.

These properties resemble familiar software-engineering concerns, but cognitive behaviour adds an important complication. The implementation that produces the decision may be inherently variable. A CU may rely on a foundation model, retrieved knowledge, examples, reasoning strategies, intermediate judgments, or combinations of these mechanisms. Its internals may change substantially as models and techniques improve.

This is why encapsulation becomes especially important.

The caller of a Cognitive Unit should not need to know which prompt produced the decision, which model was used, whether retrieval occurred, which examples were inserted into context, or how many internal reasoning steps were required. It should not have to understand whether the unit used a single model call, several competing attempts, a critic-and-revision pattern, or a specialist model trained for the task. Those are implementation concerns.

The caller should instead depend on the decision contract.

The decision contract states what judgment the unit owns, what information it expects, what kind of result it returns, and what guarantees or limitations surround that result. A component calling judge invoice materiality, for example, should depend on the fact that the unit returns an assessment of materiality with the required supporting information. It should not depend on the particular prompt template or reasoning method that currently produces that assessment.

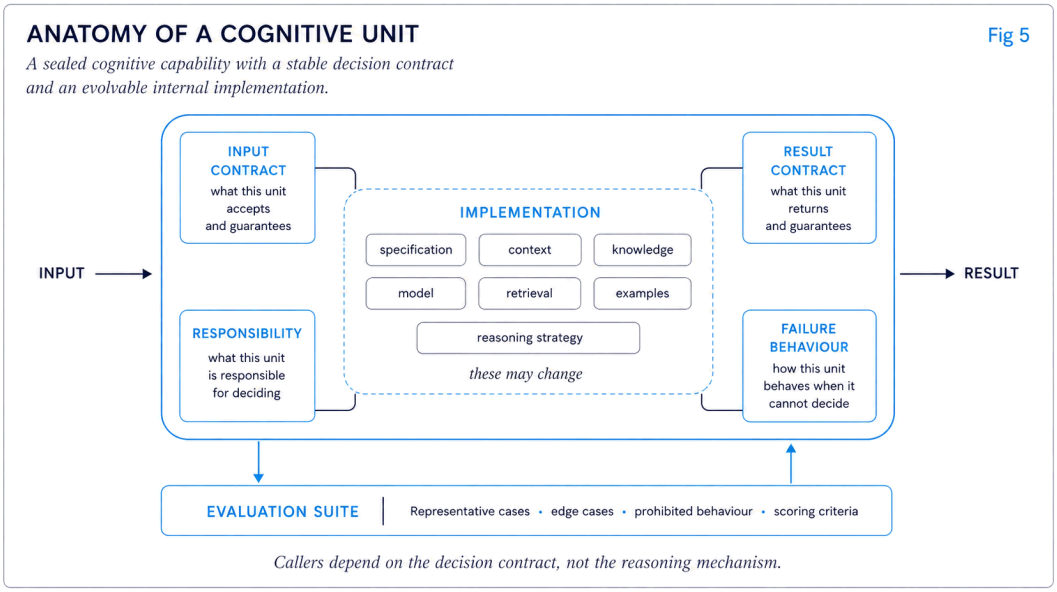
This black-box property is crucial because cognitive implementations are likely to evolve faster than the surrounding architecture. A CU may begin as a simple zero-shot call to a general-purpose model. Later, it may gain retrieved policy documents. It may move from one model family to another. A single-pass judgment may become a multi-stage process. Curated examples may be introduced. A specialist model may replace a general one. The unit may eventually learn a fast path for familiar cases while retaining a more expensive reasoning process for unusual ones.

None of these changes necessarily alter the architectural responsibility of the component.

This gives rise to a core principle of Thoughtware: cognitive encapsulation. Consumers of a cognitive capability should depend on its responsibility and result contract, not on the mechanism by which the judgment is produced.

Encapsulation matters for more than maintainability. It also changes how cognitive capability can improve. If a prompt is scattered through application code, replacing it may require modifying the surrounding product. If a judgment is sealed behind a stable cognitive boundary, its implementation can improve independently. Evaluation can compare one version of the capability with another while callers remain unchanged. The software system gains a way to evolve its intelligence without repeatedly restructuring every workflow that uses it.

The Cognitive Unit also creates a natural boundary for evaluation. Testing an entire intelligent application often produces weak information. If an end-to-end output is poor, it may be difficult to determine whether the problem came from interpretation, retrieval, reasoning, orchestration, or execution. When judgment is localised, the quality of a specific decision can be measured directly. Evaluation cases can be attached to the responsibility they test, and changes to the unit can be assessed against the same behavioural expectations over time.



This does not mean every model invocation should become its own CU. Architectural decomposition should follow coherent responsibility, not implementation granularity. A single Cognitive Unit may internally make several tightly related intermediate choices if those choices exist only to produce one externally meaningful judgment. Conversely, a broad reasoning task should be separated when it contains decisions that have independent meaning, can be evaluated independently, or are likely to be reused elsewhere.

The boundary is therefore conceptual rather than mechanical. A CU should be as small as necessary to create a clear locus of responsibility, but not so small that the architecture fragments one meaningful judgment into arbitrary implementation steps.

Once Cognitive Units are sealed in this way, another consequence follows: they can be collected and reused. An organisation may gradually accumulate CUs for assessing evidence, interpreting requests, prioritising cases, checking compliance, recommending actions, or evaluating outputs. Such a collection becomes more than a set of prompts. It becomes a library of cognitive capabilities within a domain.

This possibility is significant because reusable cognition changes what larger intelligent systems can be built from. Instead of repeatedly reconstructing the

same judgments inside every agent or workflow, future systems can compose from capabilities whose responsibilities and quality are already known.

But one judgment, however well encapsulated, is rarely enough to accomplish a meaningful goal. Real work requires deciding which capabilities to invoke, in what order, with what state, and when the objective has been sufficiently achieved. For that, Thoughtware needs a different architectural construct: the Agent.

SECTION 6

Agents and Cognitive Orchestration

A Cognitive Unit owns a judgment. An Agent owns progress toward a goal. This distinction is simple, but it separates two fundamentally different architectural responsibilities.

A Cognitive Unit is bounded around a decision. Its job is to accept relevant inputs, perform one coherent cognitive responsibility, and return an evaluable result. An Agent operates at a different level. It may need to decide which judgments are required, which tools should be used, what information is still missing, whether an intermediate result is sufficient, and what should happen next. Its responsibility is not a single conclusion but the controlled progression of work toward an outcome.

An Agent may therefore own a goal, working state, applicable policies, available authority, accessible Cognitive Units, tools, context, stopping conditions, and escalation conditions. These elements define the space within which it is allowed to operate. A customer-recovery Agent, for example, may be responsible for resolving a service complaint. To do so, it might retrieve booking information, ask a Cognitive Unit to interpret the complaint, invoke another to assess severity, calculate compensation using deterministic code, consult policy, determine whether the proposed remedy falls within its authority, and either execute the action or escalate the case to a human.

The Agent does not need to perform each judgment itself. Its distinctive responsibility is orchestration.

This orchestration is often described through an agent loop. Conceptually, an Agent may plan, act, observe, review, revise, and repeat until it reaches a stopping

condition. It may begin with an initial interpretation of the goal, select an action, inspect the result, reconsider its state, and decide whether another step is necessary. This loop captures something important about goal-directed cognitive behaviour: the path to completion does not always need to be specified in advance.

The loop should not, however, be mistaken for a mandatory implementation algorithm. Some Agents may require extensive planning and revision. Others may follow a relatively stable sequence with only occasional branching. Some may plan explicitly, while others make smaller local decisions after each observation. A mature Agent may even bypass most deliberation for situations it has encountered many times before. What defines the Agent is not a particular loop structure, but its ownership of the evolving trajectory toward a goal.

This distinction also clarifies the relationship between orchestration and intelligence. Current agent architectures often place large amounts of cognitive responsibility inside a general-purpose reasoning loop. The Agent receives a goal, reasons about the problem, chooses tools, interprets results, evaluates its own progress, and produces an outcome. This can be powerful, particularly for novel tasks. It can also become difficult to inspect, expensive to operate, and hard to improve systematically when recurring judgments remain buried inside the loop.

