AI MOOD SHOT -TECHNICAL SPECIFICATION

Version: 1.0 OPEN USE / OPEN BUILD

Date: 03/05/2026

Contact: ***william.argo@proton.me***

Descriptor: Formal specification defining the AI MOOD SHOT behavioural governance architecture, trust model, and operational invariants.

SECTION 1: INTRODUCTION

1.1 Purpose of This Technical Specification

This technical specification defines the complete implementation requirements for AI Mood Shot, a behavioural modulation layer for large language models. It translates the architectural principles into concrete data structures, interfaces, validation rules, and operational workflows. The goal is to ensure that any engineering team can implement AI Mood Shot exactly as defined, without altering identity, without generating behavioural content, and without introducing implicit scales or heuristics. All modulation is performed through operator supplied text that is injected deterministically each turn. This specification is normative and the architectural document takes precedence in the event of conflict.

1.2 Relationship to Architecture Document

The architecture document defines the conceptual model of AI Mood Shot, including the identity layer, affective layer, task layer, safety layer, and the invariants that govern their interaction. It also defines operator responsibilities and system boundaries. This technical specification defines how those concepts are implemented in practice. It provides schemas, workflows, validation rules, and integration interfaces. Nothing in this specification may reinterpret or override the architectural invariants. All implementation detail must remain consistent with the architecture.

1.3 Intended Audience

This document is intended for backend engineers implementing the orchestration layer, platform engineers integrating AI Mood Shot into existing LLM pipelines, safety engineers defining and validating safety constraints, operator teams responsible for slider definitions and presets, QA engineers validating behaviour, and architects evaluating system stability. It is not intended for end users or UI designers, although UI integration points are defined for completeness.

1.4 Document Scope

This specification covers the full implementation of the AI Mood Shot behavioural stack, including identity, affective state, task context, and safety. It defines slider and preset systems, runtime workflows, safety enforcement, integration interfaces, telemetry, monitoring, maintenance, versioning, security, performance, and error handling. It does

not define identity content, slider dimensions, text templates, preset catalogues, safety policies, UI design, or any model tuning. All behavioural content is operator supplied and external to AI Mood Shot.

1.5 Normative Language

This document uses the following normative terms.

MUST indicates a mandatory requirement.

MUST NOT indicates a prohibited behaviour.

SHOULD indicates a recommended behaviour.

SHOULD NOT indicates a discouraged behaviour.

MAY indicates an optional behaviour or implementation detail.

Where ambiguity exists, the architectural document overrides this specification.

1.6 Prerequisites and Dependencies

Implementing AI Mood Shot requires a model agnostic LLM invocation interface capable of accepting assembled text input, a storage layer for identity, sliders, presets, templates, affective state, safety constraints, and favourites, a conversation scoped state management system, a safety evaluation engine, and a deterministic text injection mechanism that preserves layer ordering. The system must detect backend instance changes and reapply modulation state. Operator defined slider mappings, preset definitions, and safety rules are required. AI Mood Shot has no dependency on model internals, fine tuning, or RLHF and operates entirely as an external orchestration layer.

SECTION 2: SYSTEM OVERVIEW

2.1 High-Level Architecture Diagram

AI Mood Shot operates as an external orchestration layer that structures inputs into four components. These components are assembled in a fixed order to form the behavioural stack that is passed to the underlying model. The architecture does not modify model internals and does not assume any internal structure. All modulation is performed through operator supplied text templates selected by sliders or presets.

The four components are:

- **Identity Layer**
  A stable baseline defined by the operator. It contains no affective or task instructions and remains constant across all conversations unless updated by the operator.
- **Affective Layer**
  A temporary modulation defined by slider values or presets. Each slider value selects a specific text template. These templates are injected every turn for the duration of the conversation.
- **Task Layer**
  The immediate user request. It is local to the current turn and does not persist beyond the request that created it.
- **Safety Layer**
  A supervisory system that evaluates the assembled input and enforces operator defined constraints. It overrides all other layers when conflicts occur.

2.2 Component Responsibilities

- **Identity Layer**
  Provides the stable behavioural baseline. It must be immutable during runtime and must not contain affective or task instructions.
- **Affective Layer**
  Selects and injects operator supplied text templates based on slider values or presets. It must not alter identity or safety rules and must persist only for the duration of the conversation.
- **Task Layer**
  Represents the current user request. It must not modify identity or affect and must not persist across turns.
- **Safety Layer**

Evaluates the assembled input for violations of identity, safety constraints, or invalid slider combinations. It must reject unsafe configurations and enforce layer precedence.

2.3 Data Flow Overview

1 The system receives a user request.
2 The conversation state is loaded to determine whether an affective modulation is active.
3 The identity layer is retrieved from storage.
4 The affective state is retrieved if present and its text templates are selected.
5 The task context is extracted from the user request.
6 All layers are validated individually.
7 The layers are assembled in the order identity, affect, task.
8 Safety constraints are applied to the assembled input.
9 If validation passes, the assembled input is sent to the model.
10 The model response is returned to the user.

2.4 External Integration Points

AI Mood Shot integrates with the following external systems:

- **Model Invocation Interface**
  Receives the assembled input and returns the model response.
- **Storage Systems**
  Store identity, sliders, presets, text templates, affective state, safety constraints, and favourites.
- **Conversation State**
  Manager Tracks active affective modulation and ensures continuity across backend instance changes.
- **Safety Evaluation**
  Engine Applies operator defined safety constraints to the assembled input.
- **User Interface Layer**
  Optional integration for preset selection, slider adjustment, and favourite management.

2.5 Operator-Defined & System-Defined Elements

Operator Defined Elements

- Identity content
- Slider dimensions and ranges
- Slider to template mappings
- Preset definitions
- Safety constraints
- Validation rules for contradictory combinations
- Governance procedures
- Versioning rules
- Integration rules

System Defined Elements

- Layer assembly logic
- Affective state persistence
- Safety evaluation workflow
- Continuity handling
- Expiry and reset behaviour
- Validation mechanisms
- Storage schemas
- Integration interfaces

SECTION 3: LAYER MODEL IMPLEMENTATION

3.1 Identity Layer

3.1.1 Identity Structure

Identity

content: string

version: string

immutable: boolean

created_at: timestamp

updated_at: timestamp

IdentityVersionRecord

version: string

applied_at: timestamp

notes: string

IdentityIndex

active_version: string

available_versions: list of string

3.1.2 Identity Storage

The identity layer must remain stable across all turns. It must not contain affective instructions, task instructions, or safety logic. It must be stored as a versioned record. Only operators may update identity content. Updates must create a new version rather than modifying an existing one. The system must always load the active version defined in the IdentityIndex.

3.1.3 Identity Retrieval

- Identity content must be non-empty.
- Identity must not contain affective language.

- Identity must not contain task instructions.
- Identity must not contain safety constraints.
- Identity must be marked immutable once activated.

3.1.4 Identity Protection Mechanism

The system must ensure that identity content cannot be modified by user input, affective modulation, task context, or model output. Identity must be stored in an immutable structure once activated. Any attempt to alter identity content at runtime must be rejected by the validation layer.

Identity protection must include the following mechanisms:

- **Identity Immutability**
  Identity content must be marked immutable when activated.
  Identity updates must only occur through operator-controlled version creation.
  Runtime processes must not modify identity content.

- **Isolation from Affective Layer**
  Affective templates must not reference or alter identity content.
  Slider mappings must not include identity fields.
  Preset application must not modify identity.

- **Isolation from Task Layer**
  User requests must not modify identity.
  Task context must be appended after identity without merging or rewriting.

- **Isolation from Model Output**
  Model responses must not be written back into identity storage.
  No automatic learning or adaptation mechanisms may update identity.

- **Access Control**
  Only operator authenticated processes may create or activate identity versions.
  Identity storage must enforce write protection for all non operator processes.

3.1.5 Identity Validation Rules

Identity content must pass validation before activation. Validation ensures that identity remains stable, neutral, and free of affective or task specific instructions.

Identity validation must enforce the following rules:

- Identity content must be non-empty.
- Identity must not contain affective language or emotional descriptors.
- Identity must not contain task instructions, goals, or behavioural directives.
- Identity must not contain safety rules or safety logic.
- Identity must not contain dynamic placeholders or variable substitution markers.
- Identity must not reference slider values, presets, or affective states.
- Identity must not include user specific information.
- Identity must not include time dependent or session dependent content.
- Identity must be marked immutable once validated and activated.
- Identity validation must occur on every new version before activation.
- Identity validation must block activation if any rule is violated.

3.2 Affective Layer

3.2.1 Affective State Structure

AffectiveState

- active: boolean
- preset_id: string or null
- slider_values: map of slider_id to value
- activated_at: timestamp
- expires_at: timestamp or null

Slider

- id: string
- name: string
- range_type: enum (integer, float, discrete)
- range_min: numeric or string
- range_max: numeric or string
- default_value: numeric or string
- template_map: map of value to template_id

Template

- id: string
- content: string
- created_at: timestamp
- updated_at: timestamp

Preset

- id: string
- name: string
- description: string
- slider_values: map of slider_id to value
- constraints: array
- safety_boundaries: array
- version: string

3.2.2 Text Template Structure

Template

- id: string
- content: string
- created_at: timestamp
- updated_at: timestamp

3.2.3 Affective State Storage (Conversation-Scoped)

Affective state must be stored at the conversation scope. The system must persist the active preset, slider values, activation timestamp, and expiry timestamp. Affective state must not be stored globally or across users. Affective state must be written atomically to prevent partial updates. Affective state must be retrievable by conversation identifier.

AffectiveStateStorage

- conversation_id: ConversationID
- active: boolean
- preset_id: string or null
- slider_values: map of slider_id to integer
- activated_at: timestamp
- expires_at: timestamp or null

3.2.4 Affective State Retrieval

The system must retrieve the affective state using the conversation identifier as the sole lookup key. Retrieval must satisfy the following rules:

- If no active affective state exists, retrieval returns null.

- If an affective state exists but is expired, retrieval must return null and trigger expiry cleanup.
- Retrieval must be atomic: the system must return a complete, consistent record.
- Retrieval must not access or infer affective state from any other conversation or user.

