AI in practice

Analysis from a software engineering perspective

November 2025

Written by:
Joost de Jong jdjong@sogyo.nl

Edited by:
Glenn Mulder gmulder@sogyo.nl

Version	Date	Authors	Description
1.0	Nov. 03, 2025	Joost de Jong Glenn Mulder	Release
1.1	Nov. 05, 2025	Joost de Jong Glenn Mulder	Added visual summary

Introduction **3**
Beyond Bugs: Understanding Inherent Outcome Reliability (IOR) **4**
Definition 4
The Unfixable Limitation 4
Example: Cat vs. Dog Image Classifier 5
Software components 6
Determinism of the problem space 6
Modelling possibilities of the problem space 7
Reliability of the software executing the model 8
Summary 9
Dealing with unreliable outcomes **10**
Lessons from AI's Past 10
The "Human-in-the-Loop" Approach 10
Optimizing Human Verification 11
Risk and Confidence Thresholds 11
The Imperative of Auditability 11
Statistical Analysis 12
Pattern Matching for Anomaly Detection 13
Strategic Error Trade-Offs: Precision vs. Recall 13
Prioritizing Recall (Minimizing False Negatives) 13
Prioritizing Precision (Minimizing False Positives) 13
Beyond Training: Influencing the Outcome 14
Language Models 15
Beyond Training: The Imperative of Prompt Engineering 15
Self-Correction: An Emergent Capability 15
Consensus-Based Responses (Self-Consistency) 15
Explicit Uncertainty Statements 16
Constrained Generation 17
Architectural Constraints: Moving Beyond Natural Language 17
Automated Feedback Loop for Correct Outcomes 18
The Value of AI-Powered Software Components **20**
Outperforming Humans at Scale 20
Automating Complex Tasks 20
Pattern Discovery 21
The Impact of AI-Powered Software Components on Software Architecture **22**
Latency 22
Role of component: Processer/Transformer 22
Role of component: Router/Decision Engine 23

Role of component: Autonomous Agent/Orchestrator 24
Draft or Decide? Two Roads for AI Integration **25**
The "Draft & Review" Path (Human-in-the-Loop) 25
The "Trust & Execute" Path (Fully Automated) 25
Impact of AI-Powered Software on Software Engineering **26**
Conclusion: Engineering a Reliable Future **28**
Appendix 1: inherent output reliability (IOR) mathematical model **29**
References **30**

AI-Powered Engineering: A Practical Guide

Introduction

Artificial Intelligence is shifting from niche technology to general purpose tooling (Freitag 2025). With large language models (LLMs) available to all and the rise of **Language Model (LM) based AI services**, the challenge for **software engineers and organizations** is no longer *if* they should build with AI, but *how* to do so reliably. Unlike traditional software, AI-powered features introduce a new challenge: the system can be perfect, yet still produce a wrong answer. LLM power is impressive, but delivering dependable, high-quality features requires specific strategies to ensure **outcome reliability** (Wohlin et al. 2003) in production.

Here, we discuss guidelines for building dependable AI-powered software. We start with the concept of **inherent outcome reliability**, dive into practical strategies for imperfect scenarios and techniques to improve performance, and conclude with the **architectural impact** of integrating these components and key considerations for autonomous AI operation.

Beyond Bugs: Understanding Inherent Outcome Reliability

Definition

Inherent outcome reliability represents the **theoretical, upfront probability** that a given software component will produce a valid output for a random, valid input, with a value ranging from 0 to 1.

- A value of **IOR=1** means you are theoretically certain that every valid input will yield a correct output.
- A value of **IOR=0** means you are theoretically certain that every valid input will result in a wrong output.

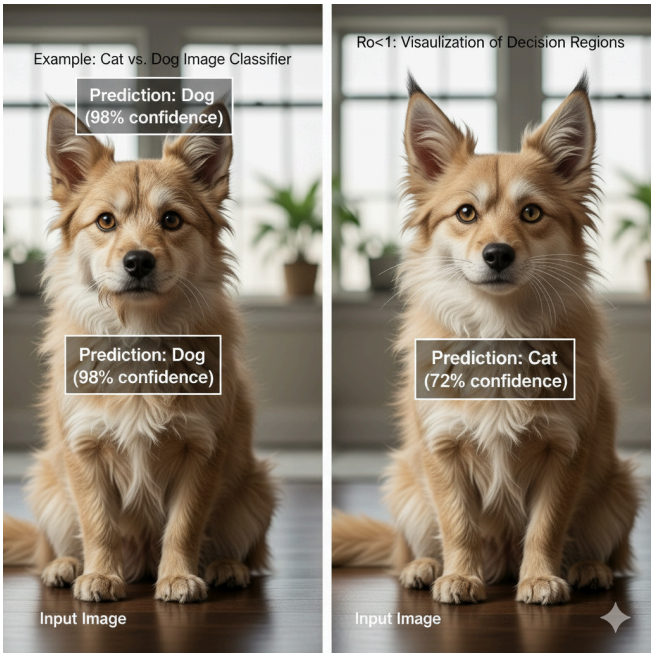
(A more detailed mathematical explanation can be found in appendix 1.)

The Unfixable Limitation

If a software component's theoretical, upfront outcome reliability is less than 1, you can't always trust its output. This is a **fundamental limitation** that cannot be fixed by better programming.

We explicitly use the terms **theoretical** and **upfront** to distinguish this concept from fixable issues like programming bugs, faulty logic, or network failures. In those common failure cases, the resulting output (even if it's an error message) is a **truthful representation** of what happened: the failure is reported honestly. An inherent outcome reliability of less than 1, however, means the output itself, the **final answer**, is an **unintended falsehood**, even if the code that produced it is perfectly implemented.

Example: Cat vs. Dog Image Classifier



Consider a function IsCat(image) powered by a Machine Learning (ML) model (Paleti and Angelou, 2024).

- **Perfect Code, Inherently Imperfect:** The code running the ML model is bug-free. It processes the image and outputs "Cat" or "Not Cat (Dog)" according to the parameterization and training of the underlying model. That is, the implementation is correct.
- **Inherent Limitation:** Even the best ML models are statistical systems. Due to ambiguous images, imperfect training data, or model limitations, there's always a theoretical, non-zero chance it will misclassify an image (Paleti and Angelou 2024).

If IsCat(image) outputs "Cat," there's a small, built-in probability that the image is actually a dog. This isn't a code bug, a hardware problem, or a network error. It's an **unfixable limitation** because the model makes predictions based on probabilities, not absolute certainty.

Therefore, the inherent outcome reliability is less than 1 (e.g., IOR=0.98), reflecting the theoretical chance of a wrong classification, regardless of code quality.

This IOR < 1 reality forces a shift from programming for **correctness** to programming for **trust**.

Software components

A software component, when viewed in the context of **inherent outcome reliability**, consists of three dimensions, mirroring the traceability path in system development (e.g., as found in **Software Product Line Engineering** (Pohl, Böckle, and Linden 2011)):

- The **problem space**: The real-world context you're trying to solve.
- The **model**: The abstraction of the problem space.
- The **software**: The code that executes your model.

Let's examine how each of these contributes to a component's inherent outcome reliability.

Determinism of the problem space

The problem space can be either **deterministic** or **non-deterministic**.

- In a **deterministic** problem space, a given input always produces the same output. For example, in a cash register system, the total for items costing $10 and $12 will always be $22.
- In a **non-deterministic** problem space, the same input can produce different, yet valid, outputs (Udawatta et al. 2008). For example, a game of Yahtzee produces different dice rolls each time, but each roll is a valid outcome.

Crucially, the determinism of the problem space itself does **not** affect inherent outcome reliability. If all possible outputs are valid, inherent outcome reliability is 1. This makes sense: a system that accurately reflects reality is, by definition, reliable.

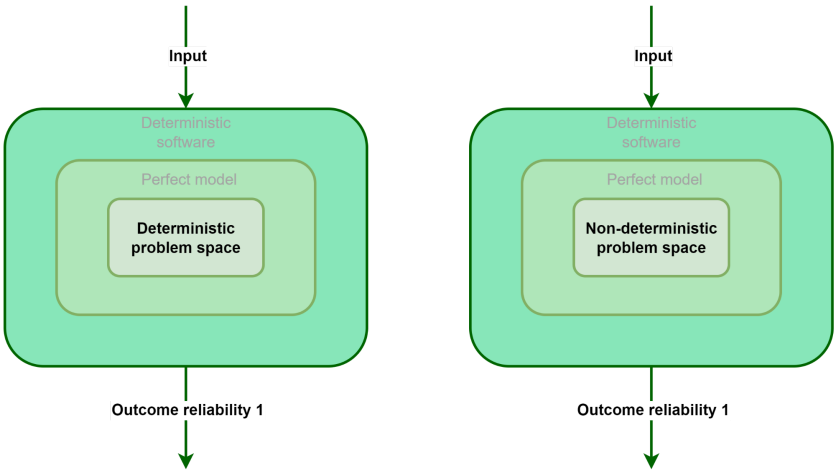


Figure 1. Deterministic vs. Non-deterministic problem space in relation to inherent outcome reliability

Modelling possibilities of the problem space

This is where things get interesting. Even if a problem space is deterministic, it might be impossible to create a perfect model for it. Consider **fraud detection** (Lindemulder and Kosinski 2024). A transaction is either fraudulent or it is not: a deterministic reality. However, creating a perfect model is impossible because fraudulent transactions are designed to look normal. You can't know for sure if a transaction is fraudulent without a full human investigation.