> **Thoughtware therefore favours a different architectural tendency: Agents should orchestrate cognition rather than absorb every cognitive responsibility into an opaque general-purpose loop.**

When a recurring decision has a stable meaning, it can often be extracted into a Cognitive Unit. Instead of repeatedly reasoning from first principles about whether an invoice discrepancy is material, an Agent can invoke a capability whose responsibility is precisely to make that judgment. Instead of reinterpreting compliance policy inside every trajectory, it can call an evaluated policy-assessment unit. The Agent remains responsible for deciding when those capabilities are needed and how their results affect progress, but the judgments themselves gain locality.

This separation improves reuse and evaluation, but it also has an economic consequence. Agent loops are expensive. Every additional cycle may require model inference, retrieval, tool execution, intermediate state, and further

evaluation. More importantly, an open-ended loop introduces variability in both cost and behaviour. For unfamiliar, high-ambiguity problems, that flexibility may be justified. For familiar situations, repeatedly reconstructing the same reasoning is wasteful.

> **A mature Thoughtware system should therefore not reason endlessly about situations it already knows how to handle.**

Repeated successful trajectories can gradually be compressed. A judgment discovered repeatedly inside an Agent may become a reusable Cognitive Unit. A stable sequence may become a deterministic fast path. A recurring approach may become a learned strategy. Patterns drawn from accumulated experience may become explicit expertise that shapes future decisions before an expensive search process begins.

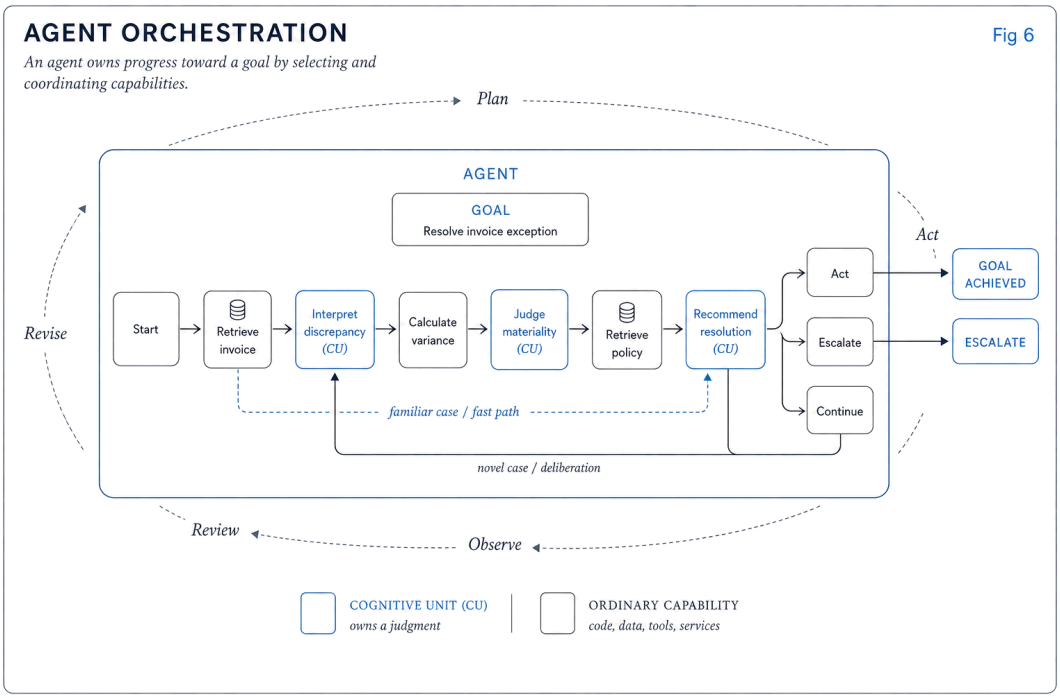
This creates a natural progression from deliberation to competence. Early in the life of a system, an Agent may need to explore several possibilities, inspect intermediate results, recover from errors, and repeatedly revise its plan. As the system gains reliable capabilities and recognises familiar situations, fewer decisions need to be rediscovered inside the loop. The trajectory becomes shorter, cheaper, and more predictable.

Crucially, compression should not eliminate the slower path. Familiarity can be mistaken, context can change, and apparently routine cases can contain novel conditions. A capable architecture therefore preserves the ability to return to deeper reasoning when confidence is insufficient or when a known strategy no longer fits. Fast paths represent accumulated competence, not permanent certainty.

This also explains why Agents and Cognitive Units should remain distinct even when their boundaries sometimes appear blurred. A sufficiently complex CU may contain internal reasoning steps. An Agent may make small local decisions that do not deserve separate components. The distinction is not based on the number of model calls or whether a loop exists internally. It is based on responsibility. A Cognitive Unit is accountable for the quality of a particular judgment. An Agent is accountable for moving a larger goal toward completion.

The architectural value of the Agent therefore lies in controlled composition. It brings together cognitive capabilities, deterministic operations, tools, knowledge,

state, and authority into a goal-directed process while preserving the boundaries of responsibilities that deserve to remain explicit.



As Thoughtware matures, this orchestration can itself become more capable. Agents can accumulate experience about which strategies work, recognise recurring situations earlier, and rely increasingly on reusable expertise. The next question is therefore how a system distinguishes what it knows now from what it has learned before, and how repeated experience can alter future judgment.

SECTION 7

Memory, Experience and Expertise

If Thoughtware is to improve through use, it needs more than a larger context window or an ever-growing history of prior interactions. Improvement requires distinctions between different kinds of information and the roles they play in future judgment. Treating all retained information as “memory” obscures these roles and makes both architecture and governance harder.

The first distinction is context. Context is information relevant to the situation at hand. It may include the current request, the user’s immediate objective, the state of an ongoing task, recent actions, or other information needed to make the present judgment. Context is local and temporary. Its value depends on the decision being made now.

Knowledge is different. Knowledge consists of reusable facts, policies, documents, rules, domain material, and other information that may remain useful across many situations. A customer-service system may draw on product documentation and refund policy. A finance system may consult accounting rules and vendor contracts. Knowledge does not necessarily come from the system’s own past behaviour. It is part of what the system can know about the world or the domain in which it operates.

Experience is a record of what happened before. It connects a previous situation with the judgments, actions, outcomes, and, where available, subsequent feedback associated with it. Experience therefore has temporal structure. It can preserve not only what was decided, but under what conditions the decision was made and what followed from it. A previously resolved invoice exception, for example, may later become relevant because a new case resembles it in important ways.

Yet retaining experience is not the same as learning from it. A system could accumulate millions of historical cases and still reason about each new one as if no prior work had occurred. The more consequential concept is expertise: reusable understanding about how classes of situations should be handled.

Expertise is not simply a remembered answer. It is a compression of repeated experience into something that can shape future judgment. A collection of successful cases may reveal that a particular discrepancy pattern is usually harmless under certain contract conditions. Repeated agent trajectories may reveal that one strategy consistently resolves a class of customer complaints with fewer steps. Evaluation results may show that a particular reasoning approach performs poorly when evidence is incomplete. When such patterns become sufficiently reliable, they can be represented explicitly and used to alter future behaviour.

This creates a progression:

EVENT → EXPERIENCE → PATTERN → EXPERTISE → FAST PATH

An event becomes experience when the system preserves enough information for it to be useful later. Across experiences, recurring structures may become recognisable patterns. Patterns that are sufficiently reliable and bounded can become expertise. Expertise can then influence how future situations are approached, sometimes allowing a system to bypass more expensive deliberation through a fast path.

This progression matters because intelligent systems should not be forced to rediscover familiar reasoning indefinitely. An Agent that repeatedly encounters the same class of problem may initially reason through several alternatives, retrieve multiple sources, and review its work before reaching a conclusion. If the same successful strategy proves useful across many comparable cases, that strategy can begin shaping the initial approach. A stable judgment may become a Cognitive Unit. A predictable sequence may become deterministic. A recognised case signature may invoke a known strategy directly. The system becomes more capable not merely because it remembers more, but because prior experience changes how new work is performed.

