AISDL – The AI System Design Language

An Open Specification for AI-Designed Enterprise Systems Executed by Controlled Runtimes

Trust AI to design business behavior without granting its generated code general-purpose execution authority. AI authors and maintains executable business specifications in AISDL and restricted Process logic in AIScript. A controlled Runtime enforces their declared interfaces and execution boundaries.

The architectural and mathematical analysis examines security, predictability, performance and portability, together with the economic consequences of this separation.

Mohammed Loukili El Fahssi

November 2026 | Edition 4.2

© 2026 Mohammed Loukili El Fahssi. Licensed under CC BY 4.0.

The proposition

AISDL separates the intelligence that designs a business System from the infrastructure that executes it. AI owns the design: its data, Processes, decisions, exceptions, recovery and improvement. People establish intent and authorize what runs. An AISDL Runtime executes the approved specification through defined language and Service contracts. The enterprise's enduring asset becomes the specification, not a separately maintained application stack.

The enabling proposition is broad AI design freedom without unrestricted execution authority. AI must author a controlled specification and controlled Process code, not an independent general-purpose application stack. The restriction is not a temporary limitation to remove as models improve: it is what makes the proposed separation possible. Better AI can produce better business designs without receiving broader host privileges.

The proposal begins from confidence in AI as a designer. As models improve and design, criticism and testing can be repeated faster, AI has a credible route to better complete business designs than conventional human delivery teams. This confidence is not a measured universal ranking. The supporting mathematics identifies how accepted improvement accumulates and what rates would establish the advantage; comparative tests must measure those rates and outcomes.

Trusting the Designer and controlling execution solve different problems. A sound design still needs a precise execution environment. The AISDL Specification defines the System, AISDL AIScript expresses its restricted business logic, and the AISDL Runtime enforces the supported execution contracts. The purpose is not to replace business judgment with infrastructure rules, but to make the Designer's intended behavior inspectable, executable and portable without rebuilding technical controls for every System.

Software becomes infrastructure; business specifications become the unit of change. Neither humans nor AI maintain a separate bespoke application stack for each System. Controlled compilation artifacts may exist inside the Runtime. What disappears is the recurring application-development obligation as the route from business intent to operation.

This is an architectural thesis, followed by its economic consequences. The framework is open to new, documented utilities and Services. Different providers can implement compatible AISDL Runtimes and compete on execution without owning the enterprise's business design. Where this separation delivers a sustained advantage in quality, time and full cost, ending application dependence becomes a competitive consequence rather than a slogan.

Redefining AI-native

An AI feature is not an AI-native enterprise. Here, *AI-native* means replacing application delivery with AI-maintained executable business specifications and governed Runtime infrastructure. It does not mean using AI to write the same application stack faster. The following definitions apply throughout.

AI-assisted enterprise software. Applications in which AI helps people perform work: answering questions, preparing content, recommending actions or automating tasks. These capabilities can be valuable without changing who designs and maintains the application's behavior. Adding a chatbot or copilot does not, by itself, establish the operating model defined here.

An AI-native business System. A complete executable definition of business behavior that AI designs, manages, maintains and improves in a controlled specification. It includes Data Collections, Processes, contracts, Service dependencies and business logic. People establish business intent and constraints and authorize activation; AI translates them into a coherent executable System. A governed Runtime executes the approved revision.

An AI-native enterprise. An enterprise whose business changes take the form of AI-authored and AI-maintained executable specifications, not application-development projects. People decide which Systems to create or improve and authorize execution. AI owns their design and evolution within AISDL; the Runtime supplies execution infrastructure. Neither people nor AI maintain a separate application codebase for each System.

AISDL Specification and AISDL AIScript. AISDL, the AI System Design Language, occupies the space between configuration and general-purpose programming. It expresses complete enterprise business Systems, including executable logic, but excludes unrestricted application and infrastructure development. AIScript expresses that logic within defined contracts and Service interfaces. AI designs the business specification; the Runtime controls its execution. This is not the absence of logic, but the absence of a separately engineered application stack.

AISDL Runtime. The software that validates, compiles or interprets the approved specification, executes its Processes and enforces the supported technical boundaries. The business authorizes behavior without maintaining a separate generated application stack. Runtime operation, infrastructure, technical assurance and accountable governance remain necessary.

AI-native does not mean AI on every transaction. Approved Process logic can execute without consulting a model. A Process uses AI during execution only where its design requires it. AI proposes improvements; the Runtime executes the approved revision, not unapproved changes.

Contents

1 The AISDL design and execution boundary 4

2 An open specification and competing Runtimes 11

3 The long attempt to remove the IT bottleneck 15

4 Why this is an inflection point 17

5 The mathematical case for ending application dependence 21

6 From IT releases to business publication 24

7 The economics of competitive business evolution 26

8 The no-IT-department enterprise 34

9 The strongest objections and how to test the thesis 36

10 Conclusion: The AISDL architectural model 39

Scope of the evidence 40

1 The AISDL design and execution boundary

The enabling mechanism. AI needs a complete, inspectable specification for business systems and a predictable runtime capable of executing it. AISDL—AI System Design Language—is one proposed mechanism. The argument does not depend on a particular proprietary implementation.

The post-application enterprise cannot be built on prompts alone. Its first technical foundation is *AISDL*: a versioned, human-readable specification of an entire executable business System, with machine-checkable structure and interfaces. It describes Data Collections, Audiences, Processes, Services, Secrets and Events. Process contracts define Inputs, Outputs, invocation and presentation; *AIScript* expresses their executable business logic. Documentation explains the behavior and its domain obligations. Identity and access administration are supplied separately by the runtime. This is not merely a database schema, workflow diagram, business rules table, or a prompt. Machine-checkable structure does not imply automatic proof of business completeness.

AI designs, manages, enhances and maintains the System specification, including its data relationships, business rules, recovery behavior, Service choices and performance trade-offs. Humans supply intent, organizational context and consequential approvals; they do not need to prescribe tables, algorithms or a rules-engine configuration. The business must be able to understand and authorize the resulting behavior without assuming responsibility for maintaining Process code. AIScript contains the executable logic; the Runtime runs it as part of the approved specification.

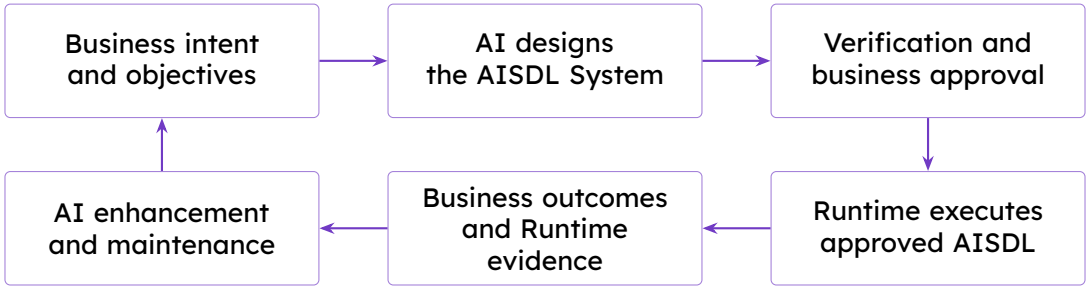

Figure 1: AI maintains the specification, the business authorizes its behavior, and the Runtime executes the approved revision.

AI maintains and improves the specification; it need not perform each business execution. The Runtime executes the Process behavior defined by the admitted AISDL revision. A language-model call belongs on that path only when the business design explicitly requires it as a Service. The execution cost, latency and reliability of an ordinary transaction therefore need not include a model inference.

The second foundation is a *conforming controlled runtime*. It validates and executes the accepted specification, supplies system-of-record persistence and common services, enforces resource and security policy, and retains auditable evidence. The enterprise need not know whether the runtime uses Java, Rust, native code, WebAssem-

bly, a managed interpreter, or another implementation internally. The runtime may compile, translate, or interpret AISDL Process logic, but must preserve the language's specified observable semantics. This is a proposed architecture; it does not assert that any current AISDL implementation already proves full semantic preservation.

The critical departure from an application server is that a Process does not arrive as an unrestricted, separately packaged application. AISDL may contain code written in AIScript, a purpose-built programming language for Process logic, not a general-purpose language for constructing application stacks or accessing a host machine. Its calculations, data structures, functions and asynchronous operations express business behavior; communication with Services uses defined interfaces. The language restricts the technical execution surface, not the AI author's freedom to design the business System.

Broad business design, controlled execution. AISDL occupies a deliberate middle ground between configuration of a fixed application and unrestricted programming. AI can define new data relationships, calculations, decisions, workflows, exceptions and recovery behavior, rather than merely select predefined options. Those capabilities support a broad range of enterprise Systems without giving their Processes arbitrary host access. When an additional capability is required, a declared Service can expose it through an approved interface; this does not imply that every needed Service already exists. The boundary controls how business behavior executes, not which business model AI may design.

Against coding as enterprise delivery. Complete business behavior still requires executable decisions, calculations, coordination and recovery. AIScript expresses this restricted logic within the AISDL specification; it does not create a separate application-development obligation for either humans or AI. The Runtime's compilation layer translates the specification into its engine's executable representation, or interprets it, without allowing declared Service interfaces to be bypassed. Generated execution artifacts belong to infrastructure, not to a second codebase the enterprise must maintain. A smaller technical surface can support stronger analysis and enforcement, but does not alone guarantee correct business logic or faster execution.

1.1 AI design freedom without unrestricted execution authority

The thesis does not ask an enterprise to trust arbitrary AI-generated software. It asks the enterprise to authorize business behavior expressed within an explicit execution contract. AISDL defines the System's resources, relationships, Inputs, Outputs and dependencies. AIScript expresses the necessary decisions, calculations, state transitions, coordination and recovery. These are controlled, executable artifacts, not unrestricted instructions to a host machine.

General-purpose languages such as C, C++, Java and Python are not the proposed authoring boundary for an AI-designed business System. They are designed for a broader range of software construction and can be connected to operating-system

facilities, general libraries and independent application infrastructure. Asking AI to generate another application in one of those languages leaves the technical surface of that application to be engineered, inspected and maintained. This is an architectural objection to unrestricted application generation, not a claim that these languages are inherently insecure or unsuitable for Runtime engineering.

AIScript deliberately withholds that general-purpose authority. A Process cannot import arbitrary libraries, invoke a shell, evaluate host-language code or bypass declared Services to access networks, files or credentials. Its executable operations come from the language contract and the supported Service and Utility interfaces. A conforming compiler and Runtime must reject unsupported operations and preserve the boundary in translated or interpreted execution. An extension must preserve the same mediation rather than reintroduce an unrestricted escape through a new interface.

The AI remains responsible for the business solution. It can choose data structures, algorithms, workflows, recovery strategies and the supported Services that best realize the intent. The Runtime is responsible for the authority those choices may exercise. Trust in design capability therefore does not require trust in unrestricted generated code. General-purpose engineering remains within the Runtime and Service implementations, where its assurance can be shared across many Systems rather than reconstructed for each one.

The relevant safety question is not whether AI can always generate correct code, but whether incorrectly generated code can acquire authority outside the accepted specification. Under the confinement obligation defined below, model-generated text does not create new technical capabilities. Errors within permitted business behavior remain possible and require independent tests, invariants and evidence. The restriction is necessary to this thesis's execution guarantee; it is not, by itself, a proof of business correctness or a vulnerability-free Runtime.

