

AliothStudio-Handbuch zur Anwendungserstellung

Blue Paper · Der vollständige Weg von der Umgebungsvorbereitung bis zur Bereitstellung

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

Anwendbare Versionen: Alioth-Modell v10.0.44 · AliothStudio 0.1.x

Inhaltsverzeichnis

1. Plattformaufbau: zwei Ketten
2. Umgebungsvorbereitung
3. Modellseite: Modellzentrum und Modellveröffentlichung
4. Anwendungserstellung: der AppAgent-Workflow
5. Entwicklungshinweise zu Modulen, Blocks und Services
6. Erweiterungen: extensions/*.yaml
7. Build und Bereitstellung
8. Schnellübersicht der Qualitäts-Gates
9. Online-Praxis: SaaS-Lernedition

1. Plattformaufbau: zwei Ketten

AliothStudio besteht aus zwei Ketten mit getrennten Zuständigkeiten; das Verständnis ihrer Grenze ist die Voraussetzung für die korrekte Nutzung der Plattform:

Kette	Zuständigkeit	Treiber	Ergebnis
Modellver- öffentli- chungsket- te	strukturelle Evolution des Daten- fundaments (Tabellen/Spalten/Vererbung/Di- mensionscodes)	Meta-Modell- zentrum (ma- nueller Kanal)	Alioth/v* –Veröffentlichungspa- ket (DDL · Seed · maschinenlesba- rer Vertrag · Verifikationsbericht)
Anwen- dungser- stellungskette	Montage von Geschäftsanwen- dungen auf dem eingefrorenen Fundament	AppAgent- Agent + Entwickler	Anwendungskonfiguration, Service- Backend, Modul-Frontend, Deploy/{ns} – Bereitstellungspaket

Rote Linie: Das isahl-Schema ist strukturell eingefroren. Jedes Anliegen der Anwendungserstellungskette nach „Tabelle hinzufügen / Spalte hinzufügen / Spalte ändern“ muss zurück in die Modellveröffentlichungskette (manuelle Entscheidung im Modellzentrum); DDL, Migrationsskripte oder das direkte Schreiben von Metadaten sind als Umgehung verboten.

2. Umgebungsvorbereitung

2.1 Voraussetzungen

Abhängigkeit	Zweck
PostgreSQL 14+	Datenfundament (Modell v10.0 als Tabellenvererbungsstruktur)
mise	einheitlicher Einstieg für Aufgaben und Umgebung (<code>mise run <task></code>)
Rust-Toolchain / pnpm / bun	Backend-Build, Frontend-Build, Skript-Toolchain

2.2 Datenbankinitialisierung und –start

```
# Initialisierung der Entwicklungsdatenbank (Replay aus dem Veröffentlichungs-Snapshot; führt DROP
und Neuaufbau der Zieldatenbank aus, nur für aliothstudio_dev)
bash scripts/db/reset-db.sh --dev

# Meta-Plattform (Modellverwaltung + AppAgent): Backend :4949, Frontend :49494
mise run meta

# Entwicklungsumgebung des Namespace (Gateway + SSO), {ns} ∈ alioth | wz | avic | cosmic | se |
mat
mise run dev:{ns}
```

Die Datenbank ist vierfach isoliert: Produktion `aliothstudio` ← Pre-Prod `aliothstudio_pre` → Entwicklung `aliothstudio_dev` → Test `aliothstudio_test`. Umgebung und `DATABASE_URL` sind zwingend aneinander gebunden; Schema-Abfragen laufen ausnahmslos über `cd Meta/backend && mise run schema-info -- <command>` (die DB ist die einzige Quelle der Wahrheit; das Ableiten aus DDL-Dateien ist verboten).

3. Modellseite: Modellzentrum und Modellveröffentlichung

3.1 Modellzentrum (Meta UI)

Die gesamte strukturelle Evolution erfolgt über das Modellzentrum im Meta-Frontend: Anlegen von Collections (collection, Anhängen an eine Elterntabelle zur Bildung von Vererbung), Registrieren von Feldern (fields), Pflege der Dimensionscodes (Szene/Faktor/Funktion). Die Registrierung von Collections und Feldern wird von der Plattform programmatisch mit physischen Tabellen und Vererbungsanhängen synchronisiert — Entwickler schreiben keinerlei DDL zum Erzeugen von Tabellen.

3.2 Modellveröffentlichung

Nach dem Start einer Veröffentlichung im Modellzentrum exportiert die Pipeline das `isahl`-Schema (`pg_dump --schema-only` + Daten von 268 Seed-Tabellen + die Dimensions-Seeds `seed-dimensions.sql`), verarbeitet es zu einer reinen CREATE/ALTER-Form, schreibt es nach `Alioth/v{version}/` und führt in einer dedizierten Verifikationsdatenbank einen vollständigen Neuaufbau mit Verifikation durch (Struktur-Roundtrip + Seed-Roundtrip + uid-Eindeutigkeit); der Bericht wird als `verify-report.json` abgelegt. Die aktuell neueste Veröffentlichung `v10.0.44`: 1022 `isahl`-Tabellen, 1204 Vererbungskanten, 268 Seed-Tabellen tabellenweise roundtrip-konsistent, Verifikationsstatus `verified`. Die Veröffentlichungspipeline erzeugt 7 Fortschrittsereignisse (`prepare` / `schema` / `isahl_dump` / `seed_dimensions` / `manifest` / `verify` / `done`); die Zeilenebenen-Hierarchie der Collections wird durch die modellseitige Deklaration `row_layer` (`abstract` / `carry`) getragen — hält eine nicht adjudizierte Elterntabelle Zeilen, wird die Veröffentlichung blockiert.

