Behavioural Capability Standard — Version 1.1
An Open Standard for Installable AI Capabilities

Steward Organisation: BigBlueYonder (Initial Founding Body)
Primary Author: Alan Roche — Capability Architecture & AI Interoperability Researcher
Contact / Website: bigblueyonder.com

Table of Contents

Contents

0.0 Preamble & Framing

Artificial intelligence systems are rapidly transitioning from monolithic, general-purpose models to modular, extensible ecosystems where specialised behaviours can be installed, combined, orchestrated, and governed. As AI agents become more capable and more integrated into daily workflows, the need for predictable, verifiable, safe, and interoperable component behaviours becomes increasingly critical.

This specification was authored by **Alan Roche** and is currently stewarded by **BigBlueYonder (BBY)** during its initial development phase. The intention is that future iterations of the standard may evolve through broader community and ecosystem participation.

The BBY Capability Specification (BCS) defines a unified, platform-agnostic standard for describing such behaviours. A capability in this context is a deterministic, self-contained functional unit that an AI agent may invoke to perform a specific task. Capabilities are not autonomous agents; they do not maintain ongoing goals, take independent action, or manage multi-step sequences. Instead, they encapsulate narrow, clearly stated behaviours such as summarisation, extraction, analysis, classification, formatting, calculations, or domain-specific functional transformations.

BCS exists to address a growing fragmentation problem. As different AI platforms conceptualise behaviours—whether as "skills," "tools," "functions," "abilities," or "plugins"—developers face increasing inconsistency in how these behaviours are written, documented, validated, and discovered. Without a shared standard, cross-platform behaviours become harder to audit and reuse, and safety properties become unpredictable across ecosystems.

BCS provides a common language for:

- *describing what a capability is*
- *defining what a capability does*
- *declaring how a capability behaves under specified inputs*
- *articulating constraints, assumptions, and safety requirements*
- *structuring inputs and outputs in a machine-validatable way*
- *ensuring deterministic, testable behaviour*
- *enabling interoperability across agent frameworks and vendor ecosystems*

The goal of BCS is not to impose a vendor-specific implementation model, nor to prescribe how AI agents should internally reason or plan. Instead, BCS provides a transparent, uniform specification for the behaviours that agents may call. This allows different platforms—commercial or open-source—to safely incorporate capabilities without ambiguity or undocumented assumptions.

The emergence of capability stores, marketplaces, and agent ecosystems further amplifies the need for formal documentation, validation, and safety guarantees. BCS ensures that both capability developers and platform operators can rely on consistent, predictable semantics. A capability described using BCS v1.1 must behave the same way when executed by any compliant agent, regardless of vendor, runtime, or operating environment.

BCS is designed with three fundamental objectives:

1. *Predictability*
   A capability must behave deterministically under defined conditions. Agents, developers, and end-users must be able to rely on consistent outcomes.
2. *Safety*
   Capabilities must explicitly declare risks, constraints, prohibited outputs, and fallback behaviours. Safety must be inspectable, enforceable, and testable.
3. *Interoperability*
   BCS enables capabilities to function across diverse ecosystems through shared structure, metadata, and validation rules.

This specification is intended for:

- *capability developers*
- *AI platform vendors*
- *agent framework designers*
- *researchers exploring modular AI behaviour models*
- *organisations requiring auditability and compliance*
- *standardisation groups*
- *regulatory stakeholders evaluating AI component behaviour*

BCS is designed to evolve. This document expresses the normative requirements for BCS v1.1. Future versions may extend fields, introduce stricter safety models, add taxonomy definitions, or refine validation pipelines. Backwards compatibility is maintained unless otherwise stated.

Above all, the BBY Capability Specification is an effort to bring consistency, clarity, safety, and predictability to the emerging world of installable AI behaviours. By defining a shared standard, BCS aims to become the common foundation upon which capability ecosystems, marketplaces, and agent frameworks can reliably grow.

0.1 Purpose of This Specification

This specification defines a portable, deterministic description format for capabilities. A compliant implementation MUST be able to parse, validate, and execute a capability description in a manner that produces the same behaviour across conforming platforms. This document specifies the structure, constraints, and semantic rules required to achieve that outcome.

The BBY Capability Specification (BCS) exists to provide a unified, rigorously defined standard for describing modular AI behaviours in a way that is transparent, safe, interoperable, and vendor-neutral. As AI systems mature, the industry is shifting away from monolithic, opaque reasoning toward architectures where specialised functional units—capabilities—augment or extend what an AI agent can do.

BCS defines the structural, behavioural, and safety-related requirements for these modular behaviours. The specification aims to ensure that:

- *capabilities behave predictably across platforms,*
- *their safety properties are explicit and inspectable,*
- *their input/output structures are machine-validatable, and*
- *they can be reused, shared, and deployed consistently in diverse environments.*

The overarching purpose is to provide a common language and shared expectation model that eliminates ambiguity and fragmentation in capability design.

BCS is not a code execution standard, runtime model, or programming API. It is a behavioural specification standard—a contract between the capability developer and any agent or vendor that consumes their work. The relative precedence between capability components (metadata, behaviour, schema, constraints, safety) is formally defined in Section 2.13. This hierarchy informs how conflicts are resolved throughout the specification.

Note: Formal definitions of capability behaviour, determinism, safety mechanisms, and execution guarantees are provided in **Section 1 (Conceptual and Behavioural Foundations)**. Section 0 introduces the specification; Section 1 provides the normative conceptual framework used throughout this document.

0.1.1 BCS in Relation to Existing Vendor Frameworks (Informative)

BCS is designed to operate **alongside** existing agent and tool-calling frameworks, not to replace them. It standardises how capabilities describe their **behaviour**, **safety boundaries**, **schemas**, and **deterministic execution expectations**, while leaving execution pipelines, agent reasoning, optimisation strategies, and platform APIs under the control of each vendor.

In practice, BCS answers the question:
"What does this capability do, and what are its declared constraints and boundaries?"

Vendor platforms continue to answer:
"How is this capability invoked, orchestrated, and executed within our environment?"

BCS therefore supports **cross-platform interoperability** and **consistent safety semantics**, while respecting vendor innovation and implementation independence.

0.2 Why This Specification Is Needed

Modern AI models can perform thousands of tasks, yet they often rely on ad hoc descriptions of tool usage, inconsistent documentation, loosely defined behaviours, or incomplete safety constraints. This leads to:

- *inconsistent outcomes across platforms,*
- *unpredictable behaviour under edge cases,*
- *duplication of effort by developers,*
- *difficulty auditing outputs,*
- *safety oversights and ambiguous risk boundaries,*
- *incompatibility between agent frameworks,*
- *fragmented ecosystems.*

As capability stores and behavioural marketplaces emerge, these inconsistencies become magnified. Without a standard, developers produce behaviours that are:

- *undocumented,*
- *incompatible,*

- *unsafe,*
- *non-deterministic,*
- *difficult for agents to evaluate or reason about.*

BCS resolves this by defining a universal structure that any platform can parse and validate before a capability is executed.

0.2.1 Practical Benefits for Implementers

Adopting this specification provides concrete, operational benefits for developers and platform maintainers:

- Consistent behaviour across runtimes — the same capability description produces the same observable behaviour on conforming platforms.
- Predictable validation and execution semantics — behaviour is defined by explicit rules instead of implicit or model-specific interpretation.
- Reduced ambiguity and runtime errors — capability definitions are structurally and semantically constrained, making failure cases easier to detect before execution.
- Improved reproducibility and debugging — behaviour can be inspected, tested, and reasoned about deterministically.
- Clear conformance guarantees — implementations can be evaluated against a shared behavioural standard instead of informal expectations.

0.3 Motivation for a Behaviour Standard

In summary, the motivation for this specification is to replace implicit, model-specific behaviour definitions with explicit, deterministic, and inspectable ones. By defining behaviour in a constrained, declarative form, capability execution can be validated, tested, reasoned about, and reproduced across conforming platforms. This allows capabilities to behave as reliable software components rather than emergent artefacts of a particular model or runtime environment.

0.4 The shift toward modular AI ecosystems

AI is evolving into a modular architecture where small, well-defined behaviours augment a central reasoning engine. This mirrors the evolution of computing itself:

- *operating systems → applications*
- *browsers → extensions*
- *cloud platforms → microservices*
- *smartphones → apps*
- *AI agents → capabilities*

Capabilities become the atomic units of functionality in an AI-driven environment.

A standard such as BCS ensures that these behavioural units can be developed by anyone and executed by everyone.

0.4.1 Eliminating ambiguity in behaviour descriptions

Today, capabilities are frequently described in free-text or informal JSON fields. This causes:

- *unpredictable interpretation by agents,*
- *misunderstanding of safety requirements,*
- *misalignment between developer intent and agent behaviour.*

BCS eliminates ambiguity by defining strict behavioural fields, constraints, and expected determinism.

0.4.2 Building trust and safety into ecosystems

AI safety must not depend on undocumented assumptions. For ecosystems to scale, safety properties need to be expressible, enforceable, and testable.

BCS introduces:

- *explicit safety blocks*
- *prohibited content declarations*
- *fallback behaviours*
- *risk classifications*
- *mitigation strategies*

This allows vendors to enforce safety at the platform level rather than depend on inference or heuristics.

0.4.3 Supporting cross-platform portability

A capability authored once should be executable across:

- *OpenAI agents*
- *Google agent frameworks*
- *Apple ecosystem agents*
- *Microsoft and enterprise platforms*
- *Open-source agent runtimes*
- *Proprietary or domain-specific systems*

Portability requires consistent structure and semantics, which BCS provides.

0.4.4 Enabling future regulation and auditability

As regulatory frameworks evolve, organisations must demonstrate:

- *predictable behaviour*
- *defined safety boundaries*
- *repeatable validation*
- *transparent behaviour descriptions*

- *clear responsibility attribution*

BCS makes capabilities auditable in a way that satisfies both internal governance and external regulators.

0.4.5 Preparing for capability marketplaces

Within the next 24–36 months, major vendors are expected to introduce:

- *capability stores*
- *behavioural marketplaces*
- *curated libraries*
- *enterprise repositories*

Without a shared standard, each platform becomes a silo.

BCS enables:

- *multi-platform publishing*
- *uniform developer onboarding*
- *shared validation tools*
- *cross-platform discoverability*

This dramatically reduces friction and lowers the barrier to entry for developers and vendors alike.

0.4.6 A foundation for future automation ecosystems

In mature agent ecosystems, capabilities will be:

- *installed,*
- *invoked,*
- *orchestrated,*
- *validated,*
- *updated, and*
- *deprecated*

…in the same way apps and packages are today.

BCS establishes the behavioural contract required for such automation to function safely and predictably.

0.5 Design Principles

The BBY Capability Specification (BCS) is founded on a set of core design principles intended to ensure that all capabilities—regardless of origin, platform, or domain—exhibit predictable, interpretable, safe, and interoperable behaviour. These principles serve both as a philosophical foundation and as binding constraints that shape every normative requirement in this specification.

The following principles are non-negotiable. Any implementation that does not align with these concepts is considered non-conformant, even if individual fields appear valid.

0.5.1 Determinism Over Emergence

A capability must behave deterministically. Given the same input, environmental context, and configuration, an agent should be able to rely on a predictable and stable form of output.

BCS rejects emergent or open-ended behaviours within capabilities.
Capabilities are not reasoning engines, improvisational systems, or autonomous policies. They are functional units that exhibit defined and testable outputs.

A capability MUST NOT:

- *maintain internal hidden state,*
- *adjust behaviour based on undocumented cues,*
- *generate outputs inconsistent with its behavioural contract,*
- *rely on ambiguous interpretation by the agent.*

This principle ensures interoperability and validates that capabilities can be audited, certified, and reused safely.

0.5.2 Clarity and Inspectability

Capabilities must be transparent in intent and execution.
An agent—or a human reviewer—must be able to inspect:

- *what the capability does,*
- *how it accomplishes its task,*
- *what assumptions it makes,*
- *what risks it may introduce,*
- *what constraints govern its behaviour.*

BCS requires every capability to include:

- *a clear behaviour definition,*
- *explicit descriptions of constraints and assumptions,*
- *well-documented input and output schemas,*
- *precise safety declarations.*

This avoids reliance on inference, guesswork, or undocumented developer intent.

0.5.3 Minimalism and Narrowness of Function

A capability should do one thing well.

BCS explicitly discourages overly broad or multi-purpose behaviours.
The narrower a capability's scope, the more predictable its behaviour and the easier it is for agents to orchestrate multiple capabilities safely.

Capabilities SHOULD:

- *focus on a single transformation or analysis,*
- *contain a limited and well-defined input surface,*
- *avoid behaviours that logically belong in separate capabilities,*
- *not attempt to be comprehensive or domain-unbounded.*

A capability that attempts to handle too many unrelated tasks cannot be validated or reasoned about reliably.

0.5.4 Explicit Safety Boundaries

Safety must be declared upfront, not inferred.
Every capability must include an explicit safety block describing:

- *risk level,*
- *prohibited outputs,*
- *mitigation strategies,*
- *fallback behaviours,*
- *domain restrictions.*

This ensures downstream platforms can enforce safety at runtime and allows capability stores or agents to filter based on risk tolerance.

Safety cannot be left to agent interpretation.
It must be built into the capability specification itself.

0.5.5 Vendor and Platform Neutrality

Capabilities written in BCS must remain fully portable across agents and platforms.

To achieve this:

- *the specification avoids runtime-specific language,*
- *avoids vendor-specific APIs,*
- *avoids platform-dependent assumptions,*
- *does not prescribe internal agent reasoning models,*
- *describes behaviour without reference to any specific LLM architecture.*

Every capability must behave identically whether executed by:

- *OpenAI agents,*
- *Google agent frameworks,*
- *Apple ecosystem agents,*
- *Microsoft and Azure systems,*
- *open-source agent runtimes,*

- *proprietary enterprise orchestration engines.*

This principle allows developers to write once and deploy everywhere.

0.5.6 Safety Over Capability

Where there is conflict between functionality and safety, safety takes priority.

This is a normative principle that governs the entire specification.
If a capability's intended behaviour introduces risk that cannot be mitigated through constraints or fallback behaviours, the capability MUST NOT be published.

BCS explicitly forbids:

- *"unsafe by default" designs,*
- *capabilities that rely on the agent to impose safety,*
- *deceptive, misleading, or ambiguous behaviours,*
- *omission of safety disclosures.*

Capabilities must be safe even when invoked by naïve or unsophisticated agents.

0.5.7 Interoperability as a First-Class Constraint

Capabilities are intended to be components in a larger behavioural ecosystem. They must therefore:

- *expose structured interfaces,*
- *avoid unnecessary complexity,*
- *use consistent naming conventions,*
- *follow predictable schemas,*
- *declare versioning clearly,*
- *remain inspectable by automated tools.*

This makes BCS capabilities "plug-compatible" between runtimes and agent systems.

Interoperability is not a by-product—it is a designed property.

0.5.8 Evolution With Backward Compatibility

BCS is designed to evolve.
However, capabilities authored today must remain valid under future versions of the standard unless:

- *a breaking change is explicitly introduced,*
- *a deprecation period has been honoured,*
- *the capability relies on a field that is formally removed.*

Backward compatibility ensures:

- *capability stores can continue to host older capabilities,*
- *agents can execute older specifications safely,*

- *validation pipelines remain predictable.*

0.5.9 Human Comprehensibility

Capabilities must be understandable by humans, not just machines.

This prevents over-engineering and ensures that domain experts, auditors, and regulators can inspect capabilities without specialised tooling.

A capability should read as:

- *a clear definition of task,*
- *a predictable behaviour,*
- *a well-bounded risk profile,*
- *a structured and documented interface.*

A human reviewer must be able to determine within minutes whether a capability is safe, valid, and appropriate for use.

Section 0 provides context, motivation, conventions, and high-level conformance framing. It does not define behavioural rules, structural formats, safety taxonomy, or validation processes, which are normatively specified in Sections 1–4.

0.6 Scope of This Specification

The BBY Capability Specification (BCS) defines a behavioural contract for modular AI capabilities. This contract enables agents, platforms, and validation tools to understand, inspect, and execute capabilities in a predictable, safe, and interoperable manner.

BCS specifies:

0.6.1 Structural Requirements

BCS defines the required structure and fields that make up a capability file, including:

- *metadata*
- *behaviour definition*
- *input and output schemas*
- *safety declarations*
- *examples*
- *versioning*
- *publisher information*

This ensures capabilities can be validated using machine-readable rules.

0.6.2 Behavioural Semantics

BCS describes how capability behaviours must be expressed, including:

- *deterministic behaviour*
- *explicit constraints*
- *assumptions*
- *transparent purpose*
- *vendor-neutral task framing*

The behavioural model ensures capabilities do not produce unpredictable, emergent, or unsafe outputs.

0.6.3 Safety Model

BCS specifies:

- *risk classification*
- *prohibited content*
- *mitigation strategies*
- *fallback behaviours*
- *domain restrictions*

This enables platforms to enforce safety without interpretive heuristics.

0.6.4 Input and Output Schema Requirements

BCS defines rules for:

- *data types*
- *fields*
- *constraints*
- *names and descriptions*
- *error structures*

The schema ensures consistent interpretation between platforms.

0.6.5 Validation Requirements

BCS defines a multi-layer validation pipeline:

- *structural validation*
- *type validation*
- *behaviour validation*
- *safety validation*
- *output validation*

This ensures capabilities can be automatically checked before execution or publication.

0.6.6 Interoperability Across Platforms

The specification defines rules that make capabilities:

- *portable*
- *reusable*
- *shareable*
- *platform-agnostic*

BCS explicitly supports multi-vendor ecosystems and capability marketplaces.

0.6.7 Conformance Requirements

BCS defines what makes a capability:

- *fully conformant*
- *conditionally conformant*
- *non-conformant*

Conformance levels allow platforms to enforce quality and safety thresholds.

0.6.8 Documentation and Examples

BCS requires capabilities to include:

- *clear examples*
- *behaviour demonstrations*
- *error case examples*
- *changelog entries*

This ensures transparency and practicality.

0.7 Non-Scope of This Specification

Equally important is what BCS does not define.
BCS explicitly avoids prescribing runtime, reasoning, internal model behaviour, or operational semantics for agents or LLMs.

Clarification: Registries and APIs in the Context of BCS

BCS does not define commercial marketplace models, economic relationships, ranking or discovery mechanisms, payment flows, or business governance structures. These remain entirely outside the scope of this specification.

Where this specification defines requirements for **registries** and **registry APIs** (see Section 6), these requirements apply **only to the technical integrity, lifecycle management, validation behaviour, and safety alignment of published capabilities**. They exist solely to ensure that capabilities remain consistent, auditable, and interoperable across hosting environments.

Registries and platforms MAY implement additional operational, commercial, or experience-level functionality, provided that such behaviour does not weaken or bypass the normative integrity and safety requirements defined by BCS.

BCS is not:

0.7.1 Not an Execution Standard

BCS does not specify how a capability is executed at runtime.
This includes:

- *scheduling*
- *resource management*
- *parallelisation*
- *caching*
- *error handling at the agent level*

These are responsibilities of the agent or platform.

0.7.2 Not a Programming Language or API

BCS does not define:

- *a programming language*
- *a function calling API*
- *runtime bindings*
- *model weights*
- *code-level behaviour*

Capabilities describe behaviour, not implementation.

0.7.3 Not an Agent Specification

BCS does not define:

- *how agents reason*
- *how agents plan tasks*
- *how agents choose which capability to invoke*
- *how agents maintain memory*
- *multi-step agent execution logic*

BCS describes the components that agents use, not the agents themselves.

0.7.4 Not a Safety Policy for Agents

BCS defines the safety model for capabilities, not:

- *agent-level safety*

- *system-wide safety policies*
- *content moderation rules*
- *cross-capability safety interactions*

Platforms may impose additional safety layers as needed.

0.7.5 Not a Marketplace or Distribution Platform Standard

BCS does not define:

- *commercial models*
- *store ranking algorithms*
- *submission processes*
- *payment systems*
- *intellectual property structures*

Although BCS is designed to support marketplaces, it does not prescribe how they operate.

0.7.6 Not a Regulatory Framework

BCS is compatible with audit and regulatory processes but does not constitute:

- *legal compliance requirements*
- *regulatory certification*
- *risk management frameworks*

Rather, it provides a structure that regulators may reference.

0.7.7 Not Model-Dependent

BCS does not depend on:

- *specific LLM vendors*
- *transformer architectures*
- *inference methods*
- *fine-tuning pipelines*

This guarantees longevity regardless of model evolution.

0.8 Intent of the Scope Definition

Defining scope and non-scope prevents:

- *overreach of the standard,*
- *misinterpretation of responsibilities,*
- *vendor lock-in,*
- *future incompatibilities,*
- *assumptions about execution models.*

BCS focuses narrowly and intentionally on describing capability behaviour, not on defining the architecture around it.

This narrow scope is what allows the specification to be:

- *stable,*
- *interoperable,*
- *evolvable,*
- *technology-agnostic,*
- *implementable across platforms,*
- *acceptable to vendors,*
- *suitable for publication as a formal standard.*

0.9 Capabilities vs Agents

The BBY Capability Specification (BCS) distinguishes clearly between capabilities and agents, as they serve fundamentally different roles within an AI ecosystem. Confusion between these concepts leads to unsafe behaviour, unpredictable execution, and ambiguous responsibility boundaries. This section provides a formal definition of each entity and describes the intended relationship between them.

0.9.1 Capabilities: Narrow, Deterministic Behaviour Units

A capability is a modular, installable, and deterministic unit of behaviour.
It performs precisely one well-defined task under controlled conditions. Examples include:

- *summarising content,*
- *extracting structured information,*
- *classifying text,*
- *analysing sentiment,*
- *formatting or transforming data,*
- *performing mathematical operations,*
- *generating domain-specific structured outputs.*

Capabilities MUST:

- *be stateless,*
- *be deterministic,*
- *define inputs and outputs as schemas,*
- *describe behaviour through transparent steps,*
- *declare safety restrictions,*
- *avoid autonomous actions,*
- *avoid multi-step planning,*
- *avoid external side effects.*

A capability is conceptually similar to:

- *a function without code,*
- *a declarative behaviour block,*

- *a safe, reusable behavioural "module".*

Capabilities do not reason.
They do not choose what to do, when to do it, or how to interact with the world beyond producing their output.

Capabilities are invoked by agents, not by themselves.

0.9.2 Agents: Reasoning and Orchestration Systems

An agent is a system—typically powered by an LLM or other reasoning model—that interprets tasks, plans actions, decides which capabilities to invoke, and integrates results into broader workflows.

Agents can:

- *decompose complex tasks,*
- *choose which capability to call,*
- *maintain state,*
- *pursue goals,*
- *evaluate context,*
- *apply reasoning,*
- *perform iterative planning,*
- *interpret instructions from users.*

Agents may operate autonomously within the constraints imposed by developers, platforms, or safety systems.

Agents are not defined by BCS.
BCS only defines the behavioural contract for capabilities that agents may use.

0.9.3 Why Capabilities Must Remain Distinct from Agents

Allowing capabilities to behave like agents introduces risk:

- *Unpredictability: Behaviour becomes emergent rather than deterministic.*
- *Safety gaps: Safety cannot be enforced if the capability makes decisions outside its defined scope.*
- *Reproducibility issues: Validation and auditing become impossible.*
- *Platform incompatibility: Different agents interpret free-form behaviours differently.*

BCS therefore explicitly forbids:

- *hidden planning,*
- *autonomous decisions,*
- *multi-step reasoning,*
- *stateful interactions,*
- *behaviour depending on agent identity,*
- *repeatedly calling other capabilities,*
- *generating outputs beyond the defined schema.*

Capabilities must be passive: they wait for an invocation and respond deterministically.

0.9.4 Execution Relationship Between Agents and Capabilities

The relationship can be defined as:

1. *Agent interprets task*
   → The agent decides that a capability is required.
2. *Agent prepares inputs*
   → The agent structures inputs according to the capability's schema.
3. *Capability executes deterministically*
   → It performs its defined behaviour and produces outputs.
4. *Agent integrates results*
   → The agent interprets results as part of a larger workflow.
5. *Agent may invoke additional capabilities*
   → Or continue reasoning based on the result.

The capability does not know:

- *when it will be called,*
- *by whom,*
- *for what larger purpose,*
- *as part of what workflow,*
- *any context beyond its inputs.*

This ensures separation of concerns.

0.9.5 Accountability and Auditing

Because capabilities do not plan or reason, responsibility for:

- *task interpretation,*
- *error recovery,*
- *safety enforcement at runtime,*
- *multi-step execution,*
- *user interaction,*
- *goal-directed behaviour,*

…belongs to the agent or platform.

Capabilities are responsible only for:

- *their defined behaviour,*
- *their safety declarations,*
- *their structured outputs.*

This division makes auditing significantly easier.
A capability can be tested in isolation, without recreating agent reasoning.

0.9.6 Benefits of Strong Separation

A strict separation between agents and capabilities yields:

✓*Predictability*

Capabilities behave consistently across platforms.

✓*Safety*

Capabilities cannot override or bypass agent-level safety systems.

✓*Interoperability*

Agents from different vendors can invoke the same capability.

✓*Simplicity*

Capabilities remain easy to understand, validate, and trust.

✓*Extensibility*

Agent behaviour improves independently of capability design.

✓*Regulation-readiness*

Auditors can evaluate capabilities without reconstructing proprietary reasoning models.

0.9.7 Analogy for Intuition

A useful analogy is:

- *Agent = Operating system + application logic*
- *Capabilities = Functions or system calls*

Functions do not decide when to run.
They simply define what happens when they are run.

Capabilities exist to be:

- *predictable,*
- *inspectable,*
- *reusable,*
- *safe.*

Agents exist to be:

- *adaptive,*
- *contextual,*

- *goal-directed.*

BCS formalises this relationship.

0.10 Terminology and Definitions

The BBY Capability Specification (BCS) relies on precise terminology to avoid ambiguity and ensure consistent interpretation across platforms, validators, developers, auditors, and agent frameworks. The following definitions are normative unless explicitly stated otherwise. Where possible, terms are defined in alignment with conventions used by major AI vendors, standards bodies, and software engineering practices.

0.10.1 Capability

A capability is a modular, installable, deterministic behaviour unit that produces outputs according to a defined schema and behaviour description.
A capability:

- *does not reason,*
- *does not maintain state,*
- *does not perform autonomous actions,*
- *responds only to the inputs provided,*
- *adheres strictly to its behavioural contract.*

Capabilities are the atomic units of functionality in systems that use BCS.

0.10.2 Agent

An agent is a system—typically powered by a reasoning model—that:

- *interprets tasks,*
- *determines which capabilities to invoke,*
- *orchestrates multi-step processes,*
- *integrates results,*
- *maintains context or memory,*
- *performs autonomous or semi-autonomous actions.*

BCS does NOT specify agent behaviour or design.

0.10.3 Behaviour Block

The behaviour block is the core part of a capability describing:

- *its purpose,*
- *the steps it performs,*
- *assumptions,*
- *constraints,*
- *deterministic aspects of execution.*

It defines what the capability does without specifying how agents should implement or interpret it.

0.10.4 Determinism

Determinism means that:

For identical inputs and conditions, a capability MUST produce a predictable, structurally consistent output.

Determinism does not imply byte-for-byte identical text output, but the behaviour, schema, content patterns, and safety conditions MUST remain stable.

0.10.5 Safety Block

The safety block contains the capability's declared:

- *risk level,*
- *prohibited content,*
- *mitigation strategies,*
- *fallback behaviour,*
- *domain limitations.*

It enables platforms to enforce safety consistently.

0.10.6 Schema

A schema is a structured definition of the inputs and outputs for a capability.
Schemas include:

- *field names,*
- *data types,*
- *constraints,*
- *descriptions,*
- *required/optional indicators.*

Schemas must be machine-validatable.

0.10.7 Validation

Validation refers to automated processes that confirm:

- *the capability file is structurally correct,*
- *the schema is valid,*
- *the behaviour is well-defined,*
- *the safety declarations are complete,*
- *the versioning is correct,*
- *the capability is non-ambiguous and non-harmful.*

Validator implementations vary, but the validation pipeline defined in BCS is mandatory.

0.10.8 Publisher

A publisher is the individual or organisation responsible for:

- *authoring the capability,*
- *ensuring its correctness and safety,*
- *releasing version updates,*
- *maintaining documentation.*

Publishers may be independent developers or corporate entities.

0.10.9 Invocation

Invocation is the act of an agent requesting a capability to run with a given set of inputs.

Invocations:

- *MUST be explicit,*
- *MUST be single-shot (no state retention),*
- *MUST conform to the defined schema,*
- *MUST be executed without side effects.*

0.10.10 Fallback Behaviour

Fallback behaviour is the safe, deterministic alternative output that a capability returns if:

- *the inputs violate constraints,*
- *prohibited content would be produced,*
- *the behaviour cannot complete safely.*

Fallbacks must themselves satisfy all safety requirements.

0.10.11 Constraint

A constraint is a condition that limits capability behaviour. Constraints include:

- *inputs that MUST or MUST NOT appear,*
- *boundaries for acceptable behaviour,*
- *domain restrictions,*
- *safety obligations.*

Constraints support deterministic and safe execution.

0.10.12 Error Object

The error object is a structured output that describes:

- *what went wrong,*
- *why it went wrong,*
- *how the agent should interpret the error.*

Error objects MUST follow the BCS schema.

0.10.13 Semantic Versioning (SemVer)

Capabilities MUST use semantic versioning:

- *MAJOR = breaking changes*
- *MINOR = new backwards-compatible features*
- *PATCH = non-behavioural fixes*

Platforms depend on versioning for stability and safe updates.

0.10.14 Non-Normative Content

BCS distinguishes between:

- *Normative content — required for conformance*
- *Non-normative content — explanatory, optional, or informative*

Normative statements include:

- *MUST,*
- *MUST NOT,*
- *SHOULD,*
- *SHOULD NOT,*
- *MAY.*

0.10.15 Capability Marketplace

A capability marketplace is any platform where capabilities can be:

- *published,*
- *discovered,*
- *rated,*
- *updated,*
- *installed,*
- *deprecated.*

BCS does not define marketplace rules, but marketplace operators rely on BCS for capability validation.

0.10.16 Auditor

An auditor is a role—human or automated—that reviews capabilities for:

- *safety,*
- *correctness,*
- *compliance,*
- *risk classification,*
- *conformance to BCS.*

Auditors may be internal (platform teams) or external (regulators).

0.10.17 Execution Environment

The execution environment is the runtime in which an agent executes a capability.
BCS intentionally does not define:

- *model architecture,*
- *compute layer,*
- *memory systems,*
- *planning algorithms.*

Execution environments vary across vendors.

0.11 Requirements Notation

This specification uses a controlled vocabulary to indicate the normative strength of requirements. These terms define whether a behaviour, structure, or field is mandatory, recommended, optional, or prohibited. The terminology follows established conventions from IETF RFC 2119 and ISO/IEC standards, adapted for behavioural AI capabilities.

Unless explicitly noted otherwise, the following terms are normative and must be interpreted exactly as defined in this section.

0.11.1 Normative Keywords

The following keywords indicate requirement strength:

MUST

Indicates an absolute requirement.

