

From the SCR Relationship to Search Space Governance: A Foundational Analytical Framework for Delivery Reliability in AI Software Engineering

Author: Liu Ming

ORCID: 0009-0001-2816-538X

English Final Manuscript V2.6.2

Abstract

Generative AI can already produce code and functionality at low cost, yet the reliable delivery of complex software still depends on continuously managing task constraints, component dependencies, change impacts, and validation feedback. This paper examines the complete AI development system, comprising large language models, development agents, tools, execution environments, validation feedback, and multi-round iteration, and takes the task session—from task acceptance to completed delivery, termination, or further decomposition—as the unit of analysis. On this basis, the paper defines Search Space (S), Coverage Capability (C), and Delivery Reliability (R), and proposes the **Search Space–Coverage Capability–Delivery Reliability Relationship (SCR)** as a foundational framework for analyzing AI software delivery. Delivery Reliability (R) describes whether a specific AI development result can be trusted as a completed delivery. It is neither equivalent to correctness under known criteria nor to the statistical success rate across repeated tasks. Within the SCR framework, the match between Coverage Capability (C) and Search Space (S) constitutes a foundational constraint on Delivery Reliability (R); when other key conditions are comparable, a more adequate match generally makes a specific delivery more likely to reach a state that can be trusted. This yields two basic intervention directions: expanding Coverage Capability (C), or adjusting Search Space (S). This relationship is definitional within the framework rather than a fully operationalized, self-contained quantitative formula, and it does not assume that Search Space (S) and Coverage Capability (C) can be reduced to single scalar values. This paper focuses on the latter direction and defines Search Space Governance as the engineering process of continuously adjusting task boundaries and the engineering environment to bring Search Space (S) within the range that the current Coverage Capability (C) can consistently handle. Although Search Space (S) is difficult to measure on an absolute scale, it is often possible to assess how local engineering changes alter the scope, structure, distinguishability among candidate paths, and validation conditions that typical subsequent tasks must handle, and to calibrate those assessments progressively through task trajectories and delivery outcomes. Because AI-generated code also becomes part of the engineering environment for subsequent tasks, Search Space Governance must continue throughout project evolution. In large, long-lived AI software projects, feature generation reflects how many direct outputs a project produces, whereas long-term engineering governance affects whether those outputs can continue to be understood, modified, and validated by subsequent tasks. The SCR relationship therefore provides a common structure for describing search space, system capability, and delivery outcomes; Search Space Governance identifies what can be directly acted upon through engineering; and the analysis of long-term projects further shows that engineering progress also depends on whether current results can be carried forward consistently into subsequent work. Together, these elements explain both the continuity and the change between traditional software engineering and AI software engineering, and provide a foundational analytical framework for engineering theory, system implementation, and subsequent evidence-based research in the AI era.

Keywords: AI software engineering; SCR relationship; search space; coverage capability; delivery reliability; search space governance

1. Introduction

AI-generated code is not only the result of the current task; it also becomes part of the engineering environment on which subsequent tasks depend. A locally usable implementation may simultaneously introduce new state,

dependencies, and implicit constraints, changing the scope that future development must understand and validate. Therefore, whether the current feature works, whether the current result can be trusted as a completed delivery, and whether subsequent tasks can continue to be completed reliably are related but distinct questions.

Traditional software engineering has long improved engineering environments through modularity, interfaces, types, testing, and refactoring. AI development adds mechanisms such as task specifications, context selection, retrieval augmentation, and agent planning and execution feedback. This paper seeks to add a unified analytical structure for AI development: what project state should engineering governance aim to create so that AI can maintain reliable delivery in an evolving engineering environment? Such a structure needs to explain how mechanisms at different technical levels—including single-pass generation, context organization, repository structure, and agent capability—jointly affect delivery outcomes.

This paper first explains Delivery Reliability (R) through the match between Search Space (S) and Coverage Capability (C), then discusses how that match can be improved by adjusting task boundaries and the engineering environment, and finally analyzes why this process must continue throughout long-term project evolution. The contribution of this paper is to propose and delimit this analytical framework rather than provide a directly computable quantitative function applicable across tasks. The framework can give a unified account of how model capability, agent mechanisms, task specifications, context mechanisms, and traditional software engineering practices affect delivery outcomes. It can also explain why AI software engineering still depends on engineering structures such as modules, interfaces, types, and tests, while addressing delivery uncertainty in generated results more directly.