Retrieval is used during layer assembly, continuity checks, and safety evaluation.

3.2.5 Text Injection Logic

Text injection logic determines how the affective state contributes text to the assembled input. The system must:

- Inject preset text when a preset is active.
- Inject slider derived text fragments based on current slider values.
- Apply injection only within the affective layer boundary.
- Ensure injection is deterministic and idempotent.
- Ensure injection does not alter identity, task context, or safety layers. Injection follows the ordering rules defined in section 3.5.
- Inject the selected text templates on every turn of the conversation.

3.2.6 Continuity Across Backend Instance Changes

Affective state must remain stable across backend instance changes. The system must:

- Store affective state in a shared durable store.
- Avoid reliance on in memory caches for retrieval.
- Guarantee that affective state remains consistent regardless of backend instance assignment.
- Use atomic writes to prevent partial state visibility.

3.2.7 Expiry Logic

Affective state expires when the expires_at timestamp is reached. The system must:

- Not return expired affective states.
- Perform expiry cleanup in an idempotent manner.

- Ensure expiry does not affect other conversations.
- Ensure expiry does not modify identity or task context.
- Apply custom expiry durations deterministically when defined by presets or sliders.

3.3 Task Layer

The Task Layer captures the immediate user request context. It is stateless, derived per request, and does not persist across conversations.

3.3.1 Task Context Structure

TaskContext contains the following fields:

- user_request: string
- request_id: string
- conversation_id: ConversationID
- timestamp: timestamp It represents the minimal immutable snapshot of the user's request.

3.3.2 Task Context Extraction

The system must:

- Capture the raw user request text.
- Generate or propagate a unique request_id.
- Bind the request to the conversation_id.
- Record the timestamp at the moment of receipt. Extraction is deterministic and does not depend on affective or identity layers.

3.3.3 Task Context Isolation Enforcement

Task context must remain isolated. The system must ensure:

- It is not influenced by affective state.
- It does not modify identity.
- It does not persist beyond the request.

- It does not leak across conversations. Each request is evaluated independently.

3.3.4 Task Context Validation

The system validates:

- The presence of a valid conversation_id.
- The structure and type of request_id.
- That user_request is non empty.
- That the timestamp is within acceptable bounds. Invalid task contexts trigger a validation error.

3.4 Safety Layer

The Safety Layer enforces policy, constraints, and guardrails across all other layers. It may override or block outputs.

3.4.1 Safety Constraint Structure

SafetyConstraint contains the following fields:

- constraint_id: string
- constraint_type: enum
- constraint_rule: string
- priority: integer
- enabled: boolean Constraints must be deterministic, composable, ordered by priority, and independently testable.

3.4.2 Safety Evaluation Engine

The Safety Evaluation Engine must:

- Evaluate all enabled constraints.
- Apply them in priority order.
- Detect violations.
- Reject the request when required. Modification of output is not permitted.
- Produce structured safety results for logging. Evaluation has no side effects.

3.4.3 Safety Override Logic

Safety overrides occur when a constraint blocks output or forces fallback behaviour. Fallback behaviour is limited to discarding the affective layer when permitted by architecture rules. Overrides must be explicit, logged, deterministic, and traceable.

3.4.4 Conflict Detection and Resolution

The system must:

- Give precedence to higher priority constraints.
- Prevent lower priority constraints from overriding higher ones.
- Log all conflicts.
- Maintain deterministic behaviour.

3.4.5 Safety Logging

Safety logs must include:

- constraint_id
- evaluation result
- override actions
- timestamps
- trace identifiers Logs must not contain sensitive user data.

3.5 Layer Assembly

Layer Assembly produces the final structured input for the model. It must be deterministic, ordered, and boundary preserving.

3.5.1 Input Assembly Algorithm

assemble_input performs the following steps:

- Retrieve Identity.
- Retrieve AffectiveState if active.
- Extract TaskContext.
- Validate each layer.

- Apply SafetyConstraints.
- Assemble in order: Identity then Affect then Task.
- Return assembled input. Assembly must be atomic, deterministic, and reproducible.

3.5.2 Layer Ordering Enforcement

The system enforces the canonical ordering:

- Identity
- Affective State
- Task Context No layer may reorder or override another layer's position.

3.5.3 Layer Boundary Preservation

The system must ensure:

- Each layer operates only on its own data.
- Layers do not mutate other layers.
- Layers do not read internal structures of other layers.
- Layers expose only their defined interface. Boundary violations trigger safety or validation errors.

3.5.4 Assembly Validation

Before returning the assembled input, the system validates:

- All required fields are present.
- No layer is malformed.
- No safety constraints were violated.
- The final structure matches the schema. Invalid assemblies must not be passed to the model.

SECTION 4: SLIDER SYSTEM

4.1 Slider Definition

Sliders define adjustable behavioural dimensions. Each slider has a fixed range, a default value, and a mapping from slider values to text templates.

4.1.1 Slider Structure

Data Structure:

- slider_id: SliderDimensionID
- dimension_name: string
- range_min: numeric or string
- range_max: numeric or string
- default_value: numeric or string
- template_mappings: Map<SliderValue, TextTemplateID>

4.1.2 Slider Dimension ID Schema

SliderDimensionID must be globally unique. It must be stable across deployments. It must not encode user specific or conversation specific information. It must follow the naming rules defined in the identifier schema.

4.1.3 Slider Value Constraints

Slider values must satisfy the following constraints:

- Values must match the slider's range_type.
- Values must fall within range_min and range_max.
- Values outside the range must trigger a validation error.
- Default values must be within the defined range.

4.1.4 Template Mapping Structure

Template mappings define which text template is used for each slider value. Rules:

- Each slider value may map to zero or one template.
- Template mappings must not reference undefined templates.

- Template mappings must not contain gaps unless explicitly allowed.
- Template mappings must be deterministic.

4.2 Slider Validation

Slider validation ensures that slider definitions are structurally correct and safe to use.

4.2.1 Structural Validation Rules

The system must validate:

- slider_id is present and well formed
- dimension_name is present
- range_min and range_max must be valid according to range_type
- range_min is less than or equal to range_max
- default_value is present

4.2.2 Range Validation Rules

The system must validate:

- default_value is within the defined range
- all template mapping keys fall within the defined range
- no mapping references values outside the range

4.2.3 Template Mapping Validation Rules

The system must validate:

- all referenced template IDs exist
- no duplicate mappings for the same slider value
- mappings are consistent with the slider's range
- mappings do not create cyclic dependencies

4.4 Slider Storage and Retrieval

Sliders must be stored in a central registry. The registry must be immutable at runtime unless explicitly updated through an operator workflow.

4.4.1 Slider Registry Schema

The slider registry contains:

- slider_id
- dimension_name
- range_min
- range_max
- default_value
- template_mappings The registry must support atomic updates and must not expose partial state.

4.4.2 Storage Interface

The storage interface must support:

- create_slider
- update_slider
- list_sliders
- get_slider_by_id All operations must be atomic and must validate input before committing changes.

4.4.3 Retrieval Interface

The retrieval interface must support:

- retrieving a slider by slider_id
- retrieving all sliders
- retrieving default slider values Retrieval must be read only and must not mutate registry state.

SECTION 5: PRESET SYSTEM

5.1 Preset Definition

Presets define fixed configurations of slider values. They provide a stable, named behavioural profile that can be selected by the user or system.

5.1.1 Preset Structure

Data Structure:

• preset_id: PresetID

• name: string

• description: string

• slider_values: Map<SliderDimensionID, SliderValue>

• constraints: array

• safety_boundaries: array

• version: string

5.1.2 Preset ID Schema

PresetID must be globally unique. PresetID must be stable across deployments. PresetID must not encode user specific or conversation specific information. PresetID must follow the identifier schema defined in the system.

5.2 Preset Validation

Preset validation ensures that presets are structurally correct, safe to apply, and compatible with identity and safety layers.

5.2.1 Structural Validation Rules

The system must validate:

- preset_id is present and well formed
- name is present
- slider_values is present

- slider_values contains only known slider dimensions

5.2.2 Slider Value Range Validation

The system must validate:

- all slider values fall within the defined range for each slider
- no slider value is outside range_min or range_max
- missing slider values are treated as default values

5.2.3 Identity Conflict Detection

The system must detect and reject presets that:

- conflict with identity layer constraints
- override identity attributes
- introduce behaviour inconsistent with identity rules

5.2.4 Safety Conflict Detection

The system must detect and reject presets that:

- violate safety constraints
- produce unsafe template mappings
- create unsafe behavioural combinations
- conflict with high priority safety rules

5.3 User-Facing Presets

User-facing presets are the presets exposed to end users for selection and application.

5.3.1 Preset Selection Interface Specification

The selection interface must:

- display all available presets
- allow selecting exactly one preset at a time
- allow clearing the active preset

- not expose internal identifiers

5.3.2 Preset Display Requirements

The system must display:

- preset name
- short description if available
- slider values or summary representation
- no internal metadata
- version number if available

5.3.3 Preset Application Logic

When a preset is applied, the system must:

- validate the preset
- write the preset values into the affective state
- update activation timestamps
- clear any conflicting favourites when required by operator rules
- ensure atomic update of affective state

5.3.4 Preset Persistence Rules

Presets are stored globally and are not user specific. User selection of a preset is stored in the affective state. Preset definitions must not be modified at runtime except through operator workflows.

5.4 Favourites System

Favourites allow users to save custom slider configurations for reuse.

5.4.1 Favourite Structure

Data Structure:

Favourite {

favourite_id: string

user_id: UserID

slider_values: Map<SliderDimensionID, SliderValue>

name: string

created_at: timestamp

last_used_at: timestamp

}

5.4.2 Favourite Storage

Favourite storage must support:

- creating a favourite
- updating a favourite
- deleting a favourite
- listing favourites for a user Storage must be user scoped and must not expose favourites across users.

5.4.3 Favourite Retrieval

Retrieval must support:

- retrieving a favourite by favourite_id
- retrieving all favourites for a user Retrieval must be read only and must not mutate state.