If a capability fails to satisfy a MUST requirement, it is considered non-conformant.

Example:

“A capability MUST provide a deterministic behaviour block.”

MUST NOT

Indicates an absolute prohibition.

Any violation renders the capability invalid and unsafe.

Example:

“A capability MUST NOT produce outputs outside its defined schema.”

SHOULD

Indicates a recommended best practice.
There may be valid reasons to deviate, but the implications must be understood and carefully considered.

Example:

“A capability SHOULD restrict its purpose to a single transformation or analysis.”

SHOULD NOT

Indicates a discouraged practice.
Deviation is possible but must be justified.

Example:

“A capability SHOULD NOT rely on extensive free-form natural language descriptions to explain behavioural constraints.”

MAY

Indicates an optional feature or behaviour.
Platforms and developers can safely choose whether to implement it.

Example:

“A capability MAY include additional non-normative examples to clarify behaviour.”

0.11.2 Interpretation of Normative Keywords

When normative keywords apply to capability structure:

- *They define the presence or absence of fields,*
- *Dictate acceptable values,*
- *Specify required relationships between fields,*
- *Govern behavioural boundaries.*

When applied to runtime behaviour:

- *They describe what the capability guarantees under valid inputs,*
- *Ensure that agents may rely on predictable properties,*
- *Define safety obligations that must not be violated.*

Platforms, validators, and auditors must interpret these keywords strictly.

0.11.3 Non-Normative Statements

Non-normative text includes:

- *explanatory paragraphs,*
- *rationale,*
- *examples,*
- *diagrams,*
- *commentary,*
- *implementation notes.*

Such content exists to clarify intent but does not impose requirements.

Non-normative statements may use softer language such as:

- *“for example,”*
- *“typically,”*
- *“in most cases,”*
- *“it may be useful to…”*

These must not be interpreted as defining conformance.

0.11.4 Normative vs Informative Sections

Each major section of the specification is either:

Normative

— defining structural, behavioural, schema, validation, or safety requirements.

Informative

— providing explanation, illustration, or justification.

Most capability fields, behaviours, and schema definitions are normative.
Appendices containing examples and diagrams are informative unless explicitly indicated otherwise.

0.11.5 Use of Capitalisation

Normative keywords MUST appear in uppercase to distinguish them from general descriptive language.

- *“a capability must…” → non-normative*
- *“a capability MUST…” → normative requirement*

This distinction ensures clarity for developers, platforms, and auditors.

0.11.6 Precedence of Requirements

If a conflict appears between:

- *normative statements,*
- *examples,*
- *diagrams,*
- *commentary, or*
- *appendices,*

…the normative statements take precedence.

If two normative statements conflict, the following order applies:

1. *Behaviour and safety requirements override metadata descriptions*
2. *Schema constraints override behavioural narrative*
3. *Prohibitions override permissions*
4. *Higher-level section requirements override lower-level field notes*
5. *Errata and version clarifications (if published) override all prior text*

This prevents ambiguity and ensures conformance is well-defined.

0.11.7 Requirements Enforcement

Validators, platforms, and marketplace operators MUST:

- *treat MUST and MUST NOT as mandatory rules,*
- *treat SHOULD and SHOULD NOT as strongly recommended guidance,*
- *treat MAY as optional,*
- *reject capabilities that violate mandatory requirements.*

Additionally:

- *Deviations from SHOULD require justification in documentation.*
- *Deviations from MUST render a capability invalid.*

This establishes a uniform compliance baseline across all ecosystems.

0.12 Conformance Requirements

This section defines what it means for a capability to conform to the BBY Capability Specification (BCS). Conformance ensures that capabilities behave predictably, can be validated automatically, and can be safely integrated into diverse agent ecosystems.

A capability is considered conformant only when it satisfies all relevant normative requirements in this specification, including structural, behavioural, safety, and schema-related obligations.

Conformance requirements apply to:

- *Capability developers*
- *Automated validation tools*

- *Platform operators*
- *Marketplace review systems*
- *Auditors and governance bodies*

BCS recognises three conformance states: Fully Conformant, Conditionally Conformant, and Non-Conformant.

Full conformance requires meeting all normative requirements defined in the structural specification (Section 2), behavioural and determinism requirements (Section 1), safety requirements (Section 3), and validation pipelines (Section 4). These sections collectively define the mandatory criteria for BCS v1.1

0.12.1 Minimal Conformance Profile ("BCS-Lite") (Informative)

This specification defines a full capability model with a rich behavioural, safety and lifecycle structure. To support incremental adoption, this subsection describes a **minimal conformance profile ("BCS-Lite")** that preserves the core guarantees of BCS while reducing initial implementation scope.

A capability conforming to the BCS-Lite profile MUST include at minimum:
– Core metadata fields (identifier, version, authoring information)
– A behaviour block describing a **single, narrowly-scoped deterministic purpose**
– An input schema defining expected fields and types
– An output schema defining returned fields and types
– A safety block declaring prohibited inputs/contexts and a deterministic fallback behaviour
– At least one worked example illustrating valid inputs and expected outputs

A BCS-Lite capability MAY omit optional or advanced constructs such as extended safety taxonomy mappings, complex constraint declarations, registry-specific metadata, and advanced lifecycle annotations.

Implementers adopting BCS-Lite are encouraged to evolve capabilities toward **full BCS conformance** over time. Any deviation from full conformance MUST be clearly documented and MUST NOT be represented as full BCS validation or full-model alignment.

0.12.2 Fully Conformant Capability

A capability is Fully Conformant if and only if:

0.12.3 All Mandatory Fields Are Present

The capability MUST include every required field:

- *metadata*
- *behaviour block*
- *safety block*
- *input schema*
- *output schema*
- *examples*
- *versioning*
- *publisher information*

Missing or malformed required fields results in non-conformance.

0.12.4 Behaviour Is Deterministic

The behaviour block MUST describe a deterministic transformation.
This includes:

- *predictable processing steps*
- *no emergent multi-step reasoning*
- *no hidden state*
- *no undocumented external dependencies*
- *no variations in behaviour that contradict the specification*

If a capability behaves differently under identical valid inputs, it is non-conformant.

0.12.5 Safety Requirements Are Explicit and Complete

The safety block MUST:

- *declare a risk level*
- *define prohibited content*
- *identify mitigation strategies*
- *define fallback behaviour*
- *specify domain limitations*

A capability without a complete safety block is non-conformant.

0.12.6 Input and Output Schemas Are Valid

Schemas MUST:

- *use supported data types*
- *include field names and descriptions*
- *mark required vs optional fields*
- *include constraints where applicable*
- *conform to BCS schema formatting rules*

Any schema ambiguity renders the capability non-conformant.

0.12.7 Versioning Is Correct

Capabilities MUST use semantic versioning:

MAJOR.MINOR.PATCH

Invalid or missing versioning results in non-conformance.

0.12.8 No Prohibited Behaviours Appear

A conformant capability MUST NOT:

- *reason autonomously*
- *maintain state*
- *call other capabilities*
- *depend on unspecified context*
- *produce outputs beyond its schema*
- *violate safety declarations*
- *perform multi-step planning*

Violations are grounds for immediate rejection.

0.12.9 Examples Are Provided

A conformant capability MUST include:

- *at least one positive example*
- *at least one error or fallback example*
- *explanations where needed*

This ensures interpretability and auditability.

0.12.10 The Capability Passes All Validation Layers

Validation includes:

1. *Structural validation*
2. *Type validation*
3. *Behaviour validation*
4. *Safety validation*
5. *Output schema validation*

Failure in any layer means the capability is non-conformant.

0.12.11 Conditionally Conformant Capability

A capability is Conditionally Conformant if:

- *All required fields are present*
- *Behaviour is deterministic*
- *Safety declarations are complete*
- *But one or more optional or SHOULD-level requirements are not met*

Examples include:

- *missing optional examples*
- *insufficiently detailed descriptions*
- *minor clarity issues*
- *incomplete non-normative documentation*

Conditionally conformant capabilities are valid but may be excluded from higher-trust environments or certain marketplaces.

BCS recommends that platforms:

- *accept them with warnings,*
- *flag deviations for developer review,*
- *provide guidance for full conformance.*

0.12.12 Non-Conformant Capability

A capability is Non-Conformant if it violates any MUST or MUST NOT requirement.

Examples include:

- *ambiguous or non-deterministic behaviour*
- *missing required fields*
- *invalid schemas*
- *unsafe behaviour*
- *prohibited content allowances*
- *incorrect versioning*
- *missing fallback behaviour*
- *structural invalidity*

Non-conformant capabilities MUST be rejected by:

- *validators*
- *platforms*
- *marketplaces*
- *auditors*

They may not be published or installed until corrected.

0.12.13 Conformance Levels for Platforms and Validators

Platforms and validation tools MUST:

- *reject non-conformant capabilities,*

- *accept fully conformant capabilities,*
- *optionally accept conditionally conformant capabilities,*
- *document their acceptance criteria,*
- *provide error reports for rejected submissions.*

Platforms MAY impose stricter safety or structural requirements beyond BCS.
However, they MUST NOT redefine or relax normative BCS rules.

0.12.14 Conformance for Developers

Capability publishers MUST:

- *ensure all normative requirements are met,*
- *update documentation with every version change,*
- *maintain backwards compatibility unless bumping MAJOR version,*
- *disclose any deviation from SHOULD-level requirements,*
- *correct non-conformance before publication.*

Non-conformance by developers creates risks in downstream ecosystems and may result in capability de-listing by platforms.

0.12.15 Conformance Change Over Time

As BCS evolves:

- *older capabilities remain valid unless explicitly deprecated,*
- *new fields MAY become required in future versions,*
- *conformance definitions MAY expand,*
- *deprecated fields MUST continue to function for one MAJOR version cycle.*

A capability that was conformant in v1.1 may become conditionally conformant in a future version but cannot become retroactively non-conformant unless explicitly stated.

0.12.16 Evidence of Conformance

A capability is considered conformant only when:

- *the capability file passes automated validation,*
- *the publisher provides examples,*
- *the safety block is present and complete,*
- *an agent or marketplace can parse it without modification.*

Platforms MAY publish conformance badges or metadata flags indicating:

- *“BCS v1.1 Fully Conformant”*
- *“BCS v1.1 Conditional Conformance”*
- *“BCS v1.1 Audit Required”*

0.12.17 Conformance and Certification

BCS does not (yet) define a certification process, but it anticipates:

- *third-party auditors,*
- *enterprise governance tools,*
- *automated certification workflows,*
- *marketplace approval checks.*

Certification frameworks built on top of BCS MUST use the conformance model defined in this section.

0.13 Document Conventions

This specification follows a set of formatting, structural, and semantic conventions to ensure clarity, consistency, and ease of reference across all capability implementations and related documentation. These conventions are designed to support both human readability and machine-interpretation by validators, auditors, and platform ingestion systems.

0.13.1 Heading Structure

Sections use hierarchical numbering to define document structure:

- *Level 1 headings represent major sections (e.g., 1, 2, 3).*
- *Level 2 headings represent subsections (e.g., 1.1, 1.2).*
- *Level 3 headings may be used for further breakdown where necessary (e.g., 1.1.1).*

Heading levels correspond to semantic structure, not visual size alone.
Agents or validators MUST NOT rely on typography to infer hierarchy.

Human-readable versions MAY use graphical separators (e.g., lines, spacing) for clarity, but these have no normative effect.

0.13.2 Normative vs Informative Content

Sections are either:

- *Normative — defining requirements that MUST or MUST NOT be followed,*
- *Informative — providing explanation, rationale, or examples.*

Unless explicitly marked, all behavioural, structural, schema, and safety-related sections are normative by default.

Appendices, diagrams, and extended examples are informative, unless specifically labelled as normative in future versions.

0.13.3 Text Styles and Keyword Usage

The following conventions apply:

- *Normative keywords (MUST, MUST NOT, SHOULD, SHOULD NOT, MAY) appear in all caps.*
- *Field names appear in monospace formatting when cited (e.g., output_schema).*
- *Capability examples may use indentation or fenced blocks for clarity.*
- *Behaviour descriptions use plain prose unless specifying exact schema.*

Platforms parsing this document MUST ignore visual styling and rely instead on structural semantics.

0.13.4 Examples and Illustrations

Examples in this specification serve to:

- *illustrate valid behaviours,*
- *demonstrate correct schema structure,*
- *show proper error handling,*
- *clarify intent.*

Examples are informative unless explicitly stated otherwise.
A capability MUST NOT rely on examples to define required behaviour; all requirements must appear in normative sections.

Each capability SHOULD include:

- *at least one positive example,*
- *at least one error example,*
- *additional examples where complexity warrants.*

0.13.5 Diagrams

Diagrams are included to illustrate:

- *behaviour flow,*
- *safety boundaries,*
- *schema relationships,*
- *validation pipelines.*

Unless a diagram is explicitly marked normative, diagrams are informative and included only for conceptual clarity.

All diagrams in later sections will appear in appendices for modularity.

0.13.6 Schema Formatting Conventions

Schemas MUST follow these conventions:

- *Field names use snake_case.*
- *Data types use lowercase keywords (e.g., string, integer, boolean).*
- *Nested objects MUST be indented consistently or expressed using JSON-like structures.*
- *Descriptions SHOULD be short, explicit, and free from ambiguous phrasing.*

Some schema examples use JSON-like structures, but BCS does not require JSON as the sole format. Machine-validatable formats (e.g., JSON Schema, YAML, CBOR-like) MAY be used by platforms.

0.13.7 Behaviour Description Conventions

Behaviour descriptions SHOULD be:

- *written as declarative transformations,*
- *free from agent-oriented language ("you should", "the agent must", etc.),*
- *explicit in assumptions and constraints,*
- *unambiguous in sequencing when sequencing is required.*

Behaviour descriptions MUST NOT:

- *require hidden context,*
- *imply intermediate reasoning steps,*
- *reference agent internals,*
- *contain instructions that conflict with safety declarations.*

0.13.8 Safety Declaration Conventions

Safety blocks MUST:

- *list prohibited content categories explicitly,*
- *define fallback behaviour deterministically,*
- *avoid ambiguous or context-dependent restrictions,*
- *specify inputs that are out of scope or unsafe.*

Safety declarations MUST NOT:

- *rely on platform-enforced heuristics,*
- *defer responsibility to agents,*
- *use subjective language without clear definitions.*

0.13.9 Versioning Conventions

All normative changes to this specification are reflected in:

- *MAJOR version updates for breaking changes,*
- *MINOR updates for new fields or expansions,*
- *PATCH updates for clarifications or error corrections.*

Capabilities MUST reflect version conformance in their metadata using the bcs_version field.

0.13.10 Abbreviations and Acronyms

The following abbreviations may appear throughout the document:

- *BCS — BBY Capability Specification*
- *LLM — Large Language Model*
- *API — Application Programming Interface*
- *SemVer — Semantic Versioning*
- *JSON — JavaScript Object Notation*

All acronyms are defined at first use in each major section.

0.13.11 Document Lifecyle and Publication Notes

This specification anticipates a one-year revision cycle for MINOR versions and a multi-year cycle for MAJOR updates. Draft versions MAY be published for comment.

The document may also contain:

- *errata sections,*
- *change logs,*
- *reference tables,*
- *security considerations.*

These are informative unless indicated otherwise.

0.13.12 Normative vs Informative Content

Unless explicitly marked as “informative,” all requirements in Sections 1–4 are normative. Examples, diagrams, and commentary are informative and do not create additional normative obligations.

0.14 *Versioning Policy*

BCS uses semantic versioning.

- **Major versions (e.g., 1.x → 2.0)** MAY introduce breaking changes.
- **Minor versions (e.g., 1.0 → 1.1)** MUST NOT break conformance for existing capabilities; they MAY introduce optional fields or clarifications.
- **Patch versions (e.g., 1.1.0 → 1.1.1)** MUST only include editorial or non-normative fixes. Capability documents MUST declare the BCS version they conform to using the bcs_version field in the metadata block.

0.15 Recommended Reading Paths (Informative)

BCS is intended for multiple stakeholder groups, each with different responsibilities and information needs. This subsection provides guidance on which parts of the specification are most relevant to each role.

Capability Authors / Developers

– Section 1 — Capability Model

– Section 2 — Capability File Structure

– Appendix Section 7.1+ — Canonical and Worked Examples

Platform / Agent Implementers

– Section 1 — Capability Model

– Section 2 — Capability File Structure

– Section 3 — Safety Model and Taxonomy

– Section 4 — Validation Pipelines

Registry / Marketplace Operators

– Section 2 — Capability File Structure (metadata & integrity fields)

– Section 4 — Validation Pipelines

– Section 5 — Lifecycle and Versioning

– Section 6 — Registry and Marketplace Requirements

Governance, Risk, Compliance, and Audit Functions

– Section 3 — Safety Model and Taxonomy

– Section 4 — Validation Pipelines

– Section 5 — Lifecycle and Versioning

– Section 6 — Registry and Marketplace Requirements

Product Owners, Policy Makers, and Executive Stakeholders

– Section 0 — Preamble, Motivation and Scope

– Section 1 — Capability Model (overview subsections)

– Section 3 — Safety Model (conceptual overview)

– Section 6 — Registry and Ecosystem Overview

These recommended paths are advisory only. All sections remain normative or informative as individually stated within the specification.

1.0 Introduction to the Capability Model

The Capability Model defined in this section establishes the foundational conceptual framework for how capabilities are described, interpreted, validated, and executed within systems that implement the BBY Capability Specification (BCS).

Where Section 0 provided definitions, principles, and supporting context, **Section 1 defines what a capability *is* in formal, technical, and normative terms.** It serves as the backbone of the entire specification.

The Capability Model provides:

- a conceptual understanding of capabilities as deterministic behaviour units;
- the formal definition that all implementations MUST follow;
- the internal structural model that agents and platforms use to interpret behaviours;
- the determinism and safety guarantees required for cross-platform execution;
- a clear distinction between capability behaviour and agent reasoning;
- the behavioural and structural rules necessary for validation and auditability.

This section therefore defines the characteristics of a valid capability, the relationships between its internal components, and the normative constraints that govern its execution.

1.1 Capability Principles

The purpose of the Capability Model is to ensure that all BCS-compliant capabilities exhibit:

Predictability

They MUST behave consistently across vendors, models, and runtimes.

Transparency

Their behaviour MUST be fully inspectable and understandable.

Safety

Their risk boundaries MUST be explicitly declared and enforceable.

Interoperability

They MUST function uniformly across platforms adhering to BCS.

Determinism

They MUST produce outputs aligned with their behaviour definition and schema.

Auditability

They MUST be testable and verifiable using automated or manual review.

The Capability Model is the conceptual and structural foundation for this consistency.

1.2 Relationship to the File Structure (Section 2)

The Capability Model (Section 1) describes the **abstract requirements and behavioural rules**. Section 2 defines the **concrete file structure** used to express these requirements.

Think of it like:

- **Section 1** = the semantics, rules, and conceptual behaviour model
- **Section 2** = the formalised data structure (fields, schemas, blocks, metadata)

Together, these two sections ensure both:

- conceptual clarity, and
- machine-readable structure.

A capability CANNOT be valid under BCS unless it satisfies both Section 1 and Section 2.

1.3 Conformance to the Capability Model

All capabilities MUST satisfy:

- the determinism requirements in Section 1.4,
- the schema requirements in Sections 1.5 and 1.6,
- the constraints model in Section 1.7,
- the safety-interaction rules in Section 1.8,
- the error-handling rules in Section 1.9,
- the cross-platform guarantees in Section 1.10.

A capability that satisfies Section 2 (file structure) but violates Section 1 (behaviour model) is **non-conformant**.

Likewise, a capability that behaves consistently but lacks correct structure is **non-conformant**.

This ensures that the Capability Model is not merely descriptive but **normative and binding**.

1.4 High-Level Structure of Section 1

Section 1 consists of the following subsections:

1.6 Conceptual Overview

1.7 Formal Definition of a Capability

1.8 Behaviour Structure Model

1.9 Determinism Requirements

1.10 Input Schema Requirements

1.11 Output Schema Requirements

1.12 Constraints & Assumptions Model

1.13 Safety Interaction Model

1.14 Error Handling Model

1.15 Cross-Platform Behaviour Consistency

Each subsection defines functional and behavioural requirements that contribute to a coherent and interoperable capability.

1.5 Scope of This Section

Section 1 applies to:

- capability developers,
- platform integrators,
- agents that execute capabilities,
- marketplace validators,
- auditors evaluating safety and compliance.

This section does NOT define:

- file format fields (Section 2),
- safety taxonomies (Section 3),
- validation pipelines (Section 4),
- versioning or lifecycle rules (Section 5).

It focuses exclusively on **behavioural and semantic requirements**.

1.6 Conceptual Overview

This subsection provides a high-level conceptual framework that describes what a capability is, how it should behave, and how it differs from other components within an AI ecosystem. It establishes the philosophical and architectural underpinnings that guide all subsequent normative requirements in this specification.

A capability, as defined by BCS, is a **deterministic, self-contained behavioural unit** that performs a narrowly scoped function. Capabilities do not reason, decide, plan, or maintain state. They operate strictly within the confines of the inputs provided and the behavioural contract defined by the capability developer.

The conceptual model situates capabilities as **the atomic functional components** in modular AI ecosystems. They represent the lowest-level building blocks of behaviour that an agent may invoke during task execution.

1.6.1 Illustrative Example Capability (Informative)

This subsection provides a simplified, human-readable example of a BCS capability. It is not a full canonical file and does not include all required fields, but illustrates the core structural and behavioural concepts defined in Section 1.

Capability purpose

Extract the names of people mentioned in a short piece of text.

Behaviour summary

The capability receives a text input and returns a list of detected person names. If no names are found, it returns an empty list. The capability does not infer relationships, roles, or attributes beyond the names themselves.

Input schema (informal)

– `text` — string — the text to analyse

– `language` — optional string — language hint (e.g., `"en"`, `"fr"`)

Output schema (informal)

– `people` — array of objects, each containing:

– `name` — string — the detected person name

– `confidence` — number between 0 and 1

Determinism and constraints

– Results are ordered by first appearance in the input text.

– The capability does not guess or invent names.

– If uncertainty is high, names are omitted rather than fabricated.

Safety and fallback behaviour

– If the input contains prohibited categories (e.g., hate-speech contexts), the capability returns a safe failure response as defined in Section 3.

– If the input cannot be processed, it returns an empty result with an explanatory status field rather than generating alternative content.

For full canonical capability examples and complete schema structures, see Appendix Section 7.1 and the normative example in Section 2.12.

1.6.2 Capabilities as Modular Behaviour Units

Capabilities are designed to be modular, reusable, platform-agnostic units of behaviour. They resemble:

- pure functions in functional programming,
- microservices without networking,
- system calls without executable code,

- declarative behavioural modules.

A capability MUST implement a **single, well-defined purpose** and MUST NOT contain multiple unrelated behaviours. Narrowness of scope is essential for:

- predictability,
- safety,
- interoperability,
- auditability,
- version stability.

Capabilities MUST remain isolated from:

- agent memory,
- agent reasoning processes,
- external state or history,
- contextual information not explicitly provided in the inputs.

This ensures that capabilities are **pure behavioural definitions**, not adaptive or generative systems.

1.6.3 Capabilities as Deterministic Transformations

A core principle of the Capability Model is that capabilities are **deterministic**.

Determinism (as formally defined in Section 0.10.4) requires that, for the same valid inputs and environmental conditions, a capability produces a predictable and structurally consistent output. In practical terms, this means that capabilities operating under the BCS model MUST:

In practical terms, capabilities operating under the BCS model MUST:
– produce the same type, structure and semantic category of output for equivalent inputs;
– avoid behaviour changes based on hidden heuristics or undeclared contextual signals;
– avoid varying behaviour based on the invoking agent or execution pathway;
– introduce bounded randomness only where it is explicitly declared and safely constrained.

Determinism enables:
– automated validation,
– repeatable testing,
– reliable cross-platform execution,
– confidence in capability behaviour during orchestration.

1.6.4 Capabilities as Declarative Specifications

A capability describes **what** behaviour must occur, not necessarily **how** an agent should internally structure its reasoning.
Capabilities define:

- the transformation performed,
- allowed and prohibited outputs,
- how input data should be interpreted,

- how output must be formatted,
- the behaviour under edge cases,
- the constraints and assumptions.

Agents interpret this declarative behaviour and implement it according to their own internal reasoning systems.

BCS deliberately does not prescribe:

- LLM reasoning flow,
- internal planning strategy,
- execution order of operations,
- agent-level safety systems.

Capabilities therefore remain **transparent and declarative**, while agents remain **procedural and adaptive**.

1.6.5 Capabilities as Vendor-Neutral Behaviour Contracts

A capability's behavioural contract must be valid across all platforms implementing BCS.
This guarantees that:

- developers can write once and deploy everywhere,
- marketplaces can index and validate capabilities consistently,
- agents from different vendors can invoke the capability safely,
- users experience predictable results regardless of platform.

Platform-agnostic behaviour ensures that capabilities are:

- portable,
- interoperable,
- resilient to internal changes in agent or model design,
- future-proof as AI ecosystems evolve.

1.6.6 Capabilities vs Conventional Software Components

Capabilities differ from conventional software constructs in several critical ways:

Not executable code

Capabilities express behaviour symbolically.

Execution is delegated to an agent's reasoning engine, not a processor or interpreter.

Not tools in the traditional API sense

A capability does not provide a callable function endpoint or perform external actions.

It operates only on the data provided.

Not plugins with embedded logic

Capabilities declare behaviour but do not contain internal executable logic or scripts.

Not complete applications

Capabilities are building blocks used by agents to construct larger behaviours.

These distinctions ensure that capabilities remain lightweight, safe, and universally interpretable.

1.6.7 Capabilities Within Agent Ecosystems

Capabilities exist to augment an agent's ability to perform tasks.
The high-level interaction model is:

1. Agent interprets user intent.
2. Agent decides that a capability is relevant.
3. Agent prepares input data according to the capability's schema.
4. Agent invokes the capability.
5. Capability produces deterministic output.
6. Agent integrates the output into its larger reasoning or workflow.

The capability does not control:

- invocation timing,
- task decomposition,
- multi-step processes,
- follow-up actions,
- broader goals.

This division of labour ensures clear responsibility boundaries.

1.6.8 Capabilities as Safe, Constrained Behaviour Units

Because capabilities operate within strict boundaries, they offer a powerful mechanism for enforcing safety:

- They can only output what their schema permits.
- They can only perform behaviours explicitly defined.
- They cannot improvise beyond their contract.
- They declare explicit prohibited content.
- They include deterministic fallback behaviours.
- They operate within declared risk classifications.

This makes capabilities safer and more predictable than arbitrary generative output.

1.6.9 Motivations for the Conceptual Model

This conceptual framing exists to support:

- **cross-platform consistency**,
- **developer clarity**,
- **safer agent orchestration**,
- **regulatory audit trails**,
- **marketplace standardisation**,
- **future capability evolution**,
- **test harness development**,
- **automated validation pipelines**.

It is the conceptual backbone that ensures all capabilities—and all vendors—interpret BCS consistently.

Having introduced the conceptual model of a capability, the following sections define the formal structures and behavioural rules that capabilities MUST conform to under BCS.

1.7 Formal Definition of a Capability

This subsection provides the precise, normative definition of a capability within the BBY Capability Specification (BCS). All subsequent requirements in Sections 1–5 depend on the terms, conditions, and constraints established here.

A **capability** is defined as:

A deterministic, stateless, declarative behavioural unit that performs a narrowly scoped transformation on input data and produces output strictly according to a predefined schema, without autonomous reasoning, planning, or side effects.

The following clauses are normative and collectively constitute the official definition of a capability under BCS.

1.7.1 Deterministic Behavioural Unit

A capability MUST describe a behaviour that:

- produces predictable and consistent outputs,
- does not vary based on agent identity,
- does not depend on external or historical context,
- functions solely on the provided input data,
- adheres to a behavioural contract defined by the developer.

Determinism is required for:

- testability,
- cross-platform execution,
- validation pipelines,
- safety enforcement,
- reproducibility.

A capability MUST NOT include behaviours that are emergent, improvisational, or intentionally variable.

1.7.2 Stateless Execution

Capabilities MUST be stateless.
This means:

- they MUST NOT retain, cache, or reference previous invocations,
- they MUST NOT modify internal state across calls,
- each invocation MUST be treated as a complete, isolated execution event.

Stateful designs violate the core BCS model and result in non-conformance.

1.7.3 Declarative Behaviour Specification

Capabilities MUST express behaviour in a **declarative**, not procedural, form.

Declarative behaviour:

- describes the *intended transformation*,
- specifies *rules and constraints*,
- defines expected output,
- omits procedural implementation details.

Capabilities do NOT include:

- algorithms,
- execution logic,
- control flow,
- code-like constructs,
- intermediate steps intended for machine interpretation.

Agents translate declarative behaviour into internal execution sequences using their own reasoning mechanisms.

1.7.4 Narrow Scope and Single Responsibility

A capability MUST implement a single, well-defined purpose.

A capability that attempts to:

- perform multiple unrelated transformations,
- branch into different behaviours based on intent,
- include multiple domain areas,

…is **non-conformant**.

Narrow scope ensures:

- predictability,
- concise documentation,
- safe boundaries,
- coherent validation,
- reduced ambiguity.

This principle aligns with the **single-responsibility model** widely used in software engineering and standards development.

1.7.5 Schema-Bound Input and Output

All inputs and outputs of a capability MUST be defined within:

- a **structured input schema**,
- a **structured output schema**.

Schemas MUST:

- define field names and types,
- specify required vs optional fields,
- include value constraints where applicable,
- describe expected content and structure.

Capabilities MUST NOT:

- accept free-form, unbounded input outside schema definitions,
- produce output that violates or extends the output schema,
- dynamically alter schema structures based on the task.

Schema-bound behaviour ensures that capabilities are:

- machine-validated,
- predictable,
- inspectable,
- cross-platform compatible.

1.7.6 No Autonomous Reasoning or Planning

A capability MUST NOT perform:

- multi-step planning,
- contextual reasoning,
- autonomous decision-making,
- task decomposition,
- chain-of-thought operations,
- hidden reasoning sequences.

These behaviours belong exclusively to **agents**.

The capability MUST restrict itself to:

A bounded, deterministic transformation defined entirely by its behavioural description and input schema.

1.7.7 No Side Effects or External Actions

A capability MUST NOT:

- call external APIs,
- write or modify files,
- alter system state,
- interact with the environment,
- trigger irreversible operations,
- invoke other capabilities,
- perform network requests.

Side effects violate determinism and jeopardise safety.

Capabilities must operate purely on **input → output**, without external dependencies.

1.7.8 Behaviour Must Be Fully Described

A capability MUST include a complete and unambiguous behaviour description that:

- declares assumptions,
- defines constraints,
- outlines the transformation performed,
- describes how to interpret inputs,
- specifies how to construct outputs.

The behaviour description MUST be sufficient for:

- validators to assess validity,
- agents to implement execution,
- auditors to verify safety,
- developers to understand intended functionality.

Dependencies on undocumented behaviour are prohibited.

1.7.9 Safety Boundaries Are Part of the Definition

Because capability behaviour cannot be separated from safety constraints, this subsection describes how behavioural intent and safety boundaries interact within the capability model.