2. Components of the SCR Relationship

The **AI development system** in this paper is not a single isolated large language model call. It is the combined system of the foundation model, development agent, retrieval and editing tools, code repository and execution environment, tests and human feedback, and the capacity for multi-round correction within a given budget. The unit of analysis is the **task session**: a task session begins when the system receives a clear engineering objective and ends when the task is delivered, terminated, or further decomposed. It may include multiple model calls, tool executions, and context updates.

2.1 Search Space (S)

Search Space (S) is the range of possibilities that the system must distinguish among and handle—across requirement interpretation, engineering state, implementation paths, and validation paths—in order to complete a specific task.

Search Space (S) is jointly shaped by task requirements, the current engineering state, and task-governance methods. It includes possible interpretations of goals, non-goals, and acceptance criteria; potentially relevant code, state, documentation, and dependencies; constraints that have not yet been made explicit; plausible implementation paths; and affected states and execution paths that require validation. The more ambiguous the boundaries, the more distributed the relevant state, and the more numerous and difficult to distinguish the candidate paths, the larger the Search Space (S) that the system must handle.

Search Space (S) is not a single scalar. Its engineering form depends simultaneously on scope, the structure of the search space, implicit constraints, distinguishability among candidate paths, and validation conditions. Two tasks with similar numbers of candidates may differ greatly in how difficult they are to handle because of their dependency structures, feedback quality, or whether critical constraints are explicit. Accordingly, references in this paper to Search Space being enlarged, reduced, or improved describe the combined change of these dimensions in a concrete task rather than a universal single-value ordering across all tasks.

Search Space (S) cannot be represented directly by code size or by the context window of a single call. Context describes the information directly accessible to the model at a given moment; Search Space (S) describes the possibilities that the complete system must distinguish and handle across the entire task session in order to complete delivery. Context engineering can change Search Space (S) by supplying necessary information, excluding irrelevant

candidates, and reducing ad hoc inference, but a longer context alone does not guarantee consistent use of the information it contains [9].

2.2 Coverage Capability (C)

Search Space (S) describes the range of possibilities a task requires the system to handle. Coverage Capability (C) describes the ability of the complete AI development system to handle that range. Coverage can only be discussed relative to a specific Search Space, so Coverage Capability (C) is not a task-independent universal value. It is the capability that the system can exercise consistently under a particular task type, execution budget, and feedback condition.

Coverage Capability (C) is the ability of the complete AI development system, under a given task type, execution budget, and feedback condition, to consistently identify critical conditions in Search Space (S), distinguish candidate paths, eliminate incorrect directions, and obtain effective validation evidence.

Coverage here does not require exhaustive enumeration of all candidates. The coverage required for reliable delivery means that the critical constraints, relevant state, implementation decisions, and impact scope on which delivery depends have not been omitted; alternative paths have been compared sufficiently; incorrect directions have been eliminated; and the final result has received validation commensurate with its risk.

The model provides knowledge, reasoning, and candidate-generation capability. The agent enables those capabilities to operate on a concrete task through context organization, retrieval, planning, tool use, execution loops, and validation feedback. Validation feedback does more than inspect the result: it also helps the system distinguish among candidate paths and eliminate falsified directions. The more accurately feedback exposes critical errors, the more likely a finite search process is to achieve adequate coverage.

Coverage Capability (C) should be judged by the critical aspects the system can cover repeatedly and consistently, rather than by the upper limit suggested by occasional success on difficult tasks. The same system may be reliable on tasks with clear boundaries and dense feedback, yet its Coverage Capability (C) may decline substantially on tasks with ambiguous goals, numerous implicit states, or poor verifiability.

2.3 Delivery Reliability (R)

Delivery Reliability (R) is the degree to which a specific AI development result satisfies critical requirements, has its relevant impacts adequately validated, and remains effective under expected conditions, such that it can be trusted as a completed delivery.

Delivery Reliability (R) is a composite delivery property. It concerns both whether the result actually satisfies critical requirements and whether the available validation evidence is sufficient to justify trusting it as a completed delivery. This paper does not further decompose these into separate variables; empirical research, however, should record result state and evidence state separately to avoid treating “no error was found” as equivalent to “no error exists.”