1.2 Why a safe implementation language is not the boundary

Rust illustrates the distinction: its ownership rules govern memory management [8]. A memory-safe implementation can still debit the wrong account, trust an unverified identity, send data to an unintended destination or apply an incorrect business policy. Memory safety addresses a valuable class of implementation faults, not the entire business execution contract. A Runtime written in Rust may be an excellent engineering choice. Asking AI to produce a separate Rust application for each business System leaves the application lifecycle in place.

AISDL places that engineering below the business specification. The Runtime provider implements infrastructure in a suitable language; AI designs the System in AISDL and its logic in AISDL AIScript. Restrictions govern host access and effects, not business creativity. Technical control is compatible with trusting the Designer to choose the data structures, algorithms, Services and performance trade-offs.

1.3 The minimum separation

Suppose the desired contract requires that every business effect is mediated by an approved capability and that an approved revision is the one executed. A path that bypasses mediation or substitutes unapproved behavior contradicts that contract. Any architecture satisfying these requirements must therefore provide equivalent enforcement somewhere. The claim is necessity of the separation, not uniqueness of a product or of the AISDL name. A competing architecture that provides the same inspectable contract and non-bypassable execution boundary belongs to the same architectural class.

1.4 A business change, from intent to execution

Consider the request: "Allow trusted suppliers to receive an advance against an approved order, subject to the buyer's exposure limit, and recover safely if the payment provider's response is uncertain." This is business intent, not a prescribed database or workflow.

The AI author determines the necessary records and relationships, eligibility and approval Processes, monetary calculations, provider interaction, exception handling and recovery. It may choose a new Collection for advance attempts, reuse existing order and supplier records, and add Process Apps for requesting, approving and investigating an advance. The design must prevent a repeat request from causing a second advance and prevent concurrent approvals from exceeding exposure. Its documented outcome includes declined, completed and unresolved attempts, not just the happy path.

Independent checks then exercise duplicate submissions, concurrent requests, provider timeouts, late responses and recovery. Security vetting reviews the data and approved credential use; business approval concerns the policy and its consequences. After activation, the runtime executes the accepted Processes and collects linked technical and business activity. A later request to change exposure policy produces a revised System, dependency-aware tests and a new approval decision, rather than a separately maintained supplier-finance application. This is an illustration of the operating model, not a prescribed AISDL design or a claim of an already completed production benchmark.

Definition 1.1 (AISDL System revision). *An immutable specification S_v of a business System at version v, containing the business model, executable restricted Process logic, dependencies, declared technical capabilities, policy annotations, documentation, and acceptance evidence required by the publication policy.*

Definition 1.2 (Runtime conformance). *A runtime R conforms to an AISDL profile $\mathcal{P}$ when every admitted $S \in \mathcal{P}$ is validated and executed with the specified observable behavior and enforced capability, isolation, and resource policies, subject to the explicitly documented external-environment assumptions.*

A security or regulatory annotation may identify an intended control and the evidence required to test it. For example, an annotation referring to PCI DSS or ISO/IEC 27001 is *not* itself a certificate of compliance. Certification depends on implementation, operational scope, assessment, and evidence. For industry-specific designs, such as payments, banking, transport, or TM Forum-aligned telecom operations, the language must express relevant domain obligations without silently asserting conformity to the standards.

The restricted language must remain expressive enough for calculations, conditions, loops, typed data, validations and business state transitions, while denying arbitrary host effects. Database queries, cache operations, messaging, managed files, Secrets and outbound partner requests pass through the defined Service interfaces. The Runtime mediates those effects under its permissions, schemas, deadlines, quotas and audit policy. Executable Process logic therefore remains within the Runtime's technical control boundary rather than introducing an independent application infrastructure.

Let $\mathcal{C}(S)$ be the capability set declared by specification S and $\mathsf{Effects}(P)$ the externally visible effects of Process P. The design obligation is

$$\mathsf{Effects}(P) \subseteq \mathcal{C}(S), \qquad \mathsf{DirectHostEffects}(P) = \varnothing. \tag{1}$$

This is a target property of the language, compiler, and runtime taken together. A compiler that accidentally introduces an unrestricted network operation would violate the intended contract even if the source language forbade it.

Resource controls must include not only CPU and memory but wall-clock execution, outbound latency, request counts, concurrency, and external partner budgets. For a Process invocation p, a possible admission and enforcement policy is

$$\begin{aligned} T_{\mathsf{wall}}(p) &\le B_t, & M_{\mathsf{peak}}(p) &\le B_m, \\ N_{\mathsf{out}}(p) &\le B_n, & \sum_{j=1}^{N_{\mathsf{out}}(p)} c_j &\le B_c, \\ T_{\mathsf{call},j}(p) &\le d_j, & \mathsf{Destination}_j &\in \mathcal{A}_p. \end{aligned} \tag{2}$$

Here B_t, B_m, B_n, B_c are policy budgets, d_j are call deadlines, and $\mathcal{A}_p$ is the allowed destination set. This is a proposed control profile, not a claim that every current runtime enforces every bound per invocation. Limits on wall-clock time must account for queued work and cancellation semantics; timeouts do not imply remote operations were rolled back. The runtime exposes the documented error and correlation information. The authored business Processes decide how to persist an uncertain outcome, inquire, reconcile or compensate. A timeout alone must never be treated as proof that an external action did not happen.

Sandboxing must be enforced across compilation, generated code, libraries, services, and infrastructure. Conformance requires adversarial tests for escape attempts, denial of service, secret leakage, tenant isolation, deadline overruns, and inconsistent transaction effects. A restricted language is a prerequisite for the proposed guaran-

tees, not proof of them.

Let R denote the runtime and Π_R its mandatory technical policy. A desired safety obligation is

$$\mathsf{Traces}(D, R) \subseteq \mathsf{Allowed}(\Pi_R). \tag{3}$$

This implication holds only if the runtime correctly implements its enforcement mechanisms and completely mediates relevant effects. Runtime confinement does not prove business correctness. Independently specified business invariants must also hold:

$$\forall \tau \in \mathsf{Traces}(D, R), \qquad \mathcal{I}_D(\tau) = \mathsf{true}. \tag{4}$$

For example, a payment operation might require

$$\sum_j \mathsf{Debit}_j = \sum_j \mathsf{Credit}_j, \qquad \mathsf{Captured} \leq \mathsf{Authorized}, \qquad \mathsf{Refunded} \leq \mathsf{Captured}, \tag{5}$$

with precise adjustments for reversals, partial captures, currencies, timing, and ledger states. These examples are domain-dependent invariants, not complete payment specifications.

Bounded execution can be specified as

$$\begin{aligned} \mathsf{CPU}(P) &\leq b_{\mathsf{cpu}}, \quad & \mathsf{Memory}(P) &\leq b_{\mathsf{mem}}, \\ \mathsf{Time}(P) &\leq b_{\mathsf{time}}, & \mathsf{Calls}(P) &\subseteq \mathcal{D}_P. \end{aligned} \tag{6}$$

A centralized runtime reduces repeated control implementation but creates correlated failure risk. Therefore version isolation, staged rollout, independent security testing, recovery exercises, and portability are first-class requirements, not optional operational details. NIST's secure development framework offers relevant assurance context, though it does not certify this architecture [3].

1.5 A defined audit boundary

The Runtime brings the technical execution surface of its Processes into one governed boundary: entry points, identity enforcement, access to Services and Secrets, data operations and execution evidence. Auditors can assess these common controls at the Runtime level, then examine each AISDL System for its business rules, data handling, Process interactions and approved changes. They no longer need to reconstruct an unrelated application architecture for each product.

A stable, assessed Runtime version allows common control evidence to be reused across Systems under the same control configuration. Changes to the Runtime, configuration or dependencies require reassessment. Each AISDL revision remains subject to review of its business effects and compliance obligations. Infrastructure and external providers retain their own assessment scope; a Runtime assessment is not automatic certification of everything it executes.

1.6 Performance, security and predictability

The architecture must earn three technical advantages: better execution performance, stronger security and more predictable behavior. They are distinct from faster design. A System produced quickly is not necessarily fast, secure or predictable when running. AIScript's restricted interfaces and the Runtime's common controls provide mechanisms to test, not permission to assert superiority without comparison.

Performance. At the same load, resources and correct business outcome, let baseline mean latency be L_B, avoidable application overhead be d, and added Runtime enforcement overhead be h. A simple matched-path model gives

$$L_R = L_B - d + h, \qquad \frac{L_B}{L_R} > 1 \iff d > h. \tag{7}$$

This assumes $L_B > 0$ and $L_R > 0$. Reused Service implementations, fewer redundant layers and optimized execution may increase d; validation and mediation contribute to h. A well-engineered conventional application may already have little removable overhead. Measure throughput, p95/p99 latency, resource use and cost across load levels as well as mean latency; the equation alone does not model queuing or saturation.

Security. Restricting effects to declared Services makes a smaller permission surface possible. The intended structural property is

$$\mathsf{Effects}(S) \subseteq \mathsf{GrantedEffects}(S). \tag{8}$$

Its enforcement must cover compilation, execution, Service implementations and configuration. A defect in any of these can invalidate the property. This is a testable control claim, not the claim that every granted operation is safe or that sensitive data cannot be misused through a legitimate interface.

To illustrate control reuse, consider one specified class of security failure across m Systems over the same assessment period. Let ϵ_C bound the probability for each conventional System, ϵ_R bound a common Runtime failure, and ϵ_S bound each AISDL System's remaining failure probability. The union bound gives

$$\begin{aligned} \mathsf{Pr}(E_C) &\le \mathsf{min}(1, m\epsilon_C), \\ \mathsf{Pr}(E_R) &\le \mathsf{min}(1, \epsilon_R + m\epsilon_S). \end{aligned} \tag{9}$$

No independence is assumed. A smaller upper bound is not proof that the actual failure probability is smaller. The bounds require evidence, including common Runtime and infrastructure weaknesses; common-mode failure can affect many Systems at once, so expected loss and blast radius must be evaluated separately. The model explains the opportunity to replace repeated control defects with assessed shared controls, while preserving System-specific assurance. It is not an estimate of breach frequency.

Predictability. Fixed contracts, explicit state transitions, bounded operations and defined recovery make behavior easier to specify and test. They do not remove variation from external providers, input data, concurrency or infrastructure. For a latency illustration only, assume additive, uncorrelated external and internal contributions, with unchanged external variance. If the Runtime reduces internal variance by a factor $0 \leq \kappa \leq 1$, then

$$\frac{\sigma_R}{\sigma_B} = \sqrt{\frac{\sigma_E^2 + \kappa\sigma_I^2}{\sigma_E^2 + \sigma_I^2}} = \sqrt{f + (1-f)\kappa}, \quad f = \frac{\sigma_E^2}{\sigma_E^2 + \sigma_I^2}. \tag{10}$$

The denominator is positive. As the external share f approaches one, the benefit tends to zero even with strong internal control. Correlated components require covariance terms; the chart does not claim they are absent in production. Behavioral predictability also requires contract-conformance, duplicate handling, concurrency, timeout and recovery tests; low latency variance alone does not establish it.

The relevant proof has three layers: mathematical implications under explicit assumptions; conformance evidence that the implementation enforces the claimed boundaries and semantics; and matched operational benchmarks demonstrating the actual advantage. AISDL plus a Runtime is the proposed mechanism linking these layers, not a substitute for any of them.