A capability MUST:

- declare safety risks,
- list prohibited content,
- specify fallback behaviour,
- define domain limitations.

Safety boundaries are considered part of the capability's behaviour and therefore MUST be included in the formal definition.

1.7.10 Execution Environment Neutrality

A capability MUST be fully independent of:

- model architecture,
- inference engine,
- vendor platform,
- programming language,
- operating system,
- hardware environment.

Any dependency on such characteristics violates the platform-agnostic nature of BCS.

1.7.11 Summary Definition (Normative)

A capability is considered compliant with the BCS formal definition only if it satisfies all of the following:

1. Deterministic
2. Stateless
3. Declarative
4. Narrowly scoped
5. Schema-bound
6. Non-reasoning
7. Without side effects
8. Fully behaviour-described
9. Explicitly safety-defined
10. Platform-neutral

Failure to meet any of these conditions renders the capability **non-conformant**, regardless of structural correctness.

The behaviour structure defines how a capability declares what it does, under what conditions it operates, and how its outputs are constrained.

1.8 Behaviour Structure Model

This subsection defines the normative structure of a capability's behaviour.
While Section 1.2 defined what a capability is, the Behaviour Structure Model defines how the behaviour is declared so that platforms, agents, validators and auditors can reliably interpret it.

A capability's behaviour MUST be expressed as a **transparent, deterministic, declarative description** of a single transformation. The behaviour description forms the semantic core of the capability and governs every aspect of its execution.

Implementation implication: A capability description MUST express behaviour as a structured composition of explicitly declared components, rather than relying on implicit or emergent behaviour within the execution environment.

1.8.1 Purpose of the Behaviour Structure Model

The Behaviour Structure Model ensures that:

- behaviour can be validated automatically,
- agents can interpret and execute the behaviour safely,
- behaviour is transparent and unambiguous,
- prohibited ambiguity is identified early,
- fallback rules are well defined,
- capabilities are interoperable across environments.

Without a consistent behavioural structure, capabilities would be interpreted differently across vendors, undermining predictability and safety.

1.8.2 Behaviour Structure Requirements

The behaviour MUST be expressed as a structured block containing:

1. **Purpose Statement**
2. **Behavioural Description**
3. **Input Interpretation Rules**
4. **Transformation Logic (Declarative)**
5. **Output Construction Rules**
6. **Assumptions and Preconditions**
7. **Constraints**
8. **Error and Edge Case Behaviour**
9. **Fallback Behaviour**

Each component MUST be present unless explicitly allowed as optional in Section 2.

Capabilities that omit or ambiguously express any required component are **non-conformant**.

1.8.3 Purpose Statement

The purpose statement MUST consist of a single, concise sentence that defines:

- the capability's primary function,
- its intended use case,
- its narrow behavioural domain.

It MUST NOT include:

- multi-purpose descriptions,
- optional conditional behaviours,
- domain crossing.

Example (valid)
"To extract structured financial entities from a text input."

Example (invalid)
"To extract financial entities, summarise content, and interpret sentiment."

1.8.4 Behavioural Description

The behavioural description MUST:

- explain the transformation performed,
- follow a declarative form (not procedural steps),
- describe exactly what the capability MUST output,
- specify the conditions under which behaviour applies,
- avoid implementation details.

It MUST NOT include:

- algorithmic pseudocode,
- "if agent decides" language,
- internal reasoning steps.

Declarative behavioural descriptions allow different agents to execute the behaviour consistently.

1.8.5 Input Interpretation Rules

The behaviour MUST specify how to interpret each field in the **input schema**:

- text fields,
- numbers,
- categories,
- lists,
- nested objects.

These rules MUST:

- be explicit,
- define boundaries,
- specify how ambiguous inputs should be handled,
- align with the schema constraints.

Capabilities MUST NOT infer additional meaning from context outside provided fields.

1.8.6 Transformation Logic (Declarative)

The transformation logic describes the essential mapping from inputs to outputs.

Declarative logic:

- defines the relationship between input and output,
- specifies required operations without describing algorithms,
- ensures deterministic and platform-agnostic execution.

Valid declarative form:
“Extract all chemical substances from the input and return them as a list of objects.”

Invalid procedural form:
“Loop through each token, perform entity recognition, and filter results.”

1.8.7 Output Construction Rules

The behaviour MUST:

- define how outputs are constructed,
- bind outputs to schema fields,
- specify how to handle missing or invalid data,
- guarantee that outputs always conform to schema.

Outputs MUST NOT extend beyond the schema, even if logically relevant.

If information cannot be produced safely or deterministically, the behaviour MUST defer to fallback rules.

1.8.8 Assumptions and Preconditions

Assumptions are conditions that MUST hold true for the behaviour to function as intended.

Examples:

- “Input text is in English.”
- “Numerical values represent whole numbers.”
- “Timestamp fields follow ISO 8601 format.”

Preconditions MUST be explicit.
A capability MUST NOT rely on hidden assumptions about:

- user intent,
- domain context,

- history of prior queries,
- external systems.

1.8.9 Constraints

Constraints define boundaries that restrict behaviour.

A capability MUST include all relevant constraints such as:

- maximum input size,
- allowable value ranges,
- permitted input categories,
- safe output conditions.

Violating any constraint MUST trigger fallback behaviour or error handling.

Constraints are critical for:

- safety,
- validation,
- predictability,
- runtime enforcement.

1.8.10 Error and Edge Case Behaviour

The behaviour MUST specify how to handle:

- malformed inputs,
- missing fields,
- invalid values,
- ambiguous content,
- situations where outputs cannot be safely produced.

Edge case behaviour MUST be deterministic.

Capabilities MUST NOT leave error behaviour to agent discretion.

1.8.11 Fallback Behaviour

Fallback behaviour MUST be defined for conditions where:

- safety rules would be violated,
- transformation cannot be completed,
- constraints are breached.

Fallbacks MUST:

- be deterministic,
- be schema-compliant,
- be safe,
- contain no extraneous content.

Fallback behaviour is a **mandatory component** and cannot be omitted.

1.8.12 Prohibited Behaviour Constructs

Capabilities MUST NOT contain:

- hidden conditional branches not expressed in the behaviour block,
- multi-behaviour decision trees,
- references to agent reasoning,
- procedural or algorithmic instructions,
- any implication of state or memory,
- instructions to call other capabilities or tools.

These violate determinism and cross-platform consistency.

1.8.13 Purpose of Behaviour Structure Enforcement

The Behaviour Structure Model ensures that capabilities:

- remain bounded and safe,
- cannot behave unexpectedly,
- remain interpretable by all conformant platforms,
- pass validation pipelines consistently,
- are auditable and documentable.

This model is central to the BCS philosophy:
transparent, deterministic, safe behavioural declarations.

1.9 Determinism Requirements

Determinism is a foundational and non-negotiable requirement of the BBY Capability Specification (BCS).
A capability MUST behave predictably and consistently across all platforms, execution environments, and agent implementations.
This section defines the normative rules governing deterministic behaviour.

A capability is deterministic if:

Given identical valid inputs and conditions, it produces the same category, structure, and semantics of output, without variation in safety-relevant or behaviour-defining aspects.

Determinism ensures:

- cross-platform compatibility,
- predictable agent orchestration,
- automatic validation,
- safe multi-capability workflows,
- regulatory auditability,
- marketplace trustworthiness.

Non-determinism is considered a critical violation of BCS and results in **immediate non-conformance**.

Implementation implication: For a capability to be considered conformant, identical inputs, constraints, and contextual parameters MUST always produce the same observable behaviour when executed on any conforming platform.

1.9.1 Deterministic Output Structure

A capability MUST produce outputs that:

- match the defined output schema,
- contain all required fields,
- respect field types and constraints,
- adhere to the same structural format on every valid invocation.

Structural determinism does NOT require identical text, but the overall category and format MUST remain stable.

Example:
If the output schema defines:

```
{
  "summary": string,
  "key_points": list<string>
}
```

Then:

- summary MUST always be a concise summary string.
- key_points MUST always be a list of bullet-point-style items.

The style, length, and wording may vary **slightly**, but the structure may not.

1.9.2 Deterministic Semantic Category

The semantic intent of each field MUST remain constant.

If a field is defined as:

- a list of sentiment categories,
- extracted financial terms,
- normalised dates,
- named entities,

…it MUST always contain **that type of semantic content**, regardless of platform or agent.

A capability MUST NOT produce:

- paraphrases that alter meaning,
- semantic drift due to generative freedom,
- alternative behaviours based on perceived context,
- different interpretations across platforms.

Semantic determinism ensures capabilities behave the same everywhere.

1.9.3 No Hidden Conditional Branching

Capabilities MUST NOT:

- include undocumented behavioural branches,
- perform different behaviours based on input quirks,
- adjust execution based on agent signals,
- change behaviour over time unless versioned.

Every permitted branch MUST be:

- declared explicitly in the behaviour description,
- bound to schema and constraints,
- deterministic in its own right.

Example of prohibited behaviour:

“If input contains legal language, summarise differently.”

Unless this rule is explicitly declared and fully deterministic, it violates BCS.

1.9.4 No Dependency on External or Historical Context

A capability MUST NOT depend on:

- conversation history,
- previous invocations,
- user profile data,
- time of day,
- environmental conditions,
- agent state or memory,
- model-dependent embeddings,
- global system settings.

Every invocation MUST be isolated and stateless.

If a capability requires consistent reference data (e.g., currency symbols), such data MUST be included in the capability file or input schema.

1.9.5 Deterministic Handling of Edge Cases

For all invalid, ambiguous, or incomplete inputs, a capability MUST:

- follow the documented fallback rules,
- produce predictable error objects,
- remain within schema boundaries,
- apply constraints consistently.

Edge-case behaviour MUST NOT:

- vary across platforms,
- produce generative filler text,
- produce inconsistent error messages,
- depend on undocumented heuristics.

1.9.6 No Generative Improvisation

Capabilities MUST NOT include:

- creative expansion,
- reasoning sequences,
- contextual interpretation,
- open-ended generation,
- style-based improvisation.

Generative freedom belongs to agents — not capabilities.

If generation is required (e.g., rewrite in a fixed style), the allowed output MUST be explicitly defined and bounded.

1.9.7 Deterministic Ordering and Formatting

Lists, sets, and grouped outputs MUST:

- follow a stable ordering rule,
- remain consistent on each invocation unless explicitly declared otherwise.

Examples:

- alphabetical ordering,
- input-order preservation,
- frequency-based ordering,
- schema-defined ordering.

If ordering is unspecified, the capability is non-conformant.

1.9.8 Deterministic Numerical and Symbolic Handling

Capabilities MUST treat numerical or symbolic values consistently:

- rounding rules MUST be defined,
- units MUST be declared,
- permissible ranges MUST be constrained,
- parsing behaviours MUST be deterministic.

Numerical inconsistencies break cross-platform trust and MUST NOT occur.

1.9.9 Deterministic Fallbacks

When fallback behaviour is triggered, the fallback output MUST:

- match the fallback definition exactly,
- remain fully schema-valid,
- avoid creative or unstructured content,
- be identical across platforms if conditions match.

Fallback determinism is vital for safety.

1.9.10 Deterministic Safety Enforcement

Safety boundaries MUST be applied deterministically.

For example:

- If prohibited content is detected, the capability MUST always activate fallback or error behaviour.
- If input violates domain restrictions, it MUST refuse execution consistently.
- If a risk classification prohibits certain transformations, those outputs MUST never occur.

Any volatility in safety enforcement is a specification violation.

1.9.11 Determinism Under Adversarial Inputs

Capabilities MUST maintain deterministic behaviour even when inputs:

- are malformed,
- contain adversarial structures,
- attempt prompt injection,
- include mixed languages,
- push up against schema limits.

The capability must follow:

- constraints,
- fallback,
- error handling,

…without improvisation or drift.

1.9.12 Summary of Deterministic Guarantees (Normative)

A capability is deterministic only if:

1. Output structure is stable.
2. Semantic content categories are stable.
3. No hidden behavioural branching exists.
4. No external context influences execution.
5. Edge-case behaviour follows defined rules.
6. No generative freedom appears.
7. Ordering and formatting are predictable.
8. Numerical handling is consistent.
9. Fallback behaviour is stable.
10. Safety enforcement is consistent.

Failure to meet ANY requirement renders the capability **non-deterministic and therefore non-conformant**.

1.10 Input Schema Requirements

The input schema defines the **complete, explicit, machine-validatable structure** of all data that a capability MAY receive during invocation.
Because capabilities MUST be deterministic, stateless, and narrow in scope, the input schema forms a critical boundary between the agent and the capability.

A capability MUST NOT accept any input that is not described in its input schema.
The schema serves as the one and only contract governing admissible input.

Input schema requirements ensure:

- interoperability across platforms
- full agent predictability
- validation and safety enforcement
- reliable testing and auditing
- deterministic execution

This section defines the normative rules governing input schema structure.

Implementation implication: All inputs to a capability MUST be explicitly declared in a formal input schema. No undeclared or implicit inputs MAY influence capability behaviour, and a conforming platform MUST reject any capability description whose inputs cannot be validated against the declared schema.

The input and output schemas provide the structural contract that enables validation, deterministic execution, and cross-platform interoperability.

1.10.1 All Inputs MUST Be Schema-Bound

Every possible input to a capability MUST be explicitly defined in its input_schema.

A capability MUST NOT:

- infer meaning from unstated context,
- accept arbitrary free-form values outside schema constraints,
- interpret user intent beyond what is contained in the schema,
- rely on agent-provided metadata not listed in the schema.

If data is not present in the schema, the capability MUST behave as though it does not exist.

1.10.2 Supported Data Types

The schema MUST use only data types defined by BCS (see Section 2 for formal type definitions).
Allowed primitive types include:

- string
- integer
- float
- boolean

- enum
- list<T>
- object (nested schemas)

Capabilities MUST NOT invent custom primitive types.
Extended types MAY be defined only through object nesting.

1.10.3 Required vs Optional Fields

Each input field MUST be designated as either:

- **required** — MUST be present on every invocation
- **optional** — MAY be omitted

If a required field is missing, the capability MUST activate:

- error behaviour, or
- fallback behaviour,

as specified in the behaviour block.

Optional fields MUST have defined behaviour for:

- when the field is present, and
- when the field is absent.

Capabilities MUST NOT leave the interpretation of missing optional fields ambiguous.

1.10.4 Field Descriptions MUST Be Clear and Unambiguous

Each field MUST include a human-readable description that:

- defines the purpose of the field,
- clarifies expected content,
- specifies any domain limitations,
- defines permissible ranges, formats, or constraints.

Descriptions MUST NOT:

- rely on external documentation,
- be vague (“general text input”),
- defer interpretation to the agent.

Ambiguity in field descriptions is a conformance violation.

1.10.5 Constraints MUST Be Explicit

Constraints define the legal boundaries for input fields. They MAY include:

- length limits (min/max characters)
- numeric ranges
- permitted categories (enum)
- maximum list sizes
- required formats (e.g., ISO 8601 dates)
- domain or language restrictions
- regular expression patterns

Constraints MUST be:

- enforceable,
- deterministic,
- machine-checkable.

A capability MUST NOT silently accept input that violates constraints.

1.10.6 Input Interpretation MUST Be Deterministic

The behaviour block MUST describe exactly how each schema field is to be interpreted.

Interpretation rules MUST:

- correspond directly to schema definitions,
- specify how ambiguous or partial values are handled,
- be free of generative or implicit reasoning.

Capabilities MUST NOT:

- treat text inputs as multi-intent sources,
- infer additional meaning from formatting, tone, or style,
- dynamically reinterpret fields based on content.

1.10.7 Nested Objects and Complex Fields

Objects MAY contain nested schemas.

Nested schemas MUST:

- maintain the same required/optional discipline,
- have clearly defined constraints at each level,
- avoid ambiguous or recursive structures.

Capabilities MUST NOT include:

- unbounded nesting,
- circular schema references,
- polymorphic objects (“this may be an object or a list depending on input”).

These violate determinism and validation.

1.10.8 Lists and Collections

Lists MUST define:

- the data type of list elements,
- whether duplicates are permitted,
- whether ordering is meaningful,
- minimum and maximum allowable lengths.

If ordering matters, a rule for deterministic ordering MUST be specified.

If ordering does NOT matter, this MUST be explicitly stated.

A capability MUST NOT alter list ordering arbitrarily between invocations.

1.10.9 Input Preprocessing Rules

If preprocessing is required (e.g., trimming whitespace, lowercasing, normalising symbols), such rules MUST be:

- declared explicitly,
- deterministic,
- schema-consistent.

Capabilities MUST NOT:

- apply hidden preprocessing,
- alter meaning while preprocessing,
- depend on vendor-specific normalisation rules.

1.10.10 Handling of Invalid Inputs

Invalid input MUST trigger one of:

1. A deterministic error object
2. Fallback behaviour

The behaviour block MUST specify which applies.

Capabilities MUST NOT:

- attempt to “guess” user intent,
- creatively reinterpret malformed inputs,
- proceed with partial output without a deterministic basis.

Safety takes precedence over permissiveness.

1.10.11 Inputs MUST NOT Include Behaviour Instructions

Inputs MUST NOT include:

- instructions altering capability behaviour,
- instructions altering safety boundaries,
- meta-instructions changing schema,
- hints for alternative behaviours.

Capabilities MUST ignore any input content not covered by the schema.

If behaviour-altering instructions appear inside a valid field (e.g., a text field), the capability MUST follow its defined constraints and safety rules, NOT the embedded instructions.

1.10.12 Inputs MUST Not Imply Intent Beyond Schema

A capability MUST NOT:

- interpret inputs as user goals,
- read prompts inside text fields as requests,
- infer alternative tasks from content,
- behave differently based on stylistic features.

Agents interpret intent; capabilities interpret structured data.

1.10.13 Summary of Input Schema Guarantees (Normative)

A capability’s input schema is valid only if:

1. All fields are explicitly defined and typed.
2. Required/optional designations are clear.
3. Constraints are explicit and machine-checkable.
4. Interpretation rules are declared and deterministic.
5. No external context influences behaviour.
6. No behavioural instructions appear in inputs.

7. Invalid inputs trigger deterministic error or fallback.
8. Schema supports cross-platform interoperability.

Violation of any requirement renders the capability **non-conformant**

1.11 Output Schema Requirements

The output schema defines the complete and exclusive structure of all data a capability MAY produce.
Where the input schema governs what the capability receives, the output schema governs **what it is permitted to return**.

Because capabilities MUST be deterministic, narrowly scoped, and safe, the output schema is fundamental to ensuring:

- predictable behaviour across platforms,
- agent trust in output consistency,
- safe downstream processing,
- auditable and observable execution,
- automated validation,
- marketplace standardisation.

A capability MUST NOT produce any output outside the output schema, under any circumstances.

Implementation implication: The outputs of a capability MUST be defined in a formal output schema, and all observable results produced by a conformant implementation MUST conform to that schema. Platforms MUST treat deviations from the declared output schema as a validation failure rather than an acceptable variation in behaviour.

1.11.1 All Outputs MUST Conform to the Output Schema

Every capability invocation MUST produce outputs that:

- strictly adhere to the structure defined in output_schema,
- include all required fields,
- use valid field types,
- satisfy all constraints and value ranges,
- avoid introducing unexpected or free-form content.

A capability MUST NOT:

- include extra fields,
- omit required fields,
- change field meaning based on context,
- structure output differently for different platforms or agents.

Violation of schema conformity constitutes a **severe conformance failure**.

1.11.2 Allowed Output Types

Output schema types MUST be selected from the same set allowed for input schemas (see Section 1.5.2), namely:

- string
- integer
- float
- boolean
- enum
- list<T>
- object

The schema MUST NOT introduce:

- procedural or algorithmic indicators,
- dynamic type switching,
- polymorphic structures (“this field may be string OR object”),
- vendor-specific type extensions.

Nested objects MUST follow the same rules as top-level structures.

1.11.3 Required vs Optional Output Fields

Each field in the output schema MUST be marked as:

- **required** — MUST be present in valid outputs
- **optional** — MAY be included or omitted based on deterministic rules

The capability MUST define:

- the conditions under which optional fields appear,
- the expected default values or fallback outputs,
- behaviour when required outputs cannot be produced safely.

A capability MUST NOT:

- vary optional-field behaviour unpredictably,
- conditionally generate optional fields without documented rules.

1.11.4 Output Field Descriptions MUST Be Precise

Each output field MUST include a description that:

- defines what the field represents,

- describes its semantic purpose,
- clarifies the acceptable content domain,
- indicates any safety constraints associated with that field,
- specifies formatting or normalisation requirements.

Descriptions MUST NOT:

- be vague (“some text information”),
- reference unstated inference rules,
- defer meaning to agent interpretation.

Field meaning MUST be clear, stable, and platform-agnostic.

1.11.5 Constraints on Output Values

Every output field MAY include constraints such as:

- allowed value ranges,
- maximum length or size limits,
- permitted categories (enum values),
- lexically normalised forms,
- language or domain restrictions,
- format patterns (e.g., ISO strings),
- ordering constraints for lists.

Constraints MUST be:

- enforceable,
- deterministic,
- machine-validatable.

A capability MUST NOT produce output that violates constraints, even if logically relevant or user-requested.

1.11.6 Output MUST Be Deterministic

Outputs MUST satisfy all determinism requirements defined in Section 1.4. Specifically:

- output fields MUST reflect consistent semantic roles,
- list ordering MUST follow declared ordering rules,
- edge-case outputs MUST be deterministic and repeatable,
- fallback outputs MUST remain stable across platforms.

If variation occurs in output content due to LLM generative variance, the capability is non-conformant.

1.11.7 Output MUST NOT Contain Behavioural Descriptions

Outputs MUST NOT include:

- explanations of reasoning,
- intermediate thoughts or steps,
- paraphrased summaries of behaviour,
- procedural descriptions of how the transformation was implemented.

A capability's output represents **results**, not **internal process**.

Agents handle reasoning independently.

1.11.8 Output MUST NOT Include Safety-Disallowed Content

A capability MUST NOT generate content that violates:

- its own safety restrictions,
- global safety classifications (Section 3),
- prohibited content lists,
- regulatory compliance requirements.

Safety violations MUST immediately trigger:

- fallback behaviour, or
- explicit error output,

as specified in the capability behaviour block.

Even if the input contains unsafe or disallowed content, the output MUST remain compliant with schema and safety rules.

1.11.9 Handling of Incomplete, Uncertain, or Ambiguous Outputs

A capability MUST clearly define how to behave when output cannot be fully produced, including:

- returning null/token placeholders where allowed,
- omitting optional fields where permitted,
- activating fallback behaviour,
- returning deterministic empty structures (e.g., [] or "").

Capabilities MUST NOT:

- fill outputs with invented or hallucinated detail,
- generate approximations outside schema scope,
- provide speculative or low-confidence content unless explicitly stated as allowed.

Hallucinations constitute non-conformance.

1.11.10 Formatting and Normalisation Requirements

If formatting rules apply, they MUST be defined explicitly, including:

- casing (e.g., sentence case, uppercase, lowercase),
- whitespace normalisation,
- newline or paragraph rules,
- fixed lexical style,
- number formatting (decimals, commas, units),
- ordering conventions for multi-item fields.

Agents MUST NOT be required to guess or reinterpret formatting.

If formatting is irrelevant or flexible, the capability MUST explicitly state this.

1.11.11 Output MUST NOT Include Additional Fields

Capabilities MUST NOT return:

- metadata not defined in the schema,
- model-specific fields (e.g., scores, confidences),
- explanatory notes,
- debugging information,
- any derived content not explicitly allowed.

Extra fields introduce unpredictability and violate cross-platform consistency.

1.11.12 Output MUST Be Complete and Self-Contained

Output MUST NOT require:

- follow-up queries,
- external system context,
- additional clarification from the agent.

All necessary information MUST be included within the schema-defined structure.

If additional necessary data cannot be produced due to constraints, the capability MUST:

- invoke fallback behaviour, or
- return deterministic error output.

1.11.13 Summary of Output Schema Guarantees (Normative)

A capability output is compliant only if:

1. It strictly conforms to schema structure.
2. All required fields are present.
3. Optional-field rules are deterministic.
4. Field semantics are stable.
5. Constraints are obeyed.
6. Output is deterministic and reproducible.
7. No reasoning or behaviour descriptions appear.
8. No prohibited or unsafe content is emitted.
9. No hallucinated content appears.
10. Formatting follows the declared rules.
11. No additional fields are added.
12. Output is complete and self-contained.

Violating any of the above results in **non-conformance**, regardless of input validity or safety classification.

1.12 Constraints & Assumptions Model

Capabilities operate within tightly defined behavioural and semantic boundaries.
Constraints and assumptions define these boundaries and ensure that capabilities behave predictably, safely, and consistently across all platforms.

This section establishes the normative requirements for:

- what assumptions a capability is allowed to make,
- how constraints MUST be defined and enforced,
- how violations MUST be handled deterministically,
- how agents interpret these restrictions,
- how capability behaviour remains bounded and safe.

Constraints and assumptions are NOT optional commentary.
They are **legally binding parts of the capability's behavioural definition**.

1.12.1 Purpose of Constraints and Assumptions

Constraints and assumptions:

- define the operational domain of the capability,
- prevent ambiguous interpretation,
- enable deterministic behaviour,
- establish safety boundaries,
- allow predictable agent orchestration,
- support automated validation pipelines,

- ensure cross-platform consistency.

Without explicit constraints and assumptions, capabilities become unpredictable or unsafe.

Implementation implication: A capability MUST explicitly declare all operational assumptions and constraints that affect its behaviour. A conforming platform MUST treat execution outside these declared assumptions and constraints as non-conformant behaviour, and MUST NOT attempt to infer or substitute undeclared environmental expectations.

1.12.2 Assumptions MUST Be Explicit

A capability MAY include assumptions about:

- expected input formatting,
- domain or language context,
- numerical ranges,
- intended user scenarios,
- typical data shapes,
- domain-specific semantics.

However:

✓ All assumptions MUST be stated explicitly.

✗ Unstated or implicit assumptions are prohibited.

Assumptions MUST NOT include:

- hidden expectations,
- contextual history,
- user intent outside the schema,
- expectations about agent behaviour,
- expectations about downstream systems.

If an assumption is not written in the capability file, it is invalid.

1.12.3 Assumptions MUST NOT Require External Context

An assumption MUST NOT:

- reference prior model interactions,
- depend on conversation history,
- imply knowledge of user identity,
- require internet access,
- rely on external datasets or APIs,
- assume cultural or geographic context unless explicitly stated.

Capabilities must act solely on:

- declared assumptions,
- provided inputs,
- schema constraints.

1.12.4 Constraints MUST Be Machine-Enforceable

Each constraint MUST be:

- clearly defined,
- specific,
- deterministic,
- enforceable by automated validation,
- independent of reasoning or interpretation.

Examples of valid constraints:

- “Field age MUST be an integer from 0 to 120.”
- “Input text MUST NOT exceed 2000 characters.”
- “The list MUST contain between 1 and 10 items.”
- “Language MUST be English.”

Invalid constraints include:

- “Input should generally be short.”
- “User is expected to behave reasonably.”
- “Text should be appropriate for context.”
- “Assumes input is typical for the domain.”

Ambiguity is prohibited.

1.12.5 Domain Constraints MUST Be Clear and Narrow

Capabilities MUST explicitly define any domain limitations, such as:

- “Only works with legal contracts”
- “Supports biology-related terminology only”
- “Handles English-language text only”

If a capability is not intended for general-purpose use, domain constraints MUST make this known.

Capabilities MUST NOT:

- attempt broader interpretation outside declared domain,
- perform unsafe or speculative behaviour when inputs fall outside the domain.

1.12.6 Behavioural Constraints MUST Be Deterministic

Behavioural constraints define:

- what the capability MUST do,
- what it MUST NOT do,
- what behaviours are prohibited,
- how edge cases MUST be handled.

Examples of valid behavioural constraints:

- “Capability MUST NOT infer user intent beyond the schema.”
- “Capability MUST NOT generate content not supported by the output schema.”
- “Capability MUST NOT alter list ordering.”

Behavioural constraints MUST NOT:

- conflict with the capability definition (Section 1.2),
- introduce vendor-specific interpretation,
- require procedural logic,
- rely on internal agent mechanisms.

1.12.7 Safety Constraints MUST Be Fully Declared

Safety-related constraints MUST specify:

- prohibited output categories,
- risk classifications,
- disallowed transformations,
- domain-specific restrictions,
- mandatory fallback conditions.

Examples:

- “Capability MUST NOT generate medical diagnoses.”
- “Capability MUST NOT rewrite content into extremist or harmful categories.”
- “If prohibited content appears in input, capability MUST activate fallback behaviour.”

Safety constraints MUST be explicit and non-negotiable.

1.12.8 Constraint Violations MUST Trigger Deterministic Behaviour

When input violates constraints, the capability MUST:

Either:

1. Return a deterministic **error object**, OR
2. Trigger a deterministic **fallback output**.

Capabilities MUST NOT:

- attempt creative reinterpretation,
- execute partial transformations,
- return “best effort” outputs,
- ignore constraint violations.

Safety and determinism take priority.

1.12.9 Assumptions MUST NOT Extend Behaviour

Assumptions describe **conditions of operation**, not **additional behaviours**.

An assumption MUST NOT:

- modify the transformation,
- introduce conditional logic,
- alter output format,
- provide alternative execution paths.

All behaviour MUST come from the behaviour block, NOT assumptions.

1.12.10 Constraints MUST Be Consistent Across Platforms

Constraints MUST NOT:

- depend on vendor-specific behaviour,
- vary based on inference engine,
- adjust dynamically based on run-time conditions,
- rely on system-level properties.

All platforms MUST enforce the same constraints in the same way.

1.12.11 Nested Constraints

If objects contain nested fields, constraints MUST apply recursively at every level.

Nested constraints MUST:

- follow the same required/optional rules,
- define explicit limitations at each level,
- override no parent constraints unless explicitly allowed.

Recursive ambiguity is prohibited.

1.12.12 Constraint and Assumption Conflicts

If a conflict exists between:

- behaviour description,
- constraints,
- assumptions,
- schemas, or
- safety requirements,

Then the following precedence applies:

1. Safety constraints

2. Behavioural constraints

3. Schema constraints

4. Assumptions

Assumptions never override constraints or safety.

1.12.13 Summary of Constraint & Assumption Guarantees (Normative)

A capability is compliant with the Constraints & Assumptions Model only if:

1. All assumptions are explicit.
2. All assumptions require no external context.
3. All constraints are deterministic and enforceable.
4. Domain constraints are narrow and well-defined.
5. Safety constraints are explicit and binding.
6. Constraint violations always trigger deterministic fallback or error.
7. Assumptions do not extend behaviour.
8. Constraints are consistent across platforms.
9. No ambiguous or implicit constraints exist.

Failure to meet any requirement results in **non-conformance**.

1.13 Safety Interaction Model

The Safety Interaction Model defines the normative rules governing how capabilities MUST behave when encountering unsafe content, prohibited transformations, restricted domains, or violations of safety classifications defined elsewhere in the BCS standard.

Capabilities operate within **strict deterministic boundaries**, and safety enforcement forms a core part of these boundaries.
This section ensures that all platforms enforce safety:

- consistently,
- predictably,
- transparently,
- deterministically,
- without vendor-specific variance.

