

AliothStudio アプリケーション構築操作マニュアル

ブルーペーパー ・ 環境準備からデプロイ公開までの完全な操作パス

バージョン v1.1 ・ 2026 年 10 月 ・ 杭州宇器科技有限公司

対応バージョン: Alioth モデル v10.0.44 ・ AliothStudio 0.1.x

目次

- プラットフォーム構成: 二つのチェーン
- 環境準備
- モデル側: モデルセンターとモデルリリース
- アプリケーション構築: AppAgent ワークフロー
- モジュール、Block、Service の開発要点
- 拡張: extensions/*.yaml
- ビルドとデプロイ
- 品質ゲート早見表
- オンライン実践: SaaS 学習版

1. プラットフォーム構成: 二つのチェーン

AliothStudio は責務の分離された二つのチェーンから構成される。その境界を理解することがプラットフォームを正しく使う前提である:

チェーン	責務	駆動主体	産出
モデルリリースチェーン	データ基盤の構造進化 (テーブル / 列 / 継承 / 次元コード)	Meta モデルセンター (人手チャンネル)	Alioth/v* リリースパッケージ (DDL・シード・機読契約・検証レポート)
アプリケーション構築チェーン	凍結された基盤の上で業務アプリケーションを組み立てる	AppAgent エージェント + 開発者	アプリ設定、Service バックエンド、モジュールフロントエンド、Deploy/{ns} デプロイパッケージ

レッドライン: isahl schema の構造は凍結されている。アプリケーション構築チェーンにおける「テーブル追加 / 列追加 / 列変更」のあらゆる要求は、必ずモデルリリースチェーン (モデルセンターによる人手の裁定) へ戻さなければならず、DDL、マイグレーションスクリプト、メタデータへの直接書き込みによる迂回は禁止する。

2. 環境準備

2.1 前提依存

依存	用途
PostgreSQL 14+	データ基盤 (モデル v10.0 はテーブル継承構造)
mise	タスクと環境の統合エントリ (mise run <task>)
Rust toolchain / pnpm / bun	バックエンドビルド、フロントエンドビルド、スクリプトツールチェーン

2.2 データベースの初期化と起動

```
# 開発 DB の初期化 (リリーススナップショットからリプレイ。対象 DB を DROP して再作成するため aliothstudio_dev 専用)
bash scripts/db/reset-db.sh --dev

# Meta プラットフォーム (モデル管理 + AppAgent) : バックエンド :4949、フロントエンド :49494
mise run meta

# ネームスペース開発環境 (Gateway + SS0) 、{ns} ∈ alioth | wz | avic | cosmic | se | mat
mise run dev:{ns}
```

データベースは四層分離である：本番 `aliothstudio` ← ステージング `aliothstudio_pre` → 開発 `aliothstudio_dev` → テスト `aliothstudio_test`。環境と `DATABASE_URL` は強制的にバインドされる；schema の照会はず `cd Meta/backend && mise run schema-info -- <command>` を通す（DB が唯一の真実源であり、DDL ファイルを読んで推論することは禁止）。

3. モデル側：モデルセンターとモデルリリース

3.1 モデルセンター（Meta UI）

すべての構造進化は Meta フロントエンドのモデルセンターを通じて行う：コレクション（collection、親テーブルに接続して継承を形成）の新規作成、フィールド（fields）の登録、次元コード（シーン / 要素 / 職能）の保守。コレクションとフィールドの登録はプラットフォームによって物理テーブルと継承接続へプログラマ的に同期される——開発者はテーブル作成 DDL を一切書かない。