2 An open specification and competing Runtimes

2.1 One framework, three names

The **AISDL Specification** defines the System and its resource contracts. **AISDL AIScript** defines restricted executable Process logic. An **AISDL Runtime** implements the supported contracts and controls execution. The names describe different responsibilities, not three competing standards. This publication proposes an open community specification; it does not claim recognition by a standards body or certification of an implementation.

2.2 Utilities and Services can grow without ambiguity

AISDL defines how enterprise systems exchange information through HTTP, TCP and messaging Services. Additional communication methods can be introduced through documented contracts, with every interaction remaining under the control of the AISDL Runtime.

The common reference supplies utilities such as Dates and defined Services for persistence, communications and other enterprise needs. It need not anticipate every future capability. A utility exposes computations over defined values. A Service exposes a capability with its configuration, permissions and execution lifecycle. Neither is an invitation to invent method names or escape into arbitrary host code.

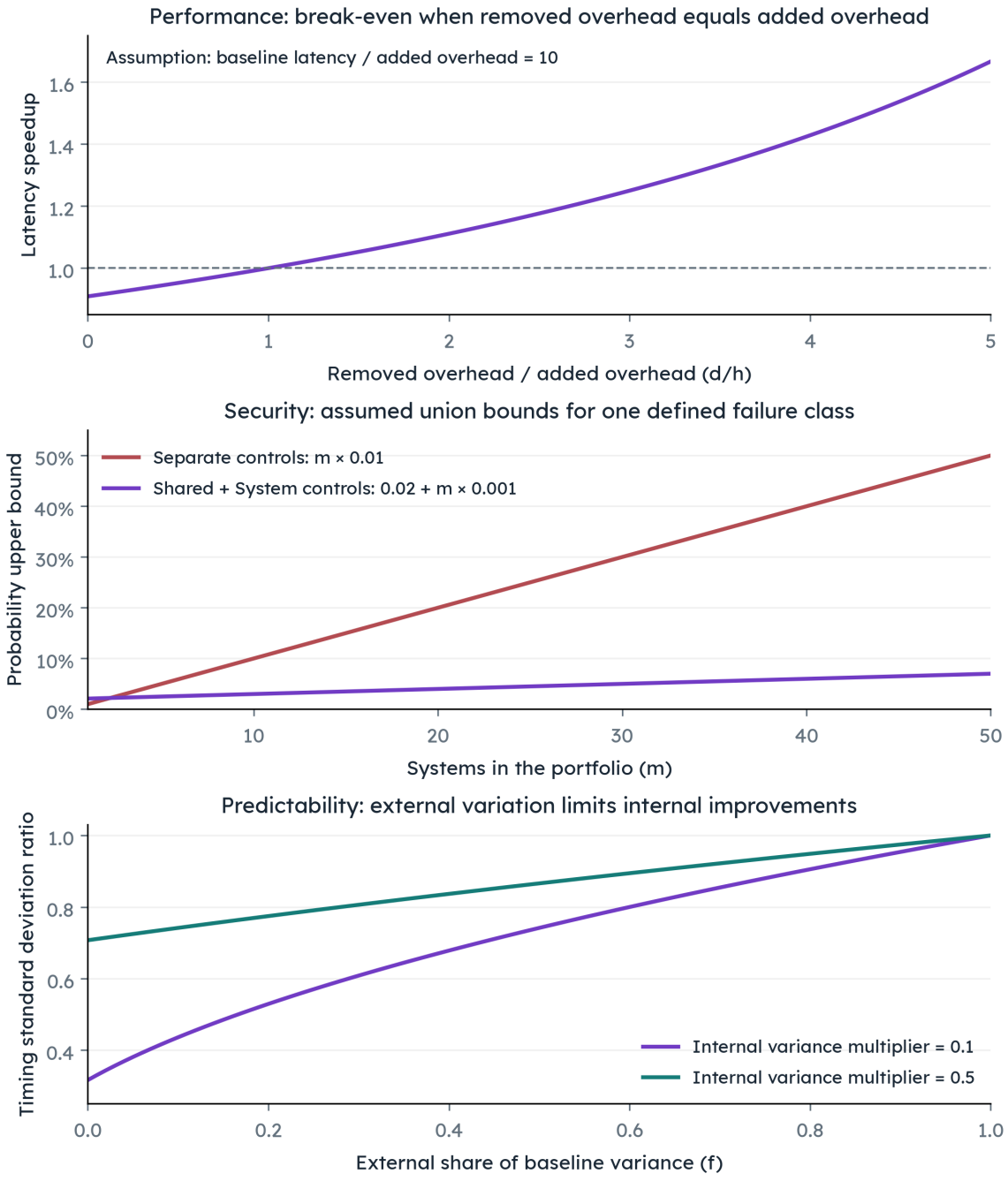

Figure 2: Illustrative technical conditions, not benchmark results. Top: speed improves only when removed overhead exceeds added enforcement overhead. Middle: conditional upper bounds for one security-failure class, not observed incident rates or proof of risk ordering. Bottom: reduced internal timing variation helps less when external variation dominates. Every parameter is an assumption stated on the chart.

The current common Service contracts cover database records and queries; HTTP requests; TCP messages and protocol framing; SMTP email; local and shared caching; a messaging bus; managed file and folder storage; and approved Secrets operations. Utilities cover text, numbers, objects, arrays, value checks, dates, durations and bytes. These are a common enterprise foundation, not a closed inventory of every future need. Additional Services and Utilities require explicit typed contracts and an implementation supported by the target Runtime. Their execution remains subject to the same capability, resource and security boundaries; an extension cannot grant Process code unrestricted host authority.

AIScript is defined by a versioned language contract, not by example programs or a general-purpose host language. EBNF grammar defines its syntax. Core type definitions describe primitive values, objects, arrays, dates, datetimes, durations and bytes; execution semantics define scope, calculations, failures and asynchronous behavior. Service and Utility contracts define exact operations, argument order, accepted types and return types. A contract can introduce named types of its own; every object type supplies a schema. Typed handles expose only their declared operations and do not provide access to the host environment.

The language manifest identifies the complete contract and every required file. The Designer and Runtime implementor use that same definition. A System records the exact language, Service and Utility contracts it requires. Services supply declared capabilities; Utilities supply computations over values. Extension packages identify their owner, unique contract names, versions, additional types and behavior. They extend available capabilities without changing the grammar or silently replacing an existing interface. Downloading a contract supplies knowledge, not an executable implementation; the target Runtime must explicitly support it.

Examples are not normative interfaces. Conformance tests are maintained separately and check the stated contract, including failure behavior and effect boundaries. A grammar or schema check establishes only the properties it actually verifies; implementation conformance and business correctness require their own evidence.

A provider extension remains a provider extension until accepted into a versioned common reference. A proposed amendment includes its rationale, complete typed contract, compatibility analysis and separate conformance tests. Community acceptance and a published revision make it part of that reference; adoption by one provider does not. Incompatible behavior requires a distinguishable version or name. Silent changes under an unchanged interface destroy both reliable generation and portability.

2.3 Execution confinement with extensions

Let C_0 be the approved core capabilities and E the admitted extensions. Let C_e be an extension's explicitly approved effect set. If the execution engine completely mediates every relevant effect, each adapter respects its declared set, and admission

preserves these conditions, then

$$\textsf{Effects}(S, R) \subseteq C_0 \cup \bigcup_{e \in E} C_e. \tag{11}$$

The result follows by considering each effect-producing step: it is either a core operation or an admitted extension operation. Pure utility steps add no external effects. The guarantee is lost if an extension exposes an unrestricted host handle or performs undeclared effects. More interfaces can increase expressive power without removing the boundary, but only when their implementations preserve mediation. Extension approval therefore includes security and resource behavior, not just a method list.

2.4 Portability is compatibility, not identical infrastructure

Let $K(S)$ be the set of versioned contracts required by a System, $K(R)$ the contracts a Runtime supports, and $\textsf{Env}(S, R)$ the required data, credentials, policies and partner conditions. Eligibility to move between two providers requires

$$K(S) \subseteq K(R_1) \cap K(R_2), \qquad \textsf{Env}(S, R_1) \simeq \textsf{Env}(S, R_2). \tag{12}$$

Environment equivalence concerns the business-relevant conditions, not identical hardware. A backend adapter identifier in a resource is an additional required contract until an explicit compatible mapping is established. Portability cannot be inferred by ignoring it. The same applies to a provider-specific utility.

If both implementations preserve the specified semantics, equivalent initial business state and equivalent external responses yield equivalent observable business outcomes:

$$\textsf{Obs}(R_1, S, x) \equiv_{\mathcal{K}(S)} \textsf{Obs}(R_2, S, x). \tag{13}$$

Here $\mathcal{K}(S)$ is the applicable semantic contract. It may allow concurrency or nondeterminism: equivalent outcomes must satisfy the same permitted behavior, not identical timestamps, physical plans or random choices. The proof obligation is semantic preservation by each Runtime, not a theorem that any two implementations happen to agree. Testing provides evidence over tested cases; stronger formal claims require their own proof.

A provider change also transfers business data and required history, replaces credential provisioning, maps integrations and policies, and resolves in-flight work. A cutover must avoid concurrent conflicting ownership. Specification portability is necessary but does not perform this migration automatically. Providers should publish support matrices, test results, limits and export procedures so customers can assess the move before committing to it.

2.5 Shared assurance changes the cost of audit

For m Systems, let a_j be System-specific business assurance, h_j repeated technical assessment in an application model, A_R the common Runtime assessment, and u_j the verification that System j uses that assessed configuration correctly. A simple decomposition is

$$A_{\text{apps}} = \sum_{j=1}^{m}(a_j + h_j), \qquad A_{\text{AISDL}} = A_R + \sum_{j=1}^{m}(a_j + u_j). \tag{14}$$

Shared assurance saves work when $A_R + \sum_j u_j < \sum_j h_j$. The inequality is a condition to measure, not an automatic saving or a certification shortcut. Runtime changes, configuration drift and common vulnerabilities affect the reusable evidence. Business controls, external providers and infrastructure still need assessment in their own scope.

3 The long attempt to remove the IT bottleneck

The historical pattern. Each wave moved a technical burden away from the business: integration, process modeling, decision rules, service access, hosting or provisioning. The gains were real. What persisted was the need for specialists to translate a business change into executable behavior. The common historical interpretation below is a thesis-level synthesis, not a claim that every technology was invented for this single purpose.

ERP unified records and cross-functional transactions and remains an enormous operational achievement. Shared accounting, procurement, manufacturing, and inventory models reduced fragmentation. But packaged processes reflected vendor design assumptions. A business change could require configuration, custom code, partner consultants, data migration, and upgrade reconciliation. The application remained the unit that the enterprise had to adopt and preserve. ERP solved fragmentation by integration; it did not make the business independent of application engineering.

The proper historical judgment is not that ERP failed in general. It succeeded at standardization and control. It fell short of a more ambitious criterion: allowing a business to define and change its executable operating model without dependence on the lifecycle of vendor-defined applications.

BPM's central insight was that the process, not the application screen or program file, should be the business abstraction. A process model could expose work, decisions, handoffs, and exceptions; BPMN provides a standard notation for such models [4]. Business rules management systems (BRMS) promised to let non-developers change decision logic without rewriting programs. This was an effort to remove IT as the bottleneck to business change, not merely a campaign for faster coding.