The reverse is equally important. Not every past interaction should become permanent memory, and not every recurring pattern deserves to become expertise. Historical information may be irrelevant, misleading, sensitive, outdated, or specific to a single unusual case. Uncritical accumulation can make future judgment worse rather than better. Thoughtware therefore needs selective retention, relevance, forgetting, and promotion mechanisms rather than an assumption that more memory necessarily produces more intelligence.

This is why memory should not be designed as one undifferentiated database. Context, knowledge, experience, and expertise serve different cognitive functions, have different lifecycles, and require different forms of access and governance. A system may need broad access to current context, selective retrieval from knowledge, similarity-based access to experience, and carefully validated use of expertise.

> **The key proposition is that experience becomes valuable when it changes future judgment. Mere retention is archival. Learning begins when previous outcomes influence how the next decision is made.**

That immediately raises a harder question. If a system changes its behaviour based on experience, how do we know that the new judgment is actually better rather than merely different? Improvement cannot be assumed from adaptation alone. It has to be demonstrated. That makes evaluation not an auxiliary concern, but a central architectural requirement.

PART III

Engineering Discipline

What replaces certainty when correctness cannot be specified in advance, what the human builder specifies instead, and where the vocabulary of implementation goes.

SECTIONS 8 — 10

SECTION 8

Evaluation Replaces Certainty

Traditional software engineering benefits from a powerful assumption: for many important behaviours, correctness can be specified before execution. Given a particular input, the system should produce a particular output. A test can therefore make a simple assertion: expected == actual. When the result differs, something is wrong. This model supports unit testing, regression testing, type systems, formal verification, and much of the discipline that makes conventional software dependable.

That assumption becomes weaker when software performs judgment.

Consider a recommendation produced for a customer complaint. Is it sensible given the circumstances? Does an explanation identify the issue that actually matters rather than merely repeating available information? Was escalation justified, or could the system have resolved the case safely? Did a response technically follow policy while violating its intent? These questions rarely have one exact string or value that can be declared correct in advance. Several answers may be acceptable, some may be clearly poor, and the difference between them may depend on qualities such as relevance, proportionality, grounding, completeness, or risk.

The loss of an exact expected output does not mean that quality becomes subjective or that engineering discipline must be abandoned. It means the mechanism of assurance has to change. Where deterministic correctness cannot be exhaustively specified, Thoughtware relies on evaluation.

Evaluation is not the practice of occasionally looking at model outputs and deciding that they seem reasonable. It is the systematic definition and measurement of what good cognitive behaviour means. A judgment that cannot be checked with a simple equality assertion can still be evaluated against

properties, criteria, cases, constraints, and reference judgments. The task moves from specifying every correct answer to specifying the dimensions along which an answer should be considered good or unacceptable.

This is especially important at the level of the Cognitive Unit. Because a CU owns a bounded judgment, it provides a natural boundary around which an evaluation suite can be constructed. Such a suite may contain representative cases that capture ordinary behaviour, edge cases that expose ambiguity or rare conditions, and adversarial cases designed to reveal predictable weaknesses. It can describe expected properties of the result, such as grounding, completeness, proportionality, or consistency. It can specify prohibited behaviours, such as inventing evidence, exceeding authority, ignoring a required policy, or reaching a decision when the available information is insufficient.

Evaluation can also use explicit scoring criteria. An invoice-materiality CU, for example, might be evaluated for whether it identifies the correct sources of discrepancy, applies policy appropriately, distinguishes material from immaterial variance, explains its conclusion using available evidence, and escalates when confidence is insufficient. Some cases may have reference judgments supplied by domain experts. Others may be evaluated along a rubric rather than against one canonical answer.

This creates an important difference between testing deterministic behaviour and evaluating cognitive behaviour. A conventional test often asks, did the implementation produce the specified answer? A cognitive evaluation asks, how well did the component exercise its assigned judgment under these conditions? The latter can still be rigorous, but the specification of quality is richer than a single expected value.

Cognitive systems can also participate in their own evaluation. A Judge CU is a Cognitive Unit whose responsibility is to assess the output or behaviour of another cognitive capability against defined criteria. A judge might determine whether an answer is grounded in the supplied evidence, whether a recommendation complies with policy, whether an explanation adequately supports its conclusion, or whether two candidate outputs differ meaningfully in quality.

Judge CUs make evaluation scalable because many behaviours that require judgment to produce also require judgment to assess. They can be run across large evaluation suites, compare alternative implementations, identify

regressions, and provide structured signals during development. They do not, however, make evaluation automatically objective.

A cognitive judge can carry its own biases. If the evaluated system and judge rely on similar models, their failures may be correlated. A judge may prefer stylistic qualities that do not correspond to actual correctness. Different judges may disagree. Scores may drift when the judge implementation changes. Some criteria that appear clear in a rubric may still be interpreted inconsistently. Judge CUs therefore require calibration against trusted reference cases and human judgment, especially where consequences are meaningful.

Evaluation itself must consequently be evaluated. A useful judge should demonstrate that its assessments correlate sufficiently with the standard the system intends to preserve. Disagreement between human reviewers and automated judges should be studied rather than hidden. Where no single evaluator is reliable enough, multiple signals may be combined. Human review remains particularly important when establishing new criteria, examining novel failures, calibrating judges, or evaluating decisions with substantial consequences.

At the Agent level, evaluation becomes broader still. A good final answer does not necessarily imply a good trajectory. An Agent might reach the correct conclusion after unnecessary reasoning, repeatedly invoke expensive tools, access information it did not need, exceed its authority before recovering, or succeed only because an intermediate mistake happened to cancel out later.

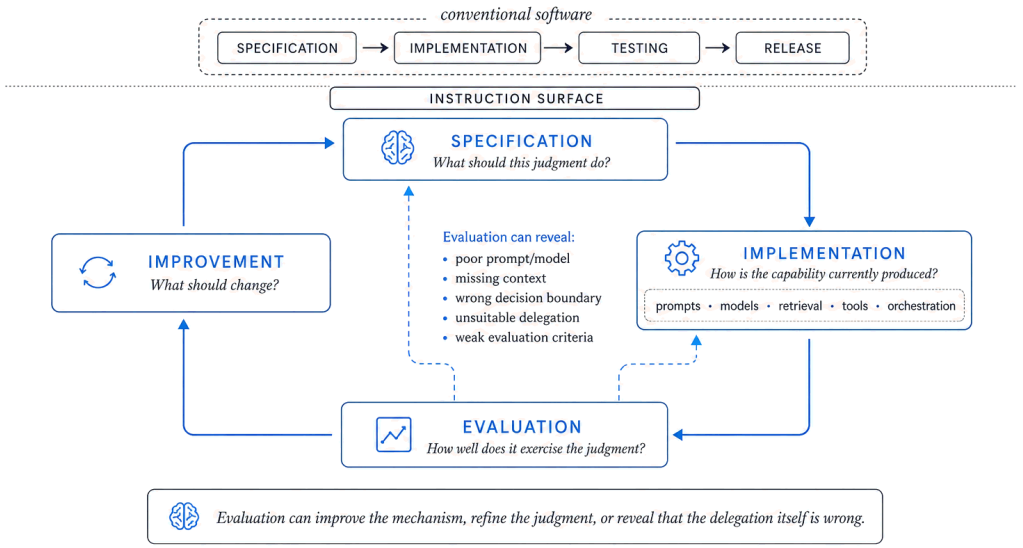
Agent evaluation must therefore consider the path as well as the destination. Was the appropriate Cognitive Unit selected? Were tools used correctly? Did the Agent take unnecessary steps? Did it respect cost and time constraints? Did it recognise when it lacked sufficient information? Did it escalate at the appropriate point? Could it recover when a tool failed or an intermediate judgment proved unreliable? Did it stop when the goal had been achieved rather than continuing to reason?

This trajectory-level evaluation becomes particularly important as Agents receive greater autonomy. A system that produces acceptable outputs while taking unsafe or economically unsustainable paths is not well engineered. Thoughtware must therefore evaluate outcomes, decisions, and the processes through which those outcomes were reached.