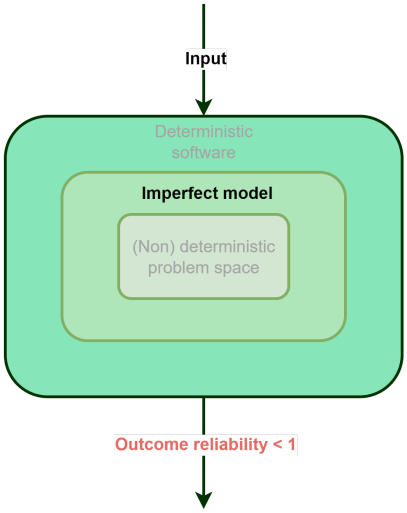


Figure 2. Imperfect modeling capabilities in relation to outcome reliability

This inability to create a perfect model for the problem space leads to an inherent outcome reliability of less than 1. The **Cynefin framework** (Snowden 1999) can help determine if your problem space is suitable for perfect modeling. In the Cynefin framework, when a problem space resides in the **Simple** or **Complicated** domains, creating a perfect model is theoretically possible (Mamonov 2023).

However, even in the **Complicated** domain, achieving a *perfect* model is often impractical. While cause-and-effect relationships are discernible and solvable by experts, the number of variables or costs and time required to map every factor can be prohibitive. As a result, complicated domains often rely on **imperfect** (yet practical) **models** that balance accuracy with feasibility.

In contrast, when the problem space moves into the **Complex or Chaotic** domain (as with dynamic financial fraud detection) the system deals with unknowable factors, meaning a perfect model is **fundamentally impossible** (Mamonov 2023).

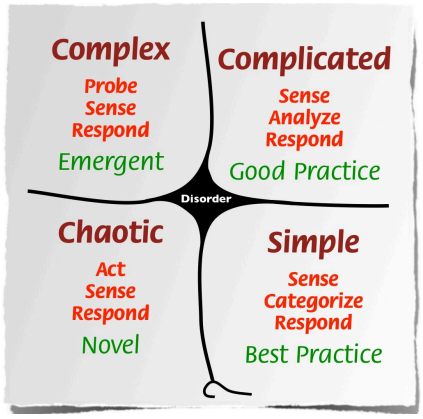


Figure 3. Cynefin framework

Reliability of the software executing the model

Traditional software execution is **dependably robust**: Modern systems and code reliably translate programming instructions into predictable, correct outcomes that are **infinitely repeatable**. Software itself does not contribute to the decay of this inherent reliability.

The challenge to reliability emerges when incorporating **Artificially Intelligent (AI) components**. Unlike conventional, AI models operate by **prediction**, leading to outputs that can be **non-compliant** or incorrect relative to the task's initial intent. In this way, leveraging AI introduces a **stochastic element** to the software's execution, thereby reducing the component's overall **outcome predictability and reliability**.

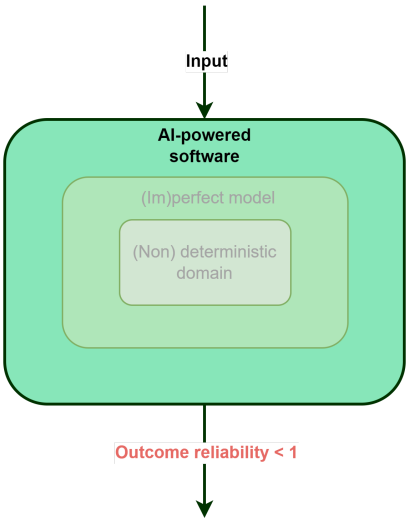


Figure 4. AI-powered software in relation to outcome reliability

You can improve outcome predictability and reliability by using techniques to ensure the outputs are valid. In some cases, you can even bring inherent outcome reliability back to 1. We'll cover that later.

Summary

In short, a software component's **inherent outcome reliability** is influenced by two factors:

1. **Ability to model the problem space.** (Refer to the Cynefin framework.)
2. **Reliability of the software executing the model.** (Especially with the use of probabilistic technologies like LLM's (Vaswani et al. 2017).)

The problem space itself has no impact on reliability. This diagram illustrates how these factors combine to determine a software component's inherent outcome reliability:

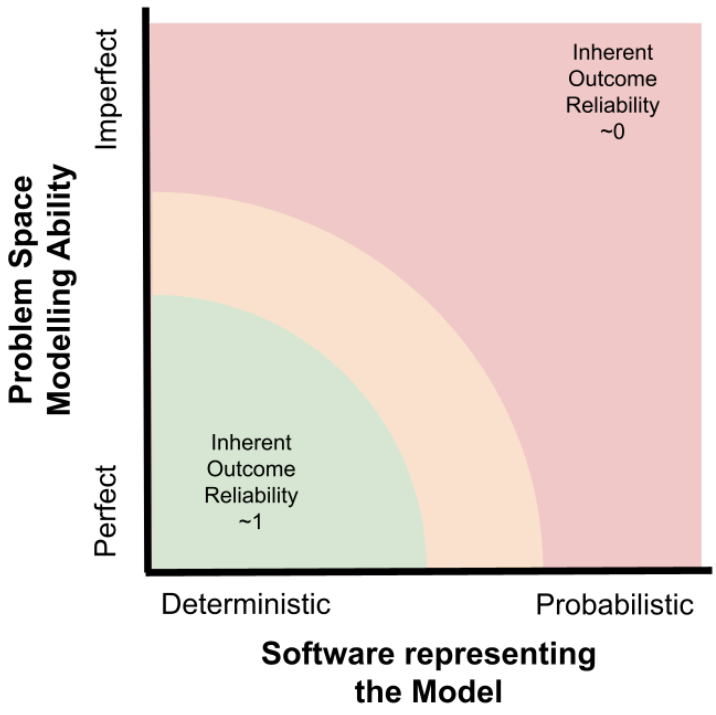


Figure 5. A software component's inherent outcome reliability factors

Dealing with unreliable outcomes

Today, **LM based AI services** create new opportunities for organizations. As discussed in the previous section, this introduces software components with an **inherent outcome reliability of less than one**, meaning they will inevitably produce **incorrect outputs**. Therefore, organizations and software engineers leveraging LM based AI services must acknowledge this fundamental uncertainty and develop strategies to manage it.

Lessons from AI's Past

Fortunately, coping with unreliable outputs is not a new challenge in software engineering. We can draw valuable lessons from domains where AI has been used for years, such as **spam and fraud detection** or **speech and image recognition**. These established fields offer proven methods for managing unpredictable, probabilistic results.

The "Human-in-the-Loop" Approach

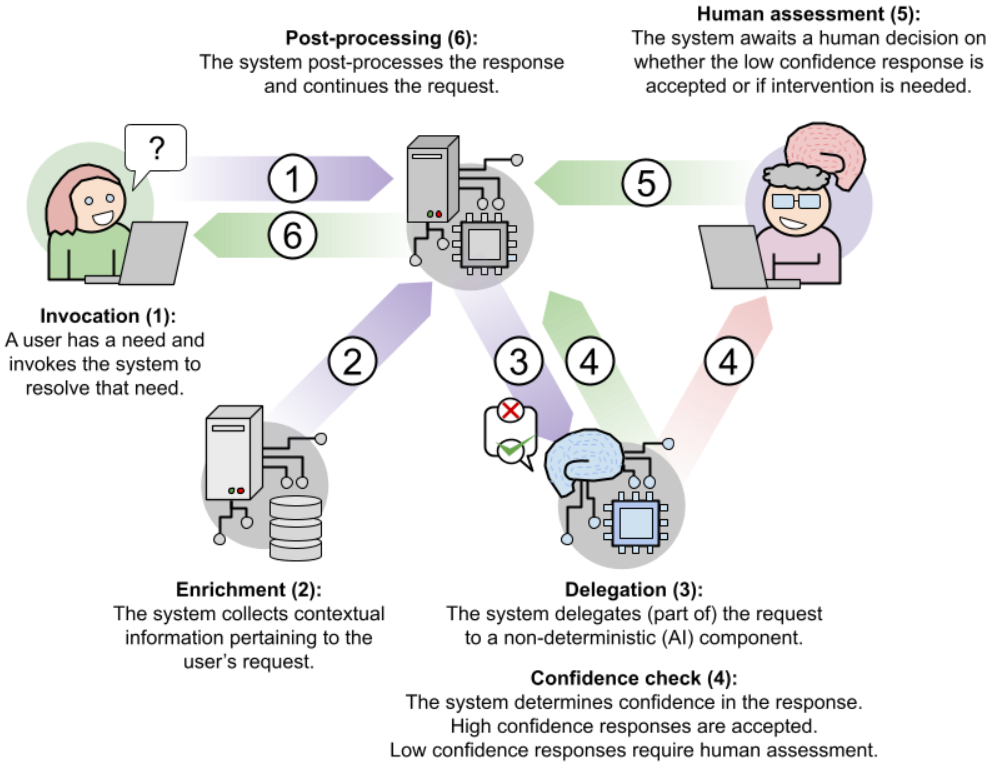


Figure 6. Human in the loop (HITL)