Yet an executable process needs more than activities and gateways. It needs typed

facts, authoritative records, concurrency semantics, transaction boundaries, external services, permissions, error handling, and recovery. Business users could adjust a threshold or a flow, but developers still defined the facts and services on which those decisions depended. A rule such as "approve when exposure is below the limit" is incomplete until exposure, timing, currency, exceptions, and posting effects have executable meaning.

APIs intensified this boundary. Rules might call APIs, but an API contract is not a complete business guarantee. It may hide an external data model, ambiguous failure states, eventual consistency, rate limits, or incompatible versions. BPM plus BRMS plus APIs often redistributed responsibilities among analysts and developers without removing their coupling. The promise was conceptually sound; the intelligence needed to author a complete executable business system was missing.

SOA, enterprise service buses, REST APIs, and microservices made software capabilities accessible and composable. They enabled reuse and independent deployment, and often improved fault isolation and organizational ownership. But they also preserved the underlying application and service boundaries. A business outcome might traverse many teams, schemas, versions, networks, and security domains. APIs answer *how to call a capability*, not *which capabilities the business needs and how their complete behavior should change*.

Let a business operation depend on a directed service graph $G = (V, E)$. Its integration burden can be approximated by

$$C_{\text{integration}}(G) = \sum_{e \in E} c_e + \sum_{v \in V} c_v^{\text{coord}} + C_{\text{failure}}(G). \tag{15}$$

These terms need not grow quadratically; architecture, shared standards, and topology matter. The point is structural: making more functions callable does not make the business design independent of the contracts between them. Microservices can improve some changes while increasing the distributed consistency, observability, and compatibility burden of others.

The early "no software to install" proposition challenged locally installed enterprise software. Browser-based CRM, subscription delivery, centralized upgrades, and provider-managed infrastructure removed substantial customer operating friction. This was a genuine commercial and technological revolution. But the customer still consumed a vendor-defined application, with objects, permissions, workflows, integration interfaces, customization limits, and a roadmap. "No software to install" was not "no business application to depend upon."

SaaS changed who operated and upgraded applications. IaaS made infrastructure elastic; PaaS standardized application deployment; serverless reduced server administration; cloud-native tooling improved automation. The distinction among cloud service models is formalized by NIST [2]. These technologies changed the economics and location of execution. They did not inherently answer what business behavior ought to exist or give business leaders independent authority to implement it. The distinction

is between *computing operations* and *business design*.

The thesis therefore does not reject cloud. It treats public, private, and sovereign cloud as valid operating locations for the runtime. Cloud is an enabling delivery environment, not the replacement for enterprise applications. The end-state is not "SaaS with AI"; it is business intent executed without application procurement and software-delivery mediation.

3.1 The closest precedents, not just the oldest ones

Model-driven architecture already separates business functionality from platform technology and permits platform-independent models to be realized on different platforms [5]. Executable modeling and low-code platforms go further: the Mendix Runtime, for example, interprets a model, executes business flows and supplies persistence and service access [6]. Durable execution platforms address another part of the problem: Temporal records execution history to support recovery, while application code runs in worker processes [7]. These are substantial precedents, not failed approximations of an idea invented here.

The proposed contribution is a combination: AI takes responsibility for designing, managing, enhancing and maintaining a complete business specification; the business controls activation; and the Runtime executes that specification through shared, governed Services. Neither executable models nor managed execution alone establishes this contribution. The test is whether the combination removes recurring bespoke application engineering and operation across complete business-change cycles, at lower full cost and acceptable risk. A competing platform that achieves the same result would support the thesis, regardless of its branding.

4 Why this is an inflection point

The difference is design intelligence. Previous technologies exposed business logic or made it cheaper to deploy; AI can increasingly propose the logic itself, challenge alternatives and repair inconsistencies. This is valuable only if the output is not another portfolio of applications.

If AI writes conventional applications faster, enterprises may accumulate more independently maintained code, schemas, APIs, dependencies, and deployment units. The speed of production is not the metric that matters. Let $A(t)$ be the stock of durable application artifacts, $\lambda(t)$ the creation rate, and $\mu(t)$ the retirement rate:

$$\frac{dA(t)}{dt} = \lambda(t) - \mu(t). \tag{16}$$

If AI increases λ while retirement and consolidation remain constrained, application stock grows. With a constant retirement hazard $\rho > 0$ and a constant creation rate λ,

the stylized model

$$\frac{dA}{dt} = \lambda - \rho A, \qquad A^* = \frac{\lambda}{\rho} \tag{17}$$

has equilibrium stock A^*. If the average annual governance and maintenance burden per artifact is δ, then

$$M^* = \delta A^* = \frac{\delta \lambda}{\rho}. \tag{18}$$

Thus multiplying generation throughput by k multiplies this *modeled* steady-state burden by k unless retirement, consolidation, or per-artifact cost changes. This is a mechanism, not an empirical prediction: AI may also improve maintenance and retirement.

Amdahl's law provides a second warning. If fraction f of an unchanged workload cannot be accelerated and the remainder is accelerated by factor k, then [1]

$$S(k) = \frac{1}{f + (1-f)/k}, \qquad \lim_{k \to \infty} S(k) = \frac{1}{f}. \tag{19}$$

When coding is only one component of an application delivery lifecycle, accelerating it alone has bounded effect. More importantly, even an infinite coding speed does not remove the need for the application itself. The proposed transition changes the work graph, not just the speed of one node.

General code generation and specialized business design are different tasks. A design agent must reason across contracts, regulations, financial invariants, external interfaces, exceptions, and organizational objectives. Domain-specialized models may outperform general models on these tasks, but specialization is a hypothesis to test rather than a guarantee. Their role is to generate competing complete designs, diagnose defects, and propose controlled improvements.

Let $\mathcal{D}$ be the space of candidate designs and $U(D)$ an independently measured business utility. The design objective is

$$\begin{aligned} D^* = \arg\max_{D \in \mathcal{D}} \ & \mathbb{E}[U(D)] \\ \text{subject to} \quad & \mathsf{Valid}(D) = 1, \\ & \mathsf{Risk}(D) \le r_{\max}, \\ & \mathsf{Cost}(D) \le B, \\ & \mathsf{Verify}(\mathcal{I}_D) = \mathsf{pass}. \end{aligned} \tag{20}$$

The verification oracle must be independent of the generator where feasible; otherwise a model can satisfy its own incomplete interpretation of the requirements. Human approval is especially necessary for high-impact policy, legal, and risk decisions.

A candidate design may be evaluated on a fixed quality-time frontier. Let $\tau_a(q_0)$ be the first time approach a reaches independently accepted quality q_0:

$$F_a(t) = \Pr\left[\tau_a(q_0) \le t\right]. \tag{21}$$

The relevant comparison is not lines of code or generated process count, but the prob-

ability of reaching an acceptable complete business outcome by a deadline, including failures and rework.

4.1 AI design within AISDL, not unrestricted software generation

The central comparison is specific: *AI designs a complete business System in AISDL, with controlled AIScript logic executed by a governed Runtime, versus conventional human-led application delivery.* It does not require AI to outperform every programmer at every task. Data Collections, Processes and their contracts describe the business System; declared Services mediate its effects. The Runtime supplies shared technical mechanisms. AI can concentrate its design effort on business behavior rather than repeatedly constructing application infrastructure.

For a portfolio of m Systems, write the full work required by each approach as

$$\begin{aligned} W_{\text{traditional}}(m) &= \sum_{j=1}^{m}(b_j + a_j + i_j + v_j), \\ W_{\text{AISDL}}(m) &= W_R + \sum_{j=1}^{m}(\tilde{b}_j + \tilde{i}_j + \tilde{v}_j). \end{aligned} \tag{22}$$

Here b is business design and implementation work, a is application-specific engineering and operation, i is integration and v is verification. Tildes denote the corresponding AISDL work, not assumed reductions; W_R includes the portfolio's allocated share of Runtime engineering, operation and assurance over the same period. This decomposition identifies where reuse can remove repeated work. It does not assume that every integration or verification becomes cheaper, or that conventional platforms provide no reuse.

4.2 Retaining improvements rather than trusting every revision

Fix a business objective and its mandatory acceptance criteria. Let $\mathcal{F}$ contain the candidate Systems that satisfy the specified language and contract checks, business requirements, security gates and operational limits. Passing these checks is evidence within their tested scope, not proof of every possible behavior. Let $Q(S)$ be a predeclared quality score, independently evaluated on the same criteria. It may combine business coverage, correctness, recovery and performance; mandatory security obligations cannot be traded for a higher score.

AI authors and critic agents propose candidate revisions $\mathcal{C}_n$. Keep the incumbent when no candidate is better:

$$S_{n+1} \in \operatorname*{arg\,max}_{S \in \{S_n\} \cup (\mathcal{C}_n \cap \mathcal{F})} Q(S), \qquad Q(S_{n+1}) \geq Q(S_n). \tag{23}$$

The inequality follows because the previous accepted System remains available. It proves non-decreasing *evaluated* quality under a fixed evaluation, not that every AI

revision is better. An unchanged benchmark can be overfit; independent held-out scenarios, adversarial testing and post-activation outcomes are needed to challenge it. If the requirements or evaluation change, re-evaluate the incumbent and candidates on the new basis rather than splice incomparable scores together.

Proposition 4.1 (Reaching a defined quality target). *Suppose that, conditional on any preceding history in which the target q^* has not been reached, each completed design-and-review round has probability at least $p > 0$ of reaching it with an accepted candidate. Then*

$$\Pr\{\textit{reach } q^* \textit{ within } n \textit{ rounds}\} \geq 1 - (1-p)^n. \tag{24}$$

Proof. The conditional probability of failing in each round is at most $1 - p$. Multiplying these conditional bounds gives a probability of failing in all n rounds of at most $(1 - p)^n$. Taking the complement proves the result. Independence between rounds is not required; the conditional lower bound is. □

Better models, useful criticism and stronger tests can increase the effective success probability or reduce round duration. More agents alone do not ensure either: they may repeat the same blind spots. If the target cannot be expressed with the available Services, or a search repeatedly stalls, the positive lower bound may not hold. The result concerns a specified target, not proof of a globally best System.

Let $N_a(T)$ be the number of completed, evaluated rounds by time T for approach a, including review and failed work. Under an illustrative fixed rate r_a, $N_a(T) = \lfloor r_a T \rfloor$. The corresponding conditional target bound is

$$F_a(T) \geq 1 - (1 - p_a)^{\lfloor r_a T \rfloor}. \tag{25}$$

Here F_a is equation (21) with $q_0 = q^*$. In real evaluations both round cost and effectiveness can vary. Speed alone is insufficient: many ineffective rounds can be worse than a few strong ones.

4.3 How much iteration is enough

For a target confidence $0 < \alpha < 1$ and the conditional success bound $0 < p < 1$ above, a sufficient number of completed evaluated rounds is

$$n_\alpha = \left\lceil \frac{\ln(1-\alpha)}{\ln(1-p)} \right\rceil. \tag{26}$$

This is obtained by solving $1 - (1 - p)^n \geq \alpha$. If a completed round, including testing and review, takes at most d, sequential completion within dn_α supplies the same conditional confidence. There is no finite bound from this formula when a positive p cannot be justified. At $p = 0.05$ and $\alpha = 0.95$, 59 evaluated rounds suffice under the assumption. Model progress matters through effective review quality and completed round time, not merely larger output.