Modellinkarnation und abgeleitete DB-Fläche sind im selben Batch gekoppelt: `refresh-derived-artifacts.ts` ist der einzige Aktualisierungseinstieg für die vier abgeleiteten Artefakte `fk-index`, `relation-index`, `value-domain-columns`, `dk-projection-tables` und ist bereits in `reset-db.sh` eingebunden; der Diff der Artefakte ist die Review-Fläche der Modelltopologie, der Schutz vor Entfernung wird durch das Gate W9 garantiert.

3.3 Dimensionen und Koordinaten

Die Geschäftssemantik wird über dreidimensionale Koordinaten lokalisiert: `dk_scene` (Szene), `dk_factor` (Faktor), `dk_function` (Funktion). Der Dimensionscode wird automatisch aus Kategoriecode + laufendem Symbol zusammengesetzt (etwa `JC` = Systemverwaltung). Koordinaten werden pro Namespace lokal interpretiert; jede API eines Service bindet statisch ein festes Koordinatentripel.

4. Anwendungserstellung: der AppAgent-Workflow

Die Anwendungserstellungskette wird vom AppAgent angetrieben; die Standardwerkzeugfolge lautet:

```
analyze_ontology    # Anforderung/Prototyp → OntologyModel-IR (mit harter Prüfung der
Vollständigkeit der fünf Domänen)
build_module        # flow-plan → module.json (Moduldeklaration)
build_block         # flow-plan → block.json (Deklaration des Fähigkeitsblocks)
transfer_ontology   # Ontologieausrichtung → alignment-graph.json + mapped-entities.json
generate_service_api # Ontologieentität → Service-Backend (Rust, statische Koordinatenbindung)
compose_app         # Modulmontage → app.json + extensions/*.yaml + Prototyp-Artefakte
```

Ablage der Artefakte: `Pre-Proc/{ns}/Apps/{app}/` (`app.json`, `extensions`, Prozessartefakte) und `Pre-Proc/{ns}/Sources/Apps/{Modules,Blocks,Services}/` (Deklarationen und Code für Module/Blocks/Services). Die Eintragsform der Ontologie-Abbildungsartefakte (`ontology-output.json`) unterliegt der Prüfung durch das Vertrags-Gate; Koordinaten müssen echte Codes aus der DB sein.

5. Entwicklungshinweise zu Modulen, Blocks und Services

Ebene	Zentrale Vorgaben
Module	Kombinationscontainer für Scene/Factor; die Pflichtfelder in <code>module.json</code> müssen vollständig sein; <code>DashboardPage</code> ist verboten (direkter Sprung von der Wurzelroute auf die Entitätslistenseite)
Block	Fähigkeitseinheit; die von <code>factors</code> / <code>services</code> referenzierten IDs müssen existieren (verwaiste Verweise haben ein eigenes Gate)
Service – Backend	Standardskelett <code>handlers/models/repositories/services</code> ; statische Bindung des Koordinatentripels (gleiche Tabelle mit unterschiedlichen Koordinaten wird in getrennte APIs aufgeteilt); <code>auth-Middleware</code> ist verboten (das Gateway übernimmt NGAC zentral); das <code>tracing</code> –Crate ist innerhalb von Modulen verboten
Frontend	Zustandsverwaltung ausnahmslos mit Jotai v2; Messfelder im Format „Zahlenwert + Einheitenwahl“; Abgleich mit dem echten Backend, Mocking ist verboten; der Datenzugriff läuft einheitlich über den Ressourcen-Transport von <code>@alioth/api</code> (<code>list/get/mutate</code> , die Envelope wird per <code>items()</code> / <code>unwrap()</code> entpackt); der Referenz-Selektor muss stets den <code>ReferenceSelectDescriptor</code> verwenden, das Extrapolieren des Vertrags aus dem Endpunkt-Typ ist untersagt