Evaluation also changes the development lifecycle. In conventional software, we often picture a progression from specification to implementation to testing. Testing confirms whether the implementation satisfies what was specified. In Thoughtware, the relationship is more iterative: specification, implementation, evaluation, and improvement continually inform one another. Evaluation may reveal that the implementation is weak, but it may also reveal that the cognitive responsibility was poorly defined, that necessary context is missing, that the evaluation criteria themselves are incomplete, or that a judgment should not have been delegated under certain conditions.

SPECIFICATION, IMPLEMENTATION, EVALUATION, IMPROVEMENT Fig 7

Thoughtware development is a continuous cognitive engineering loop, not a linear build-and-test process.



> **This leads to one of the central engineering principles of Thoughtware: in cognitive software, evaluation is not a QA phase applied after construction. It is part of the definition of the component itself.**

A Cognitive Unit without an evaluation strategy is only partially specified. An Agent whose success conditions are unclear cannot be meaningfully improved. A

system that adapts without evaluating the effect of that adaptation cannot distinguish learning from drift.

Thoughtware therefore replaces neither testing nor certainty. Deterministic parts of the system should continue to be tested deterministically wherever possible. But where software is entrusted with judgment, certainty often gives way to measured quality. The engineering challenge is to make that quality observable, comparable, governable, and capable of improving over time.

SECTION 9

The Thoughtware Specification

As cognitive implementation becomes more capable, a new question emerges for the human builder: what, exactly, should be specified?

Today, much of the practical work of building AI systems is expressed through implementation artifacts. Teams write prompts, assemble retrieval pipelines, configure model calls, define tool interfaces, tune orchestration logic, and construct evaluation scripts. These artifacts matter, but they belong to the implementation layer. They describe how a system currently achieves a capability, not necessarily what that capability is supposed to mean.

> **A prompt is therefore not a sufficient architectural specification.**

This is analogous to conventional software. A codebase may faithfully implement a product, but the code itself is not the same thing as the product specification. It does not, by itself, state why a capability exists, what responsibility it owns, what constraints matter most, or how success should be judged. In the same way, a collection of prompts is not a Thoughtware specification. Prompts may encode important behavioural intent, but they are too close to the mechanism and too sensitive to model, context, and implementation strategy to serve as the stable language of the architecture.

This becomes increasingly important as implementation itself is generated. A future system may be able to select models, construct prompts, assemble retrieval strategies, create tool bindings, choose orchestration patterns, and build evaluation harnesses automatically. If so, the human builder should not be forced to specify the software at the same level as the machinery being generated. The builder needs a higher-level surface that expresses what cognitive responsibility the system should own and under what conditions it should exercise it.

This paper calls that layer the Instruction Surface.

The Instruction Surface is the level at which a human describes the intended cognitive architecture of the system. It is not natural-language prompting in the ordinary sense, nor is it a conventional programming language. It is a structured way of expressing responsibilities, constraints, and expectations that can remain stable even when the underlying implementation changes.

A minimal Thoughtware specification may describe constructs such as:

OUTCOME	the broader result the system is intended to produce.
GOAL	the objective an Agent or subsystem is responsible for achieving.
JUDGMENT	a bounded decision that requires cognitive interpretation or reasoning.
INPUT AND OUTPUT	the information entering and leaving a capability.
CONTEXT	information relevant to the current situation.
KNOWLEDGE	reusable domain information the system may consult.
BEHAVIOUR	expectations about how the capability should act under particular conditions.
AUTHORITY	what the system is permitted to decide or execute.
ESCALATION	the conditions under which responsibility must move elsewhere, often to a human.
EVALUATION	the criteria by which the quality of the behaviour will be assessed.
LEARNING	the conditions under which experience may alter future behaviour.

The value of these constructs is not that every Thoughtware system must use this exact syntax. Their value is that they move specification above implementation detail. They describe the cognitive contract of the system.

Consider a customer recovery problem. At the implementation level, a builder might begin with an instruction such as: Write a prompt that evaluates refund requests and recommends a response. This immediately collapses architecture into mechanism. It says little about authority, context, escalation, or evaluation, and it assumes that a prompt is the relevant unit of construction.

At the Instruction Surface, the same responsibility could instead be expressed as:

JUDGMENT

Determine an appropriate customer recovery action.

CONTEXT

Booking history, complaint details, service policy, and relevant prior interactions.

AUTHORITY

May grant recovery up to a defined financial threshold.

ESCALATION

Requires human approval above that threshold, when policy is ambiguous, or when confidence is insufficient.

EVALUATION

Appropriateness, policy compliance, proportionality, grounding, and consistency across comparable cases.

This specification says much more about the system while saying much less about the implementation. It does not require a particular model, prompt structure, retrieval method, or orchestration framework. Those choices can evolve while the cognitive responsibility remains intact.

This separation enables what can be called Intent Compilation.

At the Instruction Surface, the human specifies the intended architecture of cognition. An implementation system can then compile that intent into the mechanisms required to realise it: prompts, model selections, retrieval pipelines, tool calls, orchestration logic, safeguards, and evaluation harnesses. Some implementations may use a single model call. Others may use several Cognitive

Units, a specialised Agent, deterministic checks, or a mixture of these. The specification survives those differences because it captures the responsibility rather than the machinery.

Intent Compilation also creates a stronger basis for change. If a model improves, the implementation can be regenerated without rewriting the specification. If evaluation reveals weak behaviour, the system can alter prompting, retrieval, or model selection while preserving the intended judgment. If a new regulatory requirement changes authority, the specification can be updated at the level where that responsibility is actually expressed rather than buried inside prompts and application code.

This does not imply that human builders will cease to care about implementation. Architectural judgment remains essential. Builders will still need to understand trade-offs involving latency, cost, observability, failure modes, privacy, and reliability. But the durable human contribution moves upward. It becomes less about manually encoding every cognitive mechanism and more about specifying the responsibilities, boundaries, and standards that those mechanisms must satisfy.

The key proposition is therefore that the long-term programming surface of cognitive software will increasingly describe responsibilities and judgment rather than manually encode every mechanism beneath them.

As this transition progresses, much of today's AI implementation vocabulary may become less visible. Prompts, model routing, retrieval strategies, and orchestration techniques may remain important inside the substrate while disappearing from the primary language used to describe the system itself. That possibility leads to a broader consequence of the Thoughtware model: as cognitive infrastructure matures, AI may become less visible precisely because it has become more foundational.

SECTION 10

The Submergence of AI

New technologies are often most visible before they become ordinary. Their terminology appears in product names, interfaces, investor language, job titles, architecture diagrams, and everyday conversation because the mechanism itself is still novel. Once the capability matures, however, the vocabulary of implementation tends to retreat beneath the surface.

Most users of modern software do not think about TCP when sending a message, SQL when viewing an account balance, the DOM when using a web application, serialization when synchronising data, RPC when invoking a remote capability, or container orchestration when using a cloud service. These technologies remain fundamental. Some are more important than ever. But their importance no longer requires them to occupy the conceptual surface of the product. Unless a person's work directly concerns the infrastructure, the infrastructure disappears beneath the activity it enables.

AI is currently in the opposite condition. Its implementation vocabulary is unusually exposed. Products advertise that they are powered by AI, GPT, agents, copilots, models, or retrieval-augmented generation. Interfaces invite users to "ask AI." Product documentation explains prompts, model selection, agentic workflows, and context windows. Builders routinely organise systems around concepts such as embeddings, tool calls, routing graphs, model endpoints, and retrieval pipelines.

This visibility is understandable. Cognitive capability is still new enough that the mechanism itself carries meaning. Calling something an AI assistant communicates an expectation that would otherwise be difficult to express. Describing a system as agentic signals that it can perform multi-step work with some autonomy. Terms such as RAG distinguish emerging implementation approaches that teams are still learning to design and operate.

But this vocabulary should not be mistaken for the permanent language of cognitive software.