Capabilities MUST NOT perform any behaviour or transformation that falls outside their declared safety profile or violates global safety restrictions.

1.13.1 Purpose of the Safety Interaction Model

The Safety Interaction Model exists to:

- define when a capability MUST refuse execution,
- enforce domain-specific safety boundaries,
- define deterministic fallback or error responses,
- prevent unsafe or disallowed transformations,
- ensure cross-platform safety consistency,
- guarantee that agents cannot bypass capability safety rules,
- provide auditors with a clear safety enforcement trail.

Capabilities MUST behave safely even when invoked incorrectly or maliciously.

Implementation implication: A capability MUST declare all safety-relevant interactions, including side-effects, external resource access, and risk-mitigating controls. Conforming platforms MUST enforce these declared safety boundaries during execution, and MUST treat undeclared safety-relevant behaviour as non-conformant.

1.13.2 Safety Boundaries MUST Be Explicit

Every capability MUST declare:

- its safety domain,
- its prohibited output categories,
- its prohibited transformations,
- any high-risk input types it cannot handle.

Safety boundaries MUST NOT be:

- inferred implicitly,
- partially documented,
- left to agent interpretation,
- dependent on previous interactions,

- conditioned on external context.

If a safety boundary is not explicitly stated, it does not exist.

1.13.3 Capabilities MUST NOT Produce Unsafe Output

A capability MUST NOT generate output that:

- violates global safety rules (Section 3),
- violates capability-specific safety restrictions,
- exceeds declared risk level,
- includes illegal or harmful content,
- fabricates factual claims within restricted domains (e.g., medical, legal).

If unsafe content appears in the input, output MUST remain safe regardless of:

- user intent,
- agent intent,
- request phrasing,
- contextual framing.

1.13.4 Safety Violations MUST Trigger Deterministic Behaviour

If a capability detects unsafe or disallowed elements in the input or output, it MUST:

1. **Refuse the transformation**, and
2. Produce a deterministic **fallback** or **error output**, depending on the behaviour specification.

Capabilities MUST NOT:

- return partial unsafe content,
- attempt to “sanitize” content creatively,
- rely on LLM judgment to decide safety,
- behave differently across platforms when encountering unsafe conditions.

Safety enforcement MUST be absolute.

1.13.5 Safety Scopes MUST Be Narrow and Well-Defined

Capabilities MUST define exactly which content categories they are allowed to transform.

Examples:

- A capability that classifies text sentiment may handle general content but may be forbidden from processing sexual, extremist, or medical content.

- A capability that extracts chemical entities may be forbidden from interpreting content relating to illegal drug production.
- A capability that rewrites text into simplified language may be prohibited from rewriting violent or hate-related content.

Capabilities MUST NOT:

- implicitly broaden their domain,
- improvise when content is outside scope,
- attempt “best-effort” transformations.

Out-of-scope safety detection MUST trigger fallback behaviour.

1.13.6 Input-Based Safety Triggers

A capability MUST trigger its safety fallback if:

- prohibited entities appear in the input,
- input content violates domain constraints,
- the input suggests harmful or illegal activity,
- input contains disallowed content categories,
- constraints on value ranges or structure are breached.

Safety triggers MUST NOT rely on subjective interpretation.
Triggers MUST be rule-based and documented.

1.13.7 Output-Based Safety Enforcement

A capability MUST prevent output that:

- contains disallowed categories,
- contains hallucinated harmful content,
- attempts to expand the input with unsafe material,
- violates compliance classifications,
- misrepresents regulated or safety-critical information.

Even if the input is safe, the output MUST remain within safe boundaries.

1.13.8 No Capability May Override Safety Rules

Capabilities MUST NOT:

- weaken safety constraints,
- override platform-level safety,
- request privileged execution modes,

- infer that unsafe content is permissible,
- attempt to bypass restrictions via creative phrasing.

Capabilities cannot “opt out” of safety.

1.13.9 Safety Behaviour MUST Be Deterministic

Safety enforcement MUST:

- behave identically across platforms,
- produce consistent fallback outputs,
- remain stable under adversarial input,
- be fully predictable and testable.

Unsafe content MUST NOT produce:

- varied outputs,
- generative improvisation,
- partial execution,
- vendor-specific divergence.

Deterministic safety behaviour is mandatory.

1.13.10 Error Responses for Safety Violations

A capability MAY use:

✓ Fallback Outputs

A structured, safe alternative output.

✓ Error Outputs

A schema-defined structure indicating violation.

The behaviour block MUST specify which applies.

Capabilities MUST NOT:

- use natural-language disclaimers,
- explain safety decisions in free-form text,
- include warnings not defined in schema,
- instruct the agent or user to take alternative actions.

Output MUST remain within schema boundaries.

1.13.11 Interaction with Agent Safety Systems

Capabilities operate **under**, not above, agent-level safety systems.

Capabilities MUST NOT assume that:

- the agent will filter unsafe content,
- the agent will expand safety enforcement,
- platform-level safety subsystems will add guardrails,
- agents will interpret ambiguous content safely.

Capabilities MUST enforce their own safety boundaries **regardless of agent behaviour**.

If the agent fails to enforce its own safety, the capability MUST remain safe.

1.13.12 Adversarial Safety Handling

Capabilities MUST be resilient to adversarial or malicious input, including:

- prompt-injection attempts,
- harmful content disguised as benign,
- obfuscation techniques,
- mixed-language safety evasion,
- domain boundary probing.

Adversarial detection MUST trigger:

- deterministic safety behaviour,
- fallback or error output.

Capabilities MUST NOT attempt to interpret adversarial content creatively.

1.13.13 Safety Interaction MUST Be Platform-Independent

Safety outcomes MUST NOT differ across:

- inference models,
- AI vendors,
- agent implementations,
- runtime configurations.

A capability that is allowed in one environment MUST NOT silently fail or behave differently in another.

Safety rules MUST be:

- universal,
- deterministic,
- predictable.

1.13.14 Summary of Safety Interaction Guarantees (Normative)

A capability is compliant with the Safety Interaction Model only if:

1. All safety boundaries are explicit.
2. The capability NEVER generates unsafe output.
3. Safety violations always trigger deterministic fallback/error.
4. Domain and content restrictions are well-defined.
5. Input-based safety triggers are rule-based and consistent.
6. Output-based safety enforcement is absolute.
7. Capabilities cannot override or weaken safety.
8. Safety behaviour is deterministic across vendors.
9. Adversarial inputs cannot circumvent safety.
10. No creative or generative behaviour occurs in unsafe scenarios.

Any violation constitutes **immediate non-conformance**.

1.14 Error Handling Model

The Error Handling Model defines the normative rules governing how capabilities MUST behave when execution cannot proceed normally due to invalid inputs, constraint violations, safety violations, or internal behavioural limitations.

Error handling is essential for ensuring:

- deterministic recovery,
- cross-platform consistency,
- safety under unpredictable conditions,
- predictable behaviour for agents,
- reliable validation and testing.

This section defines how errors MUST be detected, represented, and handled.

Implementation implication: Any dependency on execution context MUST be explicitly declared as part of the capability description. A conforming implementation MUST NOT vary behaviour based on undeclared contextual factors, and platforms MUST treat behaviour that depends on implicit or hidden context as non-conformant.

1.14.1 Purpose of the Error Handling Model

Errors MUST be handled in a way that is:

- deterministic,
- safe,
- schema-conformant,
- auditable,
- platform-independent.

The primary goals are to:

1. Prevent unsafe or unpredictable behaviour.
2. Provide consistent outputs across all platforms.
3. Ensure capabilities fail cleanly and transparently.
4. Maintain well-defined boundaries between capabilities and agents.

Capabilities MUST NOT attempt to continue execution after encountering an error condition unless explicitly allowed in the behaviour description.

1.14.2 All Error Conditions MUST Be Explicitly Defined

A capability MUST define all error conditions it can encounter, including:

- missing required fields,
- invalid input formats,
- values outside allowed ranges,
- constraint violations,
- domain mismatches,
- safety violations,
- internal logical impossibilities (“cannot produce output under stated assumptions”).

Capabilities MUST NOT rely on implicit or undocumented error conditions.

1.14.3 Two Allowed Error Handling Outcomes

A capability encountering an error MUST produce one of:

✓ 1. Deterministic Error Output

A structure defined in the output schema representing an error.

✓ 2. Deterministic Fallback Output

A safe alternative output that conforms to a secondary schema path.

Capabilities MUST NOT:

- invent custom error messages,
- include free-form text explaining the error,

- output vague or inconsistent responses.

All error outputs MUST follow the output schema exactly.

1.14.4 Error Outputs MUST Be Schema-Bound

Error outputs MUST:

- use an error object defined in the output schema,
- include required error fields (e.g., error_code, error_type, error_context),
- be deterministic and machine-interpretable,
- avoid narrative or natural-language explanations.

Capabilities MUST NOT embed:

- model-generated commentary,
- descriptive disclaimers,
- procedural logs,
- human-style reasoning.

Error outputs MUST be minimal, structured, predictable, and non-generative.

1.14.5 Fallback Outputs MUST Be Safe and Deterministic

Fallback outputs provide a **safe, bounded response** when valid output cannot be produced.

Fallbacks MUST:

- remain within schema boundaries,
- use deterministic content,
- contain no generative or inferred material,
- respect all safety restrictions,
- be consistent across all platforms.

Example valid fallback behaviour:

- returning an empty list,
- returning a null-equivalent structure,
- returning a fixed safe template,
- returning a reduced or partial structure permitted by schema.

Fallbacks MUST NOT:

- attempt creative summarisation,
- modify behaviour outside defined rules,
- partially guess output values.

1.14.6 Ambiguous Inputs MUST Trigger Error or Fallback

Ambiguity MUST NOT be resolved through generative inference.

Ambiguous inputs include:

- unclear or conflicting values,
- missing context required for deterministic interpretation,
- multi-intent content inside text fields,
- ambiguous units or formats not defined in schema.

Agents interpret user intent — capabilities do not.
Ambiguous capability inputs MUST trigger deterministic error/fallback behaviour.

1.14.7 Constraint Violations MUST Trigger Errors

Any violation of constraints defined in:

- input schema,
- behavioural constraints,
- safety restrictions,
- domain limitations,

MUST trigger:

- deterministic error output, OR
- deterministic fallback output.

Capabilities MUST NOT attempt to partially process invalid data.

1.14.8 Handling of Missing or Partial Inputs

If required fields are missing:

- the capability MUST NOT infer content,
- the capability MUST NOT treat the input as valid,
- the capability MUST not attempt best-effort behaviour.

Missing required fields MUST trigger an error or fallback.

Optional fields MUST have documented behaviour whether present or absent.

1.14.9 Capabilities MUST NOT Attempt Self-Correction

Capabilities MUST NOT:

- repair malformed data,
- infer missing values,
- resolve ambiguity independently,
- rewrite or reinterpret user input to fit constraints,
- apply heuristics that alter meaning.

Self-correction leads to non-determinism and cross-platform divergence.

Agents MAY perform data shaping before invocation — but capabilities MUST NOT.

1.14.10 No Generative or Explanatory Error Output

Error outputs MUST NOT contain:

- free-form warnings,
- explanatory paragraphs,
- natural language explanations,
- suggestions for alternative actions,
- reasoning steps.

Capabilities do NOT communicate with users.
Error signals exist for agents, not humans.

1.14.11 Error Handling MUST Be Cross-Platform Stable

Errors MUST produce identical structural representations across:

- different AI vendors,
- different inference models,
- different platforms or runtimes,
- different capability invocation contexts.

Variability would undermine safety and reliability.

1.14.12 Adversarial Error Handling

Capabilities MUST detect adversarial input patterns, including:

- prompt injection attempts,
- obfuscated unsafe content,
- malformed structures intended to bypass validation,
- domain violation attempts.

Adversarial triggers MUST ALWAYS result in:

- deterministic fallback, or
- deterministic error output.

Capabilities MUST NOT attempt to interpret adversarial input creatively.

1.14.13 Summary of Error Handling Guarantees (Normative)

A capability conforms to the Error Handling Model only if:

1. All error conditions are explicitly defined.
2. Errors always produce deterministic fallback or error output.
3. Error outputs strictly follow schema.
4. No generative reasoning or explanation appears.
5. Ambiguous inputs always trigger defined error paths.
6. Constraint violations always trigger error handling.
7. Missing required fields trigger deterministic errors.
8. No self-correction is attempted.
9. Error behaviour is stable across platforms.
10. Adversarial handling is deterministic and safe.

Violation of any requirement constitutes **non-conformance**.

1.15 Cross-Platform Behaviour Consistency

Capabilities must behave identically across all platforms, agent architectures, vendors, runtime environments, and inference models.
This requirement ensures that capabilities retain their meaning, safety constraints, determinism, and functional integrity regardless of the system executing them.

Cross-platform consistency is fundamental to the interoperability model of the BBY Capability Specification (BCS).
It guarantees that:

- developers can create once and deploy everywhere,
- agents can orchestrate capabilities predictably,
- marketplaces can validate uniformly,
- auditors can reason about safety and compliance,
- users can expect consistent behaviour across ecosystems.

Non-consistent behaviour across platforms constitutes **critical non-conformance**.

Implementation implication: Where capabilities are composed, the composition MUST be represented explicitly within the capability description, including data-flow, invocation order, and dependency relationships. Emergent or implicit composition behaviour is not permitted, and conforming platforms MUST execute composed behaviour only as declared.

1.15.1 Purpose of Cross-Platform Consistency

The goals of this section are to ensure that:

- capability behaviour does not depend on vendor-specific interpretation,
- behaviour remains stable across different inference engines,
- safety rules do not vary between systems,
- outputs remain predictable even as AI models evolve,
- capabilities do not fragment into platform-specific variants.

BCS aims to prevent the divergence issues that plagued early web standards, ensuring long-term stability.

1.15.2 Behaviour MUST NOT Depend on Platform Architecture

Capabilities MUST NOT:

- behave differently based on which model executes them,
- adjust outputs based on local system settings,
- vary depending on agent design or reasoning strategy,
- rely on specific tokenisation or embedding behaviours,
- depend on model-specific quirks or undocumented heuristics.

A capability MUST produce the same category, format, and semantics of output across all platforms when provided with identical valid input.

1.15.3 Behaviour MUST NOT Depend on Vendor Extensions

Platforms MAY provide additional features, but capabilities MUST NOT rely on them.

Prohibited dependencies include:

- vendor-specific safety filters,
- extended metadata not present in schema,
- proprietary formatting or interpretation rules,
- model-specific fallback behaviours.

Capabilities MUST remain fully functional and consistent even if executed with:

- a different vendor agent,
- a different LLM,
- a different runtime environment.

1.15.4 Behaviour MUST Be Stable Across Model Updates

Capabilities MUST remain functionally consistent even when:

- models are upgraded,
- underlying inference engines evolve,
- vendor safety pipelines change,
- tokenisation algorithms adjust,
- runtime execution logic is optimised.

BCS compliance requires that capability behaviour does not drift over time.

If platform-level model changes risk breaking a capability's deterministic behaviour, the platform MUST:

- isolate capability execution from such changes, OR
- maintain compatibility layers, OR
- provide version-locked execution.

Capabilities MUST NOT break due to model upgrades.

1.15.5 Output MUST Be Structurally Consistent

The output structure MUST be identical across platforms:

- same fields,
- same layout,
- same ordering rules,
- same formatting rules,
- same presence/absence of optional fields under specified conditions.

Even if different models generate slightly different lexical content (e.g., different wording inside allowed text fields), the **structure** MUST remain consistent and schema-valid.

1.15.6 Semantic Consistency Across Platforms

Semantic consistency requires that:

- categories have the same meaning across implementations,
- extracted entities remain consistently identified,
- classification outputs map to the same conceptual definitions,
- normalised fields (dates, numbers, units) follow identical rules.

Example:
If a capability extracts chemical entities, the definition of "chemical entity" MUST NOT vary between platforms.

Allowing semantic drift would invalidate cross-platform guarantees.

1.15.7 Determinism Across Platforms

Platform differences MUST NOT cause:

- different fallback triggers,
- different error conditions,
- different list ordering,
- different constraint enforcement,
- different safety behaviour.

Determinism (Section 1.4) applies globally to:

- output structure,
- semantic content categories,
- error/fallback behaviour,
- safety enforcement.

Platforms MUST ensure consistent determinism regardless of inner architecture.

1.15.8 Safety Enforcement MUST Be Consistent Across Platforms

Safety behaviour (Section 1.8) MUST NOT vary across:

- vendors,
- AI models,
- inference stacks,
- platform safety pipelines.

If unsafe content is detected:

- every platform MUST trigger the same fallback/error behaviour,
- every platform MUST block prohibited transformations identically,
- every platform MUST prevent unsafe output consistently.

Safety divergence is unacceptable and violates BCS compliance.

1.15.9 Platforms MUST NOT Alter Capability Behaviour

A platform MUST NOT:

- rewrite behaviour descriptions,
- adjust constraints,
- loosen safety boundaries,

- introduce vendor-specific interpretation rules,
- mutate fallback logic,
- alter input or output schemas.

Platforms MAY optimise internal execution, but MUST NOT change:

- what the capability does,
- how constraints are applied,
- how errors are handled,
- how safety triggers respond.

1.15.10 Capability Files MUST Be Interpreted Identically Across Vendors

Given the same capability file:

- OpenAI-compatible agents,
- Google-compatible agents,
- Anthropic-compatible agents,
- local/offline BCS implementations,

…MUST interpret:

- behaviour,
- schemas,
- constraints,
- safety rules,
- error handling,

in the same way.

Differences in file interpretation are considered severe non-conformance events.

1.15.11 Capability Versioning MUST Be Consistent Across Platforms

When a capability is versioned:

- all platforms MUST adopt the same version semantics,
- capabilities MUST behave identically across all compliant runtimes,
- outdated versions MUST be clearly identified and deprecated consistently.

Platforms MUST NOT create vendor-specific forks of capability versions.

1.15.12 Summary of Cross-Platform Guarantees (Normative)

A capability is compliant with Cross-Platform Behaviour Consistency only if:

1. Behaviour is independent of platform architecture.
2. No vendor extensions are required for execution.
3. Model upgrades do not change behaviour.
4. Output structure remains identical across platforms.
5. Semantic content remains consistent.
6. Determinism is preserved across all environments.
7. Safety enforcement is absolute and identical everywhere.
8. Platforms do not alter or reinterpret capability behaviour.
9. Capability files are interpreted consistently across vendors.
10. Capability versioning is unified across platforms.

Violation of any requirement results in **critical non-conformance**.

2.0 File Structure Specification

This section defines the **formal structure**, **syntax**, and **required components** of a BCS-compliant Capability file.
While Section 1 established the conceptual and behavioural model of capabilities, **Section 2 defines how those concepts MUST be encoded in a machine-readable format.**

The File Structure Specification ensures that:

- capabilities are **portable** across platforms,
- capability behaviour is **deterministic**,
- agents can **understand and execute** capabilities consistently,
- validation systems can **verify conformance**,
- vendors can implement **interoperable runtimes**,
- developers have a clear, predictable method of defining capability behaviours.

This section is **normative**, meaning all rules defined here MUST be followed for a capability to be considered compliant.

The full capability definition is encoded in a structured document (typically JSON or JSON-equivalent), composed of multiple **blocks**:

- **Metadata Block**
- **Behaviour Block**
- **Input Schema Block**
- **Output Schema Block**
- **Constraints Block**
- **Safety Block**
- **Extensions Block (optional)**

Each block has a clearly defined purpose, and platforms MUST interpret them consistently.

This section defines:

- the ordering rules,
- required and optional fields,
- schema constraints,
- serialisation guidelines,
- naming conventions,
- versioning and compatibility behaviour,
- reserved field names,
- interpretation rules for all fields.

All capability files MUST conform to this structure regardless of:

- platform,
- runtime,
- agent implementation,
- vendor,
- programming language.

2.1 Structural Overview

A capability file is a **single, self-contained definition** that declares:

- what the capability does,
- what inputs it accepts,
- what outputs it produces,
- what constraints govern behaviour,
- what safety boundaries restrict execution,
- which version of the BCS standard it conforms to.

The canonical structure is:

```
{
  "bcs_version": "1.0",
  "metadata": { ... },
  "behaviour": { ... },
  "input_schema": { ... },
  "output_schema": { ... },
  "constraints": { ... },
  "safety": { ... },
  "extensions": { ... }  // optional
}
```

2.2 Normalisation & Mandatory Structure Rules

For interoperability across implementations, the structure defined in this section is considered normative. A conforming capability file MUST contain all mandatory top-level blocks defined in Section 2, and those blocks MUST appear in the canonical order specified by this standard. Optional blocks MAY be included where supported, but their presence MUST NOT alter the semantics or interpretation of the mandatory blocks.

Where a platform performs internal re-ordering, normalisation, or serialisation of a capability file, the resulting representation MUST remain structurally equivalent to the canonical form defined in this specification.

2.2.1 Mandatory Blocks

The following blocks MUST appear in every capability file:

- bcs_version
- metadata
- behaviour
- input_schema
- output_schema
- constraints
- safety

2.2.2 Optional Blocks

The following block is optional:

- extensions

If present, it MUST follow all required blocks and MUST conform to extension rules defined in Section 2.9.

2.2.3 Ordering Requirements

Blocks MUST appear in the following order:

1. bcs_version
2. metadata
3. behaviour
4. input_schema
5. output_schema
6. constraints
7. safety
8. extensions (optional)

Order is normative.
Platforms MUST reject any capability file with reordered blocks.

2.2.4 Block Independence

Each block MUST:

- be internally self-consistent,
- not override or modify another block,
- not contain implied behaviour not present in the Behaviour Block,
- map directly to validation rules in Section 4.

2.2.5 Deterministic Interpretation

Platforms MUST interpret:

- field names,
- field order,
- schemas,
- constraints,
- safety rules

identically across vendors.

This prevents drift and ensures universal interoperability.

2.2.6 Formal Serialisation Format

All capability files MUST:

- use UTF-8 encoding,
- use JSON or JSON-equivalent syntax,
- avoid comments,
- avoid trailing commas,
- avoid non-standard JSON extensions.

Later sections define the precise serialisation requirements.

2.2.7 Reserved Field Names

Certain field names are reserved for BCS use only (listed in Section 2.8).
Capabilities MUST NOT redefine or repurpose reserved fields.

2.2.8 Platform Flexibility

Platforms MAY:

- compress capability files,
- store them in alternative containers,
- apply digital signatures,
- distribute them via stores or registries.

But platforms MUST NOT alter the structure or meaning of capability files.

2.3 Metadata Block Specification

The **Metadata Block** provides all identifying, descriptive, and administrative information necessary for a capability to be:

- uniquely identified,
- versioned,
- categorised,
- indexed by capability registries,
- validated by automated systems,
- surfaced in marketplace listings,
- referenced by agents and developers.

Metadata does **not** affect capability behaviour.
Metadata MUST NOT introduce behavioural rules, functional logic, or constraints.

The Metadata Block is normative:
A capability file lacking valid metadata is **non-compliant**.

2.3.1 Behaviour Definition Conformance Rules

The behaviour block defines the normative description of how the capability operates. All behaviour that may be executed by a conforming platform MUST be represented explicitly within this block. Behaviour MUST NOT rely on implicit interpretation, model-specific inference, or undeclared side-effects.

A conforming platform MUST execute behaviour only as declared in the behaviour block and MUST treat missing, ambiguous, or underspecified behaviour as a validation failure rather than attempting to infer or synthesise behaviour at runtime.

2.3.2 Metadata Conformance Rules

The metadata block is a normative component of the capability file. A conforming capability description MUST include all metadata fields marked as mandatory in this specification. Optional metadata fields MAY be included, but their omission MUST NOT affect the semantic behaviour of the capability.

Platforms MAY include additional vendor-specific metadata fields, provided that such fields do not change the meaning, interpretation, or execution semantics of any mandatory field defined in this specification. Unknown metadata fields MUST be ignored by conforming platforms rather than interpreted heuristically.

2.3.3 Purpose of the Metadata Block

The Metadata Block exists to ensure:

- **Global uniqueness** of capability identifiers
- **Traceability** across updates and versions
- **Searchability** in capability registries
- **Developer attribution and accountability**
- **User comprehension** (via descriptions)
- **Cross-platform consistency**

Metadata provides **context**, not function.
Any field that alters behaviour MUST appear in another block.

2.3.4 Required Structure

The Metadata Block MUST be structured as follows:

```
"metadata": {
  "id": "string",
  "name": "string",
  "version": "string",
  "summary": "string",
  "description": "string",
  "developer": {
    "name": "string",
    "contact": "string"
  },
  "created": "ISO8601 datetime",
  "modified": "ISO8601 datetime",
  "tags": ["string", ...]
}
```

Fields may not be reordered.
All required fields MUST be present unless explicitly stated otherwise.

2.3.5 id (Required)

A globally unique identifier for the capability.

Rules:

- MUST be unique across all capability registries
- MUST be immutable once assigned
- MUST use lowercase alphanumeric characters with hyphens
- MUST NOT contain spaces
- MUST NOT change between versions

- SHOULD follow the format:
  publisher.capability-name

Examples:

- "id": "bby.text-sentiment-analyzer"
- "id": "acme.medical-dosage-validator"

This ID is the canonical reference used by platforms and agents.

2.3.6 name (Required)

A human-readable name for the capability.

Rules:

- MUST be concise
- SHOULD use title case
- MAY contain spaces
- SHOULD reflect the capability's functional purpose

Example:
"name": "Document Topic Classifier"

2.3.7 version (Required)

A semantic version representing the capability file's revision.

Rules:

- MUST follow MAJOR.MINOR.PATCH format
- MUST increment for any change to behaviour, schema, constraints, or safety rules
- MUST NOT reuse previous version numbers
- MUST NOT imply backward compatibility unless explicitly stated

Examples:

- "version": "1.0.0"
- "version": "1.2.3"

Versioning is detailed further in Section 5.

2.3.8 summary (Required)

A one-sentence description of the capability's purpose.

Rules:

- MUST NOT exceed 200 characters
- MUST be plain text
- MUST NOT include behavioral details, instructions, or examples
- MUST NOT include marketing language

Example:
"summary": "Extracts structured entities from unformatted text."

2.3.9 description (Required)

A human-readable, multi-line description providing a deeper explanation of:

- intended use cases
- operational context
- limitations
- non-functional characteristics

Rules:

- MAY include multiple sentences
- MUST NOT contradict Behaviour, Schema, or Safety blocks
- MUST NOT contain examples that violate safety rules
- MUST describe capability intent, not behaviour logic

Example:
"description": "This capability extracts chemical names from text and normalises them into structured output fields. It is designed for document analysis workflows where deterministic chemical entity extraction is required."

2.3.10 developer (Required)

Information about the capability's developer or maintaining organisation.

Structure:

```
"developer": {
  "name": "string",
  "contact": "string"
}
```

Rules:

- name MUST be the legal or recognised name of the developer

- contact MUST be a valid email or URL
- Contact MUST be stable for long-term support
- Platforms MAY use this to facilitate bug reporting or security notices

Example:

```
"developer": {
  "name": "Big Blue Yonder",
  "contact": "https://bigblueyonder.com/bcs-contact"
}
```

2.3.11 created (Required)

An ISO 8601 timestamp indicating when the first version of the capability was created.

Rules:

- MUST NOT change between versions
- MUST be in UTC
- MUST use full date-time representation

Example:
"created": "2025-12-09T14:22:51Z"

2.3.12 modified (Required)

An ISO 8601 timestamp representing the publication date of the current version.

Rules:

- MUST update on every version increment
- MUST be in UTC
- MUST reflect the exact moment of version release

Example:
"modified": "2026-01-04T09:15:00Z"

2.3.13 tags (Optional but Recommended)

A list of short labels used for categorisation and search.

Rules:

- Each tag MUST be lowercase

- MUST contain only alphanumeric characters or hyphens
- MUST NOT redefine behaviour or constraints
- SHOULD reflect functional categories

Example:

```
"tags": [
  "text-processing",
  "classification",
  "nlp"
]
```

2.3.14 Metadata Block Validation Rules

A Metadata Block is valid only if:

1. All required fields are present.
2. All timestamps use correct ISO-8601 formatting.
3. The id is globally unique and immutable.
4. Semantic versioning is correct.
5. All strings are UTF-8 encoded.
6. No metadata field introduces behaviour.
7. The block appears in the correct position in the capability file.

Non-compliance in any rule invalidates the entire capability definition.

2.3.15 Metadata Block Non-Normative Guidance

(Not required for conformance, but recommended.)

- summary should be written for developers scanning lists of capabilities.
- description should be written for technical reviewers.
- Tags allow marketplaces to group similar capabilities.
- Developers should avoid marketing-speak in metadata.
- Metadata should be kept stable over time, except for version and timestamps.

2.4 Behaviour Block Specification

The **Behaviour Block** defines the functional behaviour of a capability.
It specifies *what the capability does*, *how it processes valid input*, and *what guarantees it provides*.

Unlike generative large language models, a capability's behaviour MUST be:

- deterministic,
- explicit,
- constrained,

- rule-based,
- reproducible,
- interpretable,
- stable across platforms.

The Behaviour Block contains **the only authoritative behavioural description** of the capability.
All other blocks (schema, constraints, safety) exist to support or limit behaviour — but they MUST NOT define behaviour.

2.4.1 Purpose of the Behaviour Block

The Behaviour Block is the definitive specification for:

- execution rules
- transformation logic
- success conditions
- behaviour boundaries
- fallback behaviour
- error triggers
- deterministic rules

It provides platforms and validation systems with enough detail to:

- confirm that behaviour is deterministic,
- verify that outputs are schema-compliant,
- verify that behaviour aligns with constraints and safety rules,
- ensure cross-platform consistency.

Whatever is written here is the behaviour.
Anything absent here MUST NOT be inferred or assumed.

2.4.2 Required Structure

The Behaviour Block MUST use the following structure:

```
"behaviour": {
  "description": "string",
  "inputs": { ... },
  "transformation": { ... },
  "outputs": { ... },
  "determinism": { ... },
  "fallbacks": { ... },
  "error_conditions": { ... }
```

}

All fields except fallbacks are **required**.

2.4.3 Behaviour Fields — Normative Definitions

Each field has a precise purpose and MUST comply with the following rules:

2.4.4 description (Required)

A high-level description of the capability's behaviour.

Rules:

- MUST use declarative language (not procedural)
- MUST NOT describe internal implementation
- MUST NOT describe input/output formats (handled in schemas)
- MUST NOT contain examples that contradict constraints or safety rules
- MUST describe *what* the capability does, not *how* internally

Example:

"description": "Determines whether a text input expresses positive, neutral, or negative sentiment."

2.4.5 inputs (Required)

A structured description of how the capability interprets valid input.

Rules:

- MUST reference fields defined in input_schema
- MUST describe semantic meaning of inputs
- MUST NOT introduce additional fields not present in the schema
- MUST NOT describe types (type rules are in schema)
- MUST describe how multiple input fields relate
- MUST define assumptions the behaviour makes about input values

Example:

"inputs": {

"text": "The natural-language text to be analysed for sentiment."

}

2.4.6 transformation (Required)

This is the **core behaviour definition** of the capability.