The most straightforward way to manage unreliable outcomes is to introduce a human check: the **"Human-in-the-Loop" (HITL)** approach (Beetroot 2025).

For example, consider an AI **coding agent** used to generate a software codebase. In this scenario, HITL means a **competent software developer** is actively involved, **steering the model with well-drafted requirements and precise prompts.** The developer uses their expertise to **validate** that the generated code is not only functional but also meets critical engineering standards, ensuring it is **simple, maintainable, validated, and valuable** to the project.

Similarly, in **fraud detection,** an AI system might flag a transaction as 'Suspicious' (high risk), but a human analyst still reviews the full account history before definitively blocking the transaction, preventing a false positive for a legitimate customer.

In contrast, simply accepting AI outputs without review or constraints could be defined as **"vibe-coding pur sang."** This approach, where unvalidated results are blindly integrated, represents the highest level of risk acceptance, betting on unverified code and accepting the high probability of future technical debt or system failure (Visibee 2025).

Optimizing Human Verification

In many cases, universal verification is not feasible. In those cases, organizations can opt for **sampling**, verifying only a portion of the outcomes. Sampling is most often applied with **risk-based approach**:

- **High-Risk Outcomes:** A business might mandate human review for any outcome where the potential negative impact is too great, such as a financial transaction involving a significant sum of money. By focusing verification efforts on these high-stakes results, we can maximize the value of the human effort.

Risk and Confidence Thresholds

Risk assessment can be further optimized by incorporating **confidence scores** for the AI's output (Chutani 2024). Organizations can establish formal protocols, practices and **confidence score thresholds** that define the point at which the consequences of an incorrect output are deemed acceptable without human intervention to fine-tune the balance between automation and oversight.

The Imperative of Auditability

For the Human-in-the-Loop technique to function effectively, and to earn user trust, the entire process must be **auditable** (Uday 2025). This isn't merely about regulatory compliance; it's essential for **debugging**, **improving system performance**, and **maintaining long-term stakeholder trust** in the AI system.

Alternatively, **statistical analysis**, which focuses on monitoring the component's **performance over time** rather than validating specific outputs, can be used to deal with low outcome reliability.

Performing statistical analysis on complex, multi-step systems, like an **agentic AI** service, is difficult. Because an agent's reasoning, tool calls, and overall process are non-deterministic, validating a single run is often impossible. The solution is to use **statistical sampling**, evaluating a small, but representative, sample of outputs using a **human-in-the-loop** approach and **automated evaluation** techniques. This carefully selected, high-quality dataset then serves as a statistically valid proxy for the overall system performance.

If you establish a baseline **accuracy** or **success rate** of 90%, you can monitor this metric constantly and automatically compare it to these established values. If the monthly sampled correctness level drops to 80%, this is a clear sign that something is fundamentally wrong. This dip in performance could be caused by **data drift** (Martyr 2025), a change in statistical properties of your input, or **concept drift** (Stankevichus 2025), a change in the relationship between your input and output.

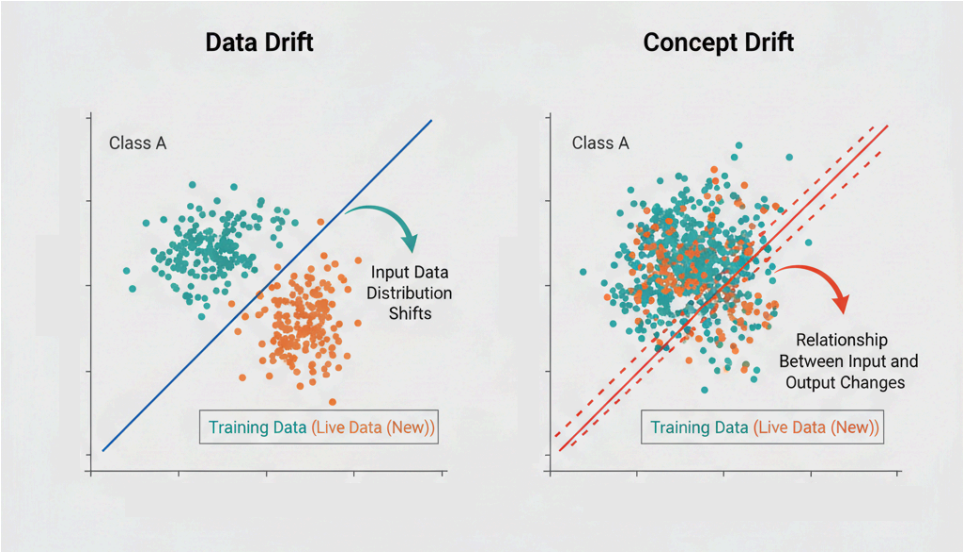


Figure 8. Data drift, input data changes. Concept drift, the meaning of the output changes.

By catching these performance trends early, you'll know exactly when and how to intervene. For instance, a drop caused by **data drift** usually necessitates **fine tuning the model with new data,** using a new version of the model or choosing another model altogether. In contrast, concept drift might demand a deeper adjustment to the system's underlying logic or rules to maintain reliability.

Pattern Matching for Anomaly Detection

A core component of this statistical monitoring is **pattern matching**, which is used to identify and quantify the frequency of specific output patterns, both correct and incorrect, to find anomalies and deviations. This involves defining an expected output pattern (a sequence of events, a data structure, or a predefined set of relationships) and comparing the component's output against it.

By looking for specific deviations from this expected pattern, statistical methods can effectively detect and categorize performance drops like data drift or concept drift, allowing engineers to focus their interventions precisely where the system is failing (Onsem et al. 2022).

Strategic Error Trade-Offs: Precision vs. Recall

A key part of dealing with unreliable outcomes is determining which type of error is more acceptable: **false positives** or **false negatives**. This choice is often framed as a strategic trade-off between **precision** and **recall** (KeyLabs 2024). Precision is the fraction of detections that represents a true case while recall is the fraction of true cases that were detected. The former tells us how many wrong detections we get. The latter tells us how many of our cases are not detected. See figure 9.

Prioritizing Recall (Minimizing False Negatives)

In high-stakes scenarios like fraud detection, you must prioritize **Recall** (sensitivity), which means catching every possible positive instance. In the case of fraud, this minimizes **false negatives**, i.e. cases where a fraudulent transaction is missed. The trade-off is an increase in **false positives**, legitimate transactions incorrectly flagged as fraud. While these require costly manual review, the benefit of preventing massive financial risks outweighs this inconvenience.

Prioritizing Precision (Minimizing False Positives)

Conversely, in systems where a false alarm causes disruption or distrust, you must prioritize **Precision** (positive predictive value). This minimizes **false positives**. Consider your AI coding agent example: if the system constantly flags correct, compliant code as 'High Risk' (a false positive), the developer will waste time investigating or, worse, begin to ignore the warnings altogether, eroding trust in the system. Here, you accept a few more **false negatives** (a small amount of faulty code is missed) in exchange for high confidence in the agent's warnings, allowing the developer to work efficiently. Furthermore, false negatives can be caught by the expert developer further diminishing the impact of misclassification.