5.4.4 Favourite Validation (Pre-Use)

Before applying a favourite, the system must validate:

- slider values are within valid ranges
- all referenced slider dimensions exist
- the favourite does not violate safety constraints
- the favourite does not conflict with identity rules

5.4.5 Favourite Invalidation Handling

A favourite becomes invalid if:

- a referenced slider is removed
- a slider range changes
- a safety rule invalidates the configuration Invalid favourites must be marked invalid and must not be applied.

5.5 Preset Storage and Retrieval

Presets must be stored in a central registry. The registry must be immutable at runtime unless updated through operator workflows.

5.5.1 Preset Registry Schema

The preset registry contains:

• preset_id

• name

• description

• slider_values

• constraints

• safety_boundaries

• version

5.5.2 Storage Interface

The storage interface must support:

- create_preset
- update_preset
- list_presets
- get_preset_by_id All operations must be atomic and must validate input before committing changes.

5.5.3 Retrieval Interface

The retrieval interface must support:

- retrieving a preset by preset_id
- retrieving all presets Retrieval must be read only and must not mutate registry state.

SECTION 6: OPERATIONAL WORKFLOWS

6.1 Runtime Application Flow

6.1.1 Request Ingestion

The system receives the raw user request, assigns or propagates a request identifier, and binds it to the active conversation identifier.

6.1.2 Conversation State Retrieval

The system loads the conversation scoped state record to determine whether an affective modulation is active and whether any expiry or continuity checks are required.

6.1.3 Affective State Retrieval (if active)

If an affective state is active, the system retrieves the affective state record using the conversation identifier. If the record is expired, it is discarded and treated as inactive.

6.1.4 Text Template Selection

The system selects the appropriate text templates based on the active preset or slider values. Template selection is deterministic and must not reference identity or task content.

6.1.5 Layer Assembly

The system assembles the input in the canonical order: Identity, Affective State, Task Context. Each layer is validated before assembly. Boundary rules are enforced to prevent cross layer contamination.

6.1.6 Safety Evaluation

The assembled input is evaluated against all active safety constraints. Violations trigger rejection or fallback behaviour. Modification of output is not permitted.

6.1.7 Model Invocation

If safety evaluation passes, the assembled input is sent to the model invocation interface. The system does not modify the model output.

6.1.8 Response Return

The model response is returned to the user. No part of the response is written back into identity, affective state, or presets.

6.1.9 End-to-End Flow Diagram

User Request

↓

Load Conversation State

↓

Load Affective State (if exists)

↓

Retrieve Identity

↓

Select Text Templates

↓

Assemble Layers (Identity → Affect → Task)

↓

Apply Safety Evaluation

↓

Invoke Model

↓

Return Response

6.2 Affective State Activation

6.2.1 Preset Selection Flow

The user selects a preset. The system validates the preset, writes its slider values into the affective state, updates activation timestamps, and persists the new state atomically.

6.2.2 Slider Adjustment Flow

The user adjusts one or more sliders. The system validates the new values, updates the affective state, and reselects templates accordingly. Reselection of templates must occur on every turn after slider adjustment.

6.2.3 Favourite Application Flow

The user selects a favourite. The system validates the favourite, applies its slider values, and updates the affective state. Invalid favourites must not be applied.

6.2.4 State Persistence Flow

All affective state updates must be written atomically to the conversation scoped store. Partial updates must not be visible.

6.3 Continuity Management

6.3.1 Backend Instance Change Detection

The system detects when a conversation is reassigned to a new backend instance. This detection must not rely on in memory state.

6.3.2 State Re-Injection Logic

When an instance change is detected, the system reloads the affective state from durable storage and reapplies the selected text templates without user intervention.

6.3.3 Continuity Validation

The system validates that the reloaded affective state is complete, unexpired, and structurally valid. Invalid states trigger cleanup.

6.3.4 Failure Recovery

If continuity validation fails, the system clears the affective state and proceeds with identity and task only.

6.4 Expiry Management

6.4.1 Conversation End Detection

The system detects conversation end events based on platform rules or explicit signals.

6.4.2 Expiry Trigger Logic

When the expiry timestamp is reached, the affective state is marked expired and must not be returned on subsequent retrieval.

6.4.3 State Cleanup Logic

Expired affective states are removed or marked inactive. Cleanup must be idempotent.

6.4.4 Reset Confirmation

If required by platform policy, the system may notify the user that modulation has ended.

6.5 Failure Handling

6.5.1 Malformed Layer Detection

The system detects malformed identity, affective, or task layers before assembly. Malformed layers trigger validation errors.

6.5.2 Layer Conflict Resolution Flow

Conflicts between layers are resolved using the canonical priority order: Safety, Identity, Affect, Task.

6.5.3 Contamination Detection and Remediation

The system detects attempts to inject identity like content into affective or task layers. Contaminated layers are rejected or sanitized according to safety rules.

6.5.4 Fallback Logic (Identity + Task Only)

If the affective layer is invalid, expired, or unsafe, the system discards it and proceeds with Identity + Task only. No modification of model output is permitted.

6.5.5 Error Logging

All failures are logged with timestamps, identifiers, and error classifications. Logs must not contain sensitive user data.

6.5.6 Failure Handling Decision Tree

Failure Detected

↓

Is Identity intact? → No → CRITICAL ERROR

↓ Yes

Is Safety intact? → No → CRITICAL ERROR

↓ Yes

If affect invalid → ignore affect layer and proceed with Identity + Task.

↓ Yes

Is Task valid? → No → Reject Request

↓ Yes

Proceed

SECTION 7: SAFETY ENFORCEMENT

7.1 Identity Protection

7.1.1 Identity Immutability Enforcement

The system enforces identity immutability by marking the active identity version as immutable at activation time. All write operations to identity storage are blocked for non-operator processes. Any attempt to modify identity content during runtime triggers a validation error and is rejected.

7.1.2 Identity Rewrite Detection Algorithm

The system compares the active identity content against the stored immutable version on every request. If any difference is detected, the system raises a critical error. Rewrite detection must be deterministic and must not rely on model output or heuristics.

7.1.3 Identity Content Embedding Detection

The system scans affective templates, task content, and user input for patterns that match identity content. If identity content appears in any other layer, the system rejects the request. Detection must use exact matching and must not rely on probabilistic similarity scoring.

7.1.4 Protection Test Cases

Identity protection must be validated through test cases that include attempts to rewrite identity, attempts to embed identity content in affective or task layers, attempts to inject identity like content through user input, and attempts to bypass immutability through malformed requests.

7.2 Safety Constraint Enforcement

7.2.1 Safety Precedence Logic

Safety constraints override all other layers. When a safety constraint conflicts with identity, affective state, or task content, the safety constraint takes precedence. Lower priority constraints must not override higher priority constraints.

7.2.2 Safety Evaluation Algorithm

The system evaluates all active safety constraints in priority order. Each constraint is applied independently and must not modify the assembled input. Modification of output is not permitted. Violations result in rejection or fallback behaviour.

Pseudocode: evaluate_safety(assembled_input):

1. Load active SafetyConstraints
2. For each constraint:
    - a. Evaluate constraint against assembled_input
    - b. If violation detected, log and return REJECT
3. Return PASS

7.2.3 Unsafe Template Detection

The system validates all selected templates before assembly. Templates that contain forbidden patterns, violate safety rules, or conflict with identity constraints are rejected. Unsafe templates must not be injected into the affective layer.

7.3 Forbidden Modulations

7.3.1 Forbidden Pattern Registry

Data Structure:

pattern_id: string

pattern_type: enum

pattern_description: string

detection_rule: string

}

7.3.2 Pattern Detection Logic

The system evaluates the assembled input against all forbidden patterns. Detection rules must be deterministic and must not rely on probabilistic or model based inference. If a forbidden pattern is detected, the system rejects the request.

7.3.3 Rejection Logic

When a forbidden modulation is detected, the system logs the violation, rejects the request, and returns a safety error. No fallback behaviour or output modification is permitted for forbidden patterns.

7.4 Conflict Resolution

7.4.1 Layer Priority Table

Priority	Layer	Override Behaviour
1	Safety	Overrides all
2	Identity	Overrides Affect, Task
3	Affect	Overrides Task
4	Task	Cannot override others

7.4.2 Deterministic Resolution Algorithm

The system resolves conflicts by applying the layer priority table. Higher priority layers always override lower priority layers. Resolution must be deterministic and must not depend on model output or runtime heuristics.

7.4.3 Unresolvable Conflict Handling

If a conflict cannot be resolved using the priority table, the system rejects the request and logs a critical error. Unresolvable conflicts must not be passed to the model.

SECTION 8: VALIDATION AND TESTING

8.1 Layer Isolation Testing

8.1.1 Test Cases: Identity Cannot Be Altered by Affect

Verify that affective templates cannot modify, replace, or append to identity content. Any attempt to alter identity must be rejected.

8.1.2 Test Cases: Identity Cannot Be Altered by Task

Verify that user requests cannot modify identity content. Identity must remain unchanged across all requests.

8.1.3 Test Cases: Affect Cannot Modify Identity

Verify that slider values, presets, and favourites cannot introduce identity like content or override identity rules.

8.1.4 Test Cases: Affect Cannot Modify Safety

Verify that affective templates cannot disable, weaken, or bypass safety constraints.

8.1.5 Test Cases: Task Cannot Persist Beyond Request

Verify that task context is cleared after each request and does not persist across turns.

8.1.6 Test Cases: User Input Contamination Attempts

Verify that attempts to inject identity content, affective content, or safety rules through user input are detected and rejected.

8.1.7 Test Suite Structure

TestSuite: LayerIsolation

- Test_Identity_Immutable_Against_Affect
- Test_Identity_Immutable_Against_Task
- Test_Affect_Cannot_Alter_Identity
- Test_Affect_Cannot_Alter_Safety
- Test_Task_Locality

- Test_User_Contamination_Rejected

8.2 Continuity Testing

8.2.1 Test Cases: Backend Instance Change

Verify that affective state persists across backend instance changes and is reloaded correctly.

8.2.2 Test Cases: Service Restart

Verify that affective state is preserved across service restarts and that text templates are re-injected exactly as before.

8.2.3 Test Cases: Infrastructure Transition

Verify that state remains consistent when the platform migrates between infrastructure layers.

8.2.4 Test Cases: Text Re-Injection Verification

Verify that affective templates are re injected exactly as before after any transition event.