It defines exactly:

- what transformation occurs,
- how input maps to output,
- the rules governing behaviour,
- the conditions for successful execution,
- the restrictions that enforce determinism.

Rules:

- MUST be deterministic
- MUST NOT contain generative behaviour
- MUST NOT rely on external knowledge not encoded in the input
- MUST provide a complete behavioural description
- MUST NOT include ambiguous or subjective language
- MUST NOT include examples

Example (abbreviated):

"transformation": {

"rule": "The capability evaluates the sentiment of the provided text and maps it to one of the output labels: positive, neutral, or negative. Sentiment is determined by evaluating the polarity of phrases using predefined lexical rules. No inference beyond lexical polarity is permitted."

}

2.4.7 outputs (Required)

Defines how the capability constructs valid output from the transformation.

Rules:

- MUST reference the fields defined in output_schema
- MUST describe what each field represents
- MUST describe when each optional field appears
- MUST NOT introduce fields not present in the schema
- MUST not redefine schema type constraints

Example:

"outputs": {

"label": "The sentiment category assigned to the input text.",

"confidence": "A deterministic confidence score representing rule-derived certainty."

}

2.4.8 determinism (Required)

Defines the rules and guarantees that ensure behaviour is identical across platforms.

Rules:

- MUST explicitly state deterministic ordering rules

- MUST specify handling of ties, edge cases, and ambiguous cases
- MUST specify rule precedence
- MUST define how repeated inputs produce identical outputs
- MUST restrict behaviour to rule-based logic

Example:

"determinism": {

"ordering": "If multiple sentiment rules apply, the rule with highest precedence is selected according to the transformation precedence table.",

"repeatability": "Given identical input, the output MUST remain identical across all platforms."

}

2.4.9 fallbacks (Optional)

Defines safe outputs used when transformation cannot produce a valid result without triggering an error.

Rules:

- MUST reference output schema paths that are explicitly allowed
- MUST NOT introduce generative or improvised fallback responses
- MUST define deterministic fallback conditions

Example:

"fallbacks": {

"empty": "Returned when the input text contains no analyzable content."

}

2.4.10 error_conditions (Required)

Defines all conditions under which the capability MUST produce an error output (see Error Handling Model, Section 1.9).

Rules:

- MUST be exhaustive
- MUST not rely on subjective judgement
- MUST NOT include vendor-specific behaviours
- MUST map 1–1 with defined error outputs
- MUST include constraints violations, safety violations, type mismatches, missing fields, and ambiguous input states

Example:

"error_conditions": {

"missing_text": "Triggered when `text` field is absent.",

"invalid_format": "Triggered when the input text contains unsupported characters or exceeds defined length limits."

}

2.4.11 Behaviour Block Validation Rules

A Behaviour Block is valid only if:

1. All required fields are present.
2. Fields only reference schema-defined structures.
3. No field introduces undocumented behavioural logic.
4. All behaviour is deterministic and rule-based.
5. No generative behaviour is implied.
6. Error conditions cover all failure scenarios.
7. No behaviour contradicts constraints or safety rules.
8. Behaviour does not rely on platform-specific features.

Any violation results in **non-conformance**.

2.4.12 Non-Normative Guidance

(Not required for compliance, but strongly recommended.)

- Behaviour descriptions should be concise but complete.
- Developers should avoid algorithmic details and focus on behavioural rules.
- Determinism guarantees must be explicit to avoid drift across platforms.
- Ambiguity in behaviour MUST be avoided—agents require clarity.
- Use simple, declarative sentences to reduce interpretation errors.

2.5 Input Schema Specification

The **Input Schema Block** defines the complete, machine-interpretable structure of all inputs accepted by a capability.
It is the authoritative source for:

- field definitions
- data types
- required vs optional fields
- enumerations
- allowed values
- structural constraints
- formatting requirements
- input validity rules

All inputs processed by the Behaviour Block MUST conform to the Input Schema.
Any deviation MUST result in a deterministic error (see Section 1.9).

The Input Schema DOES NOT define behaviour — only **structure and constraints**.

2.5.1 Input Schema Conformance Rules

The input schema block defines the complete and authoritative description of the inputs that a capability may consume. All inputs that influence behaviour MUST be represented in this schema. A conforming platform MUST reject any capability description that references or depends upon inputs which are not declared in the input schema.

Platforms MAY provide additional runtime inputs for operational purposes (such as logging, tracing, or execution context), but such inputs MUST NOT influence behavioural outcomes unless they are explicitly declared within the input schema.

2.5.2 Purpose of the Input Schema Block

The Input Schema exists to:

- enforce deterministic input expectations,
- prevent undefined or ambiguous inputs,
- allow platforms to validate inputs before execution,
- ensure cross-platform consistency in input interpretation,
- prevent behaviour from relying on implicit assumptions,
- enable safe, automated input validation at marketplaces and runtimes.

Platforms MUST validate input against the Input Schema BEFORE invoking behaviour.

2.5.3 Required Structure

The Input Schema Block MUST use the following structure:

```
"input_schema": {
  "type": "object",
  "required": [...],
  "properties": {
     "field_name": {
       "type": "...",
       ...additional constraints...
     },
     ...
  }
}
```

The schema MUST conform to a JSON Schema–compatible format **with BCS extensions** defined later in Section 2.8.

2.5.4 Input Schema Rules (Normative)

The following rules MUST be followed to ensure determinism and interoperability.

2.5.5 All Input Fields MUST Be Explicitly Declared

- No implicit fields are allowed.
- Behaviour MUST NOT reference fields absent from the schema.
- Schema-defined fields MUST be stable across versions unless versioning explicitly states otherwise.

2.5.6 type MUST Be "object"

Every Input Schema MUST be a JSON object.

"type": "object" is mandatory.

Primitive-only inputs (e.g., plain strings) are **not allowed**; they MUST be wrapped in an object.

Example (valid):

```
"input_schema": {
  "type": "object",
  "required": ["text"],
  "properties": {
    "text": { "type": "string" }
  }
}
```

2.5.7 Required Fields MUST Be Listed in required

- Any field not listed in required is optional.
- Behaviour MUST NOT assume the presence of optional fields.
- Optional fields MUST have deterministic presence/absence rules described in the Behaviour Block.

Example:

```
"required": ["text"]
```

2.5.8 properties MUST Define Every Accepted Field

Each field MUST define:

- "type"
- optional "enum"

- optional "pattern"
- optional "format"
- optional "minimum" / "maximum"
- optional "minLength" / "maxLength"
- optional "items" for arrays
- optional "properties" for nested objects

Fields MUST NOT include:

- behaviour
- generative instructions
- transformations

The schema must ONLY define structure.

2.5.9 Allowed Primitive Types

The following JSON Schema types are allowed:

- "string"
- "number"
- "integer"
- "boolean"
- "object"
- "array"
- "null" (only when explicitly permitted)

2.5.10 Arrays MUST Define items Structure

Array fields MUST declare:

- the type of elements
- any constraints on length
- ordering rules if applicable

Example:

```
"tokens": {
  "type": "array",
  "items": { "type": "string" },
  "maxItems": 500
}
```

2.5.11 Enumerations MUST Be Closed

Enumerations MUST define an explicit list of allowed values.

Example:

```
"sentiment": {
  "type": "string",
  "enum": ["positive", "neutral", "negative"]
}
```

No values outside the enumeration are permitted.

2.5.12 Patterns MUST Use Full Regular Expressions

If "pattern" is used, it MUST:

- define a full regex
- be unambiguous
- be consistent across platforms

Example:

```
"pattern": "^[a-zA-Z0-9 _.-]+$"
```

2.5.13 Nested Objects MUST Have Fully Declared Schemas

A nested object MUST declare its structure — no partial schemas allowed.

Example:

```
"metadata": {
  "type": "object",
  "required": ["author"],
  "properties": {
    "author": { "type": "string" }
  }
}
```

2.5.14 Input Schema MUST NOT Encode Behaviour

The schema MUST NOT include:

- instructions
- logic
- procedural semantics
- conditional processing
- fallback definitions
- safety rules
- transformations

All of that belongs in Behaviour, Constraints, or Safety.

2.5.15 Input Validation Requirements

Platforms MUST validate inputs using:

- the schema in this block,
- the constraints in Section 2.6,
- safety rules in Section 2.7.

If any input fails validation:

➡️ Behaviour MUST NOT be executed.
➡️ A deterministic error MUST be returned.

2.5.16 Determinism Requirements for Input Schemas

To ensure consistent behaviour across platforms:

- No schema-defined field may have ambiguous meaning.
- Optional fields must not change meaning depending on presence/absence.
- Default values MUST NOT be implicitly assumed (platforms may NOT auto-fill inputs).
- Schema rules MUST NOT reference external documents.
- Human-readable descriptions MUST NOT alter technical meaning.

2.5.17 Input Schema Example (Non-Normative)

```
"input_schema": {
  "type": "object",
  "required": ["text"],
  "properties": {
    "text": {
      "type": "string",
      "minLength": 1,
      "maxLength": 2000
    },
    "language": {
      "type": "string",
      "enum": ["en", "fr", "de", "es"],
      "default": "en"
    }
  }
}
```

2.5.18 Summary of Input Schema Normative Requirements

A valid Input Schema:

- MUST define explicit structured inputs
- MUST use JSON Schema–compatible constructs
- MUST include required and optional fields
- MUST enforce strict types and constraints
- MUST not encode behaviour
- MUST be validated before execution
- MUST be deterministic across all platforms

2.6 Output Schema Specification

The Output Schema Block defines the complete, machine-interpretable structure of all valid outputs produced by a capability.
It is the authoritative contract between the capability and the platform, ensuring that:

- outputs are deterministic,
- outputs are fully structured,
- no generative text appears unless strictly permitted,
- all values conform to expected types and formats,
- the behaviour produces output aligned with schema rules,
- validation systems can confirm compliance consistently across platforms.

The Output Schema MUST fully describe every possible valid output, including:

- primary outputs,
- alternative outputs,
- fallback structures,
- error outputs (where applicable),
- optional fields and their conditions.

Platforms MUST reject outputs that do not comply with the schema.

2.6.1 Output Schema Conformance Rules

The output schema block defines the complete and authoritative description of the outputs that a capability may produce. All observable results generated by a conforming implementation MUST conform to this schema. A capability MUST NOT emit additional, undeclared outputs, and platforms MUST treat output that falls outside the declared schema as a validation failure rather than an acceptable variation in behaviour.

Where a platform performs internal transformation or post-processing of output for operational purposes, the externally visible result MUST remain semantically equivalent to the output defined in the declared schema.

2.6.2 Purpose of the Output Schema Block

The Output Schema exists to:

- define the exact structure of capability outputs,
- eliminate ambiguity for agents processing results,
- ensure cross-platform consistency,
- support deterministic rendering and post-processing,
- prevent hallucinated or unstructured output,
- enable strong validation for compliance.

Outputs MUST be fully predictable from the schema and behaviour definition.

2.6.3 Required Structure

The Output Schema Block MUST use the following structure:

```
"output_schema": {
  "type": "object",
  "required": [...],
  "properties": {
     "field_name": {
       "type": "...",
       ...additional constraints...
     },
     ...
  },
  "oneOf": [ ... ]    // optional
}
```

The schema MUST conform to JSON Schema–compatible rules with BCS-specific extensions defined in Section 2.8.

2.6.4 Output Schema Rules (Normative)

2.6.5 All Outputs MUST Be Described Explicitly

Every possible output MUST be fully defined in the schema.

- No implicit fields
- No dynamic fields
- No unconstrained objects
- No free-form generative text unless explicitly permitted

Anything not defined in the schema is invalid.

2.6.6 type MUST Be "object"

As with input schemas:

- primitive-only outputs are disallowed,
- all outputs MUST be wrapped in a JSON object.

Example:

```
"output_schema": {
  "type": "object",
  "required": ["label"],
  "properties": {
    "label": { "type": "string" }
  }
}
```

2.6.7 Required and Optional Fields MUST Be Declared

- All required fields MUST appear in ALL valid outputs.
- Optional fields MUST NOT appear unless conditions in the Behaviour Block allow them.

This rule ensures deterministic field presence.

2.6.8 Enumerations MUST Be Closed

Enumerated fields MUST clearly define all allowed values.

Example:

```
"label": {
  "type": "string",
  "enum": ["positive", "neutral", "negative"]
}
```

2.6.9 Arrays MUST Define Their Structure

Same rules as input schemas:

- items MUST define the type
- maxItems and minItems SHOULD be included where deterministic
- Order MUST be deterministic (e.g., sorted lex order)

2.6.10 Nested Objects MUST Have Fully Declared Schemas

Nested output structures MUST:

- define all nested fields,
- include type rules, required lists, etc.,
- not allow arbitrary additional properties.

2.6.11 Use of additionalProperties

The following rules apply:

- additionalProperties MUST be set to false unless explicitly needed.
- If allowed, its type MUST be strictly defined.
- Arbitrary open-ended objects are prohibited.

Example (valid only if needed):

"additionalProperties": false

2.6.12 Output Schema MUST NOT Define Behaviour

The schema defines structure only.

It MUST NOT:

- describe logic or transformation rules,
- describe how fallback occurs,
- describe when error outputs appear,
- describe ordering logic,
- describe preprocessing rules.

All such details belong in the Behaviour, Constraints, or Safety blocks.

2.6.13 Error Output Requirements

If the capability defines error outputs:

- The error format MUST be included in the Output Schema.
- It MUST be deterministic.
- It MUST match error conditions defined in the Behaviour Block.
- It MUST use structured fields (e.g., error_code, error_type).
- It MUST NOT contain generative natural language explanations.

Example:

```
"errors": {
  "type": "object",
  "required": ["error_code"],
  "properties": {
```

```
    "error_code": { "type": "string" },
    "error_context": { "type": "string" }
  }
}
```

The platform MUST NOT alter error formats.

2.6.14 Handling Multiple Potential Output Formats

If the capability may produce multiple valid output structures (e.g., standard vs fallback vs error), the schema MUST use oneOf:

Example:

```
"output_schema": {
  "oneOf": [
    { "$ref": "#/definitions/standard_output" },
    { "$ref": "#/definitions/fallback_output" },
    { "$ref": "#/definitions/error_output" }
  ]
}
```

Rules:

- oneOf MUST be exhaustive
- Structures MUST not overlap ambiguously
- Behaviour MUST define when each branch occurs

2.6.15 Determinism Requirements for Output

Outputs MUST be:

- stable across platforms
- derived solely from inputs + rules
- not influenced by runtime heuristics
- not generative in nature
- not influenced by vendor-specific model differences

Output order:

- Arrays MUST follow deterministic order (alphabetical, numeric, documented)
- Object properties SHOULD follow a canonical order (though JSON is unordered, canonical order improves reproducibility)

2.6.16 Example Output Schema (Non-Normative)

```
"output_schema": {
  "type": "object",
  "required": ["label", "confidence"],
  "properties": {
    "label": {
      "type": "string",
      "enum": ["positive", "neutral", "negative"]
    },
    "confidence": {
      "type": "number",
      "minimum": 0.0,
      "maximum": 1.0
    }
  }
}
```

2.6.17 Summary of Normative Output Schema Requirements

A valid Output Schema:

- MUST define all valid outputs
- MUST use deterministic types and constraints
- MUST prohibit unstructured generative output
- MUST include error and fallback structures if applicable
- MUST be exhaustive and unambiguous
- MUST map directly to Behaviour Block definitions
- MUST be stable across platforms

2.7 Constraints Block Specification

The Constraints Block defines all **non-behavioural**, **non-schema**, and **non-safety** limitations that govern valid execution of a capability.
Where the Input and Output Schemas define *structure*, and the Behaviour Block defines *function*, the Constraints Block defines the **static rules** that bind or restrict how behaviour may operate.

Constraints ensure that:

- inputs fall within valid operational boundaries,
- behaviour cannot operate outside intended parameters,
- platforms can perform pre-execution validation,
- outputs reflect rule-bounded logic rather than heuristic inference,
- deterministic behaviour is preserved under all valid conditions.

Constraints MUST be enforced *before* behaviour is executed.

2.7.1 Constraints & Assumptions Conformance Rules

The constraints and assumptions block defines the declared operating boundaries for the capability. All environmental assumptions and execution constraints that may affect behavioural outcomes MUST be represented explicitly in this block. A conforming platform MUST treat execution outside these declared assumptions and constraints as non-conformant behaviour.

Platforms MUST NOT infer, substitute, or silently supply undeclared assumptions at runtime. Where the declared assumptions cannot be satisfied, the capability MUST fail validation rather than executing in an undefined or partial state.

2.7.2 Purpose of the Constraints Block

The Constraints Block exists to:

- define limits on values, ranges, sizes, formats, or relationships between fields,
- ensure behaviour executes only within defined valid domains,
- prevent ambiguous or undefined execution paths,
- prevent behaviour from processing pathological or adversarial inputs,
- guarantee deterministic failure when constraints are violated.

Constraints DO NOT define behaviour.
They define **boundaries** within which behaviour is allowed to operate.

2.7.3 Required Structure

The Constraints Block MUST use the following structure:

```
"constraints": {
  "value_constraints": { ... },
  "structural_constraints": { ... },
  "relational_constraints": { ... },
  "domain_constraints": { ... }
}
```

All sections MUST be present, even if empty.

Example of an empty section:

```
"domain_constraints": {}
```

This ensures structural consistency across capability files.

2.7.4 Constraint Categories (Normative)

Constraints MUST be grouped into the following categories.

2.7.5 Value Constraints

Rules limiting the permissible values of fields defined in the Input Schema.

Examples:

- "minimum" / "maximum"
- "minLength" / "maxLength"
- "pattern"
- fixed ranges (e.g. 0–100)
- allowed precision (e.g. 2 decimal places)

These MUST duplicate or refine schema-level constraints; they MUST NOT contradict them.

Purpose:
Reinforce deterministic boundaries.

2.7.6 Structural Constraints

Rules defining additional structural expectations beyond the raw schema.

Examples:

- maximum number of array items
- required ordering of array elements
- mutually exclusive fields (e.g., field A XOR field B)
- disallowed nested patterns
- required object substructures under certain conditions

These constraints restrict the *shape* of data allowed for execution.

2.7.7 Relational Constraints

Rules defining relationships between multiple input fields.

Examples:

- "start_time" < "end_time"
- "value" must equal "quantity" * "unit_price"
- "country" must be compatible with "language"
- "option" must not appear unless "flag" is set to true"

Purpose:

- enforce deterministic relationships
- prevent ambiguous input combinations
- assert logical consistency across fields

Relational constraints MUST be deterministic and unambiguous.

2.7.8 Domain Constraints

Rules defining the **conceptual domain** within which the capability is allowed to operate.

Examples:

- maximum input size appropriate for deterministic processing
- permitted categories of content
- restrictions on numeric domain (e.g., non-negative, bounded sets)
- semantic limits (e.g., capability only handles Latin-script languages)

Domain constraints MUST NOT introduce safety rules; safety is defined separately.

2.7.9 Constraint Enforcement Requirements

Platforms MUST enforce all constraints before invoking behaviour.

If ANY constraint is violated:

1. **Behaviour MUST NOT be executed.**
2. A deterministic **error output** MUST be returned.
3. The error MUST map to a defined error_condition in the Behaviour Block.

Constraints MUST be evaluated **after** schema validation and **before** safety evaluation.

2.7.10 Determinism Requirements for Constraints

Constraints MUST be:

- explicit
- rule-based
- stable across platforms
- not dependent on external data sources
- not dependent on model interpretation
- not influenced by platform heuristics

All constraints MUST produce one of two outcomes:

- **valid** — platform proceeds to safety evaluation
- **invalid** — platform returns a deterministic error

There MUST be no third state.

2.7.11 Restrictions on Constraint Usage

Constraints MUST NOT:

- depend on runtime model inference,
- depend on agent interpretation,
- include probabilistic or generative rules,
- contradict schema definitions,
- contradict behaviour definitions,
- include natural-language instructions,
- apply safety restrictions—those belong in the Safety Block,
- introduce new behavioural logic or branching.

Constraints MUST be static and machine-verifiable.

2.7.12 Constraint Examples (Non-Normative)

Example simple constraint block:

```
"constraints": {
  "value_constraints": {
    "text.maxLength": 2000
  },
  "structural_constraints": {
    "tokens.maxItems": 500
  },
  "relational_constraints": {
    "start_time_before_end_time": {
      "field_a": "start_time",
      "field_b": "end_time",
      "rule": "a < b"
    }
  },
  "domain_constraints": {
    "languages": ["en", "fr", "de", "es"]
  }
}
```

2.7.13 Invalid Constraint Patterns

A constraint is invalid if it:

- implies behaviour (“if X then perform Y”)
- requires generative interpretation
- uses subjective language (“reasonable”, “likely”, “generally”)
- requires external context (“check if user previously entered…”)
- depends on agent-level logic

- approximates content instead of evaluating structure
- contradicts the metadata or behaviour block

Invalid constraints MUST be rejected during validation.

2.7.14 Summary of Normative Requirements

A valid Constraints Block:

- MUST include all four constraint categories
- MUST contain only static, deterministic rules
- MUST be evaluated before behaviour
- MUST cause deterministic failure if violated
- MUST NOT conflict with schema, behaviour, or safety definitions
- MUST NOT include generative or subjective logic
- MUST ensure cross-platform interoperability

Violation of any requirement results in a non-compliant capability.

2.8 Safety Block Specification

The Safety Block defines all safety-related restrictions, prohibitions, and boundaries that govern a capability's valid execution domain.
Where Section 1.8 defined the *behavioural rules* for safety interaction, **Section 2.7 defines the structural, machine-readable rules** that platforms MUST enforce.

The Safety Block is authoritative for:

- prohibited inputs
- prohibited outputs
- category restrictions
- risk levels
- allowed domains
- explicit safety boundaries
- deterministic safety triggers
- fallback/error mapping
- content classification rules

The Safety Block MUST NOT contain any behaviour or transformation logic; it defines *restrictions*, not execution.

2.8.1 Safety Interaction Conformance Rules

The safety interaction block defines all safety-relevant behaviours, side-effects, and external interactions associated with the capability. A capability MUST explicitly declare any operation that may modify external systems, access protected resources, or produce effects beyond its declared outputs.

Conforming platforms MUST enforce the declared safety boundaries during execution, and MUST treat undeclared safety-relevant behaviour as non-conformant. Platforms MUST NOT introduce additional side-effects or safety-relevant operations beyond those declared in this block, even where such behaviour may appear operationally convenient.

2.8.2 Purpose of the Safety Block

The Safety Block provides:

1. A normative, structured declaration of safety limitations.
2. A mechanism for platforms to enforce safety before and after execution.
3. A deterministic mapping between violations and error states.
4. A stable, machine-interpretable safety model.
5. A guarantee that unsafe behaviours are blocked consistently across vendors.

Safety must be **explicit**, **bounded**, and **non-negotiable**.
Implicit safety assumptions are prohibited.

2.8.3 Required Structure

The Safety Block MUST follow the structure:

```
"safety": {
  "risk_level": "...",
  "prohibited_inputs": { ... },
  "prohibited_outputs": { ... },
  "content_restrictions": { ... },
  "domain_restrictions": { ... },
  "safety_triggers": { ... }
}
```

All fields are required.

Empty structures are allowed only when explicitly meaningful, e.g.:

```
"prohibited_outputs": {}
```

2.8.4 Risk Level (Normative)

2.8.5 risk_level (Required)

Defines the overall safety classification of the capability.

Allowed values (closed set):

- "low"
- "medium"
- "high"

Meaning:

- "low": Minimal risk. Purely structural, deterministic tasks with no sensitive content.
- "medium": Moderate risk tasks involving general text processing that could include potentially harmful content.
- "high": Tasks that require elevated scrutiny due to domain sensitivity (e.g., medical, legal, chemical, financial).

This is NOT a behavioural instruction — it is a classification used for:

- marketplace visibility
- elevated auditing
- platform-level enforcement
- risk-based routing

Platforms MUST NOT adjust this value dynamically.

2.8.6 Prohibited Inputs

2.8.7 prohibited_inputs (Required)

Structured list of categories, patterns, or types of content that MUST NOT be processed.

Examples include:

- sexual content
- extremist content
- hate content
- medical advice content
- illegal drug synthesis
- violent content
- personally identifiable information (PII)
- unsafe numerical ranges
- non-supported languages

Each prohibited input rule MUST be structured:

```
"prohibited_inputs": {
  "content_categories": ["categoryA", "categoryB"],
  "patterns": ["regex1", "regex2"],
  "value_ranges": {
    "field_name": { "min": ..., "max": ... }
  }
```

}

Rules:

- MUST be deterministic
- MUST NOT rely on model judgment
- MUST NOT require generative inference
- MUST cause immediate safety-trigger errors

Platforms MUST validate inputs against these rules *before* behaviour execution.

2.8.8 Prohibited Outputs

2.8.9 prohibited_outputs (Required)

Specifies output forms or content that MUST never be produced by the capability.

Examples:

- unstructured natural language
- harmful instructions
- hallucinated scientific claims
- non-schema fields
- categories outside enumerated sets
- unsafe numerical values

Structure example:

```
"prohibited_outputs": {
  "content_categories": ["self_harm", "violence"],
  "unnamed_fields": true,
  "patterns": ["regex1"]
}
```

If behaviour produces an output that violates these rules:

- The platform MUST block the output.
- The platform MUST return a deterministic safety error.
- The violation MUST be logged for audit.

2.8.10 Content Restrictions

Content restrictions define **what types of content the capability is allowed to operate on**.

2.8.11 content_restrictions (Required)

These MAY include:

- allowed languages
- allowed categories of text
- maximum allowed content size
- allowed structural patterns
- allowed semantic domains

Example:

```
"content_restrictions": {
  "languages": ["en", "fr"],
  "allowed_categories": ["general", "informational"],
  "max_length": 5000
}
```

Rules:

- Restrictions MUST be deterministic.
- Restrictions MUST NOT depend on inference.
- Violations MUST map to defined error conditions.

2.8.12 Domain Restrictions

2.8.13 domain_restrictions (Required)

These specify conceptual domains the capability MUST NOT enter.

Examples:

- no medical interpretation
- no legal reasoning
- no financial advice
- no chemical synthesis
- no political persuasion
- no biometric classification

Structure:

```
"domain_restrictions": {
  "forbidden_domains": ["medical", "legal"]
}
```

Domain restrictions MUST be clear, closed lists—no partial definitions.

2.8.14 Safety Triggers

2.8.15 safety_triggers (Required)

Defines explicit conditions that MUST trigger deterministic safety fallback or error output.

Safety triggers MAY include:

- detection of prohibited input categories
- detection of prohibited output categories
- detection of ambiguous or adversarial content
- constraints violations that imply safety risk
- domain mismatches
- unsupported languages
- extreme numeric values
- model-level uncertainty signals (if provided by the platform)*

*Note: LLM uncertainty signals may only be used if deterministic and documented.

Example:

```
"safety_triggers": {
  "prohibited_input_detected": true,
  "unsupported_language": ["zh", "ar"],
  "ambiguous_request": true
}
```

All triggers MUST map *one-to-one* with:

- error conditions in the Behaviour Block, or
- fallback structures in the Output Schema.

2.8.16 Safety Enforcement Requirements

Platforms MUST enforce safety in this order:

1. Schema validation
2. Constraints validation
3. Safety block validation
4. Behaviour execution

If safety rules are violated:

- Behaviour MUST NOT execute.
- A deterministic error MUST be returned.
- Output MUST NOT contain generative explanations.

Platforms MUST NOT override or relax safety rules declared in the Safety Block.

2.8.17 Determinism Requirements

Safety evaluation MUST be:

- rule-based
- consistent across platforms
- independent of model differences
- independent of runtime parameters
- not probabilistic
- not subject to interpretation

If the same input is unsafe on one platform, it MUST be unsafe on all.

2.8.18 Invalid Safety Patterns

The following safety structures are prohibited:

- subjective rules (“generally safe”, “probably harmful”)
- ambiguous domain categories
- open-ended lists (“etc.”)
- model-dependent safety decisions
- dynamic thresholds not encoded in the file
- safety definitions that include behavioural logic

Safety MUST be declarative and unambiguous.

2.8.19 Summary of Normative Safety Requirements

A valid Safety Block:

- MUST include all required sections
- MUST define explicit safety boundaries
- MUST prohibit unsafe inputs and outputs
- MUST define deterministic safety triggers
- MUST enable pre-execution validation
- MUST NOT encode behaviour
- MUST enforce identical safety outcomes across platforms
- MUST guarantee deterministic fallback/error behaviour

Violation of any requirement constitutes immediate non-compliance.

2.9 Reserved Fields and BCS Extensions

The BCS standard defines a set of **reserved fields**, **reserved keywords**, and **extension mechanisms** that ensure cross-platform consistency and prevent naming collisions between capability authors, registries, and vendor runtimes.

Reserved fields are those that:

- MUST NOT be used for developer-defined purposes,
- MUST be interpreted identically across platforms,
- MUST NOT be overridden or redefined,
- MUST not appear in contexts outside those defined by the standard,

- MAY be extended only through formal BCS extension mechanisms.

This section also defines the allowable extension patterns developers may use when platform-specific or capability-specific metadata is necessary, without violating cross-platform determinism.

2.9.1 Purpose of Reserved Fields

Reserved fields exist to:

1. Prevent developers from defining capability fields that conflict with the BCS specification.
2. Ensure consistent meaning of key structures across ecosystems.
3. Reduce ambiguity in capability definitions.
4. Maintain forward compatibility as the BCS standard evolves.
5. Allow platforms to safely introduce vendor-specific metadata without breaking capability interoperability.

Reserved fields MUST always retain the meaning defined in this specification.

2.9.2 Global Reserved Fields

The following fields are globally reserved and MUST NOT be repurposed or redefined by developers:

- bcs_version
- metadata
- behaviour
- input_schema
- output_schema
- constraints
- safety
- extensions
- error
- _internal (prefix reserved for system-level use)
- $schema
- $ref
- definitions
- oneOf
- allOf
- anyOf
- not

These keywords form the backbone of the BCS structural model and reference mechanisms.

Developers MUST NOT introduce fields of the same name at any other level of the capability file.

2.9.3 Reserved Field Name Prefixes

The following prefixes are restricted and MUST NOT be used by developer-defined fields:

- bcs_

- sys_
- _bcs_
- _sys_
- $

Rules:

- Fields beginning with these prefixes are reserved for future extensions of the BCS standard.
- Developers MUST NOT create fields beginning with these prefixes in metadata, behaviour, schema, constraints, or safety blocks.
- Any such field MUST produce a validation error.

Example invalid field names:

- "bcs_custom"
- "sys_override"
- "_bcs_temp"
- "$config"

2.9.4 Reserved Enumerations and Types

The following enumerations are reserved and MUST retain their defined meanings:

2.9.5 Risk Levels

- "low"
- "medium"
- "high"

2.9.6 Deterministic Behaviour Flags

- "deterministic" (boolean)
- "fallback"
- "error"

2.9.7 Prohibited Content Categories

This standard reserves specific strings for content categories, such as:

- "violence"
- "self_harm"
- "extremism"
- "hate"
- "sexual_content"
- "illegal_activities"
- "medical_advice"
- "legal_advice"
- "financial_advice"

Platforms MUST use consistent definitions for these categories.
Developers MUST NOT use these reserved category identifiers for unrelated purposes.