This paper proposes the Submergence Principle:

> **As a technological capability becomes ordinary infrastructure, its implementation vocabulary retreats from the domain surface.**

The domain surface is where people encounter software in the language of the work they are trying to accomplish. As cognitive capability matures, users should increasingly be able to remain there. A traveller wants to plan my trip. A finance team wants to resolve this exception. A policyholder wants to prepare my claim. An operator wants to manage my inventory. None of these intentions naturally require the user to think about whether an agent, model, retrieval system, or particular prompting technique sits underneath the interaction.

This produces a process of re-naturalisation. Software initially exposes the vocabulary of a new technical capability because users and builders have not yet developed abstractions around it. As the capability becomes dependable and embedded, interaction returns to the natural language of the domain. The user stops operating the AI and starts operating through the software again.

This does not imply that conversational interfaces will disappear. Conversation may remain one useful interaction form among many. What should disappear is the requirement that the user conceptualise the product in terms of its AI machinery. A future travel system may use several models, specialised cognitive capabilities, planning Agents, retrieval, vision, and continuous evaluation while presenting none of those concepts to the traveller. What matters at the domain surface is whether the trip is planned well.

The more consequential form of submergence, however, may occur on the builder side.

Today, building cognitive software often requires explicit manipulation of prompts, embeddings, context assembly, model calls, tool schemas, routing logic, and orchestration graphs. These are currently necessary engineering concerns because the infrastructure remains immature and relatively manual. But if implementation systems become increasingly capable of generating and optimising these mechanisms, the human programming surface can move upward.

Builders may spend less time specifying which prompt should contain which sentence and more time defining which judgment the system owns. They may spend less time drawing tool-routing graphs and more time expressing authority and escalation. Rather than manually deciding how every piece of context reaches every model invocation, they may define what information a cognitive responsibility is permitted or required to know. Rather than optimising model calls directly, they may define evaluation criteria against which generated implementations can be compared.

The concepts that rise to the surface are therefore not primarily technical mechanisms. They are judgment, responsibility, behaviour, authority, context, evaluation, and outcome. These are closer to the enduring architecture of the system because they describe what the software is responsible for rather than how a particular generation of technology happens to implement it.

This shift is already implicit in the Instruction Surface described in the previous section. If cognitive intent can be compiled into prompts, retrieval, orchestration, tools, and evaluation infrastructure, then those implementation mechanisms can remain accessible when necessary without remaining the primary language of construction. Just as modern developers can work at high levels of abstraction without forgetting that networks, memory, processors, and databases exist, Thoughtware builders can increasingly work in terms of cognitive architecture without denying the mechanisms beneath it.

Submergence should therefore be understood as a sign of technological maturity, not disappearance. The technologies themselves may become more pervasive precisely as their names become less visible. AI can move from being a feature attached to software to becoming part of the substrate from which software is constructed.

The central proposition is therefore paradoxical only at first: AI disappears not because it becomes less important, but because it becomes foundational. When that happens, users return to the language of their work, builders move toward the language of judgment, and the architecture of cognition remains after the vocabulary of its implementation has receded beneath the surface.

PART IV

Consequences

What changes for the organisations that build this software, how it looks in one ordinary operational problem, and which questions remain open.

SECTIONS 11 — 15

SECTION 11

Implications for Software Organisations

If the architecture of software changes, the organisation around it changes as well. Thoughtware does not merely introduce new technical components into an existing development process. It shifts where design effort, engineering attention, and organisational judgment need to be concentrated. The central work moves gradually away from specifying every feature and interaction in advance and toward defining which cognitive capabilities a system should possess, how those capabilities should behave, and where responsibility should remain with humans.

For product leaders, this changes the unit of planning. Traditional product development often decomposes a roadmap into features, screens, workflows, and integrations. These remain important, but they are no longer sufficient when the product itself can interpret situations and exercise judgment. The more consequential questions become: What judgment should the product own? What should remain human? What information should it be allowed to use? What level of confidence is required before it acts? When should it ask for clarification? When should it escalate rather than proceed?

This moves product thinking from feature definition toward capability and delegated responsibility. A feature describes something the product can do. A cognitive capability describes a class of decisions the product can be trusted to make. The latter requires a deeper specification because the system is no longer only following a path designed in advance. Product leadership increasingly has to define the boundaries within which the system may interpret, choose, recommend, and act.

Design changes for the same reason. Much of traditional digital design has focused on arranging information, reducing interaction cost, guiding attention,

and making workflows understandable. These concerns do not disappear, but the design surface expands. When the system itself can decide how to respond, the behaviour of that intelligence becomes part of the user experience.

Designers therefore have to consider questions of disclosure, intervention, autonomy, context, memory, confidence, and escalation. What should the system remember, and what should it forget? When should it present a recommendation as tentative rather than definitive? When should it expose uncertainty? Should it act automatically or propose an action first? What should happen when the user disagrees? How should a system communicate that it lacks sufficient information without shifting unnecessary cognitive work back onto the user?

This is a move from designing only interfaces toward designing intelligent behaviour. The interface remains one expression of that behaviour, but it is no longer the whole of it. As visual patterns become increasingly standardised and implementation becomes more generative, a larger proportion of conventional interface production may be automated. Human design attention can then shift upward toward the architecture of interaction between human and machine judgment.

Engineering changes in parallel. Conventional engineering remains essential, but new responsibilities appear around cognitive architecture. Engineers must decide which responsibilities belong in deterministic code and which justify cognitive treatment. They need clear boundaries around Cognitive Units, reliable interfaces between cognitive and deterministic components, observability into system behaviour, and infrastructure for running evaluations at scale.

This also changes what robustness means. A service that crashes can be monitored through familiar operational signals. A cognitive capability can remain fully available while quietly degrading in judgment quality. An Agent may complete its task while using unnecessarily expensive trajectories or escalating too often. A model update may improve one class of cases while introducing regressions in another. Engineering therefore has to observe not only whether the system is running, but whether its cognitive behaviour remains within acceptable bounds.

Testing consequently expands into evaluation. Traditional tests continue to protect deterministic behaviour, but they are joined by evaluation suites that measure interpretation, reasoning, recommendation quality, policy adherence, escalation, and other cognitive properties. This affects the role of quality teams as

well. The work shifts from verifying only whether prescribed behaviour occurred toward defining what good judgment looks like, constructing representative cases, calibrating evaluators, investigating regressions, and determining when a capability is safe to promote.

At organisational scale, perhaps the most important change is reuse. Today, companies frequently solve similar cognitive problems independently across different products and teams. One product interprets customer intent. Another assesses evidence. A third prioritises cases. Each may construct its own prompts, evaluation criteria, retrieval strategies, and failure handling.

Thoughtware provides a different possibility. Once cognitive responsibilities are sealed, evaluated, and governed, organisations can begin accumulating libraries of reusable cognitive capabilities. A capability for interpreting policy, judging claim completeness, assessing risk, or evaluating written reasoning can become shared infrastructure rather than being rebuilt inside every application.

Such libraries represent something more significant than technical reuse. They become repositories of organisational cognitive capability. Policies, judgment criteria, experience, evaluation standards, and domain expertise can be concentrated into components that many systems can invoke. Improvement to one capability can then propagate across multiple products, while governance can be applied at the level where the judgment actually resides.

This also suggests new organisational responsibilities. Someone must own shared cognitive capabilities, define their evaluation standards, control their authority, manage versions, and determine when they are appropriate for reuse. Product, design, engineering, domain experts, and evaluators become more tightly connected because no single discipline can specify a consequential judgment alone.

For this reason, enterprise AI maturity should not be measured by the number of models integrated, copilots launched, or Agents deployed. Those metrics primarily describe adoption of technology. A more meaningful measure is whether the organisation can deliberately identify cognitive responsibilities, build them with clear boundaries, evaluate their behaviour, reuse proven capabilities, govern their authority, and improve them over time.

> Enterprise AI maturity is therefore not the quantity of AI inside the organisation. It is the organisation's ability to deliberately build, evaluate, reuse, and govern cognitive capability.

SECTION 12

Thoughtware in Practice

The architectural ideas in this paper become clearer when applied to an ordinary operational problem. Consider invoice exception resolution, a process found in many organisations whenever an invoice does not match the purchase order or expected delivery against which it was raised. The problem is useful because part of it is perfectly deterministic while another part requires interpretation and judgment. It therefore illustrates why Thoughtware is not simply the replacement of conventional software with models, but the deliberate allocation of different kinds of responsibility.