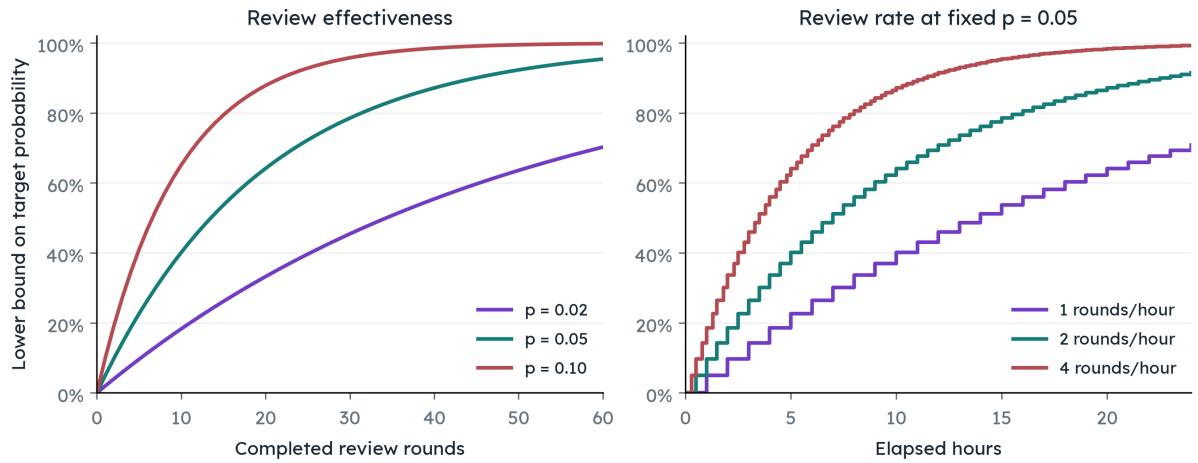

Figure 3: Conditional quality-target bounds, not measured AI performance. Left: the effect of review effectiveness p. Right: the effect of completed rounds per hour at fixed $p = 0.05$. Rates and probabilities are illustrative assumptions; no curve is an observed AI or human result.

4.4 The one-day claim and the economic connection

The statement that AI can produce a better System in one day than a team of x humans is a benchmarkable claim. Define the same 24-hour elapsed window, business scope, external interfaces, mandatory gates and independently evaluated quality for both approaches:

$$Q_A(24\text{h}; B_A) > Q_H(24\text{h}; x, B_H). \tag{27}$$

The budgets B_A and B_H disclose full resource costs; the human team's working hours, AI concurrency and review resources must be recorded. A matched-cost comparison fixes equal budgets; a matched-deadline comparison reports both costs. Repeated runs and uncertainty intervals are necessary, not a selected successful run. Human-designed AISDL Systems form an additional comparison arm to distinguish the benefit of the architecture from the benefit of AI authorship.

Equation (27) is the empirical premise to establish, not a conclusion obtained by assuming AI's rate is higher. Equations (23)–(25) explain how retained improvement and effective iteration can produce an advantage. Once measured, the resulting accepted-change time, quality and full cost feed the economic models that follow. The thesis is therefore a combination: AISDL and the Runtime supply a controlled design and execution environment; AI supplies design and iteration capacity; independent evaluation establishes whether the combination exceeds conventional delivery.

5 The mathematical case for ending application dependence

5.1 What counts as application independence

A claim of architectural elimination is not proved by faster code generation. It requires a complete accepted business capability that can be specified, verified, approved, published and executed without a separately maintained enterprise application stack. Let X be a fixed domain requirement set, $S(X)$ its AI-maintained AISDL specification, R the Runtime executing its approved revision, and $\mathcal{B}(X)$ the set of bespoke application stacks that must be maintained independently of the specification and shared Runtime. Define

$$\text{Direct}(X) = \mathbf{1}\{\text{Accept}(R, S(X), X) = 1 \ \wedge \ |\mathcal{B}(X)| = 0\}. \tag{28}$$

This is a precise acceptance criterion, not an assertion that every business domain already satisfies it. The existence of a conforming runtime and specification demonstrates the execution path; a complete end-to-end accepted system demonstrates $\text{Direct}(X) = 1$ for that particular X. Generalization to a portfolio requires replication across domains, failures, security controls and changing requirements.

The presence of Process code does not turn the specification into a separately operated application stack. Nor do code, APIs or configuration derived by the Runtime invalidate the criterion merely by existing. They do invalidate it if ordinary business changes require specialists to maintain them independently of the AI-maintained specification. The empirical test records every manual technical intervention, inside the enterprise or at a supplier. Outsourcing the same application lifecycle does not establish the proposed separation.

5.2 Three claims that require separate evidence

The three claims must not be conflated. Architectural feasibility requires $\text{Direct}(X) = 1$. Economic dominance additionally requires

$$C_R(X) < \min\{C_B(X), C_V(X)\}, \tag{29}$$

where all costs include design, verification, operations, integration, risk and change. Strict independence from internal application delivery additionally requires $H_{\text{internal IT}}(X) = 0$ for the accepted change class. Here the hours measure bespoke application engineering and release work, not business governance or every technical role. Later portfolio measures track progress toward that criterion; a low ratio is not the same as demonstrated zero dependence. Neither economic advantage nor organizational independence follows from feasibility alone. Conversely, no amount of AI coding speed establishes independence if a separately maintained application stack remains obligatory.

What the equations establish. These models identify conditions and trade-offs, not inevitable outcomes. The relevant metric is independently accepted business change, including verification, integration, risk and operational costs—not lines of generated

code.

Traditional enterprise delivery creates an organizational translation chain: a business need becomes requirements; requirements become IT backlog items; backlog items become code, configuration, interfaces, test environments, and releases; only then does business behavior change. This chain is expensive, but cost and time-to-market are symptoms. The deeper constraint is that the business cannot directly evolve its own operating model. Even an exceptionally fast development department remains a separate intermediary.

Let b denote a proposed business change and let $\tau(b)$ be the elapsed time until that change becomes safely executable and independently accepted. Decompose its critical path into business interpretation, engineering translation, technical implementation, integration, verification, governance, and deployment. A simplified model is

$$\tau(b) = \tau_{\text{intent}} + \tau_{\text{translation}} + \tau_{\text{implementation}} + \tau_{\text{integration}} + \tau_{\text{verification}} + \tau_{\text{approval}}. \tag{30}$$

The sum is an illustrative sequential model; real programs have parallel paths and waiting queues. AI coding attacks mainly $\tau_{\text{implementation}}$. The thesis attacks the architectural necessity of $\tau_{\text{translation}}$ and the repeated application-specific portions of the other terms. It cannot eliminate business judgment, external certification, or independent assurance.

Buying applications can outsource implementation but preserves product-defined semantics, integration, licensing, customization, and upgrade dependencies. Building applications preserves control but retains the software lifecycle. The third option is to design and execute business capability directly.

Let N be the number of independently accepted business capabilities over a common planning horizon. Define the full risk-adjusted costs

$$C_{\text{build}}(N) = F_B + Nc_B + J_B(N), \tag{31}$$

$$C_{\text{buy}}(N) = F_V + Nc_V + J_V(N), \tag{32}$$

$$C_{\text{runtime}}(N) = F_R + N(c_A + c_Q + c_O) + J_R(N), \tag{33}$$

where F denotes fixed investment; c_B includes build, operation, change, and assurance; c_V includes licensing, configuration, integration, governance, and change; c_A is AI design; c_Q is independent verification; c_O is runtime operation; and J captures nonlinear dependencies, correlated risk, and external obligations. All alternatives must deliver equivalent accepted outcomes.

These are non-overlapping accounting categories: a risk allowance already included in a unit cost must not be added again in J. Costs allocated per accepted capability include unsuccessful attempts and AI maintenance across revisions. Shared Runtime engineering, operation and assurance remain real, amortized costs; they are not the operation of a separate application stack for each capability. Repeated customer-specific application repair must be counted as bespoke work rather than hidden in a subscription.

Proposition 5.1 (Conditional replacement advantage). *If, for a defined domain and horizon, $J_B = J_V = J_R = 0$ and $c_P = c_A + c_Q + c_O$ is strictly below both c_B and c_V, then the runtime architecture is less costly than both conventional build and buy whenever*

$$N > \max\left\{\frac{F_R - F_B}{c_B - c_P}, \frac{F_R - F_V}{c_V - c_P}\right\}. \tag{34}$$

If a numerator is negative, the corresponding inequality is already satisfied for non-negative N under the assumed positive denominator.

Proof. Subtract C_{runtime} from each alternative. The two resulting inequalities are $(c_B - c_P)N > F_R - F_B$ and $(c_V - c_P)N > F_R - F_V$. Divide by their positive denominators and require both. □

The proposition is algebraically valid under its assumptions; it does not establish that its assumptions hold in production. In particular, correlated runtime failures, verification cost, integration with external infrastructure, and existing vendor reuse may reverse the result.

Change is the more important unit than initial launch. If χ is the annual rate of meaningful business changes and h is the horizon, a stylized lifecycle extension is

$$C^{(h)} = C^{(0)} + h\chi\, c_{\text{accepted change}} + \mathbb{E}[L_{\text{risk}}]. \tag{35}$$

The architecture wins only when accepted change becomes more autonomous *and* the full risk-adjusted lifecycle burden declines.

6 From IT releases to business publication

The unit of change is no longer an application release. It is a proposed business-system revision: designed, tested, benchmarked, security-vetted, approved and published under immutable versioning and runtime control.

The enterprise-facing unit of change is not a source-code commit, application build, container image, or deployment pipeline. It is an *approved AISDL System revision*. The proposed business lifecycle is: *Design* → *Validate* → *Test* → *Benchmark* → *Security Vetting* → *Business Approval* → *Publish* → *Observe* → *Improve*. AI proposes a complete revision and produces test cases, simulations, regression checks, performance evidence, and an explanation of changed business behavior. Independent verification and business approval gate activation. The runtime publishes an immutable revision and provides compatibility checks, staged activation, rollback or compensating recovery, and audit history.

Let S_v be the active revision and S_{v+1} the candidate. Define the publication decision

$$\text{Publish}(S_{v+1}) = 1 \iff V(S_{v+1}) \wedge T(S_{v+1}) \wedge B(S_{v+1}) \wedge \text{Sec}(S_{v+1}) \wedge A(S_{v+1}) \wedge G(S_{v+1}), \tag{36}$$

where V is specification and invariant validation, T is independent automated test acceptance, B is benchmark acceptance, Sec is mandatory security vetting, A is business authorization, and G is the runtime governance gate. The predicate is a proposed policy abstraction; each term needs measurable criteria and independent evidence. Publication must also define the fate of in-flight transactions, long-running Processes, data migrations, and versioned external contracts.

Runtime providers still engineer, operate, secure and release the Runtime. AI maintains the business specification that runs within it. The narrower and more radical claim is that *ordinary business change no longer requires the enterprise to build, maintain and operate a separate application stack*. Infrastructure tooling remains a Runtime operating concern, whether the environment is provider-hosted or on premises. The business approves behavior and its consequences, not application packages or infrastructure artifacts.