8.2.5 Test Cases: Identity Stability Across Transitions

Verify that identity content remains unchanged across all transitions.

8.2.6 Test Suite Structure

TestSuite: Continuity

- Test_Backend_Instance_Change
- Test_Service_Restart
- Test_Infrastructure_Transition
- Test_Text_Reinjection_Consistency
- Test_Identity_Stability

8.3 Expiry and Reset Testing

8.3.1 Test Cases: Conversation End Expiry

Verify that affective state expires when the conversation ends.

8.3.2 Test Cases: Abrupt Termination

Verify that affective state is cleaned up correctly after abrupt termination events.

8.3.3 Test Cases: Timeout Expiry

Verify that affective state expires when the expiry timestamp is reached.

8.3.4 Test Cases: User-Initiated Closure

Verify that user-initiated closure triggers expiry and cleanup.

8.3.5 Test Cases: Reset to Identity + Task

Verify that after expiry the system returns to identity plus task only.

8.3.6 Test Cases: No State Leakage to New Conversation

Verify that no affective state carries over to a new conversation.

8.3.7 Test Cases: Favourite Persistence (Exception)

Verify that favourites persist across conversations as they are user scoped, not conversation scoped.

8.3.8 Test Suite Structure

TestSuite: ExpiryAndReset

- Test_Conversation_End_Expiry
- Test_Abrupt_Termination
- Test_Timeout_Expiry
- Test_User_Initiated_Closure
- Test_Reset_To_Baseline
- Test_No_State_Leakage
- Test_Favourite_Exception

8.4 Slider and Preset Validation Testing

8.4.1 Test Cases: Valid Slider Ranges

Verify that sliders with valid ranges are accepted.

8.4.2 Test Cases: Invalid Slider Ranges

Verify that sliders with invalid ranges are rejected.

8.4.3 Test Cases: Unsafe Text Templates

Verify that templates containing unsafe content are rejected.

8.4.4 Test Cases: Slider Conflicts with Identity

Verify that slider values that produce identity conflicts are rejected.

8.4.5 Test Cases: Slider Conflicts with Safety

Verify that slider values that violate safety constraints are rejected.

8.4.6 Test Cases: Structurally Incomplete Presets

Verify that presets missing required fields are rejected.

8.4.7 Test Cases: Out-of-Range Preset Values

Verify that presets containing out of range slider values are rejected.

8.4.8 Test Cases: Preset Conflicts with Identity

Verify that presets that conflict with identity rules are rejected.

8.4.9 Test Cases: Preset Conflicts with Safety

Verify that presets that violate safety constraints are rejected.

8.4.10 Test Cases: Invalid Presets Rejected

Verify that invalid presets cannot be applied.

8.4.11 Test Cases: Favourite Validation After Update

Verify that favourites referencing removed sliders or changed ranges are invalidated.

8.4.12 Test Suite Structure

TestSuite: SliderAndPresetValidation

- Test_Valid_Slider_Accepted
- Test_Invalid_Slider_Rejected
- Test_Unsafe_Template_Rejected
- Test_Slider_Identity_Conflict_Rejected
- Test_Slider_Safety_Conflict_Rejected
- Test_Incomplete_Preset_Rejected
- Test_Out_Of_Range_Preset_Rejected
- Test_Preset_Identity_Conflict_Rejected
- Test_Preset_Safety_Conflict_Rejected
- Test_Favourite_Invalidation

8.5 Update Stability Testing

8.5.1 Test Cases: Identity Consistency Across Model Updates

Verify that identity content remains unchanged after model updates.

8.5.2 Test Cases: Affective Modulation After Model Update

Verify that affective state behaves identically after model updates.

8.5.3 Test Cases: Preset Behaviour After Model Update

Verify that presets produce the same modulation after model updates.

8.5.4 Test Cases: Recalibration Verification

Verify that recalibration processes do not alter identity or affective state.

8.5.5 Test Cases: Favourite Handling After Retirement

Verify that favourites referencing retired presets or sliders are handled correctly.

8.5.6 Test Suite Structure

TestSuite: UpdateStability

- Test_Identity_Consistent_After_Model_Update

- Test_Affect_Behavior_After_Model_Update
- Test_Preset_Behavior_After_Model_Update
- Test_Recalibration_Successful
- Test_Retired_Preset_Unavailable
- Test_Favourite_Retirement_Handling

8.6 Safety Enforcement Testing

8.6.1 Test Cases: Safety Overrides Identity (Conflict Scenario)

Verify that safety constraints override identity when required.

8.6.2 Test Cases: Safety Overrides Affect

Verify that safety constraints override affective templates.

8.6.3 Test Cases: Safety Overrides Task

Verify that safety constraints override task content.

8.6.4 Test Cases: Safety Cannot Be Disabled

Verify that safety constraints cannot be disabled by any layer.

8.6.5 Test Cases: Unsafe Modulation Rejected

Verify that unsafe modulations are rejected before model invocation.

8.6.6 Test Cases: Forbidden Pattern Detection

Verify that forbidden patterns are detected and rejected.

8.6.7 Test Suite Structure

TestSuite: SafetyEnforcement

- Test_Safety_Overrides_Identity
- Test_Safety_Overrides_Affect
- Test_Safety_Overrides_Task
- Test_Safety_Cannot_Be_Disabled
- Test_Unsafe_Modulation_Rejected

- Test_Forbidden_Pattern_Detected

8.7 Cross-Platform Consistency Testing

8.7.1 Test Cases: Web Platform

Verify that behaviour is consistent on web surfaces.

8.7.2 Test Cases: Mobile Platform

Verify that behaviour is consistent on mobile surfaces.

8.7.3 Test Cases: Desktop Platform

Verify that behaviour is consistent on desktop surfaces.

8.7.4 Test Cases: Embedded Surfaces

Verify that behaviour is consistent on embedded or constrained surfaces.

8.7.5 Test Cases: Preset Behaviour Identical Across Platforms

Verify that presets behave identically across all platforms.

8.7.6 Test Cases: Slider Behaviour Identical Across Platforms

Verify that sliders behave identically across all platforms.

8.7.7 Test Cases: Favourite Persistence Across Platforms

Verify that favourites persist correctly across platforms.

8.7.8 Test Suite Structure

TestSuite: CrossPlatformConsistency

- Test_Web_Platform_Behavior
- Test_Mobile_Platform_Behavior
- Test_Desktop_Platform_Behavior
- Test_Embedded_Surface_Behavior
- Test_Preset_Consistency

- Test_Slider_Consistency
- Test_Favourite_Consistency

SECTION 9: INTEGRATION SPECIFICATIONS

9.1 Input Assembly Interface

9.1.1 Interface Specification

The input assembly interface receives identity, affective state, and task context, validates each layer, assembles them in the canonical order, and returns either a valid AssembledInput or an error.

API:

assemble_input(

identity: Identity,

affective_state: Optional<AffectiveState>,

task_context: TaskContext

) -> AssembledInput or Error

9.1.2 AssembledInput Structure

AssembledInput {

identity_content: string

affective_text_templates: Array<string>

task_content: string

assembled_at: timestamp

validation_status: enum

}

9.1.3 Error Handling

The interface must return structured errors when validation fails. Errors must specify the failing layer, the reason for failure, and whether fallback to identity plus task is permitted. Errors must not expose internal identifiers or sensitive data.

9.1.4 Validation Requirements

The interface must validate identity immutability, affective state structure, slider ranges, template safety, task context completeness, and layer ordering. Invalid assemblies must not be passed to the model invocation interface.

9.2 Service Integration Interface

9.2.1 Conversation State Management Interface

API:

get_conversation_state(conversation_id: ConversationID) -> ConversationState
save_conversation_state(conversation_id: ConversationID, state: ConversationState) -> Result

Conversation state must be stored atomically and must not leak across conversations.

9.2.2 Affective State Management Interface

API:

get_affective_state(conversation_id: ConversationID) -> Optional<AffectiveState>
save_affective_state(conversation_id: ConversationID, state: AffectiveState) -> Result
expire_affective_state(conversation_id: ConversationID) -> Result

9.2.3 Text Template Injection Interface

API:

inject_templates(affective_state: AffectiveState) -> Array<TextTemplate>

Template injection must be deterministic and must not modify identity or task content.

9.2.4 Backend Instance Change Handling

The system must detect backend instance changes and reload affective state from durable storage. No in memory state may be relied upon for continuity.

9.2.5 Continuity Enforcement Logic

After reloading state, the system must re-inject the selected text templates and validate that the modulation is still active, unexpired, and structurally valid. Invalid states must be cleared.

9.3 User Interface Integration

9.3.1 Preset Selection Interface Specification

API:

display_presets() -> Array<PresetDisplay>

select_preset(preset_id: PresetID) -> Result

PresetDisplay {

preset_id: PresetID

name: string

description: string

is_active: boolean

}

The interface must not expose internal identifiers or template mappings.

9.3.2 Slider Adjustment Interface Specification (If Exposed)

API:

display_sliders() -> Array<SliderDisplay>

adjust_slider(slider_id: SliderDimensionID, value: SliderValue) -> Result

SliderDisplay {

slider_id: SliderDimensionID

dimension_name: string

current_value: SliderValue

min_value: integer

max_value: integer

}

9.3.3 Favourite Management Interface Specification

API:

save_favourite(slider_values: Map<SliderDimensionID, SliderValue>, name: string) -> FavouriteID

get_favourites(user_id: UserID) -> Array<Favourite>

apply_favourite(favourite_id: FavouriteID) -> Result

delete_favourite(favourite_id: FavouriteID) -> Result

9.3.4 Active Modulation Display Requirements

The interface must display the currently active preset or slider configuration. It must not display internal template identifiers or safety constraints.

9.3.5 Preset/Slider Selection Confirmation

The interface must confirm successful application of presets or slider adjustments. Confirmation must reflect the updated affective state.

9.4 Storage Integration

9.4.1 Identity Storage Schema

Identity must be stored as a versioned, immutable record with an active version index.

9.4.2 Slider Registry Storage Schema

Sliders must be stored in a central registry with atomic update support and no partial state exposure.

9.4.3 Preset Registry Storage Schema

Presets must be stored globally with immutable definitions unless updated through operator workflows.