3.2 モデルリリース

モデルセンターでリリースを起動すると、パイプラインは `isahl schema` をエクスポートし（`pg_dump --schema-only` + 268 のシードテーブルデータ + 次元シード `seed-dimensions.sql`）、純粋な CREATE/ALTER の形態に後処理して `Alioth/v{version}/` へ書き出し、専用の検証 DB で全量再構築検証を行う（構造往復 + シード往復 + uid 一意性）。レポートは `verify-report.json` に落ちる。現在の最新リリースは `v10.0.44`：1022 の `isahl` テーブル、1204 の継承エッジ、268 のシードテーブルがテーブルごとに往復一致、検証状態は `verified`。リリースパイプラインは 7 つの進捗イベントを産出する（`prepare` / `schema` / `isahl_dump` / `seed_dimensions` / `manifest` / `verify` / `done`）；コレクションの行レベルはモデル側の `row_layer`（`abstract` / `carry`）宣言によって担われ、未裁定の親テーブルが行を持てばリリースは即座に遮断される。

モデルの着地と DB 派生面は同批で結合される：`refresh-derived-artifacts.ts` は `fk-index`、`relation-index`、`value-domain-columns`、`dk-projection-tables` の四つの派生物の唯一のリフレッシュ入口であり、既に `reset-db.sh` に接続されている；産物の diff はすなわちモデルトポロジーのレビュー面であり、摘除防止はゲート W9 によって保証される。

3.3 次元と座標

業務意味論は三次元座標によって位置づける：`dk_scene`（シーン）、`dk_factor`（要素）、`dk_function`（職能）。次元コードはカテゴリコード + 連番記号が自動的に合成される（例：`JC`

= システム管理)。座標はネームスペースごとに局所的に解釈される；Service の各 API は固定の座標三つ組を静的にバインドする。

4. アプリケーション構築：AppAgent ワークフロー

アプリケーション構築チェーンは AppAgent が駆動し、標準のツール列は以下のとおりである：

```
analyze_ontology      # 要件／プロトタイプ → OntologyModel IR（五領域完備性のハード検証を含む）
build_module          # flow-plan → module.json（モジュール宣言）
build_block           # flow-plan → block.json（能力ブロック宣言）
transfer_ontology     # オントロジー整合 → alignment-graph.json + mapped-entities.json
generate_service_api  # オントロジー実体 → Service バックエンド（Rust、座標の静的バインド）
compose_app           # モジュール組立 → app.json + extensions/*.yaml + プロトタイプ成果物
```

成果物の配置： `Pre-Proc/{ns}/Apps/{app}/`（`app.json`、`extensions`、フロー成果物）と `Pre-Proc/{ns}/Sources/Apps/{Modules,Blocks,Services}/`（モジュール / ブロック / サービスの宣言とコード）。オントロジーマッピング成果物（`ontology-output.json`）のエントリ形状は契約ゲートによって検証され、座標は DB の実在コードでなければならない。

5. モジュール、Block と Service の開発要点

層	重要な制約
Module	Scene/Factor の組み合わせコンテナ； <code>module.json</code> の必須フィールドは完全であること； <code>DashboardPage</code> は禁止（ルートパスから実体一覧ページへ直行する）
Block	能力単位； <code>factors</code> / <code>services</code> が参照する ID は存在しなければならない（孤児参照には専用ゲートがある）
Service バックエンド	<code>handlers/models/repositories/services</code> の標準骨格；座標三つ組の静的バインド（同一テーブルでも座標が異なれば API を分割）； <code>auth</code> ミドルウェアは禁止（Gateway が NGAC を統一）；モジュール内での <code>tracing</code> crate は禁止
フロントエンド	状態管理は一律 Jotai v2；計量フィールドは「数値 + 単位の単一選択」の方式；実バックエンドで結合し、モックは禁止；データアクセスは <code>@alioth/api</code> のリソースレベル <code>transport</code> （ <code>list/get/mutate</code> 、エンベロープは <code>items()</code> / <code>unwrap()</code> でアンラップ）に統一；参照セクターは必ず <code>ReferenceSelectDescriptor</code> 記述子を使用し、エンドポイント型からの契約外推を禁止

