

# SpecVerse Architecture Guide

For the user/how-to guide, see [SPECVERSE-USER-GUIDE.md](#).

## Overview

SpecVerse is a specification language ecosystem. You write `.specly` files that describe WHAT a system does. The engines figure out HOW to implement it.

```
.specly file (WHAT)
|
v
[Parser Engine] --> SpecVerseAST
|
v
[Inference Engine] --> Controllers, Services, Events, Views, Deployments
|
v
[Realize Engine] --> Generated Code (Prisma, Fastify, React, CLI, etc.)
```

## Repository Structure

specverse-lang/	-- Orchestrator: CLI, build scripts, templates
specverse-engines/	-- Engine packages (independent, discoverable)
packages/	
types/	-- Shared types (AST, engine interfaces)
entities/	-- Entity module system + EngineRegistry
parser/	-- Parse .specly --> AST
inference/	-- Models --> full architecture
realize/	-- Spec + manifest --> generated code
generators/	-- Diagrams, UML, documentation
specverse-self/	-- Self-spec realization (dogfood)

## The Three Layers

### Layer 1: Specification (what users write)

A `.specly` file has three sections:

```
components:      # WHAT the system does
  MyApp:
    models: ...
    controllers: ...
    services: ...
    events: ...
    views: ...
    commands: ...

deployments:     # WHERE it runs
```

```

development:
  instances: ...

manifests:      # HOW it's built (technology choices)
  implementation:
    capabilityMappings: ...

```

## Layer 2: Entity Modules (extensible building blocks)

Every concept in a `.specly` file (models, controllers, services, events, views, commands, etc.) is an **entity type**. Each entity type is defined by an **entity module** with up to 8 facets:

```

entity-module/
  module.yaml      -- Manifest: name, version, dependencies, facet paths
  schema/          -- JSON Schema: what's valid in .specly syntax
  conventions/      -- Convention processor: shorthand expansion
  inference/        -- Inference rules: what to generate from this entity
  generators/       -- Code generators: instance factory templates
  behaviour/        -- Quint specs: formal invariants and rules
    conventions/    -- Behavioural conventions: human-readable -> Quint
  docs/            -- Documentation references
  tests/           -- Test references

```

**Core entities** ship with SpecVerse: models, controllers, services, events, views, deployments.

**Extension entities** are added via packages: commands, conventions, measures.

## Layer 3: Engines (independent, discoverable processors)

Each engine is an npm package implementing the `SpecVerseEngine` interface:

```

interface SpecVerseEngine {
  name: string;           // 'parser', 'inference', 'realize', etc.
  version: string;
  capabilities: string[]; // ['parse', 'validate', 'import-resolution']
  initialize(config?: any): Promise<void>;
  getInfo(): EngineInfo;
}

```

Engines are discovered at runtime by `EngineRegistry`:

```

const registry = new EngineRegistry();
await registry.discover(); // finds @specverse/engine-* packages

const parser = registry.getEngineForCapability('parse');
await parser.initialize({ schema });
const result = parser.parseContent(content, filename);

```

## How the Pipeline Works

### Step 1: Parse

The parser engine takes `.specly` content and produces a `SpecVerseAST`.

Raw YAML

```
--> YAML parser (js-yaml)
--> JSON Schema validation (pre-processing)
--> Convention processor (entity modules expand shorthand)
--> JSON Schema validation (post-processing)
--> Semantic validation (cross-entity checks)
--> SpecVerseAST
```

Key: The convention processor discovers entity types from the **entity registry**. Each entity module provides a convention processor that knows how to expand its shorthand syntax. When you add a new entity type, the parser automatically processes it.

## Step 2: Infer

The inference engine takes models from the AST and generates full architecture.

Models (from AST)

```
--> Rule loader (loads JSON rules from entity modules)
--> For each registered generator:
-->   Pattern match models against rules
-->   Generate controllers/services/events/views
--> Deployment generator (instances, channels)
--> ComprehensiveInferenceResult
```

Key: Generators are registered in a `Map<string, generator>`. Each generator loads rules from its entity module. Adding a new generator means adding it to the map and providing rules.

## Step 3: Realize

The realize engine takes an inferred spec + manifest and generates code.

AI-optimized spec + Manifest

```
--> Instance factory library (loads factory YAMLs)
--> Capability resolver (maps capabilities to factories)
--> For each capability:
-->   Resolve factory
-->   Load template generator (TypeScript file)
-->   Execute generator with context
-->   Write output file
--> Generated code (Prisma, Fastify, React, etc.)
```

Key: Instance factories are YAML files that declare what they generate. Template generators are TypeScript files that produce code. The manifest controls which factories are used. Adding a new technology means adding a new instance factory.

## How Things Connect

---

```

Entity Module (e.g., "models")
|
|--> schema/models.schema.json
|     Used by: Parser (JSON Schema validation)
|
|--> conventions/model-processor.ts
|     Used by: Parser (convention expansion)
|
|--> inference/v3.1-controller-rules.json
|     Used by: Inference Engine (rule matching)
|
|--> generators/index.ts
|     Used by: Realize Engine (code generation metadata)
|
|--> behaviour/invariants.qnt
|     Used by: Quint (formal verification)
|
|--> diagramPlugins: [{ type: 'er' }, { type: 'class' }]
|     Used by: Generators Engine (diagram generation)
|
|--> docs/index.ts
|     Used by: Documentation system
|
|--> tests/index.ts
|     Used by: Test infrastructure

```

Each facet connects to exactly one engine. The entity module is the unit of extensibility -- add a module, and all engines discover it.

## Key Design Decisions

### Double Validation

The parser validates twice: once on the raw YAML (before convention processing) and once on the expanded YAML (after convention processing). Both must pass. This means the JSON Schema must accept both the shorthand form and the expanded form for core entity types.

Extension entity types (like commands) are only included in the processed output when present in the source YAML -- they don't get empty defaults. This avoids schema type