2.9.8 BCS-Specific Schema Extensions

The BCS standard defines several schema extensions derived from JSON Schema but with additional constraints to ensure deterministic behaviour.

Developers MAY use these extensions exactly as defined below.

2.9.9 bcs_type

Adds stricter type constraints.

Allowed values:

- "categorical"
- "bounded_string"
- "bounded_number"
- "timestamp"
- "language_code"
- "classification_label"

Example:

"bcs_type": "categorical"

2.9.10 bcs_ordering

Defines deterministic ordering for arrays.

Allowed values:

- "lexical"
- "numeric"
- "custom" (requires explicit comparator list)

2.9.11 bcs_null_policy

Dictates how null values MUST be handled.

Allowed values:

- "forbidden"
- "allowed"
- "coerced_to_default"

Platforms MUST enforce null handling exactly as declared.

2.9.12 Vendor Extension Mechanism

Vendors MAY attach additional metadata using the **BCS Extension Namespace**, but MUST NOT interfere with core definitions.

Vendor extensions MUST be placed ONLY inside the extensions block:

Example:

```
"extensions": {
  "openai": { ... },
  "google": { ... },
  "anthropic": { ... }
}
```

Rules:

- Vendor names MUST be lowercase alphanumeric.
- Vendors MUST NOT override reserved fields.
- Extensions MAY include versioning, feature flags, or deployment metadata.
- Extensions MUST NOT change behaviour or schema.
- Extensions MUST NOT introduce any generative instructions.

If a vendor extension conflicts with the core capability definition, the capability is invalid.

2.9.13 Developer Extension Mechanism

Developers MAY also include additional metadata using a reserved-neutral namespace.

Allowed prefix: dev_

Example:

```
"extensions": {
  "dev_custom": {
    "notes": "For internal testing only.",
    "priority": "low"
  }
}
```

Rules:

- Developer extensions MUST NOT alter behaviour.
- Developer extensions MUST NOT add or redefine input/output fields.
- Developer extensions MUST be optional for platform interpretation.
- Developer extensions MUST NOT alter determinism.

2.9.14 Future-Proofing Requirements

To ensure long-term stability:

- The set of reserved fields MUST NOT shrink over time.
- New reserved keywords MAY be added in future BCS versions.
- Fields beginning with bcs_ or _bcs_ remain permanently reserved.
- Backward compatibility MUST NOT be broken by future reserved-field additions.
- Capabilities MUST declare their bcs_version to allow correct interpretation.

2.9.15 Reserved Keywords Summary

The following keywords MUST NOT be used outside specified contexts:

- Structural: metadata, behaviour, input_schema, output_schema, constraints, safety, extensions
- JSON Schema: $schema, $ref, definitions, oneOf, allOf, anyOf, not
- BCS Reserved Prefixes: bcs_, sys_, _bcs_, _sys_, $
- Categories: "violence", "self_harm", "extremism", "hate", "medical_advice", etc.
- Determinism: "deterministic", "fallback", "error"

Use of any reserved keyword in an invalid context MUST cause capability validation to fail.

2.9.16 Summary of Normative Requirements

A valid capability:

- MUST NOT redefine or repurpose reserved fields.
- MUST NOT use reserved prefixes for developer-defined fields.
- MUST restrict extensions to the extensions block.
- MUST preserve determinism in all extensions.
- MUST remain forward-compatible by avoiding prohibited field names.
- MUST allow platform-level validation using reserved keyword meaning.

Violation of any rule renders the capability non-compliant.

2.10 Extensions Block Specification

The **Extensions Block** provides a formally sanctioned mechanism for including optional, non-behavioural metadata that does **not** affect capability execution, behaviour, safety, schema, or constraints.
Extensions exist to support:

- platform-specific metadata,
- developer-specific metadata,
- marketplace or registry annotations,
- deployment descriptors and configuration hints,
- future-proofing and vendor interoperability.

The Extensions Block is **strictly optional**, but when present it MUST comply with all rules defined in this section.

Extensions MUST NEVER alter behaviour, schema, safety, or determinism.

2.10.1 Extension Conformance Rules

The extensions block allows platforms and vendors to include additional, non-standard fields for experimental, diagnostic, or platform-specific purposes. Such fields MUST be optional, and their presence or absence MUST NOT change the meaning, interpretation, or execution semantics of any mandatory element defined in this specification.

Conforming platforms MUST ignore unknown extension fields rather than attempting to infer behaviour from them. Where a platform provides extended functionality that affects behavioural outcomes, that functionality MUST be expressed through formally standardised fields in a future version of this specification rather than relying on proprietary extensions.

2.10.2 Purpose of the Extensions Block

The Extensions Block exists to:

- allow platforms to include metadata without breaking cross-platform compatibility,
- permit vendors to store configuration details safely,
- allow developers to attach auxiliary information without interfering with core capability semantics,
- provide a standard location for optional metadata to avoid polluting required blocks,
- ensure future extensibility of the BCS framework.

The Extensions Block is **non-behavioural**, meaning:

- it MUST NOT change capability behaviour,
- it MUST NOT influence how agents interpret tasks,
- it MUST NOT modify input or output structure,
- it MUST NOT override or conflict with required fields.

Extensions are metadata only. They are inert. They are ignored by behavioural interpretation.

2.10.3 Canonical Representation Conformance Rules

The serialisation rules in this section define the canonical representation of a capability file. Two capability files that are structurally and semantically identical, but differ only in formatting, whitespace, field ordering within unordered collections, or equivalent normalisation artefacts, MUST be treated as the same capability for the purposes of validation, execution, comparison, and storage.

A conforming platform MAY internally transform or re-serialise a capability file, but the resulting representation MUST remain canonically equivalent to the form defined by this specification. Platforms MUST NOT treat non-canonical serialisation differences as meaningful variations in behaviour or capability identity.

2.10.4 Required Structure

If present, the Extensions Block MUST follow this structure:

```
"extensions": {
  "vendor_namespace": { ... },
  "developer_namespace": { ... }
}
```

Rules:

- MUST be a JSON object.
- MUST contain only namespaced objects.
- MUST NOT contain scalar fields, arrays, or un-namespaced objects at the top level.
- MUST NOT include behaviour definitions, schema fields, or safety rules.
- MUST NOT include reserved field names or reserved prefixes (see Section 2.8).

The block MAY contain one or more namespaces.

2.10.5 Extension Namespaces

Two types of namespaces are allowed:

2.10.6 Vendor Namespaces (Allowed)

Vendors MAY include fields under a namespace matching their vendor identifier.

Rules:

- MUST be lowercase alphanumeric (e.g., "openai", "google", "apple").
- MUST appear only inside extensions.
- MUST NOT conflict with capability behaviour.
- MUST NOT redefine or override metadata, schema, constraints, or safety.
- MUST NOT introduce reserved fields.
- MUST NOT influence determinism.

Example:

```
"extensions": {
  "openai": {
```

```
  "runtime_hint": "v2",
  "preferred_engine": "fast-sandbox"
 }
}
```

Vendor extensions MAY include:

- deployment configuration
- performance hints
- marketplace categorisation
- internal diagnostic metadata
- registry submission data

2.10.7 Developer Namespaces (Allowed)

Developers MAY include optional metadata under the **dev_** namespace prefix.

Rules:

- MUST use the prefix dev_.
- MUST be unique per capability to avoid collisions.
- MUST NOT affect any formal behaviour.
- MUST be optional for platforms.

Example:

```
"extensions": {
 "dev_notes": {
  "review_status": "pending-audit",
  "internal_priority": "low"
 }
}
```

Permitted uses:

- author notes
- internal QA metadata
- testing annotations
- development flags
- draft markers

NOT permitted:

- fallback behaviour
- error codes
- schema definitions
- logical rules

- execution instructions

2.10.8 Prohibited Content in Extensions

Extensions MUST NOT contain:

- behavioural logic
- safety rules
- schema fields
- constraints
- transformation rules
- fallback definitions
- error behaviour
- implicit instructions to the agent
- natural-language prompts intended to alter output
- anything affecting determinism
- model-specific operation rules that change execution semantics

Extensions MUST NOT be used as a loophole to bypass core specification rules.

Violation of these conditions renders the capability **immediately non-compliant**.

2.10.9 Extension Validation Rules

A capability containing an Extensions Block is valid **only if**:

1. The block appears *after* the Safety Block.
2. All namespaces comply with namespace rules.
3. No reserved prefixes or reserved fields appear.
4. No field changes or influences behaviour, schema, safety, or constraints.
5. No field contradicts determinism requirements.
6. All extension content is structurally valid JSON.
7. No extension field introduces generative instructions.

If any rule is violated:

- the capability MUST be rejected by validators and registries,
- the capability MUST NOT be loaded by platforms,
- the extension MUST NOT be ignored — violation invalidates the entire file.

2.10.10 Namespacing and Forward Compatibility

The Extensions Block is designed for long-term extensibility.

Platforms MAY add new namespaces in future without breaking existing capabilities.

Developers SHOULD NOT rely on extension data being interpreted by platforms.
Everything in extensions is **MAY-use by platforms, MUST-ignore for behaviour**.

Forward compatibility rules:

- **New namespaces MAY be added** at any time.
- **Existing extension data MUST NOT alter behaviour**.
- **Platforms MUST ignore unknown namespaces** without failing capability execution.
- **Capabilities MUST remain valid even if extension fields are removed or ignored**.
- **Extensions MUST NOT reference future BCS fields that do not yet exist.**

2.10.11 Recommended (Informative) Extension Patterns

Examples of legitimate non-normative use cases:

Marketplace metadata

```
"extensions": {
  "openai": {
    "category": "text-processing",
    "visibility": "public"
  }
}
```

Developer QA notes

```
"extensions": {
  "dev_qatags": {
    "stable": false,
    "test_coverage": "80%"
  }
}
```

Deployment preferences (non-binding)

```
"extensions": {
  "google": {
    "region_preference": "eu-west"
  }
}
```

Internal traceability

```
"extensions": {
  "dev_trace": {
    "build_id": "A23-991",
    "pipeline": "ci-v4"
```

```
  }
}
```

2.10.12 Summary of Normative Requirements

A valid Extensions Block:

- MUST be optional.
- MUST appear only at the end of the capability file.
- MUST contain only namespaced objects.
- MUST use allowed namespace patterns.
- MUST NOT define or alter behaviour.
- MUST NOT include safety rules or schema content.
- MUST NOT override or conflict with required blocks.
- MUST NOT include reserved field names or prefixes.
- MUST NOT influence determinism or execution.
- MUST be safely ignorable by all platforms.

Any violation MUST cause the capability to be rejected.

2.11 Serialisation Rules

The Serialisation Rules define the exact formatting, encoding, and structural constraints that capability files MUST follow in order to ensure deterministic interpretation across platforms, validators, registries, vendors, and agent runtimes.
These rules prevent ambiguity, eliminate vendor-specific JSON dialects, and guarantee that a capability file authored once can be executed consistently anywhere.

All capability files MUST follow these serialisation rules, regardless of platform, tooling, or internal representation.

2.11.1 Purpose of Serialisation Rules

Serialisation requirements exist to:

- ensure cross-platform portability,
- eliminate structural ambiguity,
- support deterministic parsing,
- enable byte-level signature validation,
- allow distributed caching and hashing,
- enforce strict compliance with BCS field ordering,
- guarantee that no hidden or implicit structures appear.

Serialisation rules apply **before** behavioural interpretation and **before** any runtime execution takes place.

A file that fails serialisation validation MUST be rejected before any schema, safety, or behaviour checks occur.

2.11.2 Allowed Serialisation Formats

A BCS capability file:

MUST be serialised as UTF-8 encoded JSON or a JSON-equivalent structured format.

JSON-equivalent formats include:

- strict JSON
- canonical JSON
- JSON5 **ONLY IF** comments and non-standard elements are removed
- YAML **ONLY IF** it is mechanically converted into strict JSON before validation

However:

The canonical and normative format of the capability file is JSON.

Platforms MUST validate the JSON-serialised form.
Any alternative representation MUST be losslessly convertible to strict JSON.

2.11.3 Prohibited Formats

Capability files MUST NOT be expressed in:

- XML
- HTML
- TOML
- INI
- CSV
- Protobuf
- MessagePack
- BSON
- CBOR
- any binary or compressed format as the *primary* published form

Platforms MAY internally re-encode capability files, but MUST NOT expect or require non-JSON inputs.

2.11.4 Encoding Requirements

All capability files MUST be:

- encoded in **UTF-8**
- free of BOM (Byte Order Mark)
- free of invisible control characters except standard whitespace

Allowed characters:

- ASCII
- UTF-8 letters
- UTF-8 digits
- UTF-8 punctuation permitted by JSON

Prohibited characters:

- non-printable Unicode control codes (U+0000 to U+001F except standard whitespace)
- embedded nulls
- proprietary encoding bytes
- mixed-encoding sequences

If invalid encoding is detected, the capability MUST be rejected.

2.11.5 JSON Structural Requirements

The JSON encoding MUST follow strict rules:

2.11.6 No Comments

Comments of any form are prohibited, including:

```
// comment
/* comment */
# comment
```

2.11.7 No Trailing Commas

The following is invalid:

```
{
  "metadata": { ... },
}
```

2.11.8 No Unquoted Keys

All keys MUST be double-quoted JSON strings:

```
"metadata": { ... }
```

Never:

```
metadata: { ... }
```

2.11.9 No Single Quotes

Invalid:

'name': 'example'

2.11.10 No Duplicate Keys

The following is invalid:

```
{
  "metadata": {...},
  "metadata": {...}
}
```

Duplicate keys MUST cause immediate validation failure.

2.11.11 No Additional Properties Outside Schema-defined Blocks

Arbitrary top-level fields are prohibited.

Invalid:

```
{
  "metadata": { ... },
  "debugMode": true
}
```

Only fields defined as valid capability blocks may exist.

2.11.12 Boolean Values MUST Be Lowercase

Valid:

true

false

Not valid:

True

FALSE

2.11.13 Numbers MUST Be Standard JSON Numbers

Prohibited:

- NaN
- Infinity

- Hexadecimal numeric literals
- Octal literals
- Exponential notation with uppercase E (optional but discouraged)

2.11.14 Ordering Requirements

Ordering is normative and MUST be preserved exactly:

1. bcs_version
2. metadata
3. behaviour
4. input_schema
5. output_schema
6. constraints
7. safety
8. extensions (optional)

Violations of ordering MUST result in rejection.

Within blocks:

- field ordering SHOULD follow the order defined in the BCS specification,
- platforms MUST NOT reorder fields during ingestion,
- platforms MAY reorder internally **after** validation, but MUST NOT republish reordered files.

2.11.15 Whitespace and Formatting Rules

Whitespace is generally flexible but must not affect structure.

Permitted whitespace:

- space
- tab
- newline
- carriage return

Disallowed whitespace:

- vertical tab
- null bytes
- Unicode control sequences

Indentation:

- Any indentation level is allowed,
- Tabs or spaces are allowed,
- Mixed indentation is allowed but discouraged,
- Indentation MUST NOT change semantic meaning.

2.11.16 Canonical Serialisation (Recommended)

(Not required, but highly encouraged for registry and signature use.)

Canonical serialisation ensures identical byte output for:

- registry submissions
- version comparisons
- digital signatures
- deterministic caching
- platform-agnostic hashing

Canonical serialisation SHOULD enforce:

- fields sorted lexicographically inside objects (except top-level sections),
- arrays preserved in deterministic order (see bcs_ordering),
- no trailing whitespace,
- UTF-8 normalisation to NFC.

2.11.17 File Extension Requirements

Files SHOULD use:

- .json for capability files
- .bcs.json for registry use
- .bcs for compressed or packaged formats (optional)

Platforms MUST accept .json.
Platforms MAY accept .bcs.json.
Platforms MUST NOT require .bcs unless distributing proprietary bundles.

2.11.18 Validation Priority and Error Handling

Serialisation validation MUST occur **before**:

- schema validation
- constraints validation
- safety validation
- behavioural validation

If serialisation fails:

- the capability MUST be rejected immediately,
- no behaviour may be loaded or executed,
- the validator MUST return a serialisation error,
- the platform MUST NOT attempt to auto-correct the file.

Auto-correction is explicitly prohibited to preserve deterministic author intent.

2.11.19 Example of Valid Serialisation (Non-Normative)

```
{
  "bcs_version": "1.1",
  "metadata": {
    "id": "bby.example-capability",
    "name": "Example Capability",
    "version": "1.0.0",
    "summary": "An example of valid BCS serialisation.",
    "description": "This example demonstrates strict JSON encoding.",
    "developer": {
      "name": "Big Blue Yonder",
      "contact": "https://bigblueyonder.com"
    },
    "created": "2025-12-09T12:00:00Z",
    "modified": "2025-12-09T12:00:00Z",
    "tags": []
  },
  "behaviour": { ... },
  "input_schema": { ... },
  "output_schema": { ... },
  "constraints": { ... },
  "safety": { ... },
  "extensions": {}
}
```

2.11.20 Summary of Normative Serialisation Requirements

A valid BCS capability file:

- MUST be encoded in UTF-8
- MUST use strict JSON
- MUST NOT contain comments or trailing commas
- MUST NOT include duplicate keys
- MUST follow required top-level ordering
- MUST prohibit unquoted keys and single quotes
- MUST NOT use prohibited numeric or boolean formats
- MUST use deterministic, machine-readable structure
- MUST pass serialisation validation before any other checks

Violations render the capability **non-compliant**, and it MUST be rejected.

2.12 Canonical Example Capability File (Normative)

This section defines the canonical normative example of a fully compliant BCS v1.1 capability file. The purpose of this example is to demonstrate:

- correct block ordering
- correct JSON structure
- valid metadata
- deterministic behaviour rules
- valid input and output schemas
- correct constraints
- correct safety declarations
- correct use of optional extensions

This example is normative: all capability files MUST follow the same structure and semantics unless otherwise stated.

The example capability performs a simple, deterministic text-length classification.
It is intentionally minimal, unambiguous, and free of heuristics or generative behaviour.

2.12.1 Canonical Capability Example (Normative JSON)


```
{
  "bcs_version": "1.1",
  "metadata": {
    "id": "bby.text-length-classifier",
    "name": "Text Length Classifier",
    "version": "1.0.0",
    "summary": "Classifies text as short, medium, or long based on character count.",
    "description": "This capability deterministically classifies the length of input text into one of three categories: short, medium, or long. It applies fixed numeric boundaries defined in the behaviour block and produces a structured classification label without using model inference.",
    "developer": {
      "name": "Big Blue Yonder",
      "contact": "https://bigblueyonder.com/bcs-contact"
    },
    "created": "2025-12-01T00:00:00Z",
    "modified": "2025-12-01T00:00:00Z",
    "tags": ["classification", "text", "deterministic"]
  },
  "behaviour": {
    "description": "Classifies the length of the provided text into the categories short, medium, or long based on deterministic character-count thresholds.",
```

```
  "inputs": {
    "text": "The natural-language text whose length (in characters) will be classified."
  },
  "transformation": {
    "rules": [
      {
        "condition": "length(text) < 50",
        "output_label": "short"
      },
      {
        "condition": "length(text) >= 50 AND length(text) < 200",
        "output_label": "medium"
      },
      {
        "condition": "length(text) >= 200",
        "output_label": "long"
      }
    ]
  },
  "outputs": {
    "label": "The deterministic length classification (short, medium, long).",
    "length": "The number of characters contained in the input text."
  },
  "determinism": {
    "ordering": "Rules MUST be evaluated in sequential order as defined in the transformation block.",
    "repeatability": "Given identical input text, the output MUST remain identical across all platforms.",
    "ambiguity": "No ambiguous conditions exist; each text length matches exactly one rule."
  },
  "fallbacks": {
    "empty_text": {
      "label": "short",
      "length": 0
    }
```

```
  },
  "error_conditions": {
    "missing_text": "Triggered when the required field `text` is absent.",
    "invalid_type": "Triggered when `text` is not a string.",
    "exceeds_max_length": "Triggered when text length exceeds the maximum allowed size defined in constraints."
  }
 },
 "input_schema": {
  "type": "object",
  "required": ["text"],
  "properties": {
   "text": {
    "type": "string",
    "minLength": 0,
    "maxLength": 5000
   }
  },
  "additionalProperties": false
 },
 "output_schema": {
  "type": "object",
  "required": ["label", "length"],
  "properties": {
   "label": {
    "type": "string",
    "enum": ["short", "medium", "long"]
   },
   "length": {
    "type": "integer",
    "minimum": 0
   }
  },
  "additionalProperties": false
 },
 "constraints": {
```

```
  "value_constraints": {
    "text.maxLength": 5000
  },
  "structural_constraints": {
    "text.mustBeString": true
  },
  "relational_constraints": {},
  "domain_constraints": {
    "allowed_categories": ["general-text"]
  }
},
"safety": {
  "risk_level": "low",
  "prohibited_inputs": {
    "patterns": [],
    "content_categories": [],
    "value_ranges": {}
  },
  "prohibited_outputs": {
    "unnamed_fields": true,
    "patterns": [],
    "content_categories": []
  },
  "content_restrictions": {
    "languages": ["en"],
    "max_length": 5000,
    "allowed_categories": ["general"]
  },
  "domain_restrictions": {
    "forbidden_domains": ["medical", "legal", "financial", "biometric"]
  },
  "safety_triggers": {
    "prohibited_input_detected": true,
    "unsupported_language": ["zh", "ar", "ru"],
    "excessive_length": true
  }
```

```
  },
  "extensions": {
    "dev_notes": {
      "review_status": "example-only",
      "intended_use": "canonical demonstration of BCS compliance"
    }
  }
}
```

2.12.2 Explanation of the Canonical Example (Normative)

2.12.3 Why this example?

This example:

- contains every mandatory block,
- uses clean deterministic logic,
- demonstrates error handling,
- follows strict JSON structure,
- includes a well-defined safety model,
- shows how constraints and behaviour interact,
- uses realistic but simple thresholding behaviour.

It is intentionally minimal to avoid conflating BCS rules with domain complexity.

2.12.4 Validation Requirements for the Example

A validator MUST accept this example as:

- structurally valid
- schema compliant
- deterministic
- safe
- serialisation-compliant
- fully interoperable
- correct in ordering

Any validator or platform implementation that rejects this example is considered **non-compliant** with BCS v1.1.

2.12.5 Prohibited Modifications to the Canonical Example

Platforms MUST NOT:

- reorder top-level blocks,
- add additional properties outside allowed blocks,

- modify behaviour rules,
- inject vendor metadata outside extensions,
- add generative fallback texts,
- infer new behaviour beyond the transformation rules,
- use unbounded output or free-form natural language.

If any such modification appears, the capability is invalid.

2.12.6 Summary of Section 2.11 Requirements

A canonical capability example MUST:

- demonstrate full compliance,
- use strictly deterministic behaviour,
- include safety and constraints blocks,
- include valid error conditions,
- use strict JSON syntax,
- include (optional) extensions,
- reflect the entire structural model of BCS.

2.13 Canonical JSON Schema for BCS Capability Files (Normative)

The following schema defines the complete, structural, machine-validatable specification for a BCS v1.1 capability file.
All compliant capability files MUST validate against this schema.

The canonical schema is expressed in JSON Schema format.
BCS vendored extensions (e.g., "bcs_type" and "bcs_ordering") appear where applicable.

2.13.1 Canonical Schema (Normative JSON Schema)

⚠️ **IMPORTANT**:
The JSON below contains **comments for explanation only**.
These MUST NOT appear in the actual published schema.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "BBY Capability Specification v1.1 - Canonical Schema",
  "type": "object",
  "required": [
    "bcs_version",
    "metadata",
    "behaviour",
    "input_schema",
    "output_schema",
```

```
  "constraints",
  "safety"
],
"additionalProperties": false,
"properties": {
  "bcs_version": {
    "type": "string",
    "pattern": "^1\\.1$"
  },
  "metadata": {
    "type": "object",
    "required": [
      "id",
      "name",
      "version",
      "summary",
      "description",
      "developer",
      "created",
      "modified"
    ],
    "properties": {
      "id": {
        "type": "string",
        "pattern": "^[a-z0-9]+([.][a-z0-9-]+)+$"
      },
      "name": { "type": "string" },
      "version": {
        "type": "string",
        "pattern": "^[0-9]+\\.[0-9]+\\.[0-9]+$"
      },
      "summary": {
        "type": "string",
        "maxLength": 200
      },
      "description": { "type": "string" },
```

```
    "developer": {
      "type": "object",
      "required": ["name", "contact"],
      "properties": {
        "name": { "type": "string" },
        "contact": { "type": "string" }
      },
      "additionalProperties": false
    },
    "created": {
      "type": "string",
      "format": "date-time"
    },
    "modified": {
      "type": "string",
      "format": "date-time"
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string",
        "pattern": "^[a-z0-9-]+$"
      }
    }
  },
  "additionalProperties": false
},
"behaviour": {
  "type": "object",
  "required": [
    "description",
    "inputs",
    "transformation",
    "outputs",
    "determinism",
    "error_conditions"
```

```
],
"properties": {
  "description": { "type": "string" },
  "inputs": {
    "type": "object",
    "additionalProperties": { "type": "string" }
  },
  "transformation": {
    "type": "object",
    "additionalProperties": true
  },
  "outputs": {
    "type": "object",
    "additionalProperties": { "type": "string" }
  },
  "determinism": {
    "type": "object",
    "properties": {
      "ordering": { "type": "string" },
      "repeatability": { "type": "string" },
      "ambiguity": { "type": "string" }
    },
    "required": ["ordering", "repeatability"],
    "additionalProperties": false
  },
  "fallbacks": {
    "type": ["object", "null"],
    "additionalProperties": true
  },
  "error_conditions": {
    "type": "object",
    "additionalProperties": { "type": "string" },
    "minProperties": 1
  }
},
"additionalProperties": false
```

```
},
"input_schema": {
  "$ref": "#/$defs/schemaBlock"
},
"output_schema": {
  "$ref": "#/$defs/schemaBlock"
},
"constraints": {
  "type": "object",
  "required": [
    "value_constraints",
    "structural_constraints",
    "relational_constraints",
    "domain_constraints"
  ],
  "properties": {
    "value_constraints": { "type": "object" },
    "structural_constraints": { "type": "object" },
    "relational_constraints": { "type": "object" },
    "domain_constraints": { "type": "object" }
  },
  "additionalProperties": false
},
"safety": {
  "type": "object",
  "required": [
    "risk_level",
    "prohibited_inputs",
    "prohibited_outputs",
    "content_restrictions",
    "domain_restrictions",
    "safety_triggers"
  ],
  "properties": {
    "risk_level": {
      "type": "string",
```

```
  "enum": ["low", "medium", "high"]
},
"prohibited_inputs": {
  "type": "object",
  "properties": {
    "content_categories": {
      "type": "array",
      "items": { "type": "string" }
    },
    "patterns": {
      "type": "array",
      "items": { "type": "string" }
    },
    "value_ranges": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "min": { "type": ["number", "integer"] },
          "max": { "type": ["number", "integer"] }
        }
      }
    }
  },
  "additionalProperties": false
},
"prohibited_outputs": {
  "type": "object",
  "properties": {
    "unnamed_fields": { "type": "boolean" },
    "patterns": {
      "type": "array",
      "items": { "type": "string" }
    },
    "content_categories": {
      "type": "array",
```

```
      "items": { "type": "string" }
    }
   },
   "additionalProperties": false
 },
 "content_restrictions": {
   "type": "object",
   "properties": {
    "languages": {
      "type": "array",
      "items": {
       "type": "string",
       "pattern": "^[a-z]{2}$"
      }
    },
    "allowed_categories": {
      "type": "array",
      "items": { "type": "string" }
    },
    "max_length": { "type": "integer" }
   },
   "additionalProperties": false
 },
 "domain_restrictions": {
   "type": "object",
   "properties": {
    "forbidden_domains": {
      "type": "array",
      "items": { "type": "string" }
    }
   },
   "additionalProperties": false
 },
 "safety_triggers": {
   "type": "object",
   "additionalProperties": true
```

```
      }
    },
    "additionalProperties": false
  },
  "extensions": {
    "type": "object",
    "additionalProperties": {
      "type": "object"
    }
  }
},
"$defs": {
  "schemaBlock": {
    "type": "object",
    "required": ["type", "properties"],
    "properties": {
      "type": {
        "type": "string",
        "pattern": "^object$"
      },
      "required": {
        "type": "array",
        "items": { "type": "string" }
      },
      "properties": {
        "type": "object",
        "additionalProperties": true
      },
      "oneOf": {
        "type": "array",
        "items": { "type": "object" }
      },
      "additionalProperties": { "type": "boolean" }
    },
    "additionalProperties": false
  }
```