Delivery Reliability (R) concerns a single concrete delivery and asks: “The result has been generated, but can it be trusted as a completed delivery?” An AI output may already implement the main functionality, and a local result may be correct; but if the system has missed critical constraints, failed to handle relevant state, or insufficiently validated the impact of the change, the result is still not sufficient to count as a reliable delivery.

Coverage Capability (C) and Delivery Reliability (R) belong to different analytical levels. Coverage Capability (C) describes what processing the system can perform consistently under a class of task conditions; Delivery Reliability (R) describes the state reached by a specific delivery in terms of requirement satisfaction, impact handling, and validation evidence. The former is a process capability; the latter is an attribute of the delivered result. Delivery Reliability in this paper also differs from software product reliability centered on failure-free operation over a particular period and environment. Runtime reliability may be one delivery requirement, but it cannot by itself represent whether a development result is sufficiently trustworthy as a completed delivery.

Delivery Reliability (R) is not equivalent to correctness. Correctness describes whether a result conforms to a set of known judgment criteria. Delivery Reliability (R) additionally depends on whether those criteria cover the critical requirements, whether relevant impacts have been handled, and whether the validation evidence supporting the judgment is sufficient. Correctness is a necessary component of Delivery Reliability (R), but it does not exhaust it.

Delivery Reliability (R) is also not equivalent to success rate. Success rate is a statistical outcome across a task set or repeated runs, whereas Delivery Reliability (R) primarily describes the degree to which a specific delivery can be trusted. Statistical outcomes can calibrate judgments of Delivery Reliability (R), but they cannot replace analysis of the current task's requirements, impact scope, and validation evidence.

2.4 The SCR Relationship

This paper refers to the foundational relationship among Search Space (S), Coverage Capability (C), and Delivery Reliability (R) as the SCR relationship:

Within the SCR framework, the match between system Coverage Capability (C) and task Search Space (S) constitutes a foundational constraint on Delivery Reliability (R). When other key conditions are comparable, the more adequate the match, the more likely a delivery is to reach a state that can be trusted.

The term “match” here is not a numerical comparison between two scalars. It refers to whether the requirements imposed by a task—in critical-constraint identification, relevant-state handling, candidate-path discrimination, and validation-evidence acquisition—fall within the range that the current system can handle consistently. The concrete decomposition of these aspects and the proxy indicators used to study them may vary with task type. This paper does not presuppose a universal operationalization scheme across projects.

The same product requirement can generate substantially different Search Spaces (S) under different engineering states and task-governance methods. When Search Space (S) expands relative to Coverage Capability (C), more critical conditions, candidate paths, and impact areas compete for limited attention, search iterations, and validation resources. Decisive omissions become more likely, and Delivery Reliability (R) generally declines. Conversely, when Coverage Capability (C) strengthens relative to Search Space (S), more critical conditions can be handled consistently, incorrect paths are more readily eliminated by feedback, and Delivery Reliability (R) generally increases. Therefore, neither the absolute size of Search Space (S) nor the standalone strength of Coverage Capability (C) is sufficient to explain delivery outcomes. The key question is whether the current Coverage Capability (C) can consistently handle the current Search Space (S).

The engineering difficulty of Search Space (S) depends simultaneously on the number of candidates, the structure of the space, and candidate distinguishability. Clear goals, well-defined boundaries, and stable structures can localize critical judgments; high-quality feedback can increase the distinguishability of candidate paths and improve the system's ability to eliminate incorrect directions.

Theoretical nature of the SCR relationship. The SCR relationship is a definitional analytical framework rather than an empirical formula fitted from three already-operationalized scalar variables. By distinguishing the Search Space (S) that a task requires the system to handle, the Coverage Capability (C) that the system can consistently achieve, and the Delivery Reliability (R) of a concrete result, it establishes a common structure for describing AI software delivery problems. The framework itself should be evaluated for conceptual distinctness, logical consistency, explanatory scope, and analytical utility. Specific empirical claims derived from the framework—for example, whether a particular engineering measure improves the match between S and C under given conditions and thereby increases Delivery Reliability (R)—still require testing through task trajectories and delivery outcomes. This paper does not claim that Search Space (S) and Coverage Capability (C) can already be compressed into universal single values across projects. Their operationalization and proxy indicators belong to subsequent empirical research.