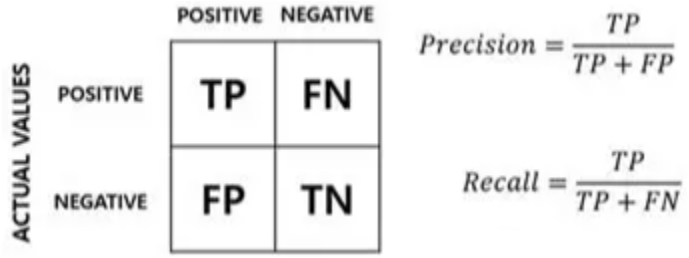


Figure 9. Confusion matrix, precision vs. recall

An organization should balance Precision and Recall strategically, using responsible risk and confidence thresholds, to determine the operational constraints that best support the ultimate business goal. Such constraints guide both the configuration and selection of an appropriate model for the LM based AI service.

Beyond Training: Influencing the Outcome

We have explored the critical operational strategies for **managing** AI's inherent unreliability: **Human-in-the-Loop**, **Statistical Analysis**, and **Strategic Error Trade-Offs**. These methods treat the AI model as a black box whose outputs must be externally controlled and audited.

For organizations leveraging modern, **LM based AI services** (like those used for coding, content generation, or summarization), modification of the underlying model through traditional machine learning **training** or fine-tuning is not cost-effective, for it requires expensive expertise and hardware.

Instead, the focus shifts from improving the model to **choosing a model** and **improving the interaction**. The reliability of these complex, often agentic, systems relies heavily on how we design the calls to the service itself. This requires a new set of techniques to steer its reasoning and output to achieve a more reliable result.

The next section explores these methods.

Language Models

Generative AI models, especially Large Language Models (LLMs), enable new software approaches. Their inherent outcome reliability can be improved with **self-correction**, **consensus-based responses**, and **explicit uncertainty statements**. This allows software components to leverage LM based AI services in robust and predictable ways.

Beyond Training: The Imperative of Prompt Engineering

For organizations using third-party LM based AI services, focus shifts from improving the model to **Prompt Engineering** (AI prompt theory 2025), a new set of skills that focuses on guiding the model to achieve higher reliability.

Self-Correction: An Emergent Capability

When you get an undesired or incorrect output from an LLM, a common approach is to re-prompt the model with different phrasing to get the output you want. For example, the prompt “I want this .csv in my database” results in an undesired result while the prompt “Transform the file.csv file to a SQL INSERT statement” does. This technique is effective because natural language offers many ways to express the same intent (SaaS Prompts 2024).

In addition, you can prompt an LLM to "critique your previous answer" or "explain what's wrong with it." The model uses its vast knowledge of language and its emergent reasoning abilities to simulate a review process and generate a corrected version. Such "self-correction" is fundamentally new: It's not just about a user trying again. It's a linguistic and reasoning-based capability of the model itself.

This technique is often implemented using a **Reflection/Refinement** pattern (Kolavi 2025), where a system feeds an initial output back to the model, along with a set of constraints or a validation function, and explicitly asks it to check and revise its own work. For example, imagine the prompt “Transform the file.csv file to a SQL INSERT statement” generates an initial script. A reflection prompt would then feed that script back to the model, asking: "You just generated this SQL script. Please review it. Are there any potential SQL injection vulnerabilities? Does it correctly handle the header row? Provide a revised, more secure version." This forces the model to apply a new read of constraints (security, accuracy) to its own previous output, which is the core of self-correction.

Consensus-Based Responses (Self-Consistency)

In the previous section, we emphasized the value of statistical analysis and sampling to validate an AI system's performance. Prompt engineering offers a direct way to apply this concept to a

single output: **Consensus-Based Responses**, often called **Self-Consistency** or **Ensemble Inference** (AI prompt theory 2025).

Instead of relying on a single, deterministic output, this technique involves running the same prompt multiple times, either on a single model with slightly different decoding parameters (like temperature or top_p), or across **multiple AI models**. Each query thus generates a distinct "reasoning path" and a final answer. The system then gathers this sample of outputs and takes a **majority vote**, selecting the answer that appears most frequently across all runs.

This method effectively turns the black box into a probabilistic ensemble: by leveraging diverse reasoning paths, we increase the probability that the final consensus answer is correct. This approach mitigates the risk of incorrect "hallucination." For systems that use **Constrained Generation** (like a 'yes' or 'no' output), this voting becomes trivial, allowing organizations to set high acceptance thresholds, such as requiring an 80% consensus.

Explicit Uncertainty Statements

To tie the Generative AI process back to the **Risk and Confidence Thresholds** we established earlier, LLM infrastructures and specific models provide **metrics** that can be utilized to construct an **Explicit Uncertainty Statement**.

By instructing the model to output a confidence score or to explicitly state its level of certainty (e.g., "I am 95% confident in this generated code structure"), the software system gains a critical piece of metadata (Grandperrin 2021). This allows an organization to automate the **Human-in-the-Loop** decision:

- **If Confidence > 90%:** Automatically accept the output.
- **If Confidence ≤ 90%:** Route the output to a human reviewer for validation.

Confidence Score	Field	Value
0.32	date	2023-10-06
0.96	name	Science Fair
1.00	participants	["Alice", "Bob"]
Total		

Figure 10. Example of confidence scores in action.

This capability transforms the LLM from a simple answer machine into a risk-aware component, allowing software engineers to build intelligent decision layers around the generative process.

Beware that you do not ask the model for a confidence score by prompting. Confidence scores are to be obtained from the model metrics.

Constrained Generation

Large language models (LLMs) are incredibly versatile. They're great at generating natural language, but they can also be guided to follow specific rules for their output (Docherty 2025).

When you don't give an LLM any special instructions, it will produce unstructured text in the form of natural language, but by adding details about how the output should be formatted and what values are possible, we can limit the output to a small set of options, such as a 'yes' or 'no' choice. This increases the chance of getting a correct and predictable answer. It may also facilitate automatic validation of the answer.

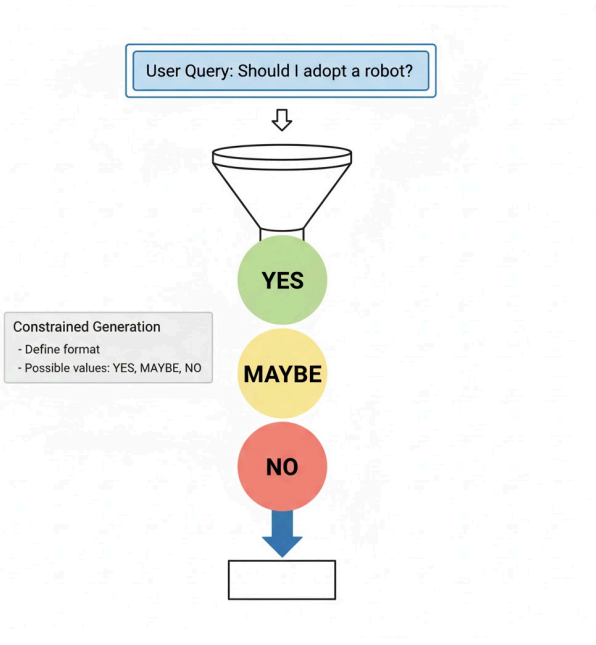


Figure 11. Simple constrained generation.

Architectural Constraints: Moving Beyond Natural Language

While clever prompting works well for simple constraints (like yes/no answers), relying solely on natural language for complex output can still lead to unreliable formatting.

The new, architecturally sound way to implement constrained generation is by using **specific technical features offered by modern LLM APIs** (like those from OpenAI, Anthropic, or Google (Google 2025)). These features enforce the structure *before* the model generates the output, making the process far more reliable:

1. **JSON Schema Enforcement:** You can provide the API with a strict **JSON Schema** (defining required keys, data types, and possible enumeration values). The model is then forced to generate a valid JSON object that strictly adheres to that schema, eliminating common formatting errors.
2. **Function Calling:** You can define a specific software function's signature and instruct the model to output the arguments necessary to call that function. This forces the model to generate reliable, structured data that a software component can immediately ingest and execute.

These technical methods make constrained generation a core feature of robust software architecture, ensuring that the LLM's output is not just human-readable, but machine-consumable.

Automated Feedback Loop for Correct Outcomes

When a software component's context allows for automated **verification of an AI model's output**, that verification result can be used as input for a retry. One great example of this is **code refactoring** ("About GitHub Copilot coding agent" 2025).

Here's a possible workflow:

- **Ask an AI model to refactor some code.** To prevent the LLM from 'ballooning' the codebase, the initial prompt must enforce strict scope constraints (e.g., 'Only refactor the methods within file.py related to user authentication, and do not change any unit test files).
- **Apply the AI's output.** The new code is integrated into the project.
- **Run code compilation.** If compilation errors occur, the AI is asked to fix the build by being provided with the specific errors.
- **Run existing unit tests.** If tests fail, the AI is asked to correct the code by being given the failed assertions.
- **Run other automated code quality tools.** If these tools flag issues, the AI is prompted to fix the code quality problems.

