

# AliothStudio Application Building Handbook

Blue Paper · The Complete Operational Path from Environment Setup to Deployment Release

Version v1.1 · October 2026 · Hangzhou Yuqi Technology Co., Ltd.

Applicable versions: Alioth Model v10.0.44 · AliothStudio 0.1.x

## Contents

1. Platform Composition: Two Chains
2. Environment Preparation
3. The Model Side: Model Center and Model Release
4. Application Building: The AppAgent Workflow
5. Development Essentials for Modules, Blocks, and Services
6. Extensions: extensions/\*.yaml
7. Build and Deployment
8. Quality Gate Quick Reference
9. Online Practice: SaaS Learning Edition

## 1. Platform Composition: Two Chains

AliothStudio consists of two chains with separated responsibilities; understanding their boundary is the precondition for using the platform correctly:

| Chain                             | Responsibility                                                                          | Driven by                         | Output                                                                                                      |
|-----------------------------------|-----------------------------------------------------------------------------------------|-----------------------------------|-------------------------------------------------------------------------------------------------------------|
| <b>Model release chain</b>        | Structural evolution of the data substrate (tables/columns/inheritance/dimension codes) | Meta Model Center (human channel) | <code>Alioth/v*</code> release package (DDL<br>· seed · machine-readable contract<br>· verification report) |
| <b>Application building chain</b> | Assembling business applications on the frozen substrate                                | AppAgent agent + developers       | Application configuration, Service backend, module frontend,<br><code>Deploy/{ns}</code> deployment package |

**Red line:** the isahl schema structure is frozen. Any request to "add a table / add a column / change a column" in the application building chain must go back to the model release chain (adjudicated by humans in the Model Center); bypassing it through DDL, migration scripts, or direct metadata writes is forbidden.

## 2. Environment Preparation

### 2.1 Prerequisites

| Dependency                  | Purpose                                                                               |
|-----------------------------|---------------------------------------------------------------------------------------|
| PostgreSQL 14+              | Data substrate (model v10.0 is a table-inheritance structure)                         |
| mise                        | Unified entry point for tasks and environments ( <code>mise run &lt;task&gt;</code> ) |
| Rust toolchain / pnpm / bun | Backend build, frontend build, scripting toolchain                                    |

### 2.2 Database Initialization and Startup

```
# Development database initialization (replayed from the release snapshot; DROPs and rebuilds the
target database – aliothstudio_dev only)
bash scripts/db/reset-db.sh --dev

# Meta platform (model management + AppAgent): backend :4949, frontend :49494
mise run meta

# Namespace development environment (Gateway + SS0), {ns} ∈ alioth | wz | avic | cosmic | se | mat
mise run dev:{ns}
```

The database has four isolation layers: production `aliothstudio` ← staging `aliothstudio_pre` → development `aliothstudio_dev` → test `aliothstudio_test`. Environments are forcibly bound to `DATABASE_URL`; all schema queries go through `cd Meta/backend && mise run schema-info --<command>` (the DB is the single source of truth — inferring from DDL files is forbidden).

## 3. The Model Side: Model Center and Model Release

### 3.1 Model Center (Meta UI)

All structural evolution is performed through the Meta frontend Model Center: creating collections (attaching to a parent table to form an inheritance), registering fields, and maintaining dimension codes (scene/factor/function). Registering collections and fields is programmatically synchronized by the platform into physical tables and inheritance attachments — developers write no table-creation DDL at all.

### 3.2 Model Release

Once a release is initiated from the Model Center, the pipeline exports the `isahl` schema (`pg_dump --schema-only` + 268 seed tables' data + the dimension seed `seed-dimensions.sql`), post-processes it into pure CREATE/ALTER form, writes it to `Alioth/v{version}/`, and performs a full rebuild verification in a dedicated verification database (structure round-trip + seed round-trip + uid uniqueness), with the report written to disk as `verify-report.json`. The latest release is currently `v10.0.44`: 1022 isahl tables, 1204 inheritance edges; all 268 seed tables verified round-trip consistent, verification status verified. The release pipeline emits 7 progress events (prepare / schema / isahl\_dump / seed\_dimensions / manifest / verify / done); the collection row layer is declared and carried by the model-side `row_layer` (abstract / carry), and an unadjudicated parent table holding rows blocks the release.

Model landing and the DB derived surface are coupled in the same batch: `refresh-derived-artifacts.ts` is the sole refresh entry point for the four derived artifacts `fk-index`, `relation-index`, `value-domain-columns`, and `dk-projection-tables`, and is wired into `reset-db.sh`; the artifact diff is the model-topology review surface, and anti-removal is enforced by gate W9.

### 3.3 Dimensions and Coordinates

Business semantics are located through three-dimensional coordinates: `dk_scene` (scene), `dk_factor` (factor), `dk_function` (function). Dimension codes are automatically composed from a category code + a serial symbol (e.g. `JC` = system management). Coordinates are interpreted locally per namespace; every API of a Service statically binds a fixed coordinate triple.

## 4. Application Building: The AppAgent Workflow

The application building chain is driven by AppAgent, with the following standard tool sequence:

```
analyze_ontology      # requirements/prototype → OntologyModel IR (with hard five-domain
completeness validation)
build_module          # flow-plan → module.json (module declaration)
build_block           # flow-plan → block.json (capability block declaration)
transfer_ontology     # ontology alignment → alignment-graph.json + mapped-entities.json
generate_service_api  # ontology entities → Service backend (Rust, static coordinate binding)
compose_app           # module assembly → app.json + extensions/*.yaml + prototype artifacts
```

Artifact locations: `Pre-Proc/{ns}/Apps/{app}/` (app.json, extensions, process artifacts) and `Pre-Proc/{ns}/Sources/Apps/{Modules,Blocks,Services}/` (module/block/service declarations and code). The entry shape of ontology mapping artifacts (`ontology-output.json`) is validated by the contract gate, and coordinates must be real DB codes.