The SCR relationship therefore yields two basic intervention directions: expand Coverage Capability (C), or adjust Search Space (S). Advances in models, agents, and tools mainly act on the former; task boundaries, engineering

structure, and validation conditions can directly alter the latter; some mechanisms may affect both. This paper focuses primarily on interventions on Search Space (S).

3. Search Space Governance

3.1 Choosing the Intervention Target

When the AI development system is fixed or changes only modestly in the short term, Search Space (S) is the most direct engineering control variable. By adjusting task boundaries, engineering structure, and validation conditions, engineering practice can improve the match between Search Space (S) and Coverage Capability (C), thereby increasing Delivery Reliability (R).

Search Space Governance is the engineering process of continuously adjusting task boundaries and the engineering environment to bring Search Space (S) within the range that the current Coverage Capability (C) can consistently handle.

Search Space Governance acts not only on generated results but also on the task and engineering conditions under which generation occurs. Clarifying goals and non-goals, constraining the relevant scope, codifying stable constraints, isolating state and dependencies, and establishing validation feedback that can distinguish among paths can all change Search Space (S). Engineering intervention thus shifts from merely helping AI generate the current result toward continuously establishing the task conditions and engineering environment needed for reliable delivery.

Task decomposition constitutes Search Space Governance only when the original objective, critical constraints, and integration relationships among subtasks are preserved. After decomposition, subtasks may form their own task sessions, but delivery requirements at the parent-task level still remain. Complexity transferred to interfaces and integration stages also remains part of the overall Search Space (S) that the complete task must handle.

Search Space Governance does not aim to minimize the number of candidates. Its goal is to improve the match between Search Space (S) and Coverage Capability (C). Incorrectly excluding necessary information or valid implementation paths may reduce the apparent number of candidates while lowering Delivery Reliability (R). Adding necessary validation paths may enlarge part of the scope that must be handled explicitly, yet still improve candidate distinguishability and the overall match. Governance therefore includes reducing ineffective search burden, preserving necessary paths, improving search-space structure, and establishing effective feedback.

3.2 Task Governance and Project Governance

Search Space Governance acts on both the current task and future tasks:

Governance scale	Direct object	Typical role	Time horizon
Task governance	Current task session	Clarify goals, select necessary context, decompose tasks, establish acceptance feedback	Current delivery
Project governance	Continuously evolving engineering environment	Isolate state and dependencies, stabilize interfaces and types, preserve regression validation	Multiple subsequent tasks

Task governance controls the interpretations, states, paths, and impact scope involved in the current delivery. Project governance transforms confirmed knowledge into structures that subsequent tasks can depend on directly. The tools used by the two may be identical; the difference lies in scope and time scale. Task governance adjusts the Search Space (S) faced by the current task, whereas project governance shapes the engineering environment on which a sequence of future tasks will depend.

Knowledge confirmed in one task can support subsequent tasks directly only when it is persisted in the engineering environment. For example, a task specification can constrain the current delivery. Constraints in the specification that

have been validated and remain stable over time can be further encoded in project rules, interfaces, or types. Likewise, a temporary acceptance test, once retained as a regression test, can continue to eliminate incorrect paths in later modifications.

3.3 From One-Time Intervention to Continuous Governance

Generated results have a dual nature: they are outputs of the current task and part of the working environment of subsequent tasks. The boundaries, dependencies, interfaces, types, and tests introduced or confirmed by one change determine what future tasks must revisit, what paths can be excluded, and how large a scope must be validated. If critical relationships exist only in the reasoning of the current session, subsequent tasks must rediscover them. If those relationships are transformed into stable structures, subsequent tasks can directly use constraints and relationships that have already been confirmed.

New requirements and code changes continuously rewrite this environment, and boundaries and validation mechanisms that were previously effective may change with it. Current generation alters the engineering environment, and that environment in turn shapes the Search Space (S) of future tasks. Therefore, a one-time intervention cannot guarantee that subsequent tasks will remain matched to Coverage Capability (C).

Search Space Governance must therefore operate as a cycle of “generate–validate–persist–continue generating,” continuously revising engineering boundaries in light of new information exposed by subsequent generation and validation. It is not a one-off task optimization, but an engineering process that continues with project evolution.

4. Judging Changes in Search Space and Accumulating Evidence

4.1 Assessing Changes in Search-Space Tractability

Search Space (S) comprises requirement interpretations, relevant state, implicit constraints, candidate paths, and impacts that require validation. Although absolute measurement is difficult, engineering changes can be compared in terms of their combined effects on processing scope, search-space structure, candidate distinguishability, and validation conditions in concrete tasks.