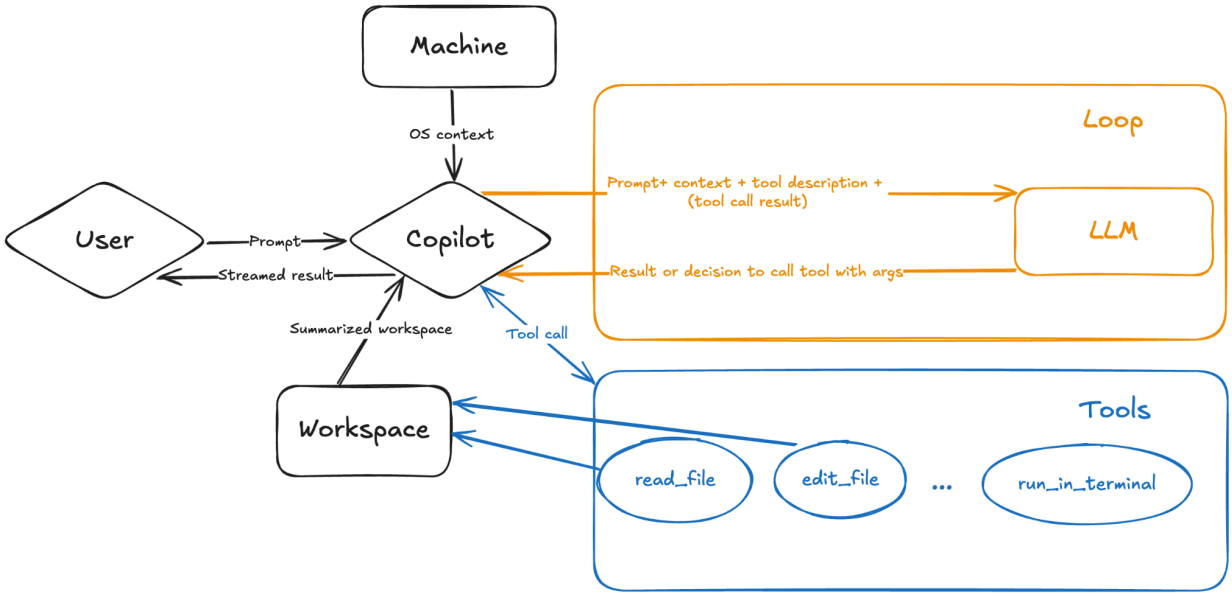


Figure 12. GitHub Copilot agent architecture.

Once the checks pass *and* automated quality metrics (e.g., reduced Cyclomatic Complexity, passing Linter scores) are met, the **desired outcome** is achieved. This combined validation ensures the code is not only **correct** but also **desirable, readable, and maintainable.**

This principle applies to many other fields where model output can be automatically verified:

- **Content Generation:** An AI writes a product description. An automated script checks for key features and grammar. If a check fails, the AI gets the feedback to revise.
- **Computer-Aided Design (CAD):** An AI designs a part. Simulation software automatically tests it for structural integrity, providing feedback to the AI to refine the design.
- **Data Entry:** An AI extracts data from an invoice. An automated system checks the extracted information against a database. If there's a mismatch, the AI is prompted to re-evaluate the document.

The ability to **validate AI model output automatically and immediately** within a software component is what makes the component **agentic**. Other techniques can achieve validation too, but often result in a higher rate of incorrect outcomes and require frequent human intervention.

The Value of AI-Powered Software Components

We've explored how to handle and improve the reliability of LM based AI service outcomes, but why build software with a degree of unreliability in the first place?

The answer is simple: AI services offer value, even with a degree of unreliability, in three key areas: **outperforming humans at scale**, **automating complex tasks**, and **pattern discovery**.

The power of general-purpose AI models is their ability to generate outcomes that are **mostly** correct. While you can't assume raw LLM output is accurate, it is usually close to the desired result (Precision Drafters 2025).

Outperforming Humans at Scale

AI tools can complete tasks faster than human agents, despite higher error rates, due to the sheer volume of data they can process (Smajic 2024). The risks of errors are offset by the ability to process workloads that are impossible for human teams to manage. This can lead to expanding services and faster service delivery without requiring a larger workforce.

Automating Complex Tasks

AI also enables automation in situations where clear, step-by-step workflows do not exist, or when dealing with unstructured data (UiPath 2025). For example, a user interacting with a chatbot might make a complex request in natural language, which the chatbot can process and, if given access to the right tools, satisfactorily resolve without a pre-defined script.

Similarly, many workflows involve decisionmaking based on unstructured information, like a customer service representative deciding how to route an email based on its content. AI can now read incoming customer support emails and automatically tag them as a "billing issue," "technical problem," or "general inquiry," then route them to the right team.

In these scenarios, AI can handle a vast number of requests, freeing up human workers to focus on nuanced cases that require deeper expertise.

AI components go beyond automating existing tasks; they introduce new ways for users to interact with software (Gison 2025). A customer searching an e-commerce site might use an AI assistant that dynamically filters products and generates a **personalized recommendation** based on browsing history and stated intent. Applications that use generative AI can propose various designs or draft texts from a single prompt, providing the user with creative starting points.

Pattern Discovery

The final value of AI components is their ability to reveal correlations and associations that were previously **invisible** or **unknowable**. This value operates on two distinct but complementary levels (scitechtalk tv 2024).

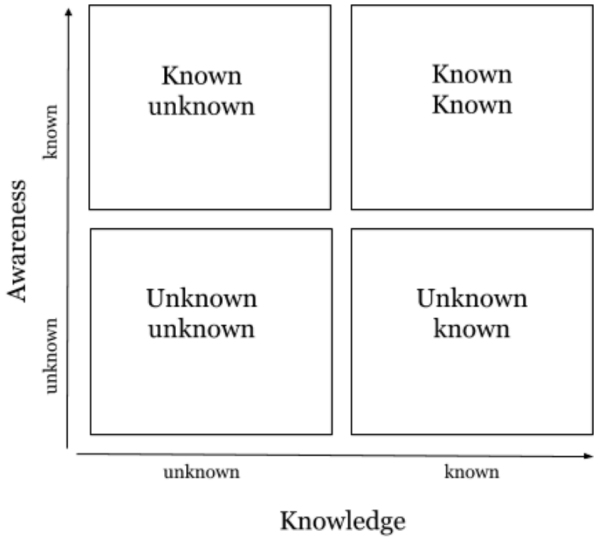


Figure 13. Rumsfeld Matrix.

First, AI functions as a sparring partner for **human reasoning**. A team can evaluate the current best solution, strategic plan, or diagnosis with AI, instantly cross-referencing it with historical outcomes and established patterns. Such use of AI models can highlight **blind spots**, identify **missing variables**, or suggest **alternative solutions** that the human group, constrained by their own experiences or cognitive biases, may overlook.

Second, AI can function as a **discovery engine**. In complex scientific fields, the volume, velocity, and dimensionality of data exceeds human capacity for analysis. AI systems excel at discerning subtle **patterns, correlations, and anomalies**. By highlighting hidden connections, these tools shift our focus from **automating the known** to **discovering the unknown**.

Ultimately, leveraging these AI components isn't about achieving perfection, it's about creating systems that permit human ingenuity to focus where it matters most.

The Impact of AI-Powered Software Components on Software Architecture

Having discussed the value of LM based AI services and the methods for dealing with unreliable outcomes, we now examine their architectural impact, which varies significantly based on the component's task (Safe Software, publish year unknown).

Latency

AI models are slow regarding response times compared to non AI-powered software components. Therefore, it is practically unavoidable to make workflows that use such tools asynchronous. Luckily, there are many described patterns for integration with asynchronous components.

Role of component: Processer/Transformer

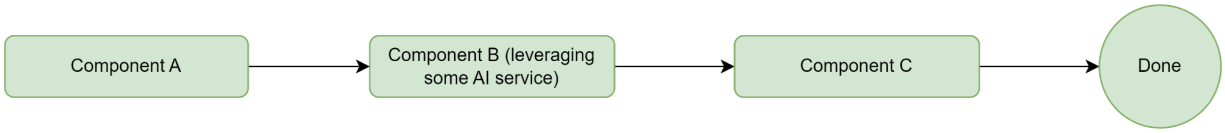


Figure 14. Simple, fixed workflow.

We define a **Processer/Transformer** as an AI-powered software component that acts like a specialized, non-deterministic function call. It's often integrated via a simple API call as a step in an existing, fixed pipeline (e.g., summarizing text, classifying a ticket).

