

CRITICAL: Update Existing Docs, Don't Create New Ones

- **NEVER** create separate/new documentation files after code changes
- **NEVER** create new audit, summary, implementation, verification, or status documents
- **ALWAYS** update existing documentation files in the unified 10-file structure
- If information doesn't fit existing files, consolidate into the most relevant existing file
- Only create new files if absolutely necessary and approved (rare exceptions)
- Historical/outdated content goes to `archive/` folder, not new files

Forbidden Documentation File Types

-  **NEVER create:** `*_SUMMARY.md`, `*_AUDIT.md`, `*_IMPLEMENTATION.md`, `*_VERIFICATION.md`
-  **NEVER create:** `STATUS_*.md`, `FINAL_*.md`, `COMPLETE_*.md`, `DELIVERY_*.md`
-  **NEVER create:** Numbered docs beyond 00-09 (e.g., `10-*.md`, `11-*.md`)
-  **ALWAYS update:** Existing docs in `docs/` folder (`00-INDEX.md` through `09-changelog.md`)
-  **ALWAYS update:** `docs/08-implementation-summary.md` for implementation status
-  **ALWAYS update:** `docs/09-changelog.md` for version history

Document Creation Policy

- **DO NOT** create new documentation files without explicit user confirmation
- **DO NOT** create new docs during discussions - discussions should happen in chat first
- **DO** update existing documentation files when requested
- **DO** ask for confirmation before creating any new markdown files
- **Exception:** Only create new docs when user explicitly requests "create a new doc" or similar

Changelog Maintenance

- **REQUIRED:** Update `docs/09-changelog.md` for any significant code changes
- Include version number, date, and description of changes
- Follow semantic versioning (MAJOR.MINOR.PATCH)
- Document breaking changes, new features, and bug fixes
- **DO NOT** create separate changelog files - update the existing one
- **DO** update changelog when making code changes that affect functionality

Implementation Summary

- **REQUIRED:** Keep `docs/08-implementation-summary.md` current with implementation status
- Update "Current Status" section when features are completed
- Document recent updates with dates and key achievements
- Maintain accurate feature list and component status
- **DO NOT** create separate status files - update the existing one
- **DO** update implementation summary when significant architectural changes are made

Documentation Structure

- Follow unified 10-file documentation structure:
 - 00-INDEX.md (navigation)
 - 01-architecture.md
 - 02-api-reference.md
 - 03-flow-documentation.md
 - 04-deployment.md
 - 05-testing.md
 - 06-configuration.md
 - 07-troubleshooting.md
 - 08-implementation-summary.md
 - 09-changelog.md
- **DO NOT** create additional numbered files (10, 11, etc.) - update existing ones

Code Changes Documentation

- When adding new features, update relevant existing documentation
- When changing API endpoints, update 02-api-reference.md (don't create new API docs)
- When changing architecture, update 01-architecture.md (don't create new arch docs)
- When changing configuration, update 06-configuration.md (don't create new config docs)
- When fixing bugs, update 07-troubleshooting.md (don't create new fix docs)

Testing Documentation

- Update 05-testing.md when adding new test strategies
- Document test coverage changes in existing testing.md
- Include test examples for new features in existing file
- **DO NOT** create separate test documentation files

Code Standards

Logging Standards (v1.2.89+)

- **CRITICAL:** All agents and services MUST use BaseService logging methods
- **REQUIRED:** Use `self.log_info()`, `self.log_error()`, `self.log_warning()`, `self.log_debug()` instead of direct logger access
- **FORBIDDEN:**
 - `self.logger.info()` - Use `self.log_info()` instead
 - `logger.info()` (module-level) - Use `self.log_info()` instead
 - `error=str(e)` in `log_error()` - Use `error=e` (exception object) instead
- **REQUIRED:** All agents inherit from BaseAgent (which inherits from BaseService)
- **REQUIRED:** Services should inherit from BaseService for observability
- **BENEFITS:** Automatic correlation ID tracking, Sentry integration, performance tracking
- **Pattern:**

#**  **CORRECT

```
self.log_info("operation_started", *job_id*=context.job_id)
self.log_error("operation_failed", *error*=exception_object)
```

#**  **WRONG

```
self.logger.info("operation_started", *job_id*=context.job_id)
logger.info("operation_started", *job_id*=context.job_id)
self.log_error("operation_failed", *error*=str(e))  *# Use exception
object*
```