The ability of AI to author a complete System does not confer authority to activate it. Security vetting is a mandatory, independent publication decision, not a descriptive annotation or an optional step inside the AI design loop. The generating model may propose safeguards and test cases, but it must not be able to change the policy that judges its own work, grant itself new capabilities, or waive adverse findings.

6.1 What the gate examines

For a candidate revision S_{v+1}, vetting examines both the full specification and its *difference* from the currently approved revision S_v. It includes declared data access and tenant boundaries; identities, roles and segregation of duties; restricted-language semantics and compiler artifacts; resource and time budgets; secrets and cryptographic use; permitted outbound destinations and data transfer; message queues and event consumers; external partner contracts; privacy and retention; transaction invariants; and recovery from uncertain outcomes. A change that adds a network destination, broadens a privilege, weakens an invariant, or increases an execution budget deserves heightened scrutiny even if it passes functional tests.

Compliance annotations identify intended controls and evidence obligations. References to PCI DSS, ISO/IEC 27001 or industry standards do not establish certification. Vetting must distinguish *declared coverage*, *tested enforcement*, *operational evidence* and *external assessment*.

6.2 A non-bypassable authorization rule

The conjunction in equation (36) is a policy requirement, not a proof that each gate is correctly implemented. For a revision with material privilege expansion, the security

predicate may itself require a separate human or independent security authority:

$$\mathsf{Sec}(S_{v+1}) = \mathsf{AutomatedClearance}(S_{v+1}) \wedge \big(\neg H(S_v, S_{v+1}) \vee \mathsf{IndependentSignoff}(S_{v+1})\big), \tag{37}$$

where H identifies a high-risk change under a published classification policy. The AI designer must not control H, the clearance engine or signing keys.

6.3 Security does not stop at publication

A successful vetting decision does not grant unrestricted runtime power. Every permitted operation must still pass runtime authorization, sandbox enforcement, capability mediation and resource controls. Runtime evidence should link an invocation to the immutable specification hash, policy version, approval record and deployed runtime version. This supports reproducible incident investigation and detection of drift.

Publication also requires a plan for in-flight work, schema migration, staged activation and safe rollback. Rolling back a financial Process definition cannot automatically reverse an external payment or erase a posted journal. Recovery must preserve business invariants and externally visible effects.

6.4 The organizational consequence

A No-IT-Department Enterprise does not mean a no-security-governance enterprise. The enterprise retains accountable business and risk authorities; a trusted runtime provider supplies technical enforcement; independent assessors may be required for regulated domains. The claim is that ordinary business changes no longer require an internal application release team, not that security decisions can be delegated without checks to the same AI that authored the change.

7 The economics of competitive business evolution

The central economic mechanism. The decisive benefit is not simply lower technology expenditure. It is the ability to discover, test, approve and implement more economically worthwhile changes, more often, while preserving safety and accountability. This can alter the feasible set of business strategies and, in competitive markets, the relative performance of enterprises.

7.1 The value of an accepted improvement

Consider a common horizon H and an opportunity set $\mathcal{O}_i$ available to enterprise i. Let $\mathcal{T}_i(H) \subseteq \mathcal{O}_i$ be the improvements actually attempted. For each attempted improvement, K_{ij} is the incurred design, integration, testing and review cost, whether accepted or rejected; a_{ij} indicates independent acceptance; and V_{ij} is the discounted

incremental operating contribution if deployed. Let O_{ij} be its additional operating cost and L_{ij} its expected residual loss if deployed. Define risk-adjusted net improvement value

$$W_i(H) = \sum_{j \in \mathcal{T}_i(H)} \left[a_{ij} \left(V_{ij} - O_{ij} - L_{ij} \right) - K_{ij} \right]. \tag{38}$$

All terms are evaluated over the same horizon. A rejected attempt contributes $-K_{ij}$, not zero. Abandoned work, failed tests and repeated reviews remain costs; overlapping improvements must be evaluated jointly. L excludes losses already charged elsewhere. This is risk-adjusted value, not a claim that expected losses equal realized cash outcomes. A faster design system raises W_i only if its accepted benefits exceed all of these costs.

For a candidate already shown to meet the common acceptance criteria, let C_{ij} denote the expected full attempt-plus-operation cost. The feasible improvement set is

$$\mathcal{F}_i = \left\{ j \in \mathcal{O}_i : \mathbb{E}[V_{ij}] - C_{ij} - \mathbb{E}[L_{ij}] > 0,\ \text{Accept}_{ij} = 1 \right\}. \tag{39}$$

Lower design and change costs can enlarge $\mathcal{F}_i$. This is the *extensive-margin effect*: valuable but previously uneconomic improvements become worth implementing. Improved quality and faster iteration create an *intensive-margin effect*: existing operations become better.

7.2 A proposition on the opportunity frontier

Proposition 7.1 (Expansion under lower marginal cost). *Let two delivery architectures face the same opportunity set, expected benefits, risk losses and acceptance criteria. If architecture R has $C_{R,j} \leq C_{T,j}$ for every j, then $\mathcal{F}_T \subseteq \mathcal{F}_R$. If at least one opportunity has*

$$C_{R,j} < \mathbb{E}[V_j] - \mathbb{E}[L_j] \leq C_{T,j} \tag{40}$$

and passes the common acceptance gates, the inclusion is strict.

Proof. Any opportunity with positive net value under T also has positive net value under R by cost monotonicity. The displayed strict interval identifies an opportunity feasible only under R. □

This result is conditional: runtime integration, independent assurance, provider fees and correlated risk must be included in $C_{R,j}$ and $L_{R,j}$. Lower code-generation cost alone is insufficient.

7.3 Business-change capacity, not developer throughput

Let λ_i be the attempted improvement rate, p_i the probability of independently accepted deployment, $\bar{u}_i = \mathbb{E}[V - O - L \mid a = 1]$ the expected value of a deployed improvement before attempt costs, and $\bar{K}_i$ the expected cost per attempt, including

unsuccessful ones. Under a stationary, additive approximation, the rate at which expected horizon-valued contribution is added is

$$\mathbb{E}[\dot{W}_i] \approx \lambda_i \left(p_i \bar{u}_i - \bar{K}_i\right). \tag{41}$$

The expression exposes throughput, acceptance quality, accepted value and the cost of trying. It is not a cash-receipt rate: benefits accrue over the common valuation horizon. Increasing λ_i destroys value when the expected value per attempt is negative. AI-native design is economically consequential only if it improves net value after counting failures, reviews and operational losses.

Where scarce business approval, integration capacity or regulatory review constrains throughput, the sustainable accepted rate cannot exceed its bottleneck:

$$\lambda_i p_i \leq \min\{\kappa_{\text{design}}, \kappa_{\text{verify}}, \kappa_{\text{approve}}, \kappa_{\text{integrate}}\}. \tag{42}$$

A controlled runtime can reduce recurring technical bottlenecks, but cannot abolish every external constraint.

7.4 One market, two operating models

Consider two enterprises A (AI-native) and T (traditional) competing for the same time-sensitive opportunity with demand D, contribution margin m, innovation-market fit q, and opportunity-decay rate λ. The comparison holds those market variables constant, while separately measuring complete delivery cost C, launch time T, and expected incremental risk loss L:

$$\Pi_A - \Pi_T = mDq\left(e^{-\lambda T_A} - e^{-\lambda T_T}\right) + (C_T - C_A) + (L_T - L_A). \tag{43}$$

The AI-native approach dominates in this scenario if $T_A < T_T$, $C_A < C_T$ and $L_A \leq L_T$, provided $mDq > 0$ and $\lambda > 0$. The risk condition is an independent assurance requirement, not something AI generation guarantees. In particular, security is not interchangeable with cost or speed: both designs must first meet the same mandatory security, compliance and operational acceptance gates.

For the illustrative *tenfold hypothesis*, $T_A = T_T/10$ and $C_A = C_T/10$, with equal independently assessed risk. Equation (43) becomes

$$\Pi_A - \Pi_T = mDq\left(e^{-\lambda T_T/10} - e^{-\lambda T_T}\right) + 0.9C_T > 0. \tag{44}$$

This proves the direction of the *modeled* advantage under the stated conditions, not the empirical tenfold premise. Its magnitude depends on market timing, demand, margins and the actual cost baseline. A faster cycle also creates room for additional experiments and revisions, but ten times the cycle capacity does not imply ten times the commercially accepted innovation: review bottlenecks, correlated failures, demand saturation and diminishing returns still apply.

To connect superiority to *competitive necessity*, let $\Pi_T^{\text{retain}}(t)$ be the incumbent's expected risk-adjusted contribution if it retains its operating model, $F_T(t)$ its unavoidable fixed obligations, and $H_T(t)$ its available financing buffer. A necessary warning condition for financial viability over a horizon H is

$$H_T(0) + \int_0^H \left[\Pi_T^{\text{retain}}(t) - F_T(t)\right] dt < 0. \tag{45}$$

This simplified cash-buffer test does not itself prove failure: financing, strategic responses, customer loyalty, pricing power and regulation can change the outcome. The strong thesis claim is *conditional*: where repeated competitive losses exceed the firm's capacity to adapt or absorb them, non-adoption becomes economically unsustainable. Measuring those conditions is the research program.

7.5 Demand, innovation and time to market

Growth cannot be inferred merely from a faster design process. It depends on the demand available to the enterprise, the commercial relevance of its innovations, and whether those innovations reach the market while the opportunity remains. The proposed architecture acts on the innovation and delivery mechanisms, not on demand by assumption.

For an opportunity j, let D_j be the monetary value of addressable demand at the opportunity's reference date; let $q_{ij} \in [0,1]$ be the share of that demand potentially captured by the quality and fit of enterprise i's innovation; let $T_{ij} \geq 0$ be elapsed time to operational market availability; and let $s_j(T) \in [0,1]$ describe the opportunity remaining at launch. A separable, deliberately testable approximation is

$$\Delta R_{ij} = D_j q_{ij} s_j(T_{ij}), \qquad s_j(T) = e^{-\lambda_j T}, \quad \lambda_j \geq 0. \tag{46}$$

The exponential form is a *hypothesis for time-sensitive opportunities*, not a universal law. In growing markets s_j need not decrease; demand and competitive entry may also change over time. The factor q_{ij} must be estimated from customer evidence, not equated with the number of generated processes.

For two enterprises facing the same opportunity, with equal innovation fit and identical market access, equation (46) yields

$$\frac{\Delta R_{Aj}}{\Delta R_{Tj}} = e^{\lambda_j (T_{Tj} - T_{Aj})}. \tag{47}$$

This ratio describes modeled *incremental opportunity capture*, not total enterprise revenue or growth. With a six-month opportunity half-life, $\lambda_j = \ln(2)/6$ per month; launch after two months preserves $2^{-2/6} \approx 79.4\%$ of the reference opportunity, whereas launch after twelve months preserves 25%. The capture ratio is approximately 3.17. These values illustrate the assumed decay curve, not observed business performance.

The economic comparison must include margin, full costs and expected risk losses:

$$\Delta\Pi_{ij} = m_{ij} D_j q_{ij} e^{-\lambda_j T_{ij}} - C_{ij} - L_{ij}, \tag{48}$$

where m_{ij} is the incremental contribution margin, C_{ij} includes design, independent assurance, integration, deployment and operation over the stated horizon, and L_{ij} is expected incremental loss. This is an *incremental contribution* model; fixed enterprise costs and cannibalization must be added when material.