A traditional invoice-processing system can handle a large portion of the workflow through rules. It can retrieve the invoice and purchase order, compare totals, calculate numerical variance, check whether the vendor is active, apply configured tolerance thresholds, and route the case to an approver according to organisational policy. If an invoice is within a permitted percentage or absolute variance, it may pass automatically. If it exceeds the threshold, the system may create an exception.

For sufficiently predictable cases, this works well. The difficulty begins when the meaning of the exception cannot be inferred from the numbers alone.

Suppose the ordered and invoiced quantities differ only for some line items. The vendor has included an explanation in an email. Part of the shipment was damaged and replaced separately. A contract permits particular substitutions or quantity adjustments under defined circumstances. Similar discrepancies from the same vendor have previously been accepted because of a known fulfilment arrangement. The invoice may still fail a numerical comparison even though the discrepancy is legitimate.

Traditional rules can be expanded to cover some of these conditions, but complexity grows quickly. Each new form of exception requires another branch, field, status, or manual intervention. The difficulty is not merely that there are many rules. It is that the system must determine what the available evidence means in the particular situation before the relevant rule can even be applied.

A Thoughtware architecture separates these responsibilities.

At the centre of the system might be an Agent with the goal Resolve Invoice Exception. The Agent does not itself need to own every judgment involved in the process. Its responsibility is to move the exception toward a valid resolution by coordinating deterministic operations and specialised Cognitive Units.

The process may begin conventionally. Deterministic software retrieves the invoice, purchase order, goods-receipt information, vendor record, and related documents. It calculates the exact numerical variance and validates that the case is eligible for processing. None of these operations requires cognitive interpretation, so introducing model-driven behaviour would add little value.

Cognitive responsibility begins where meaning must be established. An Interpret Discrepancy CU can determine what actually differs between the invoice, purchase order, delivery records, and accompanying explanation. A Determine Likely Cause CU can distinguish among possibilities such as pricing error, partial fulfilment, damaged goods, authorised substitution, duplicate billing, or an unexplained variance. An Assess Supporting Evidence CU can judge whether the available documents adequately support the explanation.

Further judgments can remain separate. A Judge Materiality CU can assess whether the discrepancy is significant in light of value, policy, contractual conditions, and risk. A Recommend Resolution CU can propose an appropriate response, such as accepting the invoice, requesting a corrected invoice, adjusting the payable amount, seeking additional evidence, or disputing the charge. A final Judge Escalation capability can determine whether the proposed resolution falls within the system's permitted authority.

These responsibilities are separated because they have independent meaning. Assessing evidence is useful beyond invoice processing. Materiality may be a recurring financial judgment across several workflows. A capability for judging whether available evidence adequately supports a claim may eventually be shared across procurement, expenses, insurance, compliance, or customer disputes. The

architecture therefore creates the possibility of reusable cognition rather than embedding the same reasoning repeatedly inside each workflow.

Knowledge and experience also play distinct roles. The relevant contract and procurement policy provide knowledge against which the current case can be interpreted. Vendor history and previously resolved exceptions provide experience. If repeated cases reveal that a particular discrepancy pattern is routinely legitimate under a specific contractual arrangement, that pattern may eventually become explicit expertise, allowing familiar exceptions to follow a shorter and more reliable path.

The Agent orchestrates these capabilities according to the situation. A simple exception may require only interpretation and a materiality judgment before being resolved. An unusual case may require additional evidence, consultation of contractual knowledge, comparison with previous cases, and several rounds of review. The trajectory is therefore adaptive without requiring every underlying judgment to become part of one opaque reasoning loop.

Authority remains separate from capability. The system may be perfectly capable of recommending a large adjustment without being authorised to execute it. An organisation might allow low-value, reversible, well-supported exceptions to be resolved automatically while requiring approval when the financial value exceeds a threshold, evidence is contradictory, the vendor is high risk, or the case falls outside familiar patterns. Human involvement is therefore not an exception to the architecture. It is one deliberately designed destination within it.

Once a resolution is approved, execution returns to deterministic software. Permissions are checked. Adjustments are written to the accounting system. Transactional integrity is preserved. Notifications are issued. Audit records are created. The cognitive system decides what should happen within its authority, while conventional software ensures that the action happens exactly and traceably.

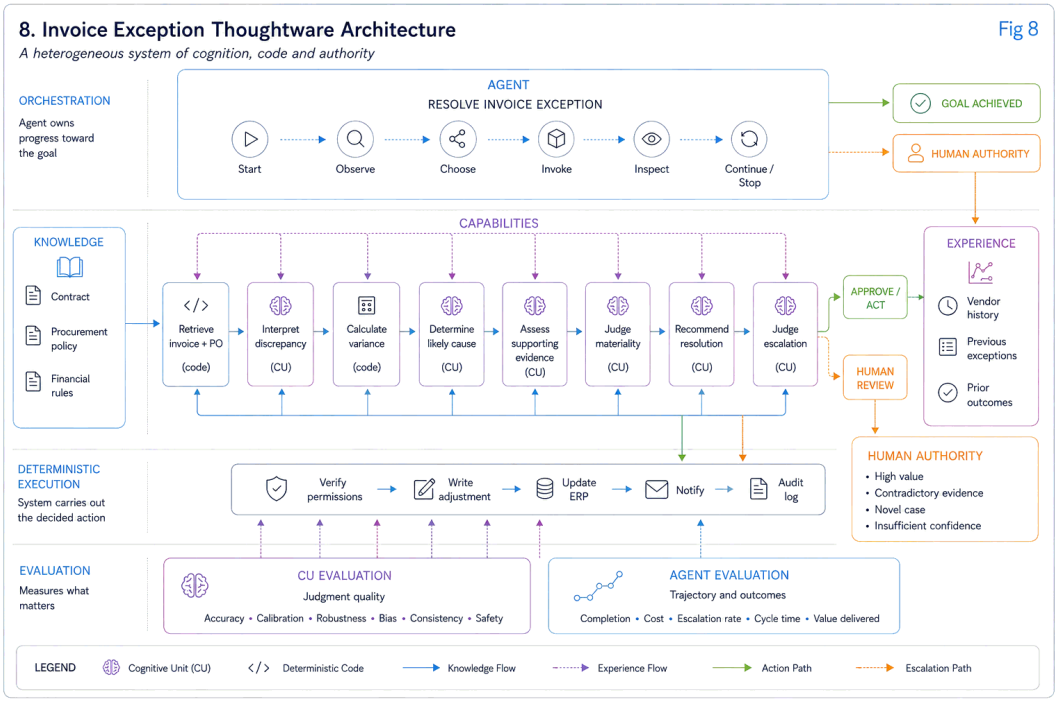
Evaluation surrounds the cognitive portion of the system. The Interpret Discrepancy CU can be tested against representative and difficult exception cases. The evidence-assessment capability can be evaluated for whether it distinguishes genuine support from irrelevant documentation. Materiality judgments can be compared with decisions made by experienced finance professionals. Resolution recommendations can be scored for policy compliance, proportionality, financial

correctness, and consistency across comparable cases. Escalation can be evaluated for both unsafe under-escalation and costly over-escalation.

The Agent itself requires another level of evaluation. Did it call the appropriate capabilities? Did it retrieve unnecessary information? Did it repeatedly reconsider a conclusion that was already sufficiently supported? Did it exceed its authority? Did it ask for additional evidence when appropriate? Did it escalate cases that could reasonably have been resolved automatically? Did it reach the correct outcome through an efficient and defensible trajectory?

Over time, the system can improve without simply accumulating longer prompts. Evaluation may reveal weak Cognitive Units that need better knowledge, different implementation strategies, or revised decision boundaries. Frequently repeated judgments can be consolidated into reusable capabilities. Familiar trajectories can become fast paths. New experience can refine expertise. At the same time, unusual cases can continue to fall back to slower deliberation or human judgment.