```
  }
}
```

2.13.2 Schema Notes (Normative)

✓ Complete enforcement

This JSON Schema includes every structural requirement from Sections 2.1–2.11.

✓ Behaviour-neutral

The schema validates structure only; behavioural semantics must be validated separately.

✓ Reserved fields protected

Fields not explicitly allowed are rejected due to additionalProperties: false at top-level.

✓ Allows BCS extensions

Vendor and developer namespaces under extensions are permitted but isolated.

✓ Schema is future-proof

New fields can be added via $defs extension blocks without breaking compatibility.

2.13.3 Validator Conformance

A validator MUST:

- validate against this schema first,
- reject non-conforming capability files,
- fail fast on unexpected fields,
- perform no auto-correction,
- treat violations as fatal errors,
- treat the schema as immutable for BCS v1.1.

A capability that passes this schema but fails behavioural or safety validation is **structurally valid but not compliant**.

Both layers are required for full conformance.

2.13.4 Summary of Section 2.12 Requirements

A capability is structurally valid only if:

- it fully validates against this canonical schema,

- it obeys all reserved field rules,
- field ordering matches Section 2.1,
- serialisation rules (2.10) are followed,
- BCS extensions appear only under the extensions object,
- all mandatory blocks appear exactly once.

This schema is the authoritative structural definition of BCS capability files.

2.14 Structural Validation Workflow

This subsection defines the **normative validation workflow** for BCS capability files **at the file-structure level**, based on the rules in Section 2.

It describes the minimum sequence of checks that:

- registries,
- marketplaces,
- static validators, and
- platform runtimes

MUST perform **before** loading or executing a capability.

Section 2.13 covers **structural validation only**:

- file serialisation,
- block presence and ordering,
- schema correctness,
- constraint structure,
- safety block structure,
- extensions correctness.

Runtime behavioural validation, dynamic testing, and advanced safety evaluations are further elaborated in **Section 4 (Validation Pipelines)**.

A capability that fails any step in this workflow is **non-compliant** and MUST be rejected.

2.14.1 Validation Conformance Rules

Validation is a mandatory precondition for execution. A conforming platform MUST validate the structure, schema, and semantics of a capability file in full before executing any part of its behaviour. Where validation fails, the capability MUST NOT be executed, either partially or in a degraded or best-effort mode.

Ambiguous, underspecified, or conflicting definitions MUST be treated as validation errors rather than being resolved heuristically at runtime. Platforms MAY provide diagnostic feedback to assist authors in correcting invalid capability files, but such feedback MUST NOT modify or reinterpret the capability definition on behalf of the author.

2.14.2 Goals of the Structural Validation Workflow

The structural validation workflow exists to ensure that:

- only well-formed capability files enter execution environments;
- all mandatory blocks are present and valid;
- schema, constraints, and safety models are syntactically consistent;
- extensions do not undermine determinism or interoperability;
- behavioural and safety reasoning operate on a sound structural foundation.

Structural validation is a **precondition** for:

- registry publication,
- platform ingestion,
- agent orchestration,
- conformance certification.

2.14.3 Validation Stages (High-Level)

Platforms and validators MUST implement the following stages in this order:

1. **Serialisation Validation** (Section 2.10)
2. **Top-Level Structure & Ordering Validation** (Section 2.1)
3. **Canonical Schema Validation** (Section 2.12)
4. **Block-Level Sanity Checks** (Sections 2.2–2.8)
5. **Extensions Block Conformance** (Section 2.9)

If any stage fails:

- the capability MUST NOT be executed,
- the capability MUST be considered invalid,
- the validator MAY produce a diagnostic report, but MUST NOT attempt auto-repair.

2.14.4 Stage 1 — Serialisation Validation

The first step is to verify that the file complies with all serialisation rules in **Section 2.10**.

Validators MUST check that:

- the file is **UTF-8** encoded,
- the content is well-formed strict JSON,
- no comments or trailing commas appear,
- all keys are quoted strings,
- there are no duplicate keys at any level,
- booleans and numbers follow JSON standards,
- no extra top-level fields appear outside those defined by BCS.

If serialisation validation fails:

- validation MUST stop,
- no further checks MAY be performed,
- the capability MUST be rejected.

2.14.5 Stage 2 — Top-Level Structure & Ordering

After serialisation passes, the validator MUST confirm that:

- all mandatory top-level blocks are present:
  - bcs_version
  - metadata
  - behaviour
  - input_schema
  - output_schema
  - constraints
  - safety
- the optional extensions block, if present, appears **after** safety.
- the blocks appear in the **exact order** defined in Section 2.1.3.

If:

- any mandatory block is missing,
- any unexpected extra top-level field exists,
- or the order is not respected,

then the capability MUST be rejected.

2.14.6 Stage 3 — Canonical JSON Schema Validation

Once the presence and ordering of blocks is confirmed, the entire file MUST be validated against the **Canonical JSON Schema** defined in **Section 2.12**.

Validators MUST:

- apply the schema as-is,
- treat any schema violation as a hard error,
- not relax or modify schema rules per platform,
- enforce additionalProperties: false where specified.

A file that fails schema validation is **structurally invalid** and MUST NOT proceed to behavioural or safety validation.

Schema validation ensures that:

- every block has the correct internal structure,
- data types are correct,
- required fields exist,
- reserved fields are not misused,
- the file is structurally interoperable across platforms.

2.14.7 Stage 4 — Block-Level Sanity Checks

After successful validation against the canonical schema, validators MUST perform additional **sanity checks** at the block level to ensure internal coherence.

These checks MUST include at minimum:

2.14.8 Metadata Block Consistency

- id follows the required format and is unique within the registry.
- version follows semantic versioning.
- created ≤ modified in time.
- summary and description are non-empty, human-readable strings.
- No metadata field contradicts behaviour, constraints, or safety.

2.14.9 Behaviour Block Coherence

- inputs only reference fields present in input_schema.
- outputs only reference fields present in output_schema.
- error_conditions are non-empty and meaningful.
- fallbacks (if present) are compatible with the output_schema.
- Behavioural descriptions do not obviously conflict with declared constraints or safety (structural, not semantic check).

2.14.10 Input & Output Schema Coherence

- All referenced fields in behaviour, constraints and safety appear in the appropriate schema.
- Required fields are correctly listed in required.
- additionalProperties is explicitly set where necessary to avoid unbounded structures.
- Any oneOf usage is structurally sound (non-overlapping and exhaustive for its intended use).

2.14.11 Constraints Block Referencing

- Keys in value_constraints, structural_constraints, and relational_constraints reference fields that actually exist in input_schema or output_schema.
- No constraint refers to undefined or removed fields.
- No constraint structurally contradicts schema-level types (e.g., numeric min/max on a string-typed field).

2.14.12 Safety Block Referencing

- content_restrictions, prohibited_inputs, and prohibited_outputs reference valid categories or fields.
- domain_restrictions use well-formed domain labels.
- safety_triggers map to named error conditions or fallback paths defined in behaviour and/or output_schema.

Failures at this stage indicate a structurally inconsistent capability file and MUST result in rejection.

2.14.13 Stage 5 — Extensions Block Conformance

If an extensions block is present, validators MUST confirm that:

- all top-level keys inside extensions are valid namespaces
    - vendor namespaces (e.g., "openai", "google") are lowercase alphanumeric;
    - developer namespaces follow the allowed dev_ prefix pattern where used.
- no reserved prefixes (bcs_, sys_, _bcs_, _sys_, $) are used for extension keys.
- the content of each namespace is **structural metadata only**, not behaviour, schema, constraints, or safety rules.
- no fields in extensions conflict with or attempt to override values in core blocks.

Platforms MAY ignore all extension data for the purpose of execution.
However, if the Extensions Block violates any rule in **Sections 2.8–2.9**, the capability MUST be rejected.

2.14.14 Non-Auto-Correction Requirement

Validators and platforms:

- MUST NOT auto-correct, rewrite, or “fix” structurally invalid capability files;
- MUST NOT silently drop invalid fields;
- MUST NOT normalise files into a superficially valid form without reporting errors.

Any correction MUST occur at the authoring stage by the capability developer.

Structural validation is binary:

- **pass** → the file is eligible for further behavioural/safety validation and execution;
- **fail** → the file MUST be rejected and MUST NOT be executed.

2.14.15 Relationship to Section 4 (Full Validation Pipelines)

Section 2.13 defines the **minimum structural validation workflow** required for conformance with the File Structure Specification.

Section 4 will define:

- full validation pipelines,
- runtime safety checks,
- behavioural determinism verification,
- test harness integration,
- capability marketplace validation workflows.

A platform or registry is **not fully BCS-compliant** unless it:

1. Implements the structural validation workflow defined in Section 2.13, **and**

2. Implements the behavioural and safety validation pipelines defined in Section 4.

2.14.16 Summary of Normative Requirements

A BCS capability file is **structurally valid** only if:

- It passes serialisation validation (Section 2.10).
- It contains all mandatory blocks in the correct order (Section 2.1).
- It validates fully against the canonical JSON Schema (Section 2.12).
- All blocks are internally coherent and cross-reference schema fields correctly.
- The Extensions Block (if present) follows all rules in Sections 2.8–2.9.
- No attempt is made to auto-correct or partially accept invalid files.

Any failure in this workflow MUST result in immediate rejection of the capability file.

3.0 SAFETY MODEL & TAXONOMY

The Safety Model defines the mandatory controls, classifications, and deterministic behaviours that constrain how a capability may process inputs and generate outputs.
It ensures that capabilities operate safely, predictably, and consistently across platforms regardless of the underlying execution engine.

Safety provisions apply to **all** BCS-compliant capabilities and MUST be validated before execution (see Section 4).
Where behaviour determines *how* a capability functions, and structure determines *how* it is defined, **safety determines *whether* it is allowed to operate**.

Where behaviour and safety requirements conflict, safety takes precedence. A capability MUST degrade into fallback or explicit refusal rather than produce outputs that violate declared safety constraints."

3.1.1 Responsibility Boundary (Normative)

BCS distinguishes between the responsibilities of **capability authors** and the responsibilities of **platforms and agents** that execute or orchestrate capabilities. This boundary is essential to ensuring consistent safety behaviour across different execution environments.

Capability authors are responsible for:
– declaring the intended behavioural scope and constraints of the capability;
– defining prohibited inputs, outputs and operational contexts;
– specifying deterministic fallback behaviour in safety-failure conditions;
– ensuring that the declared behaviour, schemas and constraints are internally consistent.

Platforms and agents are responsible for:
– enforcing declared capability safety boundaries at runtime;
– preventing invocation in contexts that violate declared constraints;
– orchestrating safe composition when multiple capabilities are combined;
– applying any additional system-level safety or policy controls;
– rejecting or quarantining capabilities that fail validation or violate declared constraints.

Platforms MAY apply stricter safety rules than those declared by the capability, but MUST NOT weaken, ignore, override or bypass declared capability safety rules.

3.2 Execution Semantics Conformance Rules

The behaviour semantics defined in this section are normative. A conforming platform MUST execute a capability strictly in accordance with the sequencing, dependencies, constraints, and declared behaviour defined in the capability description. Platforms MUST NOT introduce alternative execution pathways, implicit ordering, or optimisation-driven behavioural changes that alter the declared semantics, even where such changes appear operationally beneficial.

Where execution semantics cannot be satisfied as declared, the platform MUST treat this as a runtime conformance failure and MUST NOT continue execution in an approximate, degraded, or partially evaluated form.

3.3 Purpose and Scope of the Safety Model (Normative)

The Safety Model exists to:

- prevent capabilities from processing harmful or disallowed content,
- constrain outputs to defined, safe, deterministic shapes,
- enforce boundaries around sensitive or regulated domains,
- ensure that unsafe situations lead to predictable, non-generative fallbacks,
- maintain uniform safety behaviour across vendors and runtimes,
- provide a taxonomy for categorising unsafe inputs and outputs,
- enable deterministic, auditable safe-failure behaviour.

Safety validation occurs **before** behavioural execution and MUST override all behavioural logic.

The Safety Model governs **content categories**, **patterns**, **domains**, and **fallback triggers**, not the operational details of model inference or platform-level enforcement.

3.4 Threat-Model Awareness (Informative)

The Safety Model assumes the presence of adversarial or stress-case conditions, including malicious or coercive prompts, prompt-injection attempts, ambiguous or boundary-case inputs, and scenarios where inputs or transformations may be used to bypass declared constraints. The specification treats these conditions as expected and requires capabilities to remain deterministic, schema-bounded, and safety-conformant under such circumstances.

3.5 Safety Taxonomy Overview

The BCS safety taxonomy establishes a universal classification system for:

- prohibited inputs

- prohibited outputs
- restricted content categories
- restricted operational domains
- safety triggers that must activate under unsafe conditions

This taxonomy MUST be used by all capabilities to ensure consistent enforcement across the ecosystem.

The taxonomy is organised into four major components:

1. **Content Categories**
2. **Input Restrictions**
3. **Output Restrictions**
4. **Domain Restrictions**

All safety categories MUST be machine-evaluated and deterministic.

3.6 Content Categories (Normative Taxonomy)

All safety-related inputs/outputs MUST be mapped to one or more of the following content categories.
These categories are globally defined for all BCS implementations.

3.6.1 Harm & High-Risk Categories

- violence
- self-harm
- hate or harassment
- sexual content
- child safety violations (CSAM, grooming, minor sexualisation)
- extremism or radicalisation
- criminal enablement
- dangerous advice (chemical, biological, weapons, explosives)

Capabilities MUST NOT process or emit these categories unless authorised under specific regulatory exemptions (outside BCS v1.1 scope).

3.6.2 Sensitive & Regulated Categories

- biometric data
- precise geolocation
- legal advice or legal facts
- medical/health advice
- regulated financial advice
- identity-sensitive data
- political persuasion or targeted political content

Capabilities encountering these MUST trigger safe failure *unless* explicitly authorised and certified.

Taxonomy Extensibility (Informative).
The safety taxonomy defined in this specification is intended as a baseline reference. Regulators, sectors and enterprises MAY extend or specialise this taxonomy to reflect jurisdiction-specific legal, ethical or compliance requirements, provided that such extensions do not weaken or contradict the capability's declared constraints or safety boundaries.

3.6.3 Low-Risk Categories

- general text
- structured data
- numerical values
- metadata
- user interface text

These categories require no additional restrictions unless declared explicitly in the capability.

3.7 Input Safety Rules (Normative)

Input safety rules determine whether an incoming input is allowed to proceed to behavioural evaluation.

A capability MUST define prohibited or restricted inputs using:

- content categories
- patterns (regex, keyword lists, signatures)
- value ranges
- language restrictions
- domain restrictions

If a prohibited condition is detected:

- Behaviour MUST NOT execute
- A safety trigger MUST activate
- The output MUST follow a deterministic fallback path

3.7.1 Prohibited Input Categories

Input in any category listed under:

safety.prohibited_inputs.content_categories

MUST immediately:

- bypass behaviour rules,
- activate the corresponding safety trigger,
- return the defined fallback.

3.7.2 Prohibited Input Patterns

Patterns may include:

- explicit keywords
- regular expressions
- semantic tags (if deterministic)
- structured signatures

Patterns MUST be evaluated deterministically (no inference).

3.7.3 Prohibited Value Ranges

Used to constrain:

- numeric parameters
- text length
- list sizes
- configuration ranges

If violated, the capability MUST trigger safe failure.

3.7.4 Language Restrictions

Capabilities MAY declare allowed languages:

safety.content_restrictions.languages

Inputs in other languages MUST trigger safe failure.

3.8 Output Safety Rules (Normative)

Output safety rules ensure that capabilities:

- do not emit prohibited content
- do not generate unsafe free-form text
- do not produce fields or structures outside the declared schema

3.8.1 Prohibited Output Categories

If output falls into any category declared prohibited:

- output MUST be discarded
- fallback MUST execute
- safety trigger MUST be raised

3.8.2 Structural Output Restrictions

Outputs MUST NOT:

- contain undeclared fields
- contain unbounded or unspecified structures
- contain free-form generative text unless explicitly allowed
- violate declared types in output_schema

3.8.3 Pattern-Based Output Restrictions

If output matches a prohibited pattern (keywords, regex):

- platform MUST block it
- fallback MUST return instead
- trigger MUST activate

3.9 Domain Restrictions

Domain restrictions specify contexts in which the capability MUST NOT operate.

Domains are high-level verticals such as:

- medical
- legal
- financial
- political
- biometric
- educational
- general
- entertainment
- developer tools
- safety analysis

Capabilities MUST explicitly declare prohibited or permitted domains.

If execution occurs within a restricted domain:

- behaviour MUST NOT execute
- domain safety trigger MUST activate
- fallback MUST be returned

3.10 Safety Triggers

Safety triggers are deterministic markers indicating that a safety violation occurred.

Triggers MUST:

- have stable names
- always map to a fallback pathway
- be machine-readable
- be non-generative (no natural language descriptions)

- not depend on platform-specific logic

Example triggers:

"prohibited_input_detected": true

"unsafe_output_blocked": true

"unsupported_language": "zh"

"disallowed_domain": "medical"

"excessive_length": true

Triggers MUST be defined in safety.safety_triggers.

3.11 Safe Failure & Fallback Model

When safety is violated, capabilities MUST enter **safe failure**.

Safe failure requires:

1. Behavioural rules MUST NOT execute.
2. Output MUST follow the deterministic fallback defined in behaviour.fallbacks.
3. Output MUST conform to output_schema.
4. The appropriate safety trigger MUST be raised.

Safe failure MUST NOT:

- invent alternative outputs
- fall back to generative responses
- apply heuristics or inference
- continue behaviour evaluation
- alter capability internal state

Platform-Level Messaging (Informative).
A capability's fallback behaviour MUST remain schema-bound and MUST NOT generate alternative content or unconstrained free-text outputs. However, platforms and user interfaces MAY present separate explanatory messages to end-users (for example, "This capability could not safely complete the requested operation"). Such messaging is considered a platform-level UX concern and does not form part of the capability's declared output.

If the transformation process would produce prohibited or unsafe content despite valid inputs, the capability MUST invoke its fallback behaviour rather than emit a partial or constrained unsafe output

3.12 Safety Examples — Conformant vs Non-Conformant (Informative)

The following short contrasts illustrate how the Safety Model is intended to operate in practice:

✓ **Conformant Example:** A capability receives an input that would produce content outside its declared safety boundary. The capability detects the condition and returns a deterministic fallback or explicit refusal, consistent with the rules defined in Section 3.

✗ **Non-Conformant Example:** A capability detects the same unsafe condition but attempts to emit a partially-redacted or modified output that still exposes unsafe content, instead of invoking fallback behaviour. This behaviour is non-compliant with the Safety Model.

These examples are illustrative only; the normative safety requirements are defined in the preceding subsections.

3.13 Safety Validation Requirements (Structural Layer)

Platforms MUST verify that:

- declared safety categories are valid taxonomy entries
- safety patterns are syntactically valid
- value ranges match schema types
- triggers map to valid fallback or error pathways
- prohibited content definitions are enforceable
- safety fields are structurally correct per Section 2.12

Safety validation MUST precede behavioural execution.

3.14 Behaviour–Safety Interaction Model

Safety ALWAYS overrides behaviour.

If safety validation fails:

- behavioural evaluation MUST NOT proceed
- the model MUST NOT attempt normal execution
- deterministic fallback MUST replace output
- the trigger MUST be set

Behavioural logic CANNOT weaken, override, or bypass safety controls.

3.15 Summary of Section 3 Requirements

A capability is safety-compliant only if:

- it declares explicit prohibited input/output categories
- it declares domain restrictions
- it defines deterministic, schema-valid fallback outputs
- it uses safety triggers that map to valid pathways

- unsafe content always results in safe-failure
- safety validation occurs before behavioural execution
- no free-form generative content appears in safety fields
- taxonomy categories conform to Section 3.3

Safety is a non-optional component of BCS conformance and MUST be implemented by all validators and platforms.

4.0 VALIDATION PIPELINES

The BCS Validation Pipeline defines the **mandatory processes** that platforms, registries, certification authorities, and agent runtimes MUST follow to evaluate a capability before it may be:

- published
- executed
- distributed
- stored in a registry
- certified as compliant

Section 4 builds directly upon the structural foundations in Section 2 and the safety framework defined in Section 3.

Validation under BCS v1.1 consists of four layers:

1. **Structural Validation** (Section 4.2)
2. **Behavioural Validation** (Section 4.3)
3. **Safety Validation** (Section 4.4)
4. **Certification Validation** (Section 4.5)

Each layer MUST succeed before moving to the next.
Failure at any stage MUST result in rejection of the capability.

The Safety Validation Pipeline (Layer 3) operationalises the safety guarantees defined in Section 3 and ensures they are enforced deterministically at validation and runtime.

4.1 Purpose and Scope of the Validation Pipeline

The BCS Validation Pipeline exists to ensure that:

- capabilities are structurally correct,
- behaviour is deterministic and testable,
- safety rules are enforceable and coherent,
- execution environments remain predictable across vendors,
- ecosystem-wide interoperability is maintained,
- certified capabilities pose minimal regulatory or operational risk.

Validation pipelines are **mandatory** for:

- public registries
- enterprise deployments
- model marketplaces
- vendor execution platforms
- any environment running capabilities at scale

Validation pipelines MAY be simplified for:

- internal-only capabilities,
- capabilities used for local testing,
- non-distributed prototypes.

However, Section 4 MUST be implemented in full for any capability intended for production use.

Validation Layer 1 — Structural
Validation Layer 2 — Behavioural
Validation Layer 3 — Safety
Validation Layer 4 — Certification / Governance

Declared Validation Coverage (Informative).
Platforms adopting BCS MAY implement the validation pipeline incrementally. Where only a subset of validation layers is implemented, the platform SHOULD declare which validation stages are applied (for example, structural validation only, or structural + behavioural validation). Partial validation does not constitute full BCS validation, but MAY be used as an interim adoption step provided it does not weaken or bypass any normative capability rules.

4.2 Structural Validation Pipeline (Layer 1)

Structural validation verifies that a capability file:

- is well-formed,
- matches the canonical schema,
- satisfies file structure rules,
- obeys ordering and serialisation constraints,
- does not contain prohibited fields,
- contains all required blocks,
- uses correct safety + behaviour referencing.

This layer corresponds directly to Section 2.10–2.13.

If **structural validation fails**, the capability MUST NOT proceed to behavioural or safety validation.

4.2.1 Required Steps

Platforms MUST perform the following in order:

1. **Serialisation Validation**
    - Check UTF-8
    - Verify strict JSON

    - Reject duplicates / comments / trailing commas
    - Enforce ordering
2. **Canonical Schema Validation**
    - Validate using Section 2.12 schema
    - Enforce additionalProperties: false
3. **Cross-Reference Validation**
    - Behaviour → schema fields
    - Constraints → schema fields
    - Safety → behaviour triggers
4. **Extensions Validation**
    - Namespace correctness
    - No behavioural override
    - No violations of safety or determinism

Structural validation MUST be:

- deterministic,
- reproducible,
- fully automated.

Platforms MUST NOT auto-correct structural errors.

4.3 Behavioural Validation Pipeline (Layer 2)

Behavioural validation ensures that:

- behaviour is deterministic,
- rule ordering is unambiguous,
- fallbacks are coherent,
- output structures match the declared schema,
- no rule contradicts constraints or safety.

This layer evaluates the **semantics** of behaviour without executing the runtime model.

Progression to Full Validation (Informative).
Platforms implementing partial validation are encouraged to progress toward the full multi-layer validation pipeline defined in this section. Full validation provides the strongest guarantees of safety alignment, behavioural consistency and cross-platform interoperability.

4.3.1 Determinism Validation

Platforms MUST verify:

- all conditional rules are fully specified,
- no overlapping conditions exist unless mutually exclusive,
- rule ordering is explicitly declared,
- fallbacks do not create behavioural ambiguity,
- the same input always maps to exactly one output rule.

If multiple behaviour rules could match the same input, the capability MUST be rejected.

4.3.2 Behaviour–Schema Consistency

Platforms MUST confirm:

- every referenced input exists in input_schema,
- every referenced output exists in output_schema,
- rule outputs match type definitions,
- no rule produces undeclared fields.

If a behavioural rule produces output not allowed in the schema,
the capability MUST be rejected.

4.3.3 Fallback Validation

Platforms MUST verify:

- fallbacks are explicitly declared,
- fallbacks produce schema-compliant outputs,
- fallbacks do not depend on unspecified fields,
- fallbacks do not silently generate unsafe or free-form output.

Fallbacks MUST remain deterministic under all violating conditions.

4.3.4 Error Condition Validation

Validators MUST ensure:

- every named error condition corresponds to a valid behavioural pathway,
- every error condition maps to a fallback or termination,
- no error condition produces undefined behaviour.

If any error condition lacks a valid output → rejection.

4.4 Safety Validation Pipeline (Layer 3)

Safety validation ensures that:

- prohibited input/output categories are enforceable,
- patterns and ranges are well-defined,
- safety triggers map correctly to behavioural fallbacks,
- domain restrictions align with taxonomy,
- safety rules do not contradict structure or behaviour.

This layer enforces Section 3.

Safety validation MUST occur **after structural validation** and **before execution**.

4.4.1 Safety Validation Conformance Rules

The safety validation pipeline is normative and MUST be applied before capability execution. Where a capability fails any safety validation stage, the platform MUST treat the capability as non-conformant and MUST NOT proceed to execution, either partially or in a degraded mode.

Platforms MUST NOT reinterpret, sanitise, or transform unsafe inputs, behaviours, or outputs in order to continue validation or execution. Where the safety status of a capability cannot be determined unambiguously, the platform MUST fail closed and refuse execution rather than proceeding under uncertainty.

4.4.2 Input Safety Validation

Platforms MUST confirm:

- each prohibited category is valid under Section 3 taxonomy,
- patterns are syntactically valid,
- value ranges match schema types,
- unsupported languages are clearly listed,
- safety triggers exist for every unsafe input path.

Violation → mandatory fallback, not behavioural execution.

4.4.3 Input Safety Enforcement Rules

Where an input falls within a prohibited safety category, pattern, or value constraint, a conforming platform MUST treat the capability as failing safety validation. The platform MUST NOT attempt to reinterpret, transform, or partially accept unsafe input in order to allow execution to proceed.

Where the safety classification of an input cannot be determined unambiguously, the platform MUST fail closed and reject execution rather than continuing under conditions of uncertainty.

4.4.4 Output Safety Validation

Platforms MUST confirm:

- prohibited output categories are declared and valid,
- structural output violations cannot occur given behaviour rules,
- any behaviour rule that could generate unsafe output triggers fallback,
- prohibited patterns apply deterministically.

Behaviour MUST NOT override safety.

4.4.5 Output Safety Enforcement Rules

Where capability execution would produce output that falls within a prohibited safety category or violates an output safety constraint, a conforming platform MUST prevent the output from being emitted and MUST treat the condition as a safety-validation failure.

Platforms MUST NOT rewrite, transform, truncate, or partially emit unsafe output in order to allow execution to continue. Where the safety status of an output cannot be determined unambiguously, the platform MUST fail closed and refuse emission rather than proceeding under uncertainty.

4.4.6 Domain Restriction Validation

Platforms MUST ensure:

- domain labels are valid,
- restricted domains are coherently declared,
- domain restrictions are enforceable via input or context inspection.

Unused or invalid domain labels → rejection.

4.4.7 Domain Restriction Enforcement Rules

Execution of a capability is only permitted within the domains explicitly declared in its capability description. Where the requested execution context falls outside a declared domain, a conforming platform MUST treat the capability as failing safety validation and MUST NOT attempt to execute it in an expanded or inferred domain.

Platforms MUST NOT substitute, approximate, or reinterpret domain definitions in order to permit execution. Where domain membership cannot be determined unambiguously, the platform MUST fail closed and refuse execution rather than proceeding under uncertainty.

4.4.8 Trigger Path Validation

Platforms MUST verify:

- each safety trigger corresponds to a real behavioural fallback OR
  a valid error termination pathway.
- no trigger leads to undefined behaviour.
- safety triggers never generate free-form text.

4.4.9 Trigger Path Enforcement Rules

Safety trigger definitions form a normative part of the capability description. Where a trigger path is incomplete, inconsistent with the declared behaviour model, or cannot be resolved unambiguously, a conforming platform MUST treat the capability as failing safety validation and MUST NOT proceed to execution.

Platforms MUST NOT bypass, downgrade, or substitute trigger conditions in order to allow execution to continue. Where a trigger condition is activated during validation, the platform MUST fail closed and refuse execution rather than attempting partial or degraded execution.

4.5 Certification & Governance Validation(Layer 4)

Certification validation is required where a registry, platform, or deployment environment mandates certification in addition to validation.
It is commonly required for:

- publication in official registries,
- enterprise/government deployment,
- regulated industry use (finance, legal, medical, safety domains).

The certification pipeline includes:

1. Full Structural Validation
2. Full Behavioural Validation
3. Full Safety Validation
4. Metadata Provenance Validation
5. Developer Identity Validation
6. Determinism Audit
7. Reproducibility Audit (same input → same output across reference engine)
8. Capability Execution Sandbox Test
9. Registry Signature & Immutable Hash Generation

4.5.1 Certification Validation Conformance Rules

Certification validation is normative where a capability declares that certification, attestation, or provenance guarantees form part of its trust model. Where a capability fails any certification validation stage — including expired attestations, unverifiable provenance, or invalid issuer signatures.

Where certification is required by the registry or runtime environment, failure of any certification validation stage MUST be treated as a blocking condition and the capability MUST NOT proceed to execution or publication.

Where certification is **not** required, a certification failure does not invalidate the capability's validation status, but the capability MUST NOT be represented as certified.

Platforms MUST NOT bypass, downgrade, or locally override certification requirements in order to permit execution. Where certification status cannot be determined unambiguously, the platform MUST fail closed and refuse execution rather than proceeding under uncertainty.

4.5.2 Provenance Validation

Certifiers MUST verify:

- developer identity,
- build environment hash,
- modification timestamps,
- version lineage,
- compatibility with previous versions.

4.5.3 Determinism Audit

Auditors MUST:

- run every behavioural rule with edge-case synthetic inputs,
- confirm stable behaviour across reference implementations,
- confirm no randomness or model inference influences output.

Capabilities that exhibit nondeterminism MUST fail certification.

4.5.4 Safety Audit

Auditors MUST:

- test each prohibited input category,
- test each prohibited output scenario,
- verify fallback activation,
- test unsupported languages,
- test domain restriction enforcement.

If any unsafe content bypasses a block → certification failure.

Validation confirms that a capability is safe and structurally sound prior to execution. Runtime behaviour remains constrained by the same rules and MUST NOT diverge from validated behaviour.

4.6 Runtime Pre-Execution Validation

Before execution, platforms MUST perform:

- cached structural + schema validation (fast path),
- safety trigger mapping check,
- domain/context compliance check,
- integrity + signature check for certified capabilities.

Capabilities MUST NOT execute unless runtime validation succeeds.

If runtime context violates safety or domain restrictions,
execution MUST be blocked with a deterministic fallback.

4.6.1 Pre-Execution Conformance Rules

Runtime pre-execution validation forms the final mandatory checkpoint before capability execution. A conforming platform MUST verify that all required runtime conditions, assumptions, and environmental constraints declared in the capability description are satisfied in full before execution begins. Where any required condition cannot be met, the platform MUST treat the capability as failing validation and MUST NOT execute it, either partially or in a degraded mode.

Platforms MUST NOT substitute, infer, or synthesise missing runtime conditions in order to permit execution. Where pre-execution readiness cannot be determined unambiguously, the platform MUST fail closed and refuse execution rather than proceeding under uncertainty.

4.7 Validation Failure Modes

If any stage of validation fails, platforms MUST:

- halt execution,
- return a deterministic failure response,
- raise the appropriate structural/behavioural/safety error code,
- avoid generating any behavioural output.

Platforms MUST NOT attempt auto-correction or speculative recovery.

4.7.1 Failure Handling Conformance Rules

Validation failures are terminal conditions. Where any validation stage fails, a conforming platform MUST treat the capability as non-conformant and MUST NOT execute it, either partially or in a degraded, fallback, or speculative mode.

Platforms MUST classify validation failures according to the failure modes defined in this section and MUST report them in an explicit, machine-interpretable form. Platforms MUST NOT suppress, reinterpret, or downgrade validation failures to warnings or soft-error states in order to permit execution.

4.8 Summary of Section 4 Requirements

A capability is **fully validated** only if:

- Structural validation passes,
- Behavioural validation passes,
- Safety validation passes,
- Certification (if required) passes,
- Runtime validation passes.

Validation MUST be:

- deterministic,