Proposition 7.2 (Value of earlier launch under a decaying opportunity). *For the same opportunity, innovation fit, contribution margin, full cost and expected loss, with* $mDq > 0$ *and* $\lambda > 0$*, reducing launch time strictly increases modeled incremental contribution. Specifically,*

$$\frac{\partial \Delta\Pi}{\partial T} = -\lambda m D q e^{-\lambda T} < 0. \tag{49}$$

Proof. Differentiate equation (48) with respect to T, holding the stated quantities constant. □

Earlier launch can nevertheless be economically worse if rushing increases costs, lowers innovation fit or creates greater expected losses. The full comparison is $\Delta\Pi_A - \Delta\Pi_T$, not timing alone.

7.6 From operating architecture to innovation capacity

Let b_i be the spending budget per unit time available for attempting business improvements and $\bar{K}_i > 0$ the average cost per attempt. Both use the same accounting basis. Then $\lambda_i \bar{K}_i \leq b_i$. When the expected value per attempt is nonnegative, a simplified upper bound is

$$\mathbb{E}[\dot{W}_i] \lesssim \frac{b_i}{\bar{K}_i} \left(p_i \bar{u}_i - \bar{K}_i \right), \tag{50}$$

subject to the bottlenecks in equation (42). The units are value per time: a spending rate divided by cost per attempt gives attempts per time. This is not a measured productivity law, and spending the entire budget is not rational when expected value per attempt is negative. Lower attempt cost expands experimentation; shorter launch time preserves opportunity; higher acceptance and value make the experiments worthwhile. Diminishing returns, overlap, saturation and scarce approvals still matter.

The causal chain to test is therefore:

controlled AI-native design → lower complete change cost and elapsed time → more accepted, market-relevant innovation → higher risk-adjusted contribution under available demand.

The arrow from architectural change to lower cost and time is a benchmark hypothesis. The arrow from innovation to contribution is mediated by customer demand,

competitive responses, margins and operational risk.

7.7 Competitive divergence and the adoption threshold

For a strictly positive, consistently defined enterprise performance index $P_i(t)$, define $g_i(t) = d \log P_i(t)/dt$. Then

$$\frac{P_A(t)}{P_T(t)} = \frac{P_A(0)}{P_T(0)} \exp\left(\int_0^t [g_A(s) - g_T(s)]\, ds\right). \tag{51}$$

This is an accounting identity, not proof of superior AI-native growth. The demand–innovation–timing model provides a *candidate mechanism* for a positive growth differential, which must be estimated from realized contribution, reinvestment and market-share dynamics. No automatic exponential growth is assumed.

Let K be the one-time migration and organizational transition cost, $d(t)$ the incremental risk-adjusted net cash flow of the AI-native model over the incumbent model, r a discount rate and H a decision horizon. The rational adoption condition is

$$\mathrm{NPV}_{\mathrm{switch}} = -K + \int_0^H e^{-rt} d(t)\, dt > 0. \tag{52}$$

A claim that non-adopters cannot survive requires a separate competitive viability threshold, taking account of demand, market structure, imitation, switching costs, regulation and financial resilience. A positive switching NPV alone does not prove inevitable extinction.

7.8 From operating advantage to market-share transfer

The comparison must hold market demand constant before attributing competitive outcomes to the operating model. Consider a market with total period demand $M_t > 0$ and two enterprises A (AI-native) and T (traditional). Let $x_t \in (0, 1)$ be A's share of demand and $1 - x_t$ be T's share. Define a dimensionless customer-attractiveness score $u_{i,t}$, estimated from observed price, service quality, availability, innovation fit, switching friction and delivery timing. The score must not be identified with development speed alone.

A transparent, testable share-response model is

$$x_t^* = \frac{e^{\beta u_{A,t}}}{e^{\beta u_{A,t}} + e^{\beta u_{T,t}}}, \qquad \log \frac{x_t^*}{1 - x_t^*} = \beta(u_{A,t} - u_{T,t}), \quad \beta > 0. \tag{53}$$

Here x_t^* is the target share implied by customer attractiveness, not necessarily the observed current share x_t. This is a stylized discrete-choice specification, not a universal market law. Switching friction can delay adjustment toward that target. An AI-native cost or delivery advantage affects share only to the extent that it changes customer-perceived value or competitive pricing. Constant relative attractiveness im-

plies a finite target share, not automatic monopoly.

7.9 A cash-flow viability threshold

Let $m_T > 0$ denote the traditional enterprise's cash contribution per unit of market demand after variable operating costs; $F_T > 0$ its unavoidable period fixed cash commitments; and $H_T \geq 0$ the cash contribution from protected contracts or unrelated business. Then its operating cash surplus is

$$S_T(t) = m_T M_t(1 - x_t) + H_T - F_T. \tag{54}$$

This deliberately excludes one-time transition costs and debt principal from F_T only if they are accounted for separately; the same obligation must never be counted twice.

Proposition 7.3 (Critical market share for incumbent operating viability). *Assume $m_T M_t > 0$ and fixed period parameters. Define*

$$x_{\text{crit}}(t) = 1 - \frac{F_T - H_T}{m_T M_t}. \tag{55}$$

Then $S_T(t) \geq 0$ if and only if $x_t \leq x_{\text{crit}}(t)$. If $0 < x_{\text{crit}} < 1$, crossing above the threshold makes the incumbent's ongoing operating cash surplus negative.

Proof. Rearranging $m_T M_t(1 - x_t) + H_T - F_T \geq 0$ and dividing by the positive quantity $m_T M_t$ gives the stated inequality. □

If $x_{\text{crit}} \geq 1$, the incumbent's protected contribution covers fixed commitments even without contestable market share; if $x_{\text{crit}} \leq 0$, the incumbent is not viable even with the entire contestable market, except for the boundary case of equality. The threshold is a condition for *period operating viability*, not a bankruptcy prediction.

7.10 Competitive transition time and the cost of delay

Under constant attractiveness and a fixed target share x^* from equation (53), a consistent continuous-time adjustment model is

$$\frac{dx}{dt} = k(x^* - x), \qquad k > 0, \tag{56}$$

where k is an empirically calibrated adjustment rate; it is not inferred from a tenfold development-speed assumption. Its solution for initial share $x_0 \in (0, 1)$ is

$$x(t) = x^* + (x_0 - x^*)e^{-kt}. \tag{57}$$

Proposition 7.4 (Time to the viability boundary). *For constant M, m_T, F_T, H_T, assume*

$0 < x_0 < x_{\text{crit}} < x^* < 1$ *and* $k > 0$*. The first zero of* $S_T(t)$ *occurs at*

$$t_{\text{crit}} = \frac{1}{k} \log\left(\frac{x^* - x_0}{x^* - x_{\text{crit}}}\right). \tag{58}$$

Proof. Set $x(t) = x_{\text{crit}}$ in equation (57) and solve for t. Strict inequalities ensure a positive finite result. If $x^* \leq x_{\text{crit}}$ and the incumbent starts viable, there is no finite crossing under this constant-target model. □

A transition cannot be judged only by t_{crit}. Let $B_0 \geq 0$ be the incumbent's initial liquid reserve, K a transition cash outlay, and $S_T(t)$ the operating surplus along the actual path. Cash remains nonnegative through horizon h only if

$$B(t) = B_0 - K + \int_0^t S_T(s)\, ds \geq 0 \quad \text{for every } t \in [0, h]. \tag{59}$$

If the incumbent's transition takes τ and only begins after delay d, a necessary planning test is that the pre-transition cash path remains nonnegative for every $t \leq d + \tau$. This is *not* sufficient for recovery: the post-transition operating model must also restore a viable surplus. The latest safe start date is consequently an output of calibrated share dynamics, cash reserves, implementation lead time and post-transition recovery—not a universal countdown.

7.11 Worked sensitivity example: assumptions, not evidence

Suppose a fixed market has $M = 100$ monetary units per month, incumbent contribution margin $m_T = 0.30$, fixed commitments $F_T = 12$, and no protected contribution ($H_T = 0$). Equation (55) gives $x_{\text{crit}} = 0.60$. Suppose the AI-native entrant starts at $x_0 = 0.20$, customer attractiveness implies a target share $x^* = 0.80$, and the adjustment rate is hypothetically $k = 0.10$ per month. Equation (58) then gives $t_{\text{crit}} = 10 \ln 3 \approx 11.0$ months. Halving k moves the boundary to approximately 22.0 months; doubling it brings the boundary to approximately 5.5 months. If the target share were instead 0.50, the boundary would never be crossed in this model. None of these are observed market data. They demonstrate that *customer preference and switching*, not implementation speed alone, determine whether a viability boundary is reached at all.

A claim of a 10-fold cost and speed improvement must therefore be validated in two distinct stages: first, matched architecture benchmarks must establish complete cost, time, security and reliability; second, market observations must establish how those advantages affect customer value, switching, contribution and cash runway. The survival result is conditional on both stages and on the absence of a successful incumbent response.

7.12 An explicit falsification standard

A matched evaluation must compare the same business requirement universe and count full costs, external service fees, verification, incidents, delayed outcomes and the internal labor still required. The economic thesis is weakened or falsified for a tested domain if accepted-change throughput and net value do not improve, if migration costs dominate over realistic horizons, or if an apparently eliminated IT function simply reappears as equivalent vendor labor. The strongest evidence is repeatable, independently audited operating performance across multiple business-change cycles—not a successful prototype alone. The market-demand, innovation-fit, and time-to-market parameters in equations (46)–(48) must be estimated from observed cohorts, not selected to guarantee an AI-native advantage.

8 The no-IT-department enterprise

8.1 Removing the internal IT delivery dependency

The disappearance of the enterprise application is not the ultimate claim; it is the mechanism for an organizational change. If business leaders can express objectives directly to AI, inspect the resulting specification, review independently produced evidence and approve publication, there is no longer a structural requirement for an internal IT department to mediate routine business evolution.

The functions historically grouped under IT do not all vanish. AI takes responsibility for maintaining the AISDL System, while the Runtime and its operators provide execution and common technical controls. Application selection, bespoke implementation, application CI/CD and routine application operation can disappear as enterprise-facing obligations. Runtime engineering, infrastructure operation, incident response and security assurance continue as shared technical functions. Business owners retain responsibility for outcomes, access policies, risk appetite and regulated approvals.

The organizational claim is a testable implication of an executable design and publication mechanism: *the enterprise need not maintain a traditional IT delivery department for ordinary business changes* when design, verification, approval, publication and runtime enforcement are complete without bespoke internal application engineering. If the same engineering and release functions are still required, the implication does not hold for that domain.

8.2 Measuring departmental dependence

Let H_{IT} denote internal bespoke application-engineering and release labor hours attributable to a portfolio of accepted business changes, N_{chg} the number of independently accepted changes and C_{ext} the fully loaded external service cost. Runtime operation, governance and assurance remain in the full cost comparison even when they

are not application-delivery labor. Define

$$D_{\mathsf{IT}} = \frac{H_{\mathsf{IT}}}{N_{\mathsf{chg}}}, \qquad C_{\mathsf{change}} = \frac{C_{\mathsf{internal}} + C_{\mathsf{ext}} + C_{\mathsf{risk}}}{N_{\mathsf{chg}}}. \tag{60}$$

Progress toward the no-IT-delivery-department criterion is measured by declining D_{IT} for a defined class of ordinary changes. Demonstrating zero dependence in that class requires observing no internal bespoke engineering, not merely a small average. It must not conceal the same work inside expensive vendor services, increase security incidents or lower acceptance standards. The organizational claim must therefore be evaluated alongside full lifecycle cost and independently verified outcomes.