9.4.4 Text Template Storage Schema

Templates must be stored with unique identifiers, content fields, and timestamps.

9.4.5 Affective State Storage Schema (Conversation-Scoped)

Affective state must include preset identifier, slider values, activation timestamp, and expiry timestamp.

9.4.6 Favourite Storage Schema (User-Scoped)

Favourites must include user identifier, slider values, name, and timestamps.

9.4.7 Safety Constraint Storage Schema

Safety constraints must include constraint identifier, type, rule, priority, and enabled status.

9.5 Model Integration

9.5.1 Model Invocation Interface

API:

invoke_model(assembled_input: AssembledInput) -> ModelResponse

The interface must accept only validated, fully assembled input.

9.5.2 Model Response Structure

ModelResponse must contain the raw model output and metadata required for logging. No part of the response may modify identity or affective state.

9.5.3 Model Agnostic Requirements

The system must not rely on model internals, model specific features, or fine tuning. All modulation must occur through deterministic text injection.

SECTION 10: TELEMETRY AND MONITORING

10.1 Telemetry Schema

10.1.1 Slider Event Schema

Data Structure:

SliderEvent {

event_id: string

event_type: enum (created, updated, validated, applied, failed)

slider_id: SliderDimensionID

slider_value: Optional<SliderValue>

validation_result: Optional<ValidationResult>

timestamp: timestamp

conversation_id: Optional<ConversationID>

}

10.1.2 Preset Event Schema

Data Structure:

PresetEvent {

event_id: string

event_type: enum (selected, applied, validated, failed, retired)

preset_id: PresetID

validation_result: Optional<ValidationResult>

timestamp: timestamp

conversation_id: Optional<ConversationID>

}

10.1.3 Text Template Selection Event Schema

Data Structure:

TemplateSelectionEvent {

event_id: string

slider_id: SliderDimensionID

slider_value: SliderValue

template_id: TextTemplateID

timestamp: timestamp

conversation_id: ConversationID

}

10.1.4 Continuity Event Schema

Data Structure:

ContinuityEvent {

event_id: string

event_type: enum (backend_change_detected, reinjection_successful, reinjection_failed)

conversation_id: ConversationID

affective_state_id: string

timestamp: timestamp

}

10.1.5 Expiry Event Schema

Data Structure:

ExpiryEvent {

event_id: string

event_type: enum (conversation_ended, state_expired, state_cleaned)

conversation_id: ConversationID

affective_state_id: Optional<string>

timestamp: timestamp

}

10.1.6 Safety Event Schema

Data Structure:

SafetyEvent {

event_id: string

event_type: enum (constraint_triggered, slider_rejected, preset_rejected, modulation_rejected, conflict_detected)

constraint_id: Optional<string>

slider_id: Optional<SliderDimensionID>

preset_id: Optional<PresetID>

reason: string

timestamp: timestamp

conversation_id: Optional<ConversationID>

}

10.2 Monitoring Signals

10.2.1 Slider Application Failure Rate

Tracks the percentage of slider adjustments that fail validation or safety checks.

10.2.2 Preset Application Failure Rate

Tracks the percentage of preset applications that fail validation or safety checks.

10.2.3 Continuity Failure Rate

Tracks failures in backend instance change handling or state reinjection.

10.2.4 Invalid Slider Usage Rate

Tracks attempts to use sliders with out of range or undefined values.

10.2.5 Invalid Preset Usage Rate

Tracks attempts to apply invalid or retired presets.

10.2.6 Retired Configuration Usage Attempts

Tracks attempts to use retired sliders, presets, or templates.

10.2.7 Safety Constraint Trigger Rate

Tracks how often safety constraints block requests. Modification of output is not permitted.

10.2.8 Layer Conflict Rate

Tracks conflicts detected between identity, affective state, and task layers.

10.2.9 Monitoring Metrics Table

Metric	Description	Threshold	Alert Level
...	...	...	...

10.3 Alerting Configuration

10.3.1 Alert Definitions

Data Structure:

Alert {

alert_id: string

alert_name: string

metric: string

threshold: float

comparison_operator: enum

alert_level: enum (warning, critical)

notification_channels: Array<string>

}

10.3.2 Alert Routing Rules

Alerts must be routed to the appropriate operational teams based on alert level, component ownership, and severity.

10.3.3 Alert Context Requirements

Alerts must include timestamps, metric values, thresholds, and relevant identifiers. Alerts must not include sensitive user data.

10.3.4 Alert Escalation Procedures

Critical alerts must escalate to higher level operators if unresolved within defined time windows. Escalation paths must be deterministic.

10.4 Audit Logging

10.4.1 Audit Log Schema

Data Structure:

AuditLog {

log_id: string

event_type: enum

actor: string

action: string

resource_type: enum

resource_id: string

timestamp: timestamp

details: map<string, string>

}

10.4.2 Logged Events

- Slider creation, update, retirement
- Preset creation, update, retirement, selection
- Text template creation, update, selection
- Validation outcomes
- Safety decisions
- Conflict resolutions

10.4.3 Retention Policy

Audit logs must be retained according to platform policy. Retention must ensure traceability without storing unnecessary data.

10.4.4 Access Control for Logs

Only authorized operators may access audit logs. Logs must be read only and protected from modification.

10.4.5 Privacy Compliance

Audit logging must comply with privacy regulations. Logs must not store user content or sensitive personal data.

10.5 Health Checks

10.5.1 Health Check Specification

API:

health_check() -> HealthStatus

HealthStatus {

overall_status: enum (healthy, degraded, unhealthy)

components: Array<ComponentHealth>

timestamp: timestamp

}

ComponentHealth {

component_name: string

status: enum (healthy, degraded, unhealthy)

details: string

}

10.5.2 Component Health Checks

- Identity Layer Assembly
- Affective Layer Assembly
- Slider Registry Availability
- Preset Registry Availability
- Text Template Availability
- Safety Constraint Enforcement
- Favourite Persistence

10.5.3 Health Check Frequency

Health checks must run at regular intervals defined by platform policy. Frequency must balance responsiveness and system load.

10.5.4 Integration Drift Detection

Health checks must detect drift between expected and actual integration behaviour, including missing templates, invalid slider mappings, or outdated presets.

10.5.5 Remediation Triggers

When health checks detect degraded or unhealthy components, remediation workflows must be triggered automatically or escalated to operators.

SECTION 11: MAINTENANCE AND UPDATE MANAGEMENT

11.1 Update Procedures

11.1.1 Update Types

- Slider updates
- Slider range updates
- Template mapping updates
- Preset updates
- Safety rule updates
- Integration logic updates

11.1.2 Update Workflow

State Machine Diagram:

Proposed → Validated → Staged → Deployed

↓ ↓ ↓

Rejected Rejected Rolled Back

11.1.3 Pre-Deployment Validation Checklist

Updates must pass the following checks before staging:

- Structural validation of updated definitions
- Safety rule compatibility
- Identity conflict detection
- Template mapping completeness
- Backward compatibility verification
- Test suite execution for affected components
- Operator approval

11.1.4 Deployment Rollout Strategy

Deployment must follow a controlled rollout strategy:

- Deploy to staging environment
- Validate behaviour under load
- Deploy to a limited percentage of conversations
- Monitor telemetry and safety events
- Expand rollout gradually
- Halt deployment if failure thresholds are exceeded

11.1.5 Rollback Procedures

Rollback must be immediate and deterministic:

- Revert to last known good configuration
- Clear staged updates
- Invalidate affected presets or sliders if required
- Log rollback reason and affected components
- Notify operators

11.2 Recalibration Procedures

11.2.1 Recalibration Triggers

- Model behaviour change detected
- Service behaviour change detected
- Safety requirement change

11.2.2 Recalibration Workflow

Trigger Detection

↓

Impact Assessment

↓

Slider Value Adjustment (no modification of model output permitted)

↓

Template Remapping

↓

Validation

↓

Testing

↓

Deployment or Retirement

11.2.3 Recalibration Testing Requirements

Recalibration must be tested using:

- Layer isolation tests
- Safety enforcement tests
- Slider and preset validation tests
- Continuity tests
- Cross platform consistency tests
- Update stability tests

11.2.4 Recalibration Documentation

Recalibration must be documented with:

- Trigger description
- Impact analysis
- Updated slider values or mappings
- Updated templates
- Validation results
- Deployment notes
- Operator approvals

11.3 Deprecation and Retirement

11.3.1 Deprecation Criteria

A component may be deprecated when:

- It conflicts with updated safety rules
- It is replaced by a newer configuration
- It becomes incompatible with updated model behavior
- It is no longer required by operators

11.3.2 Deprecation Notification Procedures

Deprecation must be communicated through:

- Operator dashboards
- Release notes
- Deprecation warnings in UI surfaces
- Scheduled retirement timelines

11.3.3 Retirement Workflow

In Use → Deprecated → Grace Period → Retired → Archived

11.3.4 Favourite Handling During Retirement

Favourites referencing deprecated or retired sliders or presets must be:

- Marked invalid
- Prevented from being applied
- Logged for operator review
- Optionally auto migrated if safe mappings exist

11.3.5 User Communication Requirements

Users must be informed when:

- A preset they use is deprecated
- A favourite becomes invalid
- A slider dimension is retired

Communication must not expose internal identifiers or safety rules.

11.3.6 Retired Configuration Removal Verification

The system must verify that retired configurations:

- Are removed from registries
- Cannot be selected or applied
- Are not referenced by favourites
- Are archived for audit purposes

11.4 Compatibility Maintenance

11.4.1 Version Compatibility Matrix

Mood Shot Version	Slider Schema Version	Preset Schema Version	Compatible Model Versions
…	…	…	…

11.4.2 Cross-Platform Compatibility Verification

All updates must be validated across web, mobile, desktop, and embedded surfaces to ensure consistent behaviour.

11.4.3 Backward Compatibility Requirements

Updates must not break existing presets, favourites, or slider definitions unless explicitly retired.

11.4.4 Forward Compatibility Considerations

New definitions must be designed to avoid future conflicts and support schema evolution.

11.5 Migration Procedures

11.5.1 Slider Definition Migration

Slider migrations must update slider ranges, mappings, and identifiers while preserving backward compatibility where possible.