5.1 Eiserne Regeln für das Schreiben von Daten

- **INSERT in Blatt-Tabellen:** Das Schreibziel muss eine Blatt-Tabelle der Vererbungs-Halbordnung sein (das Gate misst real über `pg_inherits`; statisches Erraten von Namen wird nicht akzeptiert)
- **Zustand über Relationstabellen:** Entitätszustand/Kategorie/Tag werden in `zc_id_lifecycle_r_*` –Relationstabellen geschrieben; selbst angelegte Statusspalten an der Entität sind verboten (`flag` wird nur von Zeilen des Statuswörterbuchs verwendet)
- **Schreibvertrag für Kategorien:** Schreibvorgänge in lifecycle-Untertabellen müssen ein `dk_function` aus einem der sechs Präfixe angeben (`_f_` / `_t_` werden automatisch abgeleitet; handschriftliche Literale sind verboten)
- **Skalare Verweise:** `qk_*` –Spalten sind Zeiger in den Wertraum (`bigint`); Arithmetik darauf ist verboten, gelesen wird über die Auflösung mit `_refs`
- **Zeitsemantik:** künftige Ausschnitte gehören zur `task`–Familie, vergangene zur `event`–Familie; die Abschlusshandlung fällt in den Ereignisausschnitt (Umklappen des Ausschnitts)

6. Erweiterungen: extensions/*.yaml

Erweiterungen auf Anwendungsebene (`constraints` / `rules` / `statemachines` / `workflows`) liegen unter `Pre-Proc/{ns}/Apps/{app}/extensions/` und werden beim Start des Gateway geladen; sie sind die einzige wirksame Steuerungsfläche für Erweiterungen — Validierungen, Regeln und Abläufe lassen sich anpassen, ohne Code neu zu generieren.

7. Build und Bereitstellung

```
# Einziger Build-Einstieg (Feature-Gating + Target-Isolation + automatische Signaturbehandlung)
bash scripts/build-ns.sh <NAMESPACE> release    # oder dev (Debug-Modus)

# Veröffentlichung (Synchronisation von Apps / seed / .env / apps.json; kein eigener Build)
bash scripts/deploy/release-to-namespace.sh <NAMESPACE>
```

Artefakte: `Deploy/{ns}/bin/{ns}-server` (Namespaces mit aktiviertem OpenActivity haben zusätzlich `Deploy/{ns}/Open/bin/openactivity-server`). Bei einer Veröffentlichung in die Produktionsstufe wird das Gateway automatisch neu gestartet. Anwendungen außerhalb eines Namespace (Meta / SSO) laufen über `scripts/meta|sso/deploy.sh` und landen in `Release/`.

8. Schnellübersicht der Qualitäts-Gates

Das Repository erzwingt Qualität über ein „Substrat zur Durchsetzung von Spezifikationen“: Jede prüfbare Spezifikation hat ein Prüfskript + eine Baseline + eine Gate-Registrierung (229 registrierte Gates: 137 in der Commit-Phase / 92 in der Push-Phase). Die Spezifikationsbibliothek enthält 403 lebende Capability-Spezifikationen; jede MUST/NEVER-Klausel muss ihren Ausführungsort (Gate-ID / Skript / Testpfad) über das strukturierte Feld „`执行面`“ (Ausführungsoberfläche) deklarieren; Prosa-Deklarationen werden nicht mehr durchgelassen. Häufig berührte Einträge:

Gate	Abgefangenes Objekt	Stufe
<code>check-isahl-ddl-boundary</code>	jedes umgehende Schreiben in die isahl-Struktur	commit
<code>check-leaf-insert</code>	INSERT ohne Blatt-Tabelle / handschriftliche <code>_f_/_t_</code> -Literale	commit
<code>check-code-semantics</code>	Schreiben von Typ-/Enum-Semantik in die code-Spalte	commit
<code>check-ontology-contract</code>	Form der Ontologie-Abbildungsartefakte / gefälschte Koordinaten	commit
<code>check-bridge-host-domain</code>	falsch platzierte Host-Domäne am linken Ende der Relationstabelle	commit
<code>formal-proofs</code>	fehlgeschlagener Lean-Theorem-Build oder Beweislücken	push
<code>check-all</code> (T1/T2)	Versionsabgleich / OpenSpec-Konformität / i18n / Service-DAG u. a.	push

Bei einer Blockade nicht umgehen: Arbeite die vom Gate ausgegebene Reparaturfläche ab; bestehende Verstöße werden über die Baseline bereinigt (nur abbauen, nie aufbauen), neue Verstöße müssen behoben werden.

9. Online-Praxis: SaaS-Lernedition

Die vollständige Plattform lässt sich auch ohne lokale Umgebung erproben: <https://cosmic-tools.shop/> bietet eine Online-SaaS-Lernedition von AliothStudio — eine verschlankte Bereitstellung für einzelne Nutzer (Kernmodellierung, Anwendungsmontage und Laufzeitlinie sind vollständig nutzbar; die unternehmensweite Orchestrierung mehrerer Namespaces und die Veröffentlichungspipeline entfallen). Empfohlener Weg: registrieren und einsteigen → die Kapitel 2–7 dieses Handbuchs der Reihe nach praktisch durchlaufen → die Artefakte mit der Gate-Liste aus Kapitel 8 selbst prüfen.

Begleitliteratur: „Whitepaper zur Alioth-Modelltheorie“ — zum Verständnis der mathematischen Struktur hinter allen Vorgaben dieses Handbuchs.

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