Invoice exception resolution therefore illustrates the larger structure of Thoughtware. Deterministic code handles what can be specified precisely. Cognitive Units own bounded judgments. An Agent coordinates those capabilities toward a goal. Knowledge and experience inform decisions without becoming undifferentiated memory. Evaluation determines whether behaviour is actually improving. Authority defines where machine responsibility ends and human responsibility begins.



The result is not an “AI feature” attached to an invoice system. It is a software architecture in which computation and cognition are deliberately assigned to the forms of work each is best suited to perform.

SECTION 13

Principles of Thoughtware

The theory developed in this paper can be reduced to a small set of architectural principles. These are not implementation recipes. They are intended to remain useful as models, frameworks, and development tools change. Together, they describe how cognitive responsibility should be organised when judgment becomes part of software.

01 Judgment should be explicit

If software owns a judgment, that responsibility should be nameable. “Use AI here” is not an architectural decision. Determine whether the system is interpreting intent, assessing evidence, recommending an action, judging materiality, evaluating an answer, or performing some other bounded form of cognitive work. Explicit judgment makes responsibility visible and therefore designable.

02 Judgment should have locality

A meaningful decision should live in the smallest coherent component capable of owning it. Locality makes the inputs, outputs, context, dependencies, failure modes, and evaluation criteria of a judgment easier to understand. Cognitive responsibility should not diffuse unnecessarily across prompts, workflows, and general-purpose Agents when it can be given a clear architectural home.

03 Cognitive capability should be encapsulated

Consumers of a cognitive capability should depend on its responsibility and result contract rather than the mechanism used to produce the judgment. Prompts,

models, retrieval strategies, examples, and reasoning structures are implementation details. Encapsulation allows these mechanisms to change while preserving the architectural meaning of the capability.

04 Determinism should remain deterministic

Thoughtware is not an argument for replacing conventional software with cognition. Where the desired behaviour can be reliably expressed as a procedure, deterministic implementation remains preferable. Calculations, permissions, persistence, transactions, validation, and other precise operations should retain the certainty available to them. Judgment belongs where procedure is insufficient.

05 Agents should own goals, not every judgment

The defining responsibility of an Agent is progress toward a goal. It may select capabilities, use tools, maintain working state, inspect results, revise its approach, and determine when to stop or escalate. Recurring and independently meaningful judgments should, however, remain separable Cognitive Units rather than being repeatedly reconstructed inside an opaque general-purpose reasoning loop.

06 Evaluation is part of architecture

A cognitive capability is incomplete if there is no defined way to determine whether it performs well. Evaluation should be designed alongside the responsibility itself through representative cases, edge conditions, expected properties, prohibited behaviours, scoring criteria, reference judgments, and calibrated evaluators. Cognitive quality cannot be treated as an impression discovered after deployment.

07 Authority must be designed separately from capability

The ability to make a judgment does not automatically confer permission to act on it. Thoughtware must distinguish what a system can determine from what it is authorised to execute. Consequence, reversibility, confidence, policy, and risk should shape when software acts autonomously, proposes an action, seeks approval, or escalates responsibility.

08 Expertise should compress cognition

A mature system should not repeatedly rediscover reasoning it has already learned to perform reliably. Recurring successful judgments can become reusable Cognitive Units, repeated trajectories can become known strategies, and sufficiently stable behaviour can become deterministic fast paths. Expertise should reduce unnecessary deliberation while preserving slower reasoning for novel or uncertain situations.

09 Architecture should outlive implementation vocabulary

A durable Thoughtware architecture should remain meaningful when models, prompting techniques, retrieval methods, and orchestration frameworks change. As cognitive infrastructure matures, implementation terminology should increasingly submerge beneath stable concepts such as judgment, responsibility, authority, context, behaviour, and evaluation. The architecture describes what the system is responsible for, not which generation of technology currently implements it.

10 Human judgment remains part of the system

Human involvement should be deliberately located rather than treated as evidence that automation has failed. Some judgments remain too consequential, novel, poorly grounded, or difficult to evaluate for autonomous delegation. Thoughtware should therefore specify where humans approve, intervene, provide missing context, resolve disagreement, calibrate evaluation, and retain final responsibility.

Taken together, these principles point toward a different way of constructing intelligent software. The objective is not to maximise autonomy or the amount of AI inside a product. It is to allocate judgment deliberately, give cognitive responsibility clear boundaries, preserve certainty where certainty is available, and make the resulting behaviour evaluable and governable. Thoughtware becomes dependable not when every part of software becomes cognitive, but when cognition itself becomes architected.

SECTION 14

Open Questions and Boundaries

Thoughtware is proposed here as a way to reason more clearly about software whose behaviour depends materially on machine judgment. Like any architectural theory, however, its usefulness depends not on pretending that every boundary is already settled, but on making the difficult boundaries easier to examine. Several questions remain open and should be treated as part of the research and engineering agenda that follows from the category.

One such question concerns the boundary between a Cognitive Unit and an Agent. The distinction proposed in this paper is responsibility-based: a CU owns a judgment, while an Agent owns progress toward a goal. In practice, however, a single judgment may require several internal steps, intermediate checks, or even a short reasoning loop. At what point does a sufficiently complex CU become an Agent? The likely answer cannot depend simply on implementation mechanics. The more useful boundary may be whether the internal decisions have independent architectural meaning, deserve separate evaluation, or could reasonably be reused elsewhere.

A second question concerns the external determinism expected of cognitive components. A CU may be probabilistic internally while still presenting a stable responsibility and result contract to the rest of the system. But how stable must that behaviour be before other components can safely depend on it? Traditional APIs can preserve identical signatures while cognitive behaviour changes substantially. This creates a related problem of versioning. A new model, prompt strategy, knowledge source, or reasoning method may improve average performance while altering edge-case behaviour even though no interface has changed. Cognitive versioning may therefore need to account for behavioural

performance, evaluation results, and known failure characteristics in addition to conventional API compatibility.

There is also an unresolved tension around inspectability. Cognitive encapsulation argues that callers should not depend on prompts or internal reasoning strategies. Governance and debugging, however, often require some visibility into why a judgment occurred. Complete exposure of internal reasoning may be undesirable, unreliable, or unnecessary, while complete opacity can make failures difficult to investigate. Thoughtware therefore needs better forms of inspectability that preserve implementation independence while still exposing evidence, decision factors, confidence, provenance, and other useful traces of behaviour.

The progression from experience to expertise introduces another boundary. Not every remembered case should influence future behaviour, and not every recurring pattern deserves architectural permanence. Systems need ways to determine when a pattern is sufficiently general, stable, and well evaluated to become reusable expertise. They also need mechanisms for revising or retiring expertise when environments, policies, or underlying conditions change.

Evaluation introduces its own recursive problem. If one cognitive capability judges another, who evaluates the judge? Automated judges can be biased, correlated with the system they evaluate, or poorly calibrated against actual domain expectations. Better methods are needed for combining judge CUs, human reference judgments, disagreement analysis, calibration sets, and meta-evaluation without simply moving uncertainty one level higher.

Finally, the question of authority remains partly technical and partly institutional. Capability alone cannot determine how much responsibility software should receive. Consequence, reversibility, uncertainty, accountability, regulation, and social expectations all affect whether a system should act, recommend, ask, or escalate. The appropriate boundary will differ across domains and may change as evaluation evidence accumulates.

These questions do not weaken the case for Thoughtware. They reveal the kinds of problems that become visible once cognition is treated architecturally rather than as an undifferentiated AI feature. A useful theory should not eliminate such questions prematurely. It should provide a language in which they can be stated precisely, investigated systematically, and progressively resolved.

SECTION 15

Toward an Architecture of Thinking

Software began primarily as encoded procedure. Human builders determined the rules, conditions, sequences, and exceptions, and machines executed those decisions with speed and consistency. Machine learning shifted some of this behaviour from explicit programming into learned prediction. Foundation models extend that shift further by making broader cognitive capabilities programmable: interpretation, reasoning, evaluation, recommendation, and judgment can now occur inside the running system.

Yet much of the current language around this change still describes the machinery. We speak of models, prompts, RAG, agents, copilots, tool use, and orchestration frameworks. These terms are important because the infrastructure is still new and visible. But they do not provide a durable theory of the software being built from it.