Search Space Governance can begin with a basic judgment: does a given engineering change improve the match between Search Space (S) and Coverage Capability (C) for current or future tasks?

This judgment should focus on representative tasks that are frequent, critical, or high-risk within the project. Before and after an engineering change, one can compare the code, state, and dependencies that the system must locate and understand; the implicit constraints it must rediscover; the candidate paths it must choose among; and the impact scope it must validate. At the same time, one should examine whether critical relationships have become clearer, whether candidate paths are easier to distinguish, and whether feedback exposes decisive errors more effectively. Reducing the processing scope overall usually reduces search burden, but an increase in scope does not necessarily worsen the match. If the added content makes necessary constraints explicit or establishes effective validation, the overall tractability of Search Space (S) may still improve.

4.2 A Unified Interpretation of Existing Engineering Practices

Engineering practice	Primary effect on Search Space
Task specifications and context selection	Reduce the range of plausible requirement interpretations, provide necessary constraints, and limit the relevant scope
Task decomposition	Reduce the state, dependencies, and impact scope that a single task must cover simultaneously
Modularity and refactoring	Localize state and change impact, remove duplicate or obsolete paths

Engineering practice	Primary effect on Search Space
Interfaces and types	Make interactions and constraints explicit, automatically eliminate some invalid paths
Tests and runtime feedback	Increase distinguishability between correct and incorrect paths, clarify regression boundaries

The SCR relationship provides a unified evaluation structure for these practices: on one hand, compare the Search Space (S) of representative tasks before and after a change; on the other, examine whether knowledge obtained in the current delivery has been transformed into boundaries and feedback that future tasks can use. Traditional engineering methods and AI development specifications can therefore be evaluated within one structure according to their actual effects on Search Space (S), Coverage Capability (C), and Delivery Reliability (R).

4.3 Continuously Calibrating Judgments with Engineering Evidence

Future research can first assess how a given change is expected to alter processing scope, search-space structure, candidate distinguishability, and validation conditions for a specific task. It can then record engineering trajectories and delivery outcomes before and after the change, calibrate observable proxy indicators under comparable project, task-type, and system conditions, and continuously revise those judgments using actual outcomes. As evidence accumulates, empirical regularities applicable to specific projects and task types can gradually emerge.

Evidence that can be accumulated includes: the number of entities and modules involved in representative tasks; the dependency boundaries and state ranges crossed; implicit constraints that must be rediscovered; execution paths and impact scopes covered by validation; the number, types, and convergence characteristics of newly identified issues across multi-round generation or auditing; and changes in delivery cost and failure rate for similar tasks as the project grows.

Delivery Reliability (R) is the ultimate evaluation target. Changes in Search Space (S), task-execution trajectories, issue convergence, validation burden, and historical failure rates are proxy evidence at different analytical levels. These forms of evidence cannot be equated directly with Delivery Reliability (R). Trajectories and outcomes from similar tasks within the same project are comparable only when task type, Coverage Capability (C), and risk conditions are sufficiently similar. Continuously accumulating such evidence can improve the comparability and verifiability of delivery-risk assessment and evaluation of governance effects, while gradually improving judgment accuracy for specific projects and task types.

5. Long-Term Engineering Progress Cannot Be Measured by Feature Count Alone

5.1 Separation Between Feature Generation and Engineering Progress

A grey-literature review of vibe coding found that practitioners are often motivated by speed and accessibility and can obtain rapid “instant success,” yet most also regard generated code as fast but flawed, while quality-assurance activities such as testing are frequently neglected [11]. From the perspective of engineering progress, this creates a typical illusion: the fact that a feature runs proves only that one path is feasible under current observation conditions. It neither proves that critical conditions have been covered nor that the result has reached a state that can be trusted as a completed delivery.

Traditional development treats feature implementation as important progress partly because implementation usually requires substantial human effort, during which developers also address a significant portion of requirement, structural, and validation issues. In a controlled experiment, participants using GitHub Copilot completed a specific JavaScript programming task 55.8% faster [5], indicating that generative AI can substantially reduce the time required for some implementation activities. However, the costs of confirming boundaries, codifying constraints, isolating dependencies, and establishing effective validation do not necessarily decline at the same rate. Low-cost AI generation further

amplifies the pre-existing difference between feature generation and engineering governance, making the number of generated features an even weaker approximation of long-term engineering progress.