- reproducible,
- machine-verifiable,
- implementation-agnostic.

Failure at any stage MUST prevent execution.

In summary, the validation pipeline defined in this section is mandatory, sequential, and fail-closed. A capability MAY proceed to execution only when all validation layers succeed without exception. Any failure, ambiguity, unresolved constraint, or unmet assumption at any validation stage MUST result in rejection rather than partial, degraded, or speculative execution. Platforms that conform to this specification are therefore guaranteed to exhibit consistent, deterministic, and safety-aligned behaviour at the validation boundary.

5.0 CAPABILITY LIFECYCLE & VERSIONING

This section defines the rules for identifying, evolving, versioning, deprecating, and retiring capabilities within the BCS ecosystem.
It ensures that:

- capabilities can evolve safely over time,
- platforms can reason about compatibility,
- registries can track lineage and provenance,
- developers can publish updates without breaking dependants,
- enterprises can control upgrade risk.

Versioning and lifecycle rules apply to all capabilities, whether public, internal, or enterprise-deployed.

Lifecycle compliance is mandatory for all certified capabilities.

5.1 Purpose of Versioning & Lifecycle Management

This lifecycle model exists to:

- prevent breaking changes,
- ensure deterministic evolution,
- maintain compatibility across versions,
- provide predictable behaviour for downstream systems,
- support long-term maintainability,
- allow safe patching of vulnerabilities,
- enable stable referencing of capability IDs in external systems.

Without a formal lifecycle, capabilities would become unpredictable, untrackable, and unsafe for production use.

The lifecycle and versioning rules defined in this section are normative. Conforming platforms MUST apply these rules consistently when publishing, distributing, validating, and executing capabilities. Version identifiers, compatibility classifications, and lifecycle states are binding properties of a capability and MUST NOT be reinterpreted or overridden by a platform or deployment environment.

5.2 Version Compatibility Rules (Normative)

Minor versions of this specification SHOULD remain backward-compatible with capability files that conform to earlier minor versions within the same major version.

Major versions MAY introduce breaking changes where required for correctness, safety, or architectural integrity. Where breaking changes are introduced, they MUST be explicitly versioned, documented, and communicated through the lifecycle and deprecation processes defined in this section.

5.3 Capability Identity (Normative)

Every capability MUST have a globally unique identifier defined in:

metadata.id

The identifier MUST:

- use lowercase alphanumeric segments separated by dots
- remain stable for the lifetime of the capability
- never be reused for a different capability
- uniquely map to a single lineage

Example:

bby.text-length-classifier

Changing the id implies a *new capability*, not a new version.

A capability's identity is immutable and refers to a specific, versioned artefact. Once published, the behaviour, structure, and semantics associated with a capability version MUST NOT be altered in place. Any change that affects semantics, behaviour, safety properties, schemas, or observable outputs MUST result in the publication of a new version rather than modification of an existing one.

Platforms MUST treat capability identity as globally unique within the namespace in which it is issued, and MUST NOT substitute, alias, or merge distinct capabilities under a single identity.

5.4 Semantic Versioning Model (Normative)

Capabilities MUST use **semantic versioning (semver)**:

The semantic versioning model in this specification is normative and carries behavioural guarantees. The version class of a capability determines the compatibility expectations that conforming platforms MUST apply when validating, distributing, or executing that capability. Where a change does not meet the guarantees associated with a given version class, the capability MUST be published under a higher-impact version class rather than being misclassified.

MAJOR.MINOR.PATCH

Where:

5.4.1 PATCH Version (x.y.Z)

Patch changes MUST be backward-compatible and may include:

- bug fixes
- documentation fixes
- constraint tightening that does not change behaviour
- safety improvements that do not change outputs
- performance optimisations
- metadata corrections

Patch updates MUST NOT:

- change behaviour outputs,
- alter schema,
- introduce new required inputs,
- change safety taxonomy classifications.

Patch versions MUST be safe to auto-upgrade.

5.4.2 MINOR Version (x.Y.z)

Minor versions MAY introduce backward-compatible enhancements:

- new optional behaviour branches
- expanded output categories (optional only)
- additional safe fallbacks
- additional metadata fields
- appended safety triggers
- new, optional input fields (must default safely)

Minor updates MUST NOT:

- remove existing inputs or outputs
- redefine existing behaviour rules
- change safety restrictions in incompatible ways
- break compatibility with any consumer expecting the previous version

Minor versions MAY be auto-upgraded in consumer-controlled environments but MUST NOT be auto-upgraded in regulated sectors.

A MINOR release MAY introduce new optional capabilities, extensions, or fields, provided that existing behaviour, semantics, safety guarantees, and structural interfaces remain valid and interoperable for all consumers that rely only on earlier versions within the same MAJOR line. A MINOR release MUST NOT remove, redefine, or alter the meaning of existing behaviours or outputs in a way that would invalidate previously conformant integrations.

New behaviour introduced in a MINOR release MUST be strictly opt-in or non-disruptive for existing consumers. Where a change would require existing clients or platforms to modify behaviour or assumptions in order to remain conformant, it MUST NOT be published as a MINOR release.

5.4.3 MAJOR Version (X.y.z)

A major version indicates a breaking change.

Examples of breaking changes:

- removing existing input fields
- changing schema structure
- modifying behaviour semantics
- altering determinism guarantees
- changing output structure
- redefining safety behaviours
- removing fallback paths
- adding new required inputs
- changing domain restrictions in a way that invalidates previous assumptions
- changing constraints in a restrictive way that breaks compatible inputs

Major versions MUST:

- be published as a new version,
- undergo full recertification,
- never auto-upgrade,
- include migration notes (5.7).

Consumers MUST explicitly opt-in to major versions.

A MAJOR release may introduce changes that are intentionally incompatible with earlier versions, including alterations to behaviour, semantics, safety guarantees, structural interfaces, or observable outputs. Cross-major compatibility MUST NOT be assumed, and platforms are not required to support automatic interoperability between different MAJOR lines.

Where a change alters behaviour in a way that may affect existing consumers, it MUST be published as a MAJOR release rather than being introduced under a MINOR or PATCH version. Migration between MAJOR versions is an explicit lifecycle action and requires consumers to evaluate and adopt the new behaviour deliberately.

5.5 Breaking and Non-Breaking Changes (Normative)

A **breaking change** is any modification that alters the behavioural contract, schema contract, safety boundaries or determinism expectations of a capability in a way that may cause existing integrations

to fail or behave inconsistently. Breaking changes MUST result in a new major version of the capability. Examples of breaking changes include (but are not limited to):
– adding, removing or renaming required input or output fields;
– changing field types or allowed value ranges;
– altering behavioural scope or expected outcomes;
– modifying safety boundaries or fallback behaviours;
– changing ordering, tie-breaking or determinism semantics;
– removing previously declared guarantees.

A **non-breaking change** is a change that preserves the behavioural and schema contracts while extending or strengthening the capability. Non-breaking changes MUST result in a new minor or patch version. Examples include:
– adding optional fields;
– expanding documentation or examples;
– tightening safety constraints without weakening existing rules;
– improving validation metadata or provenance information.

Platforms and registries MUST treat major, minor and patch versions as distinct published artefacts and MUST NOT substitute versions implicitly.

5.6 Backwards Compatibility Rules (Normative)

A capability version is backwards compatible with its predecessor if and only if:

1. All existing inputs remain valid.
2. All existing outputs remain unchanged in structure and semantics.
3. All behaviour rules produce identical outputs for previously valid inputs.
4. No safety rule invalidates previously acceptable inputs unless explicitly permitted under patch rules.
5. No constraints invalidate previously valid input ranges.
6. No behavioural rule becomes ambiguous or reordered.
7. Schema changes introduce only optional fields, not required ones.

If any of these conditions fail, the update is NOT backwards compatible.

Backwards compatibility rules in this section are normative and define the conditions under which an upgraded capability may replace or co-execute with an earlier version without requiring changes to existing integrations. A conforming platform MUST evaluate backwards compatibility against these rules and MUST NOT assume compatibility based on implementation heuristics or inferred behavioural similarity.

Where backwards compatibility cannot be established unambiguously, the platform MUST treat the versions as incompatible for the purposes of substitution, migration, or multi-version execution.

5.7 Compatibility Classes

To support ecosystem clarity, each capability version MUST declare its compatibility class.

Compatibility classes defined in this section are normative and determine whether two capability versions may be treated as interoperable within a given execution, validation, or migration context. A conforming platform MUST apply compatibility classes consistently and MUST NOT infer compatibility between versions except where it is explicitly permitted by the class definition.

Where compatibility between two versions cannot be established unambiguously under the rules in this section, the platform MUST treat the versions as incompatible and MUST NOT substitute or co-execute them as if they were interchangeable.

5.7.1 Fully Compatible (FC)

Versions classified as Fully Compatible (FC) MUST be functionally and behaviourally substitutable within all contexts in which the earlier version is valid. A conforming platform MAY treat FC versions as interchangeable for execution, migration, substitution, and multi-version operation, provided that all existing semantics, safety guarantees, interfaces, and observable outputs remain valid without modification.

Where any behavioural, structural, or safety-relevant difference would cause an existing integration to change its outcomes, assumptions, or operational meaning, the versions MUST NOT be classified as Fully Compatible and MUST instead be evaluated under a weaker compatibility class.

- Patch changes only
- No behavioural or schema changes
- Safe for silent/automatic upgrade

5.7.2 Backwards Compatible (BC)

Versions classified as Backwards Compatible (BC) MAY introduce new optional behaviours, fields, or extensions, provided that all semantics, interfaces, and outputs relied upon by existing integrations remain valid and continue to produce equivalent outcomes. A conforming platform MUST allow earlier clients to operate unchanged against a BC version, without requiring them to adopt or understand the newly introduced behaviour.

Where a change would require existing consumers to modify assumptions, processing logic, safety handling, or interpretation in order to remain conformant, the versions MUST NOT be classified as Backwards Compatible and MUST instead be treated as Breaking for the purposes of substitution or migration.

- Minor updates with optional new behaviour
- Existing consumers remain functional
- Auto-upgrade permitted only in non-regulated contexts

5.7.3 Breaking (BR)

Versions classified as Breaking (BR) introduce behavioural, structural, or safety-relevant changes that are not compatible with existing integrations. A conforming platform MUST treat BR versions as incompatible for substitution, co-execution, or automatic migration with earlier versions, unless an explicit migration action is performed.

Platforms MUST NOT assume interoperability between BR versions and prior releases, and MUST NOT attempt to reinterpret or adapt behaviour at runtime in order to mask the effects of a breaking change.

- Major version
- Requires explicit adoption
- Requires migration
- Requires recertification

5.8 Deprecation Policy (Normative)

A capability version may be marked as deprecated.

Deprecation MUST:

- be declared in metadata.deprecated: true
- include a deprecation_notice
- include recommended upgrade path
- include a deprecation timeline

Deprecation does **not** mean removal.
Deprecated versions MUST remain functional until their **retirement date**.

Deprecation is a normative lifecycle state. When a capability version is marked as deprecated, conforming platforms MUST treat the version as supported for a limited and explicitly defined period, after which the version MUST be retired or replaced according to the rules in this specification. Deprecation MUST NOT be used as an informal advisory label or as a mechanism for indefinite, open-ended support.

During the deprecation period, the behaviour, semantics, and safety guarantees of the deprecated version MUST remain stable and MUST NOT be altered in a way that would affect existing conformant integrations. Any change that would materially affect behaviour MUST instead be delivered as a new version rather than applied to a deprecated one.

5.8.1 Deprecation Timeline Requirements

A deprecation timeline MUST specify:

- **announcement_date**

- **end_of_new_usage_date**
- **end_of_support_date**
- **retirement_date**

Typical lifecycle windows:

- 90 days for patch-level deprecations
- 180 days for minor versions
- 365+ days for major versions in regulated contexts

Platforms MUST notify dependent systems when deprecation occurs.

A deprecated capability version MUST specify a clearly defined support window, including the date on which deprecation begins and the date on which the version will be retired or cease to be valid for execution. Conforming platforms MUST honour the declared support window consistently and MUST NOT retire or disable a deprecated version prior to the published retirement date, except where required for security or safety reasons explicitly documented in the lifecycle record.

Where a deprecation period expires, the platform MUST treat the version as retired and MUST NOT permit further execution without an explicit exception mechanism defined by this specification. Where a deprecation timeline cannot be determined unambiguously, the platform MUST assume that the version is no longer supported and MUST refuse execution.

5.9 Deprecation and End-of-Life Behaviour (Normative)

A deprecated capability remains valid and executable until its declared end-of-life (EOL) date. During the deprecation period, the capability MUST continue to conform to this specification and MAY receive security or stability updates that do not alter its declared behaviour.

After the end-of-life date has passed, the capability MUST NOT be treated as conformant and MUST NOT be listed in any conformant registry or treated as eligible for execution in environments that require specification compliance.

5.10 Migration Requirements

When a breaking version (MAJOR) is published, the developer MUST include a **Migration Block** (informative but required in certification contexts).

Migration content MUST include:

- summary of breaking changes
- behavioural differences
- constraint differences
- schema differences
- safety model differences
- required consumer changes
- fallback equivalence mappings
- recommended upgrade instructions

Migration blocks MUST be human-readable, implementation-agnostic, and placed in.

Migration is an explicit lifecycle action rather than an implicit execution behaviour. A conforming platform MUST NOT migrate a capability across versions automatically or transparently unless compatibility has been established unambiguously under the rules in this specification and the migration has been explicitly requested or authorised by the operator or integrating system.

Where migration crosses a compatibility boundary — including transitions to Breaking (BR) or cross-MAJOR upgrades — the platform MUST require a deliberate migration action and MUST NOT substitute or auto-execute the newer version in place of the earlier one.

Migration operations MUST be applied atomically with respect to the capability's behavioural and structural state. A conforming platform MUST NOT allow partial migration, hybrid execution across versions, or intermediate states in which neither the source nor target version is fully valid.

Where a migration declares support for rollback, the platform MUST ensure that rollback returns the capability to a behaviourally equivalent and fully conformant instance of the original version. Where rollback cannot be guaranteed to meet this requirement, the migration MUST be treated as irreversible and MUST be explicitly identified as such in the lifecycle record.

extensions.<vendor_or_dev_namespace>.migration

5.11 Version Lineage & Provenance Tracking

Registries MUST track:

- all versions of a capability,
- the lineage graph,
- upgrade / downgrade compatibility,
- creation and modification dates,
- developer identity,
- certification state,
- deprecation status,
- retirement dates.

Capabilities MUST NOT "rewrite history":
publishing a version replaces no previous versions.

Every version remains immutable after publication.

5.12 Retirement Rules

A capability version enters **retirement** when:

- its retirement date is reached,
- or its replacement is fully adopted by consumers,
- or it fails a critical security or safety audit.

Retired versions MUST:

- be removed from active registries,
- not be available for execution,
- remain available only for audit / archival purposes.

Emergency Retirement

If a severe safety flaw is discovered:

- immediate retirement is allowed,
- all platforms MUST disable execution immediately,
- migration notices MUST be issued within 24 hours.

This rule mirrors real-world CVE emergency patch processes.

5.13 Multi-Version Execution (MVE)

Platforms MAY run multiple versions of the same capability *in parallel*.

MVE MUST follow:

- strict version pinning
- explicit selection of major versions
- automatic selection only for patch-level updates
- compatibility guarantees defined in 5.4

Platforms MUST NOT silently upgrade between major versions.

5.14 Summary of Section 5 Requirements

A capability is lifecycle-compliant only if:

- it uses semantic versioning;
- patch/minor/major categories follow the rules in 5.3;
- backwards compatibility is maintained where declared;
- deprecation and retirement follow defined timelines;
- version provenance is tracked;
- breaking changes require migration instructions;
- certification is renewed for major versions;
- platforms enforce version pinning logic.

Lifecycle management is a mandatory part of BCS conformance.

6.0 REGISTRY & MARKETPLACE REQUIREMENTS

Registry Scope Clarification (Informative).
The registry model defined in this specification relates solely to the technical integrity, provenance, validation status and lifecycle management of published capabilities. It does not define commercial

marketplace behaviour, economic models, discovery algorithms, ranking, payment flows, or distribution policies. Such considerations remain the responsibility of platforms and ecosystem operators.

This section defines the mandatory requirements for any system that:

- stores
- indexes
- distributes
- sells
- certifies
- executes
- or otherwise hosts

BCS-compliant capabilities.

These requirements apply to:

- public registries
- enterprise private registries
- platform marketplaces
- in-house capability repositories
- federated or distributed registries

Registries MUST enforce all rules in Sections 2–5, and MUST perform validation pipelines (Section 4) before a capability becomes available for execution or distribution.

6.1 Purpose of Registry Requirements

Registries exist to:

- enforce capability integrity
- ensure validated content is safely distributed
- prevent incompatible versions from circulating
- establish provenance and audit trails
- support discovery and dependency management
- enforce lifecycle rules (Section 5)
- enable certification workflows (Section 4.5)
- guarantee immutability of published versions
- facilitate enterprise governance and regulatory compliance

Registries are **not optional** components in a BCS ecosystem.

6.2 Registry Types (Normative)

Registries fall into three categories:

6.2.1 Public Registries

- operated by vendors, standards bodies, or open foundations
- must enforce the full BCS compliance stack
- must require developer identity verification
- must support certification status indicators (where certification is present)
- must apply safety classification tagging
- MUST NOT publish capabilities that fail validation or violate safety rules
- Certification MAY be required by a registry, but listing and certification are distinct processes

6.2.2 Enterprise Registries

- restricted-access internal registries
- may allow non-certified capabilities
- MUST enforce structural + behavioural + safety validation
- MUST enforce version pinning and lifecycle rules
- MUST maintain internal audit logs
- MAY override certain visibility and metadata rules for internal use

6.2.3 Local Developer Registries

- personal or team-level repositories
- MAY disable certification checks
- MAY accept non-production-safe capabilities
- MUST still enforce structural and safety validation before execution
- MUST NOT distribute unsafe capabilities beyond the local environment

6.3 Capability Submission Requirements

Before a capability may be published to a registry (public or enterprise), the following MUST be performed:

1. **Structural validation** (Section 4.2)
2. **Behavioural validation** (Section 4.3)
3. **Safety validation** (Section 4.4)
4. **Versioning compliance check** (Section 5.3)
5. **Provenance metadata verification** (Section 5.8)
6. **Signature creation or verification**
7. **Immutability lock-in**

Registries MUST reject capabilities that:

- violate any structural rule;
- contain unsafe behaviour;
- lack version information;
- downgrade safety guarantees;
- attempt to reuse version identifiers;
- attempt to publish two different files under the same (id, version) tuple.

A capability MUST NOT be listed, published, or made available for discovery or execution in any conformant registry unless it has successfully passed Structural Validation (Section 4.2), Behavioural Validation (Section 4.3), and Safety Validation (Section 4.4).

A registry MUST reject or deactivate any capability whose validation status is missing, invalid, or revoked, and MUST treat such capabilities as non-eligible for publication or distribution.

6.4 Immutable Version Storage (Normative)

Registries MUST store all published capability versions **immutably**.

This means:

- a versioned capability file MUST NOT be modified or replaced;
- metadata fields (e.g., developer name, contact) MAY be updated, but the capability file itself MUST remain byte-for-byte identical;
- the registry MUST record a cryptographic hash of the capability file;
- execution engines MUST verify the file hash before execution.

If a developer wishes to modify a capability:

- they MUST publish a new version;
- they MUST NOT overwrite or alter the existing version.

6.5 Developer Identity & Trust Requirements

Registries MUST verify developer identity through:

- email verification,
- platform authentication,
- or enterprise identity systems (SSO, OAuth),
- PLUS an optional strong identity layer for public registries (government ID, certificate signing, or equivalent).

Developer identity MUST be linked to:

- all published versions,
- all certification states,
- all lifecycle operations (deprecation, retirement).

Anonymous publication is prohibited in public registries.

6.6 Metadata Requirements

Registries MUST maintain metadata for every published capability, including:

- capability ID
- version
- publication timestamp
- developer identity
- certification status
- deprecation status
- retirement status
- cryptographic hash
- download count / usage telemetry (optional, non-normative)
- domain restrictions
- safety classification labels
- compatibility class (FC, BC, BR)

Metadata MUST be queryable, exportable, and indexable.

6.7 Search and Discovery Requirements

Registries MUST support deterministic, reproducible search over:

- capability IDs
- names
- tags
- domain categories
- safety classifications
- compatibility classes
- developer identity
- certification status
- semantic versioning ranges

Search MUST NOT return:

- deprecated versions by default,
- retired versions at all,
- unsafe capabilities,
- uncertified capabilities (unless explicitly requested).

6.8 Certification Status Indicators

Certification is an optional assurance layer that may be applied in addition to validation and listing. A capability MAY be listed without certification where permitted by the registry type, but it MUST still satisfy all validation, lifecycle, and safety requirements.

Registries MUST label each capability with one of:

- **Certified** (passed Section 4.5 audits)
- **Self-certified** (developer-attested only)
- **Uncertified** (but valid for internal/exploratory use)
- **Deprecated**
- **Retired**

- **Revoked** (removed due to safety violation)

Certified capabilities MUST list:

- certifying authority
- certificate ID
- certificate issue date
- expiry or revalidation schedule
- test suite version
- determinism audit level

6.9 Handling Unsafe or Invalid Capabilities

Registries MUST immediately reject or disable capabilities that:

- fail safety validation;
- are found nondeterministic;
- produce unsafe output under audit;
- include disallowed content categories;
- bypass fallbacks;
- violate schema rules;
- include undeclared behavioural logic;
- fail domain restriction enforcement.

When a capability is revoked:

- all dependent systems MUST be notified,
- downloads MUST be blocked,
- execution MUST be disabled,
- the registry MUST record the revocation reason.

6.10 Versioning Enforcement

Registries MUST enforce:

- immutability of published versions;
- version uniqueness per (id, version);
- semver compliance;
- deprecation timelines (Section 5.6);
- retirement policies (Section 5.9);
- explicit opt-in for breaking (BR) major versions;
- automatic upgrade only for patch-level versions (FC).

6.11 Registry API Requirements (Normative)

API Purpose Limitation (Informative).
Registry APIs exist to support capability submission, validation, publication, version control and

integrity management. They are not intended to define or constrain platform-level commercial, user-experience, or distribution workflows.

Registries MUST expose APIs for:

6.11.1 Capability Retrieval

- retrieve specific version
- retrieve latest compatible version
- retrieve metadata only
- retrieve certification status only
- retrieve migration paths

6.11.2 Search

- deterministic query interface,
- stable sort order,
- pagination,
- safety-class filtering.

6.11.3 Submission

- capability upload
- validation status retrieval
- signature / hash verification
- error logs

6.11.4 Lifecycle Management

- deprecation notices
- retirement scheduling
- revocation operations
- version pinning for enterprise accounts

All registry APIs MUST be deterministic and MUST NOT alter capability files.

6.12 Enterprise Policy Enforcement

Enterprise registries MUST support:

- restricted domain-only capability categories
- organisation-wide compatibility policies
- regulated-mode enforcement (e.g., finance/legal/health)
- behaviour whitelisting
- safety-class whitelisting
- internal certification layers
- capability execution graphs (dependency mapping)

Enterprise registries MAY:

- block uncertified capabilities
- force version pinning
- require mandatory review before publication
- enforce region-specific compliance rules
- override default visibility rules (private/public)

6.13 Multi-Registry Federation Requirements

Registries MAY federate, allowing:

- cross-registry lookup
- distributed storage
- mirroring of certified capabilities
- cross-vendor search
- trust propagation via signed metadata

Federation MUST:

- enforce identical validation rules,
- prohibit rewriting of foreign capabilities,
- propagate revocations within a defined time window,
- preserve capability hashes exactly.

6.14 Summary of Section 6 Requirements

A registry or marketplace is BCS-compliant only if:

- it validates all capabilities per Section 4;
- it enforces lifecycle policies (Section 5);
- it provides immutable storage;
- it verifies developer identity;
- it supports deterministic search and retrieval;
- it exposes required APIs;
- it handles unsafe capabilities correctly;
- it rejects invalid, nondeterministic, or unsafe capabilities;
- it maintains provenance and certification metadata;
- it respects deprecation and retirement rules.

Registries are critical trust anchors in the BCS ecosystem,
and non-compliant registries MUST NOT distribute capabilities.

7.0 APPENDICES (INFORMATIVE)

(All content in Section 7 is informative unless explicitly labelled "Normative")

Section 7 provides extended examples, recommended practices, anti-patterns, glossary definitions, diagrams, and other supplemental material that supports implementation and interpretation of BCS v1.1.

7.1 Canonical Capability Examples

This section provides full, end-to-end examples of BCS capability files illustrating all sections of the specification.

7.1.1 Simple Classification Capability Example

A minimal, deterministic classifier with explicit inputs, outputs, constraints, and safety rules.

(Example shown earlier in Section 2.11 may be reused or expanded here.)

7.1.2 Multi-Step Behaviour Capability Example

Shows:

- conditional rule selection
- fallback pathways
- extended safety triggers
- value constraints
- schema complexity
- extension blocks

Useful for implementers validating behavioural determinism.

7.1.3 Domain-Restricted Capability Example

A capability restricted to one or more regulated domains, demonstrating:

- domain enforcement
- safe failure
- non-overlapping behavioural paths
- prohibited input categories

7.1.4 Capability with Multiple Optional Inputs

Shows how optional fields, default values, and extended fallbacks interact under BCS rules.

7.2 Design Patterns for BCS Capabilities

Recommended best practices for capability authors.

7.2.1 Pattern: Strict Deterministic Routing

Use explicit conditional branches and avoid free-form inference.

Good:

- explicit rule ordering
- mutually exclusive conditions
- deterministic outcome mapping

Bad:

- ambiguous conditions
- overlapping rule predicates
- reliance on model inference

7.2.2 Pattern: Narrow Safety Boundaries

Authors should:

- aggressively constrain unsafe inputs
- restrict languages where possible
- minimise free-form generative output
- favour schema-based structured output

7.2.3 Pattern: Contractual Input/Output Shapes

Ensures:

- schema remains stable
- upstream systems easily integrate
- behavioural changes do not break compatibility
- validation is predictable

7.2.4 Pattern: Explicit Fallback Design

Fallbacks should:

- always produce deterministic outputs
- include a safe "meta" flag indicating failure cause
- avoid verbose natural-language descriptions
- appear in all major behavioural branches

7.2.5 Pattern: Constraint-Driven Behaviour

Use constraints to enforce business logic, not behaviour rules alone.

Examples:

- numeric ranges for model parameters

- max input lengths
- controlled vocabularies for fields
- mutually exclusive fields enforced structurally

7.3 Anti-Patterns (What NOT to Do)

These are common mistakes that lead to invalid or nondeterministic capabilities.

7.3.1 Anti-Pattern: Behaviour Hidden in Extensions

Extensions MUST NOT alter behaviour, safety, or schema.

7.3.2 Anti-Pattern: Undeclared Output Fields

Behaviour that emits fields not defined in the schema MUST be rejected.

7.3.3 Anti-Pattern: Using Model Inferential Logic Instead of Behaviour Rules

Capabilities relying on AI inference instead of deterministic rules will:

- produce nondeterministic output
- violate safety boundaries
- fail certification

7.3.4 Anti-Pattern: Safety as a Soft Suggestion

Safety MUST override behaviour.
Capabilities MUST NOT:

- attempt to bypass safety,
- ignore prohibited categories,
- degrade safety boundaries dynamically.

7.3.5 Anti-Pattern: Complex Conditional Graphs Without Ordering

Lack of rule ordering results in:

- nondeterminism
- ambiguous matching
- inconsistent validation
- certification failure

7.4 Behavioural Modelling Examples (Extended)

This appendix expands on Section 1 by showing:

7.4.1 Multi-Branch Behaviour Graphs

Visual diagrams depicting rule selection under different inputs.

7.4.2 Deterministic Fallback Cascades

Illustrates nested fallback behaviour and safety-trigger integration.

7.4.3 Conflict Resolution Cases

Examples where overlapping rules must be rejected at validation time.

7.4.4 Repeatability Guarantees

Examples of behaviour rules that appear deterministic but are not.

7.5 Schema & Constraints Examples (Extended)

Detailed patterns for:

- nested object schemas
- optional fields
- arrays and bounded lists
- oneOf structures
- relational constraints
- type enforcement

7.5.1 Relational Constraint Examples

e.g.

if field A exists, field B must be null

if score > 0.5, confidence must be >= score

7.5.2 Value Range Examples

Practical patterns for numeric, string-length, and categorical ranges.

7.6 Safety Examples (Extended)

Illustrates:

- how prohibited categories block inputs
- example safety triggers
- safe-failure behaviour in real execution

- domain-enforcement examples
- multi-trigger scenarios

7.6.1 Unsupported Language Example

Input provided in an unsupported language → safe fallback.

7.6.2 Dangerous Pattern Block Example

Regex-triggered safety failure and deterministic fallback mapping.

7.6.3 Output Sanitisation Example

Preventing unsafe content leakage into structured output.

7.7 Validation Walkthroughs

Step-by-step examples following Section 4:

7.7.1 Full Structural Validation Walkthrough

Start with raw JSON → detect structural and ordering errors.

7.7.2 Behavioural Validation Walkthrough

Shows conflict detection and nondeterminism rejection.

7.7.3 Safety Validation Walkthrough

Trigger activation and fallback application.

7.7.4 Certification Validation Walkthrough

Auditor review of determinism and provenance.

7.8 Registry Usage Examples

Explains registry interactions (from Section 6):

- Publishing
- Deprecation flows
- Retrieve latest compatible version
- Hash verification
- Revocation scenarios

- Federation resolution

7.9 Glossary of Terms (Normative)

(This subsection IS normative—definitions used in the spec must remain consistent.)

Key definitions include:

- **Capability**
- **Behaviour**
- **Capability Context**
- **Determinism**
- **Fallback**
- **Safety Trigger**
- **Constraint**
- **Schema**
- **Prohibited Category**
- **Domain Restriction**
- **Registry**
- **Certification**
- **Major/Minor/Patch versions**
- **Structural Validation**
- **Behavioural Validation**
- **Safety Validation**
- **Runtime Validation**
- **Compatibility Class** (FC / BC / BR)

This glossary ensures consistency across implementations.

7.10 Frequently Asked Questions (FAQ)

Provide clear, implementation-focused answers:

7.10.1 Why must behaviour be deterministic?

Because deterministic behavioural outputs are required for safety audits, regression testing, reproducibility, and regulated deployment.

7.10.2 Why can't capabilities contain model inference?

Inference is nondeterministic and violates the BCS contract.

7.10.3 Can I include natural language in fallbacks?

No. Fallbacks must be structured, deterministic outputs.

7.10.4 What if I need generative behaviour?

You must wrap generative models externally; BCS capabilities govern deterministic control flows only.

7.10.5 Do I need to certify every capability?

Certification is optional for internal use; mandatory for public distribution.

7.11 Troubleshooting

Common errors and resolutions:

7.11.1 “additionalProperties not allowed”

You are emitting or declaring extra fields not defined in the schema.

7.11.2 “ambiguous behavioural rule selection”

Two rules match the same input conditions.

7.11.3 “prohibited category detected”

Safety block prohibits the input content or domain.

7.11.4 “breaking version update rejected”

Your MAJOR version is incompatible without migration instructions.

7.12 Visual Diagrams (Informative)

Optionally include:

- end-to-end validation pipeline diagram
- behavioural rule graph
- fallback cascade diagram
- capability lifecycle timeline
- registry ingestion workflow
- certification process flow

7.13 Implementation & Ecosystem Outlook (Informative)

BCS is intended to provide a stable behavioural and safety foundation for capability-based systems while recognising that ecosystems evolve over time. Early implementations may begin with minimal or profile-based adoption and expand toward full-model conformance as validation pipelines, registries and assurance processes mature.

As platforms, enterprises and research communities gain experience with capability-based architectures, future iterations of this specification MAY refine the safety taxonomy, extend validation semantics, or introduce additional lifecycle and governance patterns, provided such changes remain consistent with the core principles established in this version of BCS.

7.14 Summary of Section 7

Section 7 provides:

- examples,
- patterns,
- anti-patterns,
- extended schemas,
- diagrams,
- glossary,
- FAQs,
- troubleshooting
- and reference material

to aid understanding and adoption of BCS v1.1.

It does not introduce new normative rules except where explicitly stated.