5.1 データ書き込みの鉄則

- **INSERT は葉テーブルへ**: 書き込み先は継承半順序の葉テーブルでなければならない（ゲートは `pg_inherits` で実測し、静的な名前の推測は受け付けない）
- **状態は関係テーブル経由**: 実体の状態 / カテゴリ / タグは `zc_id_lifecycle_r_*` 関係テーブルへ書き込む。実体に状態列を自作することは禁止（`flag` は状態辞書の行のみで使用）
- **クラス書き込み契約**: lifecycle 子テーブルへの書き込みは、六つの接頭辞のいずれかの `dk_function` を必ず提供する（`_f_` / `_t_` は自動派生であり、リテラルの手書きは禁止）
- **スカラー参照**: `qk_*` 列は値空間ポインタ（bigint）であり、これに対する算術演算は禁止。読み取りは `_refs` を通じて解決する
- **時間意味論**: 未来のスライス（task 族）、過去のスライス（event 族）を通り、完了動作はイベントスライスへ落ちる（スライス反転）

6. 拡張: extensions/*.yaml

アプリケーションレベルの拡張（`constraints` / `rules` / `statemachines` / `workflows`）は `Pre-Proc/{ns}/Apps/{app}/extensions/` に配置し、Gateway の起動時にロードされる。拡張が有効になる唯一の駆動面である——コードを再生成することなく、検証、ルール、フローを調整できる。

7. ビルドとデプロイ

```
# 唯一のビルド入口（feature ゲート + target 分離 + 署名の自動処理）
bash scripts/build-ns.sh <NAMESPACE> release    # または dev（debug モード）

# リリース（Apps / seed / .env / apps.json を同期；自前ではビルドしない）
bash scripts/deploy/release-to-namespace.sh <NAMESPACE>
```

成果物: `Deploy/{ns}/bin/{ns}-server`（OpenActivity を有効化したネームスペースには別途 `Deploy/{ns}/Open/bin/openactivity-server`）。production ステージへのリリース時は Gateway を自動的に再起動する。非ネームスペースアプリ（Meta / SSO）は `scripts/meta|sso/deploy.sh` を通り、`Release/` に落ちる。

8. 品質ゲート早見表

リポジトリは「規約実行基板」によって品質を強制する：検査可能な各規約にはチェックスクリプト + ベースライン + ゲート登録が付く（現行登録ゲート 229 条：commit 段階 137 / push 段階 92）。規約ライブラリには 403 個の capability 活規約があり、各 MUST / NEVER 条項は構造化フィールド「執行面」（enforcement surface）で実行位（ゲート id / スクリプト / テストパス）を宣言しなければならない、散文の宣言はもはや許容されない。接触頻度の高い項目：

ゲート	遮断対象	段階
check-isahl-ddl-boundary	迂回された isahl 構造書き込みのすべて	commit
check-leaf-insert	葉テーブルに落ちない INSERT / 手書きの <code>_f_/_t_</code> リテラル	commit
check-code-semantics	型 / 列挙の意味論を code 列へ書き込むこと	commit
check-ontology-contract	オントロジーマッピング成果物の形状 / 座標の偽造	commit
check-bridge-host-domain	関係テーブル左端のホストドメインのずれ	commit
formal-proofs	Lean 定理のビルド失敗、または証明プレースホルダの混入	push
check-all (T1/T2)	バージョン整合 / OpenSpec 準拠 / i18n / Service DAG など	push

遮断されたときは迂回しない：ゲート出力の修正面に従って処理する；既存の違反はベースライン帰線で扱い（減のみで増は不可）、新規の違反は必ず修正する。

9. オンライン実践：SaaS 学習版

ローカル環境なしで完全なプラットフォームを体験できる：<https://cosmic-tools.shop/> は AliothStudio のオンライン SaaS 学習実践版を提供する——個人ユーザー向けの簡素化デプロイである（コアのモデリング、アプリケーション組立、運行動線は完全に利用可能であり、エンタープライズ向けのマルチネームスペース編成とリリースパイプラインは省略する）。推奨パス：登録して入る → 本マニュアル第 2～7 章の順に実践する → 第 8 章のゲート一覧で成果物を自己点検する。

関連文献：『Alioth モデル理論ホワイトペーパー』——本マニュアルのすべての制約の背後にある数学的構造を理解するために。

© 2025–2026 杭州宇器科技有限公司（CosmicTools.ltd）。