## 5. Development Essentials for Modules, Blocks, and Services

| Layer           | Key constraints                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Module          | Scene/Factor composition container; required fields of <code>module.json</code> complete; <code>DashboardPage</code> forbidden (a root path that jumps straight to an entity list page)                                                                                                                                                                                                                                                                                                                 |
| Block           | Capability unit; the IDs referenced by <code>factors</code> / <code>services</code> must exist (orphan references have a dedicated gate)                                                                                                                                                                                                                                                                                                                                                                |
| Service backend | Standard skeleton of <code>handlers/models/repositories/services</code> ; static binding of the coordinate triple (split APIs for the same table under different coordinates); auth middleware forbidden (the Gateway handles NGAC uniformly); the <code>tracing</code> crate forbidden inside modules                                                                                                                                                                                                  |
| Frontend        | State management is always Jotai v2; measurement fields follow the "numeric value + unit radio" format; integrate against the real backend, mocks forbidden; data access uniformly goes through the <code>@alioth/api</code> resource-level transport (list/get/mutate, with the envelope unwrapped via <code>items()</code> / <code>unwrap()</code> ); reference selectors always use the <code>ReferenceSelectDescriptor</code> descriptor — extrapolating contracts from endpoint types is forbidden |

### 5.1 Iron Rules for Data Writes

- **INSERT lands on a leaf table:** the write target must be a leaf table of the inheritance partial order (the gate measures this with `pg_inherits`; static name guessing is not accepted)
- **State goes through relation tables:** entity state/category/tags are written to `zc_id_lifecycle_r_*` relation tables; entities may not create their own state columns (`flag` is used only by state dictionary rows)
- **Class write contract:** writes to lifecycle sub-tables must provide a `dk_function` from one of the six prefixes (`_f_` / `_t_` are derived automatically; hand-written literals are forbidden)
- **Scalar references:** `qk_*` columns are value-space pointers (bigint); arithmetic on them is forbidden, and reads go through `_refs` resolution
- **Time semantics:** future slices go to the task family and past slices to the event family; a completion action lands on an event slice (slice flip)

## 6. Extensions: extensions/\*.yaml

Application-level extensions ( `constraints` / `rules` / `statemachines` / `workflows` ) are placed in `Pre-Proc/{ns}/Apps/{app}/extensions/` and loaded when the Gateway starts. This is the only driver surface through which extensions take effect — validation, rules, and flows can be adjusted without regenerating code.

## 7. Build and Deployment

```
# The only build entry point (feature gating + target isolation + automatic signing)
bash scripts/build-ns.sh <NAMESPACE> release    # or dev (debug mode)

# Release (syncs Apps / seed / .env / apps.json; does not build by itself)
bash scripts/deploy/release-to-namespace.sh <NAMESPACE>
```

Artifacts: `Deploy/{ns}/bin/{ns}-server` (namespaces with OpenActivity enabled additionally have `Deploy/{ns}/Open/bin/openactivity-server`). Releasing to the production stage automatically restarts the Gateway. Non-namespace applications (Meta / SSO) go through `scripts/meta|sso/deploy.sh` and land in `Release/`.

## 8. Quality Gate Quick Reference

The repository enforces quality through a "specification execution substrate": every checkable specification is paired with a check script + baseline + gate registration (currently 229 items: 137 in the commit stage / 92 in the push stage). The specification library now holds 403 capability living specifications; every MUST/NEVER clause must declare its enforcement point (gate id / script / test path) through the structured field "enforcement surface" — prose declarations are no longer passed. High-frequency items:

| Gate                                  | What it blocks                                                                 | Stage  |
|---------------------------------------|--------------------------------------------------------------------------------|--------|
| <code>check-isahl-ddl-boundary</code> | Any isahl structure write that bypasses the channel                            | commit |
| <code>check-leaf-insert</code>        | INSERT not landing on a leaf table / hand-written <code>_f/_t_</code> literals | commit |
| <code>check-code-semantics</code>     | Type/enum semantics written into a code column                                 | commit |
| <code>check-ontology-contract</code>  | Ontology mapping artifact shape / forged coordinates                           | commit |
| <code>check-bridge-host-domain</code> | Misplaced host domain on the left end of a relation table                      | commit |
| <code>formal-proofs</code>            | Lean theorem build failure or proof placeholders                               | push   |
| <code>check-all</code> (T1/T2)        | Version alignment / OpenSpec compliance / i18n / Service DAG, etc.             | push   |

When blocked, do not bypass: address the fix surface reported by the gate; existing violations go through baseline normalization (only decreasing, never increasing), while new violations must be fixed.

## 9. Online Practice: SaaS Learning Edition

No local environment is needed to experience the full platform: <https://cosmic-tools.shop/> offers an online SaaS learning edition of AliothStudio — a streamlined deployment for individual users (core modeling, application assembly, and runtime workflows are fully available; enterprise-grade multi-namespace orchestration and the release pipeline are omitted). Suggested path: sign up and enter → follow Chapters 2–7 of this handbook in order → self-check your artifacts against the gate list in Chapter 8.

Companion document: "Alioth Model Theory White Paper" — understand the mathematical structure behind every constraint in this handbook.

© 2025–2026 Hangzhou Yuqi Technology Co., Ltd. (CosmicTools.ltd).