No Hardcoding Rule

- **CRITICAL:** NO hardcoded values, defaults, or fallbacks in agent code
- **ALL configuration** must come from via `agent_config` JSON or top-level fields
- **FORBIDDEN:**
 - Hardcoded default values (e.g., `.get('field', 100)` - remove the `100`)
 - Hardcoded field names that should be configurable

- Hardcoded business logic thresholds or limits
- Hardcoded multipliers, caps, or calculation constants
- **REQUIRED:** If a value is missing from Sanity, raise `AgentExecutionError` with clear message
- **EXCEPTIONS** (only these are allowed):
 - Mathematical constants (e.g., `math.pi`, `0` for empty collections)
 - System-level constants (e.g., HTTP status codes, file paths from env vars)
 - Type conversion defaults (e.g., `int(value)` where value is guaranteed to exist)
- **Pattern:** Use `config.get('field')` and validate with `if field is None: raise AgentExecutionError(...)`
- **Never use:** `config.get('field', default_value)` - this hides missing configuration

API Consistency

- Use `/api/v1/ai` prefix for core endpoints
 - Use `/api/v1/brand` prefix for brand analysis endpoints
 - Maintain consistent error response format
 - Include proper API key authentication
-

Environment Variables

Required Variables

- `API_KEY` - Service API key (REQUIRED)
- `INTEGRATION_SERVICE_URL` - Integration service endpoint (REQUIRED)

Optional but Recommended

- `SUPABASE_URL` - For job history
 - `WORKFLOW_SERVICE_URL` - For callbacks
-

Supabase Database Management

CRITICAL: Use Supabase MCP, Not Manual Migrations

- **NEVER** create manual migration files or execute SQL migrations directly
- **ALWAYS** use Supabase MCP tools for all database schema changes
- **Schema files** are stored in `supabase/schema/` for reference only
- **DO NOT** execute schema files manually - they are documentation/reference only
- **USE** `mcp_supabase_apply_migration` tool for all DDL operations
- **USE** `mcp_supabase_execute_sql` tool for DML operations (data queries only)

Schema Folder Structure

- `supabase/schema/` - Contains SQL schema files for reference/documentation
- Schema files are NOT executed manually - they serve as documentation
- All schema changes must go through Supabase MCP tools
- Schema files should be kept in sync with actual database state (for reference)

Database Operations

- **Schema Changes:** Use `mcp_supabase_apply_migration` with migration name and SQL query
 - **Data Queries:** Use `mcp_supabase_execute_sql` for SELECT/INSERT/UPDATE/DELETE
 - **Type Generation:** Use `mcp_supabase_generate_typescript_types` to generate TypeScript types
 - **Never:** Run SQL files directly or use migration CLI tools
-

Testing Requirements

- All new features must include tests
 - Maintain test coverage above 75%
 - Update test documentation when adding new test cases
 - Run tests before committing: `pytest tests/ -v`
-

Git Workflow

- Update changelog before committing significant changes
 - Keep implementation summary current
 - Document breaking changes clearly
 - Follow semantic versioning for releases
-

Naming and Structure Standards

Reference Guide: See `App/docs/all_service/CodeStructure/08-NAMING-AND-STRUCTURE-GUIDE.md` for complete standards.

Sanity Document Types

- **Format:** camelCase (e.g., `moduleConfig`, `agentPrompt`, `llmProviderConfig`)
- **Rules:** Singular nouns, descriptive names, starting with lowercase

API Endpoints

- **Format:** kebab-case with version prefix (e.g., `/api/v1/entity/analyze/mini`)
- **Pattern:** `/api/v1/{resource}/{action}/{variant}`
- **Rules:** Always lowercase, use hyphens, plural resource names

Python Code Naming

- **Files:** snake_case (e.g., `model_validator.py`)
- **Variables/Functions:** snake_case (e.g., `module_code`, `validate_module_code()`)
- **Classes:** PascalCase (e.g., `ModelValidator`)
- **Constants:** UPPER_SNAKE_CASE (e.g., `MAX_RETRIES = 3`)

Analysis Type References

- **In Module Codes:** `MINI`, `FULL` (uppercase)
- **In API Endpoints:** `mini`, `full` (lowercase)
- **In Code Variables:** `analysis_type = "mini"` (lowercase)
- **In Documentation:** "Mini" or "Full" (title case) for readability

Consistency Checklist

- ☐ Module codes use UPPER_SNAKE_CASE
 - ☐ Sanity document types use camelCase
 - ☐ API endpoints use kebab-case
 - ☐ Python files use snake_case
 - ☐ Constants use UPPER_SNAKE_CASE
 - ☐ All references are consistent across files
-

Last Updated: December 2025

Enforced: Yes