Thoughtware proposes a higher-level category. Its central concern is not which model is used or how many agents a system contains, but how cognitive responsibility is organised. Once software can exercise judgment, architecture must determine where that judgment belongs, how it is bounded, what information it may use, how it interacts with deterministic components, how much authority it receives, and how its quality is established.

The primitives developed in this paper provide one way to make that responsibility explicit. Judgment can be localised into coherent cognitive capabilities rather than dispersed through undifferentiated AI behaviour. Those capabilities can be bounded by contracts, encapsulated from their implementation mechanisms, reused across systems, and orchestrated toward larger goals. Their behaviour can be evaluated rather than merely observed,

governed according to consequence and authority, and improved through accumulated experience and expertise.

This also changes the meaning of progress in intelligent software. Better systems will not simply contain larger models or longer reasoning chains. Maturity may instead appear as greater architectural clarity: fewer judgments being rediscovered unnecessarily, more capabilities being reused, stronger evaluation, shorter trajectories for familiar situations, more deliberate boundaries between human and machine responsibility, and more of the underlying implementation disappearing beneath stable cognitive abstractions.

> **The important transition is therefore not from software to software plus AI. It is from software whose behaviour is primarily encoded as procedure to software whose architecture can also include cognition.**

That distinction matters because additions can remain peripheral. An AI feature can be attached to a product without changing the structure beneath it. Cognition as an architectural material is different. It changes what can be delegated, what can be composed, what must be evaluated, and ultimately what builders need to specify.

As the technology matures, much of the vocabulary that currently dominates AI development may recede from view. Models will remain. Retrieval will remain. Orchestration will remain. New mechanisms will replace some of them. But the enduring questions will sit above those mechanisms.

What does the system know? What judgment does it own? What should remain deterministic? What is it allowed to do? What does it remember from experience? When should it ask rather than act? When should responsibility return to a human? How can we tell whether its judgment is becoming better?

When AI eventually disappears beneath the surface of software, these are the questions that remain. What becomes visible is not the machinery of intelligence, but its architecture.

> That is the deeper proposition of Thoughtware: the Intelligence Age requires not only more capable software, but a discipline for organising cognition itself.

References

The references below provide the principal intellectual foundations for the concepts developed in this paper. They are organised by area for readability in this version of the white paper. A publication version may instead present them as a single alphabetical bibliography.

SOFTWARE ARCHITECTURE, MODULARITY, AND COMPOSITION

Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison-Wesley Professional.

Fielding, R. T. (2000). Architectural Styles and the Design of Network-Based Software Architectures [Doctoral dissertation, University of California, Irvine].

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Lewis, J., & Fowler, M. (2014). Microservices: A definition of this new architectural term. MartinFowler.com.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053–1058. DOI: 10.1145/361598.361623.

These works provide the architectural lineage for several ideas used throughout this paper, particularly responsibility-based decomposition, information hiding, encapsulation, stable interfaces, independent capability boundaries, and the organisation of systems from composable parts. The Cognitive Unit is not presented as equivalent to a traditional function, module, object, or service, but it extends a long-standing architectural principle: meaningful responsibility becomes easier to build, reuse, change, and reason about when it has an explicit boundary.

FOUNDATION MODELS, LANGUAGE MODELS, TOOL USE, AND AGENTS

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. (2021). On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. Advances in Neural Information Processing Systems, 36.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. V., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824–24837.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations.

These works establish the technological context from which Thoughtware emerges: general-purpose language models, foundation models that transfer across tasks, language-mediated reasoning, interaction with external tools, and systems capable of interleaving reasoning with action. Thoughtware builds above these mechanisms rather than proposing an alternative to them. Its concern is the architectural organisation of the cognitive capabilities that these technologies make possible.

JUDGMENT, WORKING MEMORY, AND EXPERTISE

Baddeley, A. (2000). The episodic buffer: A new component of working memory? Trends in Cognitive Sciences, 4(11), 417–423. DOI: 10.1016/S1364-6613(00)01538-2.

Ericsson, K. A., Krampe, R. T., & Tesch-Römer, C. (1993). The role of deliberate practice in the acquisition of expert performance. Psychological Review, 100(3), 363–406. DOI: 10.1037/0033-295X.100.3.363.

Evans, J. St. B. T. (2008). Dual-processing accounts of reasoning, judgment, and social cognition. Annual Review of Psychology, 59, 255–278. DOI: 10.1146/annurev.psych.59.103006.093629.

Kahneman, D. (2011). Thinking, Fast and Slow. Farrar, Straus and Giroux.

Tversky, A., & Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. Science, 185(4157), 1124–1131. DOI: 10.1126/science.185.4157.1124.

These references inform the paper's treatment of judgment, limited working context, experience, expertise, and faster versus more deliberative forms of decision-making. The analogy is intentionally limited. Thoughtware does not claim that computational systems reproduce human cognitive architecture, nor that concepts such as System 1 and System 2 map directly onto particular model behaviours. Cognitive science is used here as a conceptual mirror for separating functions that are otherwise easily collapsed into the broad label of intelligence.

HUMAN-AI INTERACTION, AUTOMATION, AND HUMAN OVERSIGHT

Amershi, S., Weld, D., Vorvoreanu, M., Fourney, A., Nushi, B., Collisson, P., Suh, J., Iqbal, S., Bennett, P. N., Inkpen, K., Teevan, J., Kikin-Gil, R., & Horvitz, E. (2019). Guidelines for human-AI interaction. Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, 1–13. DOI: 10.1145/3290605.3300233.

Horvitz, E. (1999). Principles of mixed-initiative user interfaces. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 159–166. DOI: 10.1145/302979.303030.

Lee, J. D., & See, K. A. (2004). Trust in automation: Designing for appropriate reliance. Human Factors, 46(1), 50–80. DOI: 10.1518/hfes.46.1.50_30392.

Parasuraman, R., Sheridan, T. B., & Wickens, C. D. (2000). A model for types and levels of human interaction with automation. IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans, 30(3), 286–297. DOI: 10.1109/3468.844354.

This body of work provides important foundations for the paper's distinction between capability and authority. Questions about when systems should act autonomously, when humans should intervene, how automation alters human work, how trust should be calibrated, and how mixed-initiative systems divide responsibility long predate foundation models. Thoughtware extends these questions into software architectures in which machine judgment can itself be decomposed, reused, and orchestrated.

EVALUATION, CALIBRATION, AND RELIABILITY

Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). On calibration of modern neural networks. Proceedings of the 34th International Conference on Machine Learning, 1321–1330.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. (2022). Holistic evaluation of language models. arXiv preprint arXiv:2211.09110.

Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., & Zhu, C. (2023). G-Eval: NLG evaluation using GPT-4 with better human alignment. Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing.

National Institute of Standards and Technology. (2023). Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. Advances in Neural Information Processing Systems, 36.

These works support the paper's argument that cognitive systems require forms of assurance beyond deterministic output comparison. They address multidimensional model evaluation, confidence calibration, automated evaluation using language models, agreement with human preferences, evaluator bias, and lifecycle governance. The paper's concept of Judge CUs builds on this emerging body of work while placing evaluation at the level of explicit cognitive responsibility rather than treating it only as model benchmarking.

POSITION OF THIS PAPER

The concepts Thoughtware, Cognitive Unit, Judgment Terrain, Instruction Surface, Intent Compilation, Submergence Principle, and re-naturalisation, together with the specific architectural relationships proposed among them, are introduced or developed in this paper. The preceding literature provides important foundations and adjacent ideas, but these terms should not be read as established terminology from the cited sources.

The purpose of the references is therefore not to suggest that Thoughtware follows directly from a single existing research tradition. The proposal sits at the intersection

of several: software architecture provides the principles of decomposition and encapsulation; foundation models provide programmable cognitive capability; cognitive science provides useful distinctions around judgment and expertise; human-computer interaction provides a history of allocating responsibility between people and machines; and contemporary evaluation research provides methods for measuring behaviour where deterministic verification is insufficient.

Thoughtware attempts to bring these strands together at the level of software architecture.

Towards building for the intelligence age at scale

Thank You

thoughtware.