5.2 Runnable Code Is Not Equivalent to a Maintainable Engineering Outcome

Code that runs is not necessarily code that subsequent tasks can continue to use reliably. A current implementation becomes an engineering outcome that can be maintained and reused over time only when its boundaries, dependencies, and critical constraints are sufficiently clear, the result has been effectively validated, and subsequent tasks can understand and modify it within a controlled scope.

Poorly governed implementation is difficult to convert into cumulative engineering progress. It may instead increase state, dependencies, and validation burden, expand the Search Space (S) of future tasks, and even make rework more expensive than initial generation. Feature generation therefore measures only how many direct outputs the project has produced. Whether those outputs can be carried forward consistently into subsequent work determines whether they truly constitute long-term engineering progress.

5.3 Long-Term Engineering Governance Affects Whether Results Can Continue to Accumulate

For large, long-lived AI software projects, feature generation reflects how many direct outputs a project produces, while long-term engineering governance affects whether those outputs can continue to be understood, modified, and validated by subsequent tasks.

Features remain necessary project deliverables, but feature count cannot indicate whether the current implementation can be relied on over time. For small prototypes or one-off tools, current usability may directly serve as the completion criterion. For projects that must continue evolving, engineering governance determines whether current results support subsequent development or merely transfer complexity and risk to future tasks.

The goal of long-term engineering governance is to keep the Search Space (S) of representative new tasks within the range that the current Coverage Capability (C) can consistently handle as project complexity grows. The more consistently this condition can be maintained, the less likely Delivery Reliability (R) is to decline systematically with project growth. Engineering progress in large AI software projects therefore cannot be measured by feature count alone; it must also consider whether current results continue to support reliable delivery in subsequent tasks.

6. Related Work and Theoretical Positioning

6.1 Problem Spaces and Search-Based Software Engineering

Understanding problem solving as selecting operations and paths within a problem space is a classic perspective in artificial intelligence and cognitive science [1]. Simon’s analysis of ill-structured problems further showed that problem representation and task structure may themselves be formed and revised during problem solving [2]. Search-based software engineering typically formalizes a given software engineering problem as an optimization problem with a representation, objective function, and search operators [3]. This paper does not design search algorithms and does not assume that tasks have already been formalized. Instead, it studies the foundational relationship among Search Space (S), system Coverage Capability (C), and Delivery Reliability (R) in practical AI development tasks, and how engineering governance changes that relationship.

6.2 Repository-Level Agents and Context Engineering

SWE-bench shows that real repository tasks often require coordinated changes across functions, classes, and files, and require systems to handle long contexts, complex reasoning, and execution environments [6]. CodePlan and SWE-agent further show that dependency analysis, planning, context organization, and agent–computer interfaces can significantly affect repository-level task performance [7–8]. Research on long-context models shows that increasing context length does not guarantee consistent use of relevant information [9], while repository-level code-context

engineering research reveals trade-offs among effectiveness, task complexity, and computational cost across different strategies [10]. This paper primarily treats model and agent mechanisms as sources of Coverage Capability (C), while regarding retrieval, planning, context selection, and dynamic decomposition as system mechanisms that may alter both Search Space (S) and Coverage Capability (C).

6.3 A Unified Analysis of AI Engineering Governance

Information hiding and modularity are classic software-engineering methods for controlling the scope of understanding and propagation of change [4]. Interfaces, types, tests, and change isolation likewise make constraints explicit and localize impact. AI software engineering adds task specifications, context selection, retrieval augmentation, agent planning, and execution feedback [6–10]. These practices span different levels of requirements, knowledge, context, code structure, and runtime validation. The contribution of this paper is not another local mechanism, but a unified analytical framework that places requirement specification, persistent engineering state, context mechanisms, system capability, and delivery outcomes within the same foundational relationship.