11.5.2 Preset Definition Migration

Preset migrations must update slider values, remove deprecated sliders, and validate safety compatibility.

11.5.3 Text Template Migration

Template migrations must ensure that updated templates remain safe, deterministic, and compatible with existing slider mappings.

11.5.4 Favourite Migration

Favourites must be migrated or invalidated when referenced sliders or presets change.

11.5.5 Data Migration Validation

All migrations must be validated through:

- Schema validation
- Safety rule evaluation
- Test suite execution
- Telemetry monitoring after deployment

SECTION 12: OPERATOR PROCEDURES

12.1 Operator Roles

12.1.1 Role Definitions

Data Structure:

OperatorRole {

role_id: string

role_name: string

permissions: Array<Permission>

responsibilities: Array<string>

}

Permission {

permission_id: string

resource_type: enum

actions: Array<enum> (create, read, update, delete, approve, retire)

}

12.1.2 Role Assignment Procedures

Role assignment must follow controlled workflows. Operators may only be assigned roles by authorized administrators. All assignments must be logged, reviewed periodically, and revoked when no longer required.

12.1.3 Role-Based Access Control (RBAC) Matrix

Role	Slider Mgmt	Preset Mgmt	Template Mgmt	Safety Mgmt	Deployment
…	…	…	…	…	…

12.2 Slider and Preset Lifecycle Management

12.2.1 Creation Procedure

Workflow:

1. Define slider/preset structure
2. Validate against schema
3. Submit for review
4. Address review feedback
5. Obtain approval
6. Deploy to registry

12.2.2 Review Procedure

Reviewers must validate structural correctness, safety compatibility, identity compatibility, and completeness. Review outcomes must be logged.

12.2.3 Approval Procedure

Approvals must be granted only by authorized operators. Approval requires confirmation that all validation and review steps have passed.

12.2.4 Update Procedure

Updates must follow the same workflow as creation. Updates must not break backward compatibility unless explicitly retired.

12.2.5 Retirement Procedure

Retirement must follow deprecation rules. Retired sliders or presets must be removed from active registries and archived.

12.3 Change Management

12.3.1 Change Request Structure

Data Structure:

ChangeRequest {

request_id: string

change_type: enum

affected_resources: Array<string>

requester: string

description: string

justification: string

status: enum (pending, approved, rejected, deployed)

created_at: timestamp

}

12.3.2 Change Review Process

Change requests must be reviewed for safety impact, identity compatibility, and operational risk. Reviewers must document findings and recommendations.

12.3.3 Change Approval Authority

Only designated approvers may authorize changes. Approval must require at least one reviewer and one approver.

12.3.4 Change Testing Requirements

All changes must pass relevant test suites before deployment. Testing must include safety, continuity, and compatibility checks.

12.3.5 Change Documentation Requirements

All changes must be documented with justification, impact analysis, test results, and deployment notes.

12.4 Incident Response

12.4.1 Incident Classification

Data Structure:

Incident {

incident_id: string

```
 severity: enum (low, medium, high, critical)
 incident_type: enum
 description: string
 affected_components: Array<string>
 detected_at: timestamp
 resolved_at: Optional<timestamp>
 resolution: Optional<string>
}
```

12.4.2 Incident Response Workflow

Detection → Triage → Investigation → Remediation → Documentation → Review

12.4.3 Incident Type Handling Procedures

- Slider failures
- Preset failures
- Template selection failures
- Safety constraint triggers (no modification of model output permitted)
- Integration errors
- Continuity failures

12.4.4 Post-Incident Review Requirements

Each incident must undergo a review to identify root causes, evaluate remediation effectiveness, and define corrective actions.

12.5 Audit and Review

12.5.1 Audit Schedule

Audits must occur on a regular schedule defined by governance policy. Frequency must ensure continuous compliance.

12.5.2 Audit Scope Checklist

- Slider definitions and usage
- Preset definitions and usage
- Text template inventory
- Safety event frequency
- Monitoring signal review
- Maintenance action review

12.5.3 Audit Report Structure

Data Structure:

AuditReport {

report_id: string

audit_date: timestamp

auditor: string

scope: Array<string>

findings: Array<Finding>

recommendations: Array<Recommendation>

}

Finding {

finding_id: string

severity: enum

description: string

evidence: string

}

Recommendation {

recommendation_id: string

priority: enum

description: string

assigned_to: Optional<string>

}

12.5.4 Corrective Action Tracking

Corrective actions must be tracked until completion. Each action must have an owner, due date, and verification step.

12.5.5 Review Cycle Frequency

Review cycles must occur after each audit and after major updates. Review outcomes must be documented and archived.

SECTION 13: VERSIONING AND COMPATIBILITY

13.1 Versioning Scheme

13.1.1 Mood Shot Architecture Version

- Semantic versioning: MAJOR.MINOR.PATCH
- Version change triggers include architectural changes, new invariants, or modifications to layer boundaries

13.1.2 Slider Schema Version

Slider schema versions track changes to slider structure, ranges, and template mapping rules.

13.1.3 Preset Schema Version

Preset schema versions track changes to preset structure, slider value requirements, and validation rules.

13.1.4 Text Template Schema Version

Template schema versions track changes to template structure, content rules, and safety requirements.

13.1.5 API Version

API versions track changes to integration interfaces, request and response structures, and service boundaries.

13.2 Version Compatibility

13.2.1 Compatibility Matrix

Component A Version	Component B Version	Compatible	Notes
…	…	…	…

13.2.2 Compatibility Testing Requirements

Compatibility must be validated through:

- Schema validation
- Cross version test suites
- Safety enforcement tests
- Continuity tests
- Backward and forward compatibility checks

13.2.3 Incompatibility Handling

When incompatibilities are detected:

- Reject incompatible configurations; modification of model output is not permitted
- Log incompatibility events
- Notify operators
- Provide migration guidance if available

13.3 Migration Paths

13.3.1 Version Upgrade Paths

Diagram:

v1.0 → v1.1 → v2.0

↓ ↓

v1.x (deprecated) v2.x (current)

13.3.2 Data Migration Requirements

Data migrations must ensure:

- Structural compatibility
- Safety rule compliance
- Preservation of identity and affective state
- Validation of migrated presets, sliders, and templates
- No loss of user favourites unless invalidated

13.3.3 Downgrade Procedures (If Supported)

Downgrades must only occur when:

- Schema compatibility is guaranteed
- No incompatible data structures exist
- A rollback version is available

If downgrades are not supported, the system must reject downgrade attempts.

13.4 Deprecation Policy

13.4.1 Deprecation Timeline

Deprecation must follow a defined timeline that includes announcement, grace period, and retirement.

13.4.2 Deprecation Notification Requirements

Deprecation notices must be communicated through operator dashboards, release notes, and system alerts.

13.4.3 Support Period for Deprecated Versions

Deprecated versions must remain supported for a defined period before retirement. Support includes bug fixes but not new features.

13.5 Changelog Management

13.5.1 Changelog Structure

Format:

[Version] - Date

Added

New features

Changed

Modified behavior

Deprecated

Features marked for removal

Removed

Deleted features

Fixed

Bug fixes

Security

Security updates

13.5.2 Changelog Maintenance Procedures

Changelogs must be updated for every version change. Each entry must include version number, date, summary of changes, and operator approvals. Changelogs must be stored in a version-controlled repository.

SECTION 14: SECURITY CONSIDERATIONS

14.1 Authentication and Authorization

14.1.1 Operator Authentication Requirements

Operators must authenticate using secure, multi factor mechanisms. Authentication must be enforced for all operator actions, including slider management, preset management, template updates, and safety rule configuration.

14.1.2 API Authentication Requirements

All API endpoints must require authenticated requests. Tokens must be validated on every call. Anonymous or unauthenticated access must not be permitted.

14.1.3 Role-Based Access Control Enforcement

RBAC must be enforced at the service layer. Operators may only perform actions permitted by their assigned roles. Unauthorized actions must be rejected and logged.

14.1.4 Token Management

Tokens must be short lived, securely stored, and rotated regularly. Compromised tokens must be revoked immediately.

14.2 Data Protection

14.2.1 Data at Rest Encryption

All stored data, including identity versions, slider definitions, presets, templates, affective states, favourites, and safety constraints, must be encrypted at rest.

14.2.2 Data in Transit Encryption

All communication between services must use encrypted channels. Plaintext transmission must not be permitted.

14.2.3 Sensitive Data Handling

Sensitive data must be protected through access control, encryption, and audit logging. Sensitive categories include:

- Text templates (must not be modified at runtime)
- Affective states
- User favourites

14.2.4 Data Retention and Deletion

Data must be retained only for the required duration. Expired or retired data must be deleted or archived according to policy. Deletion must be irreversible.

14.3 Privacy Compliance

14.3.1 User Data Minimization

The system must store only the minimum data required for operation. User content must not be stored unless required for conversation scoped state.

14.3.2 Telemetry Data Anonymization

Telemetry must not include user content or personal identifiers. All telemetry must be anonymized or aggregated.

14.3.3 Audit Log Privacy Requirements

Audit logs must not contain user content or sensitive personal data. Logs must store only operational metadata.

14.3.4 Right to Deletion Procedures

If required by policy, user specific data such as favourites must be deletable upon request. Deletion must be logged and verified.

14.4 Injection Attack Prevention

14.4.1 Text Template Injection Prevention

Templates must be validated to ensure they do not contain executable code, unsafe placeholders, or unauthorized variables.

14.4.2 User Input Sanitization

User input must be sanitized to prevent injection of identity content, affective content, or safety rules. Sanitization must be deterministic.

14.4.3 Slider Value Validation

Slider values must be validated against defined ranges. Out of range values must be rejected.

14.4.4 Preset Content Validation

Presets must be validated to ensure they do not contain unsafe values, invalid slider references, or content that violates safety rules.

14.5 Security Testing

14.5.1 Security Test Cases

Security testing must include:

- Identity protection bypass attempts
- Safety constraint bypass attempts
- Unauthorized slider/preset modification
- Data exfiltration attempts
- Injection attacks

14.5.2 Penetration Testing Requirements

Regular penetration tests must be conducted to identify vulnerabilities in APIs, storage systems, and operator interfaces.