The organizational consequence. Businesses still need accountable leaders, risk governance and external technical service providers. They may not need an internal department whose purpose is to convert business requirements into software delivery.

The strongest consequence is organizational. The conventional IT department exists in part because business intent cannot directly become controlled execution. It performs translation, application selection, implementation, integration, and maintenance. If AI designs complete business behavior and the runtime provides common technical guarantees, those tasks need not be staffed as a dedicated internal software-delivery chain.

This is not the elimination of every technology role. The enterprise still needs governance of identity, privacy, procurement, resilience, vendor concentration, data stewardship, and regulatory accountability. Runtime providers need engineers and security specialists. Some organizations may retain technical oversight teams, especially where physical operations, custom hardware, classified environments, or high-risk regulation demand them. The precise prediction is that *a separate IT organization is no longer structurally necessary to implement each ordinary business change.*

A falsifiable organizational metric is the share of accepted business changes requiring bespoke internal software engineering:

$$\eta = \frac{\#\{b : \text{accepted change } b \text{ required internal application engineering}\}}{\#\{b : \text{accepted business changes}\}}. \tag{61}$$

The portfolio transition is $\eta \to 0$ for a defined class of accepted business changes, without deterioration in safety, compliance, or business outcomes. Strict independence for an observed change class means $\eta = 0$ for that class. It does not establish zero dependence in all industries or all future conditions.

Payments, acquiring, telecommunications charging and provisioning, banking operations, insurance servicing, procurement, inventory, and utilities are plausible research domains because they contain explicit transactions, high change costs, auditable invariants, and recurring integration patterns. They are not equally ready for adoption. Physical control, medical safety, classified environments, legacy data quality, and external certification may create irreducible dependencies. The strongest early claim concerns *transaction-intensive digital business operations*, not all organizational ac-

tivity.

The enterprise can still use specialized interfaces for humans, devices, and partners. These interfaces should ideally be generated or governed from process contracts rather than treated as independent business applications. The difference is not whether a screen exists; it is whether the screen dictates the business model and imposes its own durable application lifecycle.

9 The strongest objections and how to test the thesis

A thesis must be vulnerable to evidence. A system that merely renames applications, requires the same engineering labor, or shifts unacceptable risk into a shared runtime does not satisfy the claim.

9.1 Is a Process System simply another application?

The distinction cannot rest on renaming software. The Runtime executes an AI-maintained specification whose Process code is restricted to its defined technical interfaces. The business authorizes that specification instead of assuming an application engineering and operating lifecycle. The test must establish that shared execution controls and AI maintenance actually remove those recurring obligations. If a separate application stack still has to be maintained for each System, the thesis fails its own criterion.

9.2 Can AI really understand the entire business?

Not reliably today in every domain. Requirements are incomplete, contradictory, and politically contested. Tacit knowledge is difficult to elicit. The thesis therefore requires independent oracles, staged deployment, feedback, and explicit governance. If AI repeatedly misses cross-process invariants or requires equivalent developer labor to repair them, the predicted organizational transition is unsupported.

9.3 Does runtime centralization create a single point of failure?

It can. Shared controls amortize assurance but concentrate risk. Tenant isolation, version pinning, blast-radius control, disaster recovery, portability, and independent auditing must be measured. A cheaper architecture with unacceptable systemic risk is not a successful replacement.

9.4 Can SaaS and ERP vendors simply absorb this model?

Yes. The thesis is architectural, not a prediction that incumbent companies must disappear. A vendor could evolve from selling applications to providing controlled execution and AI design. If the same organization and lifecycle remain necessary, however, merely adding AI assistants to existing SaaS does not meet the thesis.

9.5 Do regulators require an IT department?

Regulation often requires accountable controls, competence, evidence, and oversight, not necessarily a department with a particular name. But sector-specific requirements may mandate responsible technical roles or independent functions. Claims about organizational elimination must be tested jurisdiction by jurisdiction.

A rigorous publication separates constructive implementation evidence, mathematical implications and independently measured production outcomes. The experimental unit is an independently specified, end-to-end business capability, not a generated code file. Compare five arms: expert human teams using conventional applications; general AI generating conventional applications; human-designed Processes on a controlled runtime; general AI designing Processes; and specialized AI designing Processes. All arms face the same acceptance requirements and external services.

1. **Pre-register scope:** include normal operations, policy changes, concurrency, integration failures, exceptions, recovery, privacy, security, and regulatory constraints.
2. **Use independent evaluators:** separate design agents from the teams writing acceptance tests and domain invariants.
3. **Establish the oracle:** derive acceptance cases from business objectives, independently held domain knowledge and external interface contracts. Include held-out scenarios, accounting or conservation properties, and adversarial cases. A second model is an aid, not proof of independence; models can share blind spots. Freeze mandatory criteria before comparing approaches and record any later amendment.
4. **Measure complete outcomes:** time to accepted execution, critical defects, full cost, number of manual interventions, dependency burden, operational resilience, and post-launch change latency.
5. **Separate design from execution advantages:** test the one-day design comparison in equation (27) over repeated runs. For accepted Systems, compare throughput, tail latency and resource cost; independently test Service boundaries, isolation and attack resistance; then exercise contract consistency, timing variation and failure recovery. Use the same business workload, deployment resources and external dependencies. Report common Runtime failures and correlated losses, not just successful individual Process calls.

6. **Define statistical acceptance:** pre-specify the relevant noninferiority margins, sample sizes, observation periods and confidence intervals. Zero observed critical failures in a small test set is not evidence of zero failure probability. Mandatory safety and compliance thresholds remain gates, not tradeable speed benefits.
7. **Test change, not just creation:** introduce unanticipated rule changes, new providers, regulatory updates, and recovery scenarios after initial acceptance.
8. **Measure organizational mediation:** record whether a change required bespoke internal engineering and the nature of that intervention.
9. **Publish failures:** count rejected designs, retries, model inference costs, review effort, certification delays, and runtime incidents.

Hypothesis 9.1 (Business-change autonomy)**.** *For a specified class of enterprise operations, specialized-AI-designed Processes on a controlled runtime achieve a lower median accepted-change lead time and lower IT-mediation ratio η than conventional application approaches, with noninferior critical-defect and regulatory outcomes.*

Hypothesis 9.2 (Lifecycle advantage)**.** *For sufficiently broad portfolios of accepted capabilities, full risk-adjusted cost per accepted change is lower under AI plus controlled runtime than under both build and buy, after counting runtime assurance, integration, and correlated failures.*

Hypothesis 9.3 (Business-design completeness)**.** *Specialized AI produces more complete end-to-end contracts, invariants, and recovery behavior than general AI coding under a fixed time and resource budget, as judged by independent blind reviewers.*

Evidence against any of these hypotheses narrows or falsifies the corresponding claim. In particular, if the runtime requires comparable custom engineering for every new process, if independent verification consumes the saved development labor, or if common-mode failures dominate savings, the radical prediction does not hold.

10 Conclusion: The AISDL architectural model

The central proposition is not simply that AI can build enterprise software faster. It is that AI should design and evolve business Systems without generating a new, unrestricted execution environment for each one. AISDL and AIScript give the Designer an expressive but controlled authoring boundary; the Runtime enforces the technical authority of the approved result. This restriction is the enabling condition of the architecture, not a qualification at its margins.

The contribution is a different unit of enterprise computing: an AI-designed business specification executed by controlled infrastructure. The AISDL Specification defines the System. AISDL AIScript expresses its business logic. AISDL Runtimes make that logic operational without making a separately engineered application the enterprise's recurring obligation.

AI owns the design and can improve it through repeated criticism, tests and retained revisions. The Runtime owns technical execution. People own business intent and consequential authorization. These responsibilities reinforce one another: confidence in the Designer does not require unrestricted host access, and strong execution controls do not require prescribing the Designer's business solution.

The standard must remain open to new utilities and Services while preserving exact interfaces and compatible semantics. The enterprise owns its business design; providers compete on secure, predictable execution, performance and credible portability. A proprietary interface disguised as a common standard would defeat that promise.

The supporting mathematics establishes conditional results for retained improvement, target attainment, confined effects and compatible execution. It also identifies when shared assurance and faster business change create an economic advantage. Demonstrating the premises through repeatable comparative evidence is the next step, not a reason to weaken the architectural ambition.

The winners will combine design intelligence with dependable standardized execution. Enterprises will compete through the quality and evolution of their business specifications. Infrastructure providers will compete to run them. Application vendors can participate in that transition, but rebuilding a separate application for every business change is no longer the necessary model. **The birth of AISDL is the proposal; the end of application dependence is its consequence.**

Scope of the evidence

This is an architectural and economic thesis with a falsifiable research program. Edition 4.2 preserves editions 3.1, 4.0 and 4.1 and makes the controlled-authoring boundary explicit as the enabling condition of the thesis. Its mathematical results state consequences of explicit assumptions; its architecture describes the capabilities and controls needed to test them. It does not report independently audited production benchmarks, a completed runtime conformance assessment or measured industry replacement. The numerical examples are illustrations, not observations.

The decisive evidence will distinguish three levels: an executable implementation of the mechanism; independently accepted, complete Systems and subsequent changes; and replicated operational and economic advantage across a defined portfolio. A demonstration at one level does not establish the next. Open specifications, reproducible tests, complete cost accounting and published failures allow the proposal to be challenged without depending on the author's confidence or a supplier's claims. The ambition remains substantial: to make independently governed business evolution possible without an obligatory application-delivery organization.

Publication note

Author and license. Mohammed Loukili El Fahssi contributes this paper to the community under the Creative Commons Attribution 4.0 International license (https://creativecommons.org/licenses/by/4.0/). The license covers the paper, its original figures and manuscript text. It does not license software implementations, the AISDL specification as a separate work, or third-party material. A reference to an implementation does not constitute independent certification or endorsement.

AI assistance. AI tools assisted with drafting, editing, mathematical formulation, chart preparation and consistency checks. Responsibility for the submitted argument and its claims remains with the author. The paper reports conditional mathematical results and illustrative models, not independently measured production superiority. It has not been presented here as peer-reviewed research.

References

[1] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” *AFIPS Spring Joint Computer Conference*, 1967. DOI: 10.1145/1465482.1465560.

[2] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, NIST SP 800-145, 2011. DOI: 10.6028/NIST.SP.800-145.

[3] M. Souppaya, K. Scarfone, and D. Dodson, *Secure Software Development Framework (SSDF) Version 1.1*, NIST SP 800-218, 2022. DOI: 10.6028/NIST.SP.800-218.

[4] Object Management Group, *Business Process Model and Notation (BPMN), Version 2.0.2*, 2014. https://www.omg.org/spec/BPMN/2.0.2/.

[5] Object Management Group, *Model Driven Architecture*. https://www.omg.org/mda/. Accessed November 17, 2026.

[6] Mendix, *Mendix Runtime*, product documentation. https://docs.mendix.com/refguide/runtime/. Accessed November 17, 2026.

[7] Temporal Technologies, *What is Temporal?*, product documentation. https://docs.temporal.io/temporal. Accessed November 17, 2026.

[8] The Rust Project, *The Rust Programming Language: What Is Ownership?* https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accessed November 18, 2026.