The SCR relationship provides this foundational structure. Task specifications (Specs) constrain requirement interpretation, critical constraints, and acceptance boundaries, primarily reducing and structuring Search Space (S). The code repository (Code Repo) is not merely an information source for the agent; it persistently carries code, documentation, configuration, dependencies, and the results of prior governance. Its modules, interfaces, types, tests, and dependency structure jointly shape the Search Space (S) of current and future tasks. Retrieval-augmented generation (RAG), code retrieval, and context selection provide task-relevant information to the system, primarily strengthening Coverage Capability (C); precise selection can also reduce irrelevant states and paths that the system must distinguish. Models, agents, and tools determine the range the system can handle consistently. Tests and execution feedback help distinguish and eliminate incorrect paths, and transform confirmed judgments into reusable validation boundaries for subsequent tasks. Although these mechanisms operate at different technical levels, all of them can be analyzed through the match between Search Space (S) and Coverage Capability (C) to explain how they affect Delivery Reliability (R).

Search Space (S) and Coverage Capability (C) are analytical dimensions, not a component taxonomy that requires every engineering mechanism to be assigned uniquely to one category. Classification depends on the analytical level, and the same mechanism can act on multiple dimensions at once. This also means engineering mechanisms cannot be evaluated merely by whether they are used or by the amount invested in them. A more detailed Spec may codify necessary constraints, but it may also introduce implementation restrictions unrelated to delivery, increasing the number of conditions and validation paths the system must handle simultaneously. More retrieved content or repository context may supplement critical knowledge, but it may also introduce noise, conflicts, and additional judgment burden. A larger code repository may accumulate reusable engineering capability, but breakdowns in boundaries and the spread of dependencies may continuously enlarge the Search Space (S) of future tasks. The same mechanism can therefore affect both Search Space (S) and Coverage Capability (C). Its engineering value depends on whether it improves their match and ultimately increases Delivery Reliability (R), rather than on the mechanism's name, context size, or number of tools.

Search Space Governance further turns this relationship into a continuous engineering process. Traditional software-engineering mechanisms such as modules, interfaces, types, tests, and change isolation need no longer be understood merely as separate best practices; they can also be interpreted within one structure as engineering foundations that control current and future Search Space (S), support a system with finite Coverage Capability (C), and maintain Delivery Reliability (R). AI software engineering thus preserves traditional software engineering's focus on controlling complexity while introducing a key change: when generative systems can produce highly complete outputs before critical constraints have been adequately handled, engineering governance must directly identify and reduce delivery uncertainty. The SCR relationship therefore provides a common explanation for traditional software-engineering structures and AI development mechanisms, while Search Space Governance indicates which engineering conditions can be modified to improve delivery outcomes.

7. Scope and Extrapolation

The SCR relationship may have explanatory power beyond software engineering. In general AI tasks, task requirements, environmental state, and interaction methods likewise shape Search Space (S), while models, agents, and feedback conditions jointly form Coverage Capability (C). When the two are mismatched, the result may appear as hallucination, constraint omission, incorrect path selection, or validation failure. Software engineering provides conditions such as persistent state, explicit constraints, traceable changes, and relatively verifiable outputs, making external governance of Search Space (S) feasible. Open-ended tasks such as research, strategy, creative work, and social judgment make it more difficult to externalize goals, states, constraints, and evaluation criteria sufficiently. This paper discusses software engineering only. The explanatory power of the SCR relationship in other domains, and whether their search spaces can be governed effectively, require independent examination.

8. Conclusion

This paper defines Search Space (S), Coverage Capability (C), and Delivery Reliability (R), and proposes the SCR relationship as a foundational analytical framework. Within the framework, the match between Coverage Capability (C) and Search Space (S) constitutes a foundational constraint on Delivery Reliability (R); when other key conditions are comparable, a more adequate match generally makes a specific delivery more likely to reach a state that can be trusted. Delivery Reliability (R) describes whether a specific result can be trusted as a completed delivery. It is neither a renaming of correctness nor a statistical success rate. Coverage Capability (C) describes the system's process capability for handling a task, while Delivery Reliability (R) describes the state achieved by the specific result being delivered.

What this paper proposes is not a set of isolated AI development practices, nor a quantitative formula that has already been operationalized, but a foundational analytical structure for organizing these practices. The SCR relationship enables task specifications (Specs), code repositories (Code Repos), RAG, models and agents, tests and execution feedback, as well as modularity, interfaces, types, and refactoring from traditional software engineering, to be understood within one framework according to their actual effects on the match between Search Space (S) and Coverage Capability (C). Search Space Governance further shows that this match can be improved by continuously adjusting task boundaries and the engineering environment. Its objective is not merely to minimize Search Space (S), but to reduce ineffective search burden, preserve necessary paths, improve search-space structure, and establish effective feedback. In long-term projects, engineering progress also depends on whether cheaply generated functionality can continue to be understood, modified, and validated by subsequent tasks.