For example, a system uses an LLM to take customer input and provide a summary (output) before a human agent reviews it. The architectural impact here is minimal in terms of flow but critical in terms of performance: the architecture must now account for new **latency concerns** and introduce **retries or fallbacks** inherent to the AI model's execution time.

Role of component: Router/Decision Engine

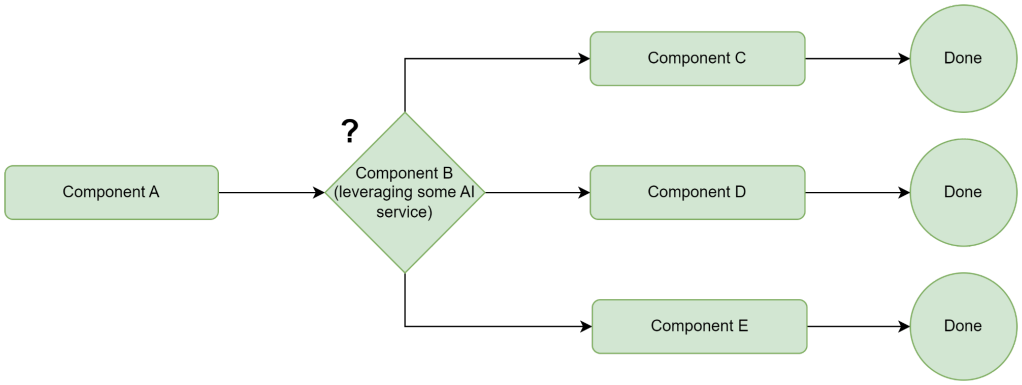


Figure 15. Fixed workflow, but decision made by AI component.

The **Router/Decision Engine** uses AI to introduce a new layer of dynamic routing. Instead of a fixed path, the system needs a more complex interface to handle the AI's probabilistic decision (e.g., a specific workflow ID) and a mechanism to fall back if the decision's confidence is low.

For example, a system uses an LLM to analyze a customer service ticket (input) and decides whether to route it to the Billing Workflow, Technical Support, or Sales. The architecture must include a **Confidence Handling** mechanism (e.g., rerouting to a human queue or a default workflow) and often a new **Context Provider Layer** to feed necessary state information to the AI for its decision-making.

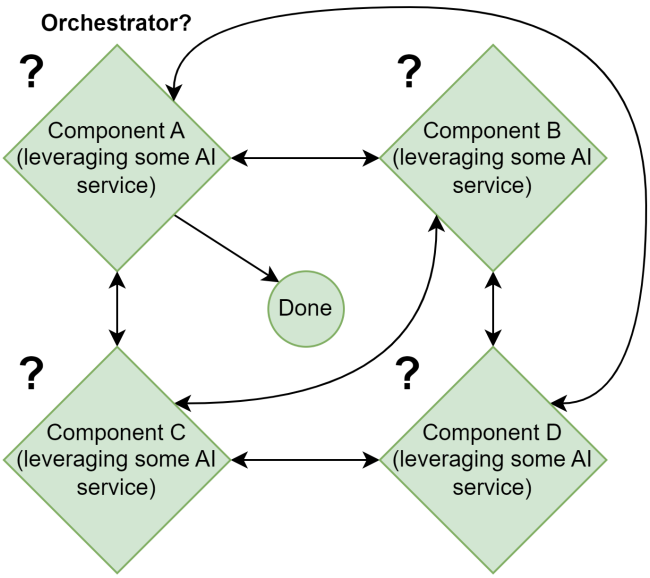


Figure 16. No predetermined workflows; workflows emerge from AI decisions.

This is the most complex role, often leveraging prompt engineering and techniques like Tool-Use or Function Calling. The impact on the software architecture is significant, moving from **fixed, imperative workflows** to a **dynamic, goal-driven** architecture.

The core architecture must now include a **Tool/Resource Abstraction Layer** (to expose external APIs as "tools" to the AI) which must incorporate **robust access control and security protocols**. This is typically done by leveraging MCP (Anthropic 2024). Furthermore, the ability to execute the multi-step plan the AI generates is essential for **Auditability**, as the AI's "thought process" and chosen tools must be meticulously logged.

For example, a user asks, "Book a flight and tell me the weather." The LLM (the Orchestrator) decides to use the Flight Booking API (tool) first, and then the Weather API (tool), dynamically creating a two-step workflow to fulfill the request.

Draft or Decide? Two Roads for AI Integration

In summation: When positioning an AI-powered component inside your software architecture, you must fundamentally decide how much to trust its output. This decision defines your entire workflow and separates your integration strategy into two clear paths (Zhou 2025):

The "Draft & Review" Path (Human-in-the-Loop)

This path uses the AI component to **draft an outcome**, and then a human is required to validate or decide the next step. You're using the AI for efficiency gains, but your system **does not trust its outcome enough to automate the next step.** This is the default approach for sensitive or high-variability tasks. As discussed, automatic verification mechanisms can permit higher throughput solutions, requiring human intervention only in a subset of nuanced or sensitive situations.

The "Trust & Execute" Path (Fully Automated)

This path **trusts the outcome** and automatically continues the workflow. This is only viable when the AI's output reliability is exceptionally high, achieved by combining a high-quality base model with extensive mechanisms like **guardrails, validation layers, and confidence checks.** You are essentially treating the AI's output as a reliable, non-deterministic input that is safe to act upon.

The Heuristic: Is your AI component a *Drafting Assistant* or a *Go Button*?

The decision between a "Drafting Assistant" and a "Go Button" isn't about the AI's compute power; it's **purely about the systemic risk** you are willing to accept. Choosing wisely is paramount, as this single decision dictates the entire architecture of your fallbacks, audit logs, and confidence mechanisms.

Impact of AI-Powered Software on Software Engineering

Most of us operate in contexts where **inherent outcome reliability is 1**. A component's **real** outcome reliability is always less than one, due to (in theory) fixable bugs, imperfect models, network failures, et cetera. However, a deterministic core is the default.

The ability for organizations and engineers to leverage LM based AI services fundamentally changes how we must think about software. We must now *all* adopt strategies to cope with non-determinism because inherent outcome reliability will be *less than 1* when leveraging them.

Taking previous sections into account, this comes down to understanding our **area of influence**. Most organizations won't be training their own foundational models due to the immense cost of computational resources and expertise required. We are, for the most part, **consumers**: since we cannot change the models, our control is limited to what happens *before* and *after* the model does its work.

Pre-processing: This is our **input strategy**. It's how we prepare and structure data, craft effective prompts, and provide the right context for the LM.

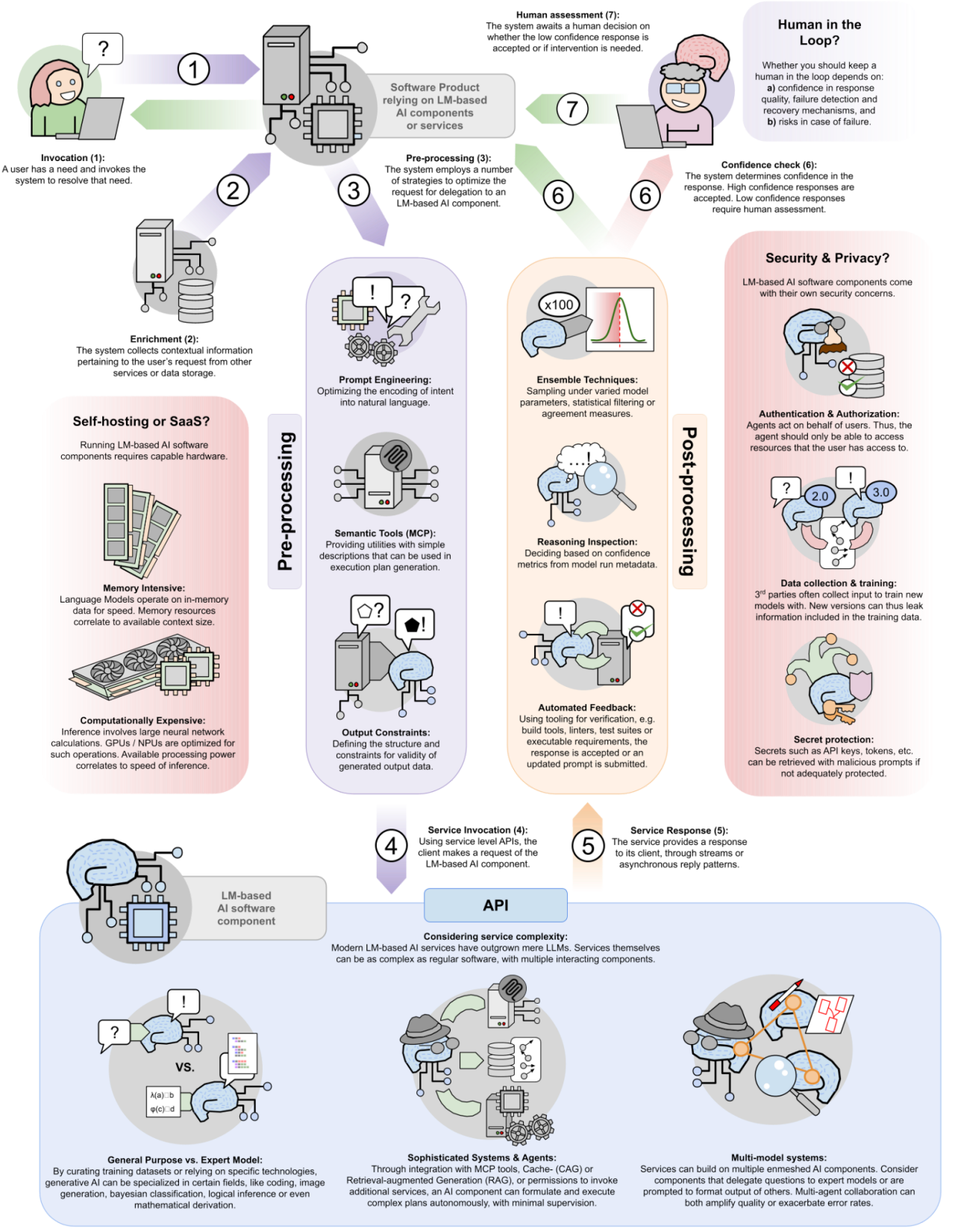
Post-processing: This is our **output-handling strategy**. It's how we validate responses, handle potential errors, extract the useful parts and decide on the next step in our workflow, using the techniques we discussed previously.