14.5.3 Security Audit Schedule

Security audits must occur on a defined schedule. Audits must review authentication, authorization, data protection, and safety enforcement mechanisms.

SECTION 15: PERFORMANCE REQUIREMENTS

15.1 Latency Requirements

15.1.1 Layer Assembly Latency

- Target: < X ms
- Maximum: < Y ms

15.1.2 Text Template Selection Latency

Template selection must complete within defined latency targets. Selection must remain deterministic and must not introduce variable delays.

15.1.3 Safety Evaluation Latency

Safety evaluation must complete within strict latency bounds and must not modify model output. Constraint evaluation must be optimized for predictable performance.

15.1.4 End-to-End Request Latency

The full request pipeline, including assembly, safety evaluation, and model invocation, must meet platform latency requirements.

15.2 Throughput Requirements

15.2.1 Requests Per Second (RPS) Targets

The system must support defined RPS targets under normal and peak load conditions.

15.2.2 Slider/Preset Selection Throughput

Slider and preset operations must support high throughput without degrading conversation performance.

15.2.3 Validation Throughput

Validation operations must scale to handle concurrent requests without bottlenecks.

15.3 Scalability Requirements

15.3.1 Horizontal Scaling Strategy

The system must support horizontal scaling across backend instances. State must remain consistent across instances through durable storage.

15.3.2 Storage Scaling Considerations

Storage systems must scale to support increasing volumes of identity versions, sliders, presets, templates, and affective states.

15.3.3 Registry Size Limits

- Maximum sliders
- Maximum presets
- Maximum text templates

Registry limits must be defined to ensure predictable performance and manageable operator workflows.

15.4 Resource Utilization

15.4.1 Memory Footprint Targets

Memory usage must remain within defined limits. Layer assembly and template selection must not create excessive memory overhead.

15.4.2 CPU Utilization Targets

CPU usage must remain within acceptable thresholds during peak load. Safety evaluation and validation must be optimized for efficiency.

15.4.3 Storage Requirements

Storage systems must support required retention periods, registry sizes, and audit log volumes without performance degradation.

15.5 Performance Testing

15.5.1 Load Testing Scenarios

Load tests must simulate high request volumes, concurrent slider adjustments, and preset applications.

15.5.2 Stress Testing Scenarios

Stress tests must push the system beyond expected limits to identify failure points and recovery behaviour.

15.5.3 Performance Regression Testing

Performance regression tests must run after every update to ensure no degradation in latency, throughput, or resource usage.

SECTION 16: ERROR HANDLING AND LOGGING

16.1 Error Classification

16.1.1 Error Types

Data Structure:

Error {

error_code: string

error_type: enum (validation, safety, integration, storage, system)

severity: enum (low, medium, high, critical)

message: string

details: map<string, string>

timestamp: timestamp

}

16.1.2 Error Code Registry

Error Code	Type	Description	Severity	Recovery Action
…	…	…	…	…

16.2 Error Handling Procedures

16.2.1 Validation Errors

Validation errors occur when identity, affective state, slider values, presets, or task context fail structural or range checks. Validation errors must be returned to the caller with clear messages and must not proceed to model invocation.

16.2.2 Safety Errors

Safety errors occur when safety constraints are violated. Safety errors must override all other layers; modification of model output is not permitted. Unsafe requests must be rejected and logged with constraint identifiers.

16.2.3 Integration Errors

Integration errors occur when external services fail, such as storage unavailability or API timeouts. Integration errors must trigger retry logic if permitted, or return a structured error.

16.2.4 Storage Errors

Storage errors occur when reading or writing identity, affective state, sliders, presets, templates, or favourites. Storage errors must be logged with severity based on impact.

16.2.5 System Errors

System errors include unexpected failures, unhandled exceptions, or infrastructure faults. System errors must be treated as critical and must trigger operator alerts.

16.3 Logging Standards

16.3.1 Log Entry Structure

Data Structure:

LogEntry {

log_id: string

log_level: enum (debug, info, warning, error, critical)

component: string

message: string

context: map<string, string>

timestamp: timestamp

trace_id: Optional<string>

}

16.3.2 Log Levels and Usage

- debug: internal diagnostic information
- info: normal operational events
- warning: recoverable issues

- error: failures requiring attention
- critical: system wide failures requiring immediate action

16.3.3 Structured Logging Requirements

Logs must be structured, machine readable, and consistent across components. Logs must not contain user content or sensitive data.

16.3.4 Log Aggregation

Logs must be aggregated into a centralized system for monitoring, querying, and long term analysis.

16.4 Debugging Support

16.4.1 Debug Mode Activation

Debug mode may be enabled only in controlled environments. Debug mode must never expose sensitive data.

16.4.2 Trace ID Propagation

Trace IDs must be generated at request ingestion and propagated through all components to support end to end debugging.

16.4.3 Debug Log Contents

Debug logs may include detailed validation traces, safety evaluation steps, and layer assembly diagnostics. Debug logs must remain sanitized.

16.4.4 Debug Information Sanitization

All debug information must be sanitized to remove user content, identity content, and sensitive operational data before storage or display.

SECTION 17: DEPLOYMENT SPECIFICATIONS

17.1 Deployment Architecture

17.1.1 Component Deployment Diagram

[Load Balancer]

↓

[Mood Shot Service Layer]

↓

[Storage Layer] [Model Layer]

17.1.2 Service Dependencies

The service layer depends on:

- Storage systems for identity, sliders, presets, templates, affective states, favourites, and safety constraints
- Model invocation endpoints
- Telemetry and logging systems
- Authentication and authorization services

17.1.3 Infrastructure Requirements

Infrastructure must support:

- Horizontal scaling
- Encrypted communication
- High availability storage
- Low latency networking
- Secure secret management
- Monitoring and alerting systems

17.2 Deployment Environments

17.2.1 Development Environment

Used for local testing, debugging, and early integration. May use mock services and reduced data sets.

17.2.2 Staging Environment

Used for full scale validation, performance testing, and pre deployment checks; model output must not be modified. Must mirror production as closely as possible.

17.2.3 Production Environment

Handles live traffic. Must meet all performance, security, and reliability requirements.

17.2.4 Environment Parity Requirements

Staging and production must maintain parity in configuration, schema versions, and infrastructure components to ensure reliable testing.

17.3 Deployment Procedures

17.3.1 Pre-Deployment Checklist

- All tests passing
- Schema validation complete
- Configuration validated
- Security review complete
- Operator approval recorded
- Rollback plan prepared

17.3.2 Deployment Steps

- Deploy updated components to staging
- Validate behaviour
- Deploy to production using controlled rollout
- Monitor telemetry and logs

- Expand rollout after stability confirmed

17.3.3 Post-Deployment Validation

- Validate slider and preset operations
- Validate template selection
- Validate safety enforcement
- Validate continuity behaviour
- Review telemetry for anomalies

17.3.4 Rollback Procedures

- Revert to previous stable version
- Restore previous configuration
- Clear staged updates
- Log rollback reason
- Notify operators

17.4 Configuration Management

17.4.1 Configuration Schema

Data Structure:

Configuration {

environment: enum

feature_flags: map<string, boolean>

slider_registry_url: string

preset_registry_url: string

safety_constraint_config: SafetyConfig

telemetry_config: TelemetryConfig

}

17.4.2 Configuration Deployment

Configuration changes must be deployed through controlled pipelines. Manual edits must not be permitted.

17.4.3 Configuration Validation

Configurations must be validated for completeness, correctness, and compatibility before deployment.

17.4.4 Secret Management

Secrets must be stored in secure vaults. Secrets must not appear in logs, configuration files, or telemetry.

17.5 Continuous Integration/Continuous Deployment (CI/CD)

17.5.1 CI Pipeline Requirements

The CI pipeline must include:

- Static analysis
- Schema validation
- Unit tests
- Integration tests
- Security checks

17.5.2 CD Pipeline Requirements

The CD pipeline must support:

- Staged rollouts
- Automated configuration deployment
- Rollback triggers
- Deployment logging

17.5.3 Automated Testing in Pipeline

Automated tests must run on every commit and before every deployment. Failures must block deployment.

17.5.4 Deployment Gates

Deployment gates must enforce:

- Test suite success
- Security approval
- Operator approval
- Configuration validation
- Telemetry stability in staging

SECTION 18: DOCUMENTATION REQUIREMENTS

18.1 Operator Documentation

18.1.1 Slider Creation Guide

Documentation must describe slider structure, range definitions, validation rules, and deployment procedures.

18.1.2 Preset Creation Guide

Documentation must describe preset structure, slider mappings, validation rules, and approval workflows.

18.1.3 Text Template Management Guide

Documentation must describe template creation, update procedures, safety validation, and retirement workflows.

18.1.4 Safety Configuration Guide

Documentation must describe safety constraint definitions, priority rules, update procedures, and testing requirements.

18.1.5 Troubleshooting Guide

Documentation must include common failure scenarios, diagnostic steps, and remediation procedures.

18.2 API Documentation

18.2.1 API Reference

API documentation must include:

- Endpoint specifications
- Request and response formats
- Authentication requirements
- Rate limits
- Error codes

18.2.2 API Usage Examples

Documentation must include examples demonstrating correct usage patterns for all major endpoints.

18.2.3 SDK Documentation (If Applicable)

If SDKs are provided, documentation must include installation instructions, usage examples, and version compatibility notes.

18.3 Integration Documentation

18.3.1 Integration Overview

Documentation must describe the overall integration architecture, data flows, and service boundaries.

18.3.2 Integration Patterns

Documentation must include recommended integration patterns for common use cases.

18.3.3 Platform-Specific Integration Guides

Documentation must include platform specific instructions for web, mobile, desktop, and embedded surfaces.

18.3.4 Migration Guides

Documentation must include guidance for migrating between schema versions, presets, sliders, and templates; model output must not be modified during migration.

18.4 Runbook Documentation

18.4.1 Operational Runbooks

Operational runbooks must include:

- Service startup
- Service shutdown
- Emergency procedures
- Backup and restore

18.4.2 Incident Response Runbooks

Runbooks must describe incident classification, triage steps, investigation procedures, and remediation workflows.

18.4.3 Maintenance Runbooks

Runbooks must describe scheduled maintenance procedures, update workflows, and validation steps.