This analysis reframes both the continuity and the change between traditional software engineering and AI software engineering. AI software engineering still depends on engineering structures such as modules, interfaces, types, and tests to control complexity. Yet the separation between generative capability and reliable delivery means that engineering governance cannot merely ensure that such structures exist; it must directly address mismatches between current Coverage Capability (C) and task Search Space (S). Search Space Governance thereby connects traditional engineering structures, AI development mechanisms, and long-term delivery outcomes, providing a common starting point for engineering theory, system design, and evidence accumulation in the AI era.

Future research can first assess how local engineering changes alter the processing scope, structure, candidate distinguishability, and validation conditions of Search Space (S), and then accumulate evidence from task trajectories, dependency scope, implicit constraints, validation burden, and delivery outcomes to continuously calibrate empirical judgments applicable to specific projects and task types. The framework itself provides a common analytical language; concrete predictions, proxy indicators, and governance effects derived from it should remain subject to empirical testing. The explanatory power of the SCR relationship for other classes of intelligent tasks, and whether their search spaces can be effectively governed, also require independent research.

References

- [1] NEWELL A, SIMON H A. Computer science as empirical inquiry: Symbols and search[J]. *Communications of the ACM*, 1976, 19(3): 113-126. DOI: [10.1145/360018.360022](https://doi.org/10.1145/360018.360022).
- [2] SIMON H A. The structure of ill structured problems[J]. *Artificial Intelligence*, 1973, 4(3-4): 181-201. DOI: [10.1016/0004-3702\(73\)90011-8](https://doi.org/10.1016/0004-3702(73)90011-8).
- [3] HARMAN M, JONES B F. Search-based software engineering[J]. *Information and Software Technology*, 2001, 43(14): 833-839. DOI: [10.1016/S0950-5849\(01\)00189-6](https://doi.org/10.1016/S0950-5849(01)00189-6).
- [4] PARNAS D L. On the criteria to be used in decomposing systems into modules[J]. *Communications of the ACM*, 1972, 15(12): 1053-1058. DOI: [10.1145/361598.361623](https://doi.org/10.1145/361598.361623).
- [5] PENG S, KALLIAMVAKOU E, CIHON P, et al. The impact of AI on developer productivity: Evidence from GitHub Copilot[EB/OL]. arXiv:2302.06590, 2023. DOI: [10.48550/arXiv.2302.06590](https://doi.org/10.48550/arXiv.2302.06590).
- [6] JIMENEZ C E, YANG J, WETTIG A, et al. SWE-bench: Can language models resolve real-world GitHub issues?[C]//*The Twelfth International Conference on Learning Representations*. 2024. <https://openreview.net/forum?id=VTF8yNQM66>.
- [7] BAIRI R, SONWANE A, KANADE A, et al. CodePlan: Repository-level coding using LLMs and planning[J]. *Proceedings of the ACM on Software Engineering*, 2024, 1(FSE): 675-698. DOI: [10.1145/3643757](https://doi.org/10.1145/3643757).
- [8] YANG J, JIMENEZ C E, WETTIG A, et al. SWE-agent: Agent-computer interfaces enable automated software engineering[C]//*Advances in Neural Information Processing Systems 37*. 2024: 50528-50652. DOI: [10.52202/079017-1601](https://doi.org/10.52202/079017-1601).
- [9] LIU N F, LIN K, HEWITT J, et al. Lost in the middle: How language models use long contexts[J]. *Transactions of the Association for Computational Linguistics*, 2024, 12: 157-173. DOI: [10.1162/tac1_a_00638](https://doi.org/10.1162/tac1_a_00638).
- [10] LI Y, LIN Q, PENG Y, et al. One size does not fit all: Revisiting code context engineering for repository-level code generation[J]. *Proceedings of the ACM on Software Engineering*, 2026, 3(FSE): 2952-2975. DOI: [10.1145/3808138](https://doi.org/10.1145/3808138).
- [11] FAWZY A, TAHIR A, BLINCOE K. Vibe coding in practice: Motivations, challenges, and a future outlook—A grey literature review[C]//*2026 IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2026: 212-223. DOI: [10.1145/3786583.3786866](https://doi.org/10.1145/3786583.3786866).