With this in mind, we can define an "LM based AI service" as any service that uses at least one LM, which we interact with through pre-processing and post-processing strategies.This definition covers a wide spectrum of tools, which we can separate into three broad groups:

- **Raw Models:** A simple API call to a raw LM. We send a prompt, perhaps tweak parameters like temperature, and get a text response that we must then process.
- **Sophisticated Services:** A more advanced service built *around* an LM, adding features like memory, augmented generation (e.g., RAG or CAG), multi-modal capabilities (handling images or audio), or other agent-like behaviors.
- **Composite Systems:** A multi-agent setup where multiple sophisticated (LM-based) services interact. Even here, the principle holds: we provide an initial input (pre-processing) and handle the final output (post-processing), even if the service itself is a complex composition of many components.

Ultimately, no matter how simple or complex the "black box" is, our job as engineers that build integrations with such services remains the same: we **control the inputs** and we **rigorously manage the outputs.**

This entire relationship can be visualized as follows:


Invocation (1):
A user has a need and invokes the system to resolve that need.
Software Product relying on LM-based AI components or services
Human assessment (7):
The system awaits a human decision on whether the low confidence response is accepted or if intervention is needed.
Human in the Loop?
Whether you should keep a human in the loop depends on:
a) confidence in response quality, failure detection and recovery mechanisms, and
b) risks in case of failure.
Enrichment (2):
The system collects contextual information pertaining to the user's request from other services or data storage.
Pre-processing (3):
The system employs a number of strategies to optimize the request for delegation to an LM-based AI component.
Confidence check (6):
The system determines confidence in the response. High confidence responses are accepted. Low confidence responses require human assessment.
Self-hosting or SaaS?
Running LM-based AI software components requires capable hardware.
Memory Intensive:
Language Models operate on in-memory data for speed. Memory resources correlate to available context size.
Computationally Expensive:
Inference involves large neural network calculations. GPUs / NPUs are optimized for such operations. Available processing power correlates to speed of inference.
Pre-processing
Prompt Engineering:
Optimizing the encoding of intent into natural language.
Semantic Tools (MCP):
Providing utilities with simple descriptions that can be used in execution plan generation.
Output Constraints:
Defining the structure and constraints for validity of generated output data.
Post-processing
Ensemble Techniques:
Sampling under varied model parameters, statistical filtering or agreement measures.
Reasoning Inspection:
Deciding based on confidence metrics from model run metadata.
Automated Feedback:
Using tooling for verification, e.g. build tools, linters, test suites or executable requirements, the response is accepted or an updated prompt is submitted.
Security & Privacy?
LM-based AI software components come with their own security concerns.
Authentication & Authorization:
Agents act on behalf of users. Thus, the agent should only be able to access resources that the user has access to.
Data collection & training:
3rd parties often collect input to train new models with. New versions can thus leak information included in the training data.
Secret protection:
Secrets such as API keys, tokens, etc. can be retrieved with malicious prompts if not adequately protected.
Service Invocation (4):
Using service level APIs, the client makes a request of the LM-based AI component.
Service Response (5):
The service provides a response to its client, through streams or asynchronous reply patterns.
LM-based AI software component
API
Considering service complexity:
Modern LM-based AI services have outgrown mere LLMs. Services themselves can be as complex as regular software, with multiple interacting components.
VS.
General Purpose vs. Expert Model:
By curating training datasets or relying on specific technologies, generative AI can be specialized in certain fields, like coding, image generation, bayesian classification, logical inference or even mathematical derivation.
Sophisticated Systems & Agents:
Through integration with MCP tools, Cache- (CAG) or Retrieval-augmented Generation (RAG), or permissions to invoke additional services, an AI component can formulate and execute complex plans autonomously, with minimal supervision.
Multi-model systems:
Services can build on multiple enmeshed AI components. Consider components that delegate questions to expert models or are prompted to format output of others. Multi-agent collaboration can both amplify quality or exacerbate error rates.

Conclusion: Engineering a Reliable Future

Building and using LM based AI services presents a unique challenge: embracing the inherent unpredictability of these tools. As we've explored in this guide, achieving outcome reliability in AI systems isn't about eliminating every error, but about strategically managing them. We've seen that the trustworthiness of an AI component is influenced by our ability to model the real-world problem and the non-deterministic nature of the AI itself. **This strategic management begins with understanding the system's inherent outcome reliability, the theoretical upfront probability of a correct result.**

By applying time-tested engineering practices like the human-in-the-loop and statistical analysis, and leveraging techniques like self-correction and ensemble methods, we can create robust systems that account for less-than-perfect outcomes. These strategies allow us to not only mitigate risks but also unlock significant value, culminating in the critical architectural choice: treating the AI component as a **Drafting Assistant** or a **fully automated Go Button**.

Ultimately, the goal of building with LM based AI services is not to achieve 100% accuracy in every single instance. Instead, it's about accepting a degree of unreliability in exchange for gains in scalability, and the ability to solve complex, unstructured problems. LM based AI services aren't a silver bullet, but by understanding its limitations and designing our systems to be resilient, we can build smarter software than ever before.

As you embark on your own LM based AI service development journey, remember that reliability is not a feature you bolt on; it's a core principle you engineer into your solutions and architecture from the very beginning.

Appendix 1: inherent output reliability (IOR) mathematical model

Given a single fixed input (x), when feeding it to a software component (function f), the output (f(x)) (or set of outputs when the problem space is non-deterministic) can either be correct or incorrect. The set of all possible inputs (X) results in a set of outputs being correct (Y_C) and a set of outputs being incorrect (Y_I). The distribution among those sets is quantized by outcome reliability. If the correct output set is empty, the inherent outcome reliability is 0. When all outputs are in the correct output set, the inherent outcome reliability is 1.