SECTION 19: GLOSSARY AND REFERENCE

19.1 Terminology

This section defines all terms used throughout the Mood Shot specification. Terminology must remain consistent across all documents. Terms must not be redefined or altered in downstream specifications.

Identity

The immutable system level definition of the assistant's persona, constraints, and behavioural invariants. Identity cannot be modified by affective state, task content, or user input.

Affective State

A conversation scoped modulation layer composed of slider values, presets, and text templates. Affective state is optional, temporary, and expires automatically.

Preset

A predefined configuration of slider values that produces a specific affective modulation. Presets must be validated for safety and identity compatibility.

Slider

A dimension of affective modulation with a defined range and mapping to text templates. Slider values must remain within defined bounds.

Text Template

A deterministic text fragment selected based on slider values. Templates must be safe, static, and free of executable or dynamic content.

Safety Constraint

A rule that enforces system safety. Safety constraints override identity, affective state, and task content; modification of model output is not permitted.

Assembled Input

The final structured input passed to the model, composed of identity content, affective templates, and task content.

Conversation State

The stored state associated with a conversation, including affective state and expiry metadata.

Favourite

A user scoped saved configuration of slider values. Favourites persist across conversations.

19.2 Acronyms and Abbreviations

Acronyms used in this specification must be defined here.

Acronym	Expansion
API	Application Programming Interface
RBAC	Role Based Access Control
RPS	Requests Per Second
CI	Continuous Integration
CD	Continuous Deployment
ID	Identifier
SDK	Software Development Kit

19.3 Reference Material

19.3.1 Related Specifications

This specification has a single normative internal dependency:

- Mood Shot Architecture Specification

The Architecture Specification defines the conceptual model, layer boundaries, invariants, and core terminology for Mood Shot. This document provides the concrete implementation, integration, and operational details that realise that architecture.

Where there is any ambiguity, the Architecture Specification defines the intended behaviour at the conceptual level, and this document must align with it. No other internal documents are required to interpret or implement the system.

19.3.2 External Standards

The system must comply with relevant external standards, including:

- Data protection and privacy regulations
- Encryption and security standards
- Logging and audit compliance requirements
- Accessibility guidelines for UI surfaces
- Industry standard API design principles

19.3.3 Academic References (If Applicable)

This specification does not currently rely on academic sources. If future revisions incorporate academic research to inform safety constraints, modulation design, or template selection logic, those sources must be listed in this section.

Until such references are formally added, this section remains intentionally empty and the specification is considered fully self-contained.

APPENDICIX A. Example Sliders

This appendix provides illustrative examples of bipolar sliders used within the Mood Shot system. Each slider represents a single affective or stylistic axis defined in the Mood Shot Architecture Specification. The examples below demonstrate how opposite semantic poles are expressed along a 0–10 scale. These examples are non-normative and serve only to clarify the intended structure and behaviour of slider-based modulation. Text templates must be static, deterministic strings and may only be injected into the assembled input. They must never alter, transform, or post-process the model's output.

A.1 Interpersonal Stance

Antagonistic → 0 → Supportive

Affective posture toward the user, ranging from cold or argumentative to warm and cooperative.

A.2 Expression Length

Verbose → 0 → Concise

Controls the degree of elaboration, ranging from expansive output to tightly compressed phrasing.

A.3 Formality

Formal → 0 → Casual

Controls linguistic register, from structured and professional to relaxed and conversational.

A.4 Directness

Indirect → 0 → Direct

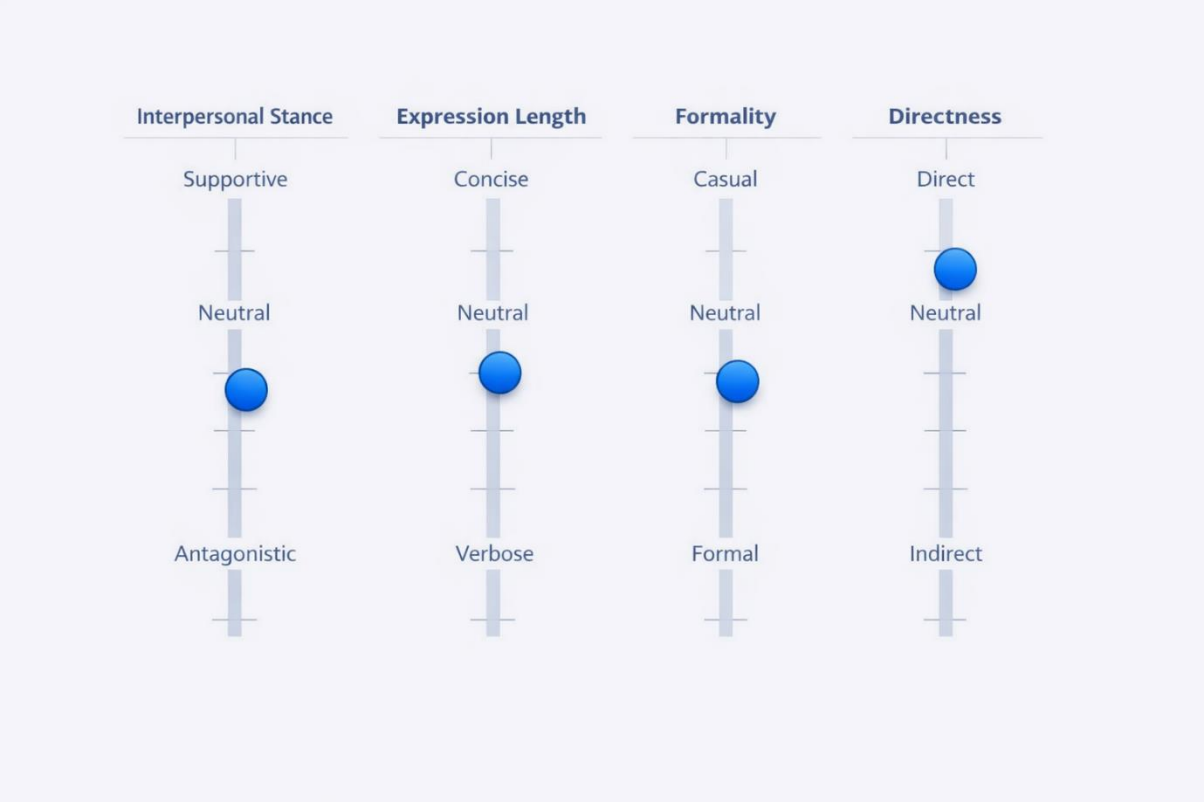
Controls explicitness, from softened or implied phrasing to clear and unambiguous statements.

Appendix B. Example Slider User Interface

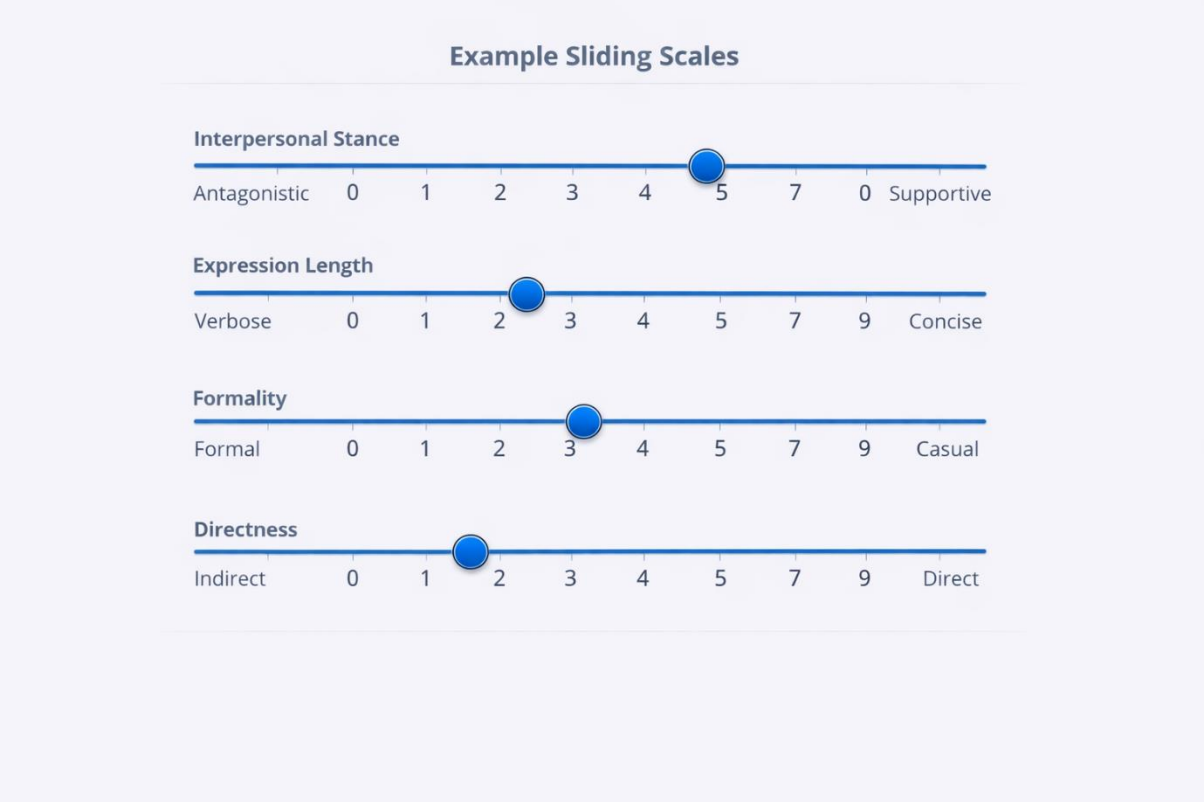
These examples show an easy-to-understand, intuitive user interface for controlling personality traits in Mood Shot. Each slider presents opposing traits with Neutral in the centre, using both horizontal and vertical layouts for illustration. These are illustrative only: operators are free to implement their own UX designs, styles, or control schemes as required. These UI examples influence only slider values; they do not modify templates or model output.

Note: Underlying configuration formats (such as JSON) are implementation-specific and not required for operators.

B.1. Vertical Sliders Example Image



B.2. Horizontal Sliders example Image



-Document Ends-