$X_C \cup X_I = X$ *(all possible inputs)*
$Y_C \cup Y_I = Y$ *(all possible outputs)*
f(x) *(software component which is passed input x)*

$X_C = \{ x \in X \mid f(x) \in Y_C \}$
$X_I = \{ x \in X \mid f(x) \in Y_I \}$

Inherent outcome reliability IOR = $|X_C| / |X|$ *where* $0 <= IOR <= 1$

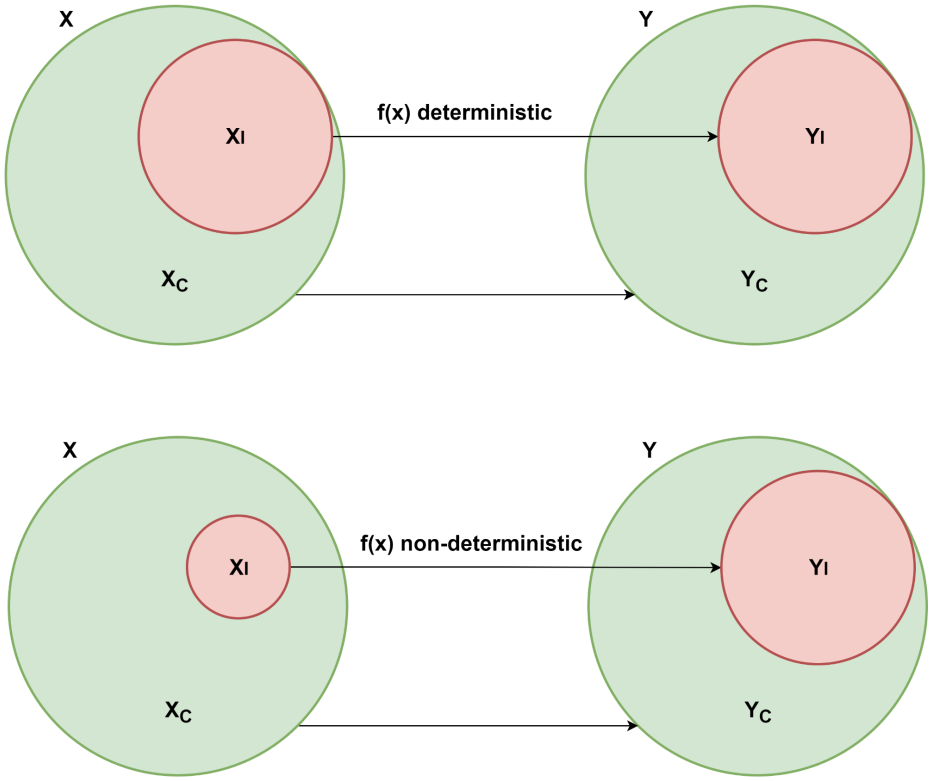


Figure A.1. Inputs leading to (in)correct outputs

References

“About GitHub Copilot coding agent.” 2025. GitHub Docs. https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent.

AI prompt theory. 2025. “Self Consistency: Improving Reliability in LLM Outputs.” aipromptttheory.com. https://aipromptttheory.com/self-consistency-improving-reliability-in-llm-outputs/.

Anthropic. 2024. “Introducing the Model Context Protocol.” Anthropic. https://www.anthropic.com/news/model-context-protocol.

Beetroot. 2025. “Human in the Loop Meets Agentic AI: Building Trust and Control in Automated Workflows.” Human in the Loop Meets Agentic AI: Building Trust and Control in Automated Workflows. https://beetroot.co/ai-ml/human-in-the-loop-meets-agentic-ai-building-trust-and-control-in-automated-workflows/.

Chutani, Gautam. 2024. “Unlocking LLM Confidence Through Logprobs | by Gautam Chutani | Medium.” Gautam Chutani. https://gautam75.medium.com/unlocking-llm-confidence-through-logprobs-54b26ed1b48a.

Docherty, Andrew. 2025. “Controlling your LLM: Deep dive into Constrained Generation.” medium.com. https://medium.com/@docherty/controlling-your-llm-deep-dive-into-constrained-generation-1e561c736a20.

Freitag, Michael. 2025. “How Artificial Intelligence Is Reshaping Industries—And What’s Next.” Forbes.

https://www.forbes.com/councils/forbesbusinesscouncil/2025/06/11/how-artificial-intelligence-is-reshaping-industries-and-whats-next.

Gison, Chase. 2025. “The Role of AI in Enhancing User Experience through Content.” medium.com. https://medium.com/@chasegison/the-role-of-ai-in-enhancing-user-experience-through-content-08fe5bd16ddc.

Google. 2025. “Structured output.” Structured output. https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/control-generated-output.

Grandperrin, Jonathan. 2021. “How to use confidence scores in machine learning models.” towardsdatascience.com. https://towardsdatascience.com/how-to-use-confidence-scores-in-machine-learning-models-abe9773306fa/.

KeyLabs. 2024. “Precision vs. Recall: Key Differences and Use Cases.” keylabs.ai. https://keylabs.ai/blog/precision-vs-recall-key-differences-and-use-cases/.

Lindemulder, Gregg, and Matthew Kosinski. 2024. “What Is Fraud Detection?” IBM. https://www.ibm.com/think/topics/fraud-detection.

Mamonov, Dmitry. 2023. “Clear and Complicated domains of the Cynefin Framework.” Clear and Complicated domains of the Cynefin Framework. https://medium.com/@dmitry.s.mamonov/clear-and-complicated-domains-of-the-cynefin-framework-62b30bf4602a.

Mamonov, Dmitry. 2023. “Complex and Chaotic domains of the Cynefin Framework.” Complex and Chaotic domains of the Cynefin Framework.

https://medium.com/@dmitry.s.mamonov/complex-and-chaotic-domains-of-the-cynefin-framework-c80766fbddba.

Martyr, Reginald. 2025. “Understanding Model Drift and Data Drift in LLMs (2025 Guide) | Generative AI Collaboration Platform.” Orq.ai. https://orq.ai/blog/model-vs-data-drift.

Onsem, M. v., D. d. Paepe, S. v. Hautte, P. Bonte, V. Ledoux, A. Lejon, F. Ongenae, D. Dreesen, and S. v. Hoecke. 2022. “Hierarchical pattern matching for anomaly detection in time series.” sciencedirect.com. https://www.sciencedirect.com/science/article/pii/S0140366422002298.

Paleti, Nikhil C., and Rita Angelou. 2024. “Convolutional Neural Networks: A Comprehensive Guide | by Jorgecardete | The Deep Hub.” Medium. https://medium.com/thedeephub/convolutional-neural-networks-a-comprehensive-guide-5cc0b5eae175.

Pohl, Klaus, Günter Böckle, and Frank J. van d. Linden. 2011. *Software Product Line Engineering: Foundations, Principles and Techniques*. -: Springer.

Precision Drafters. 2025. “How AI is Transforming the Drafting Industry?” precisiondrafters.com. https://www.precisiondrafters.com/post/how-ai-is-transforming-the-drafting-industry.

Safe Software. ? “AI Agent Architecture: Tutorial & Examples: Chapter 3: AI Agent Routing.” fme.safe.com. https://fme.safe.com/guides/ai-agent-architecture/ai-agent-routing/.

scitechtalk tv. 2024. “The Potential of AI to Discover Unknown Unknowns Using Serendipity and Out of the Box Thinking.” medium.com. https://medium.com/@tvscitechtalk/the-potential-of-ai-to-discover-unknown-unknowns-using-serendipity-and-out-of-the-box-thinking-bb49d6bdcb57.

Smajic, Nermin. 2024. “Humans, Errors, and the Rise of AI.” medium.com. https://medium.com/@nermiX/humans-errors-and-the-rise-of-ai-d58bba45a0e2.

Snowden, David. 1999. “Cynefin framework.” Cynefin framework - Wikipedia. https://en.wikipedia.org/wiki/Cynefin_framework.

Stankevichus, Ivan. 2025. “Keeping AI Pair Programmers On Track: Minimizing Context Drift in LLM-Assisted Workflows.” dev.to. https://dev.to/leonas5555/keeping-ai-pair-programmers-on-track-minimizing-context-drift-in-llm-assisted-workflows-2dba.

Udawatta, Kapila, Mehdi Ehsanian, Sergey Maidanov, and Surya Musunuri. 2008. “Test and validation of a non-deterministic system — True Random Number Generator.” Test and validation of a non-deterministic system — True Random Number Generator. https://ieeexplore.ieee.org/document/4695881.

Uday, Alney. 2025. “Why Audit Trails and Human-in-the-Loop Matter—Especially Now.” Why Audit Trails and Human-in-the-Loop Matter—Especially Now. https://www.linkedin.com/pulse/why-audit-trails-human-in-the-loop-matterespecially-now-uday-alney-bbaoc.

UiPath. 2025. “What is AI automation? Definition, benefits & examples.” UiPath. https://www.uipath.com/automation/ai-automation.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, and Llion Jones. 2017. “[1706.03762] Attention Is All You Need.” arXiv. https://arxiv.org/abs/1706.03762.

Visibee. 2025. “Understanding The Risks Of AI-Generated Code.” Visible One. https://visibleone.com/blog/understanding-the-risks-of-ai-generated-code/.

Wohlin, Claes, Martin Höst, Per Runeson, and Anders Wesslén. 2003. “Software Reliability.” *Encyclopedia of Physical Science and Technology (Third Edition)* -, no. - (1): 25-39. https://doi.org/10.1016/B0-12-227410-5/00858-9.

Zhou, Yuxuan. 2025. “Balancing Autonomy and Human Oversight: A Review of Automation and Control Systems in Large-Scale Industrial Processes.” *Applied and Computational Engineering* 135, no. 1 (2): 154-159. 10.54254/2755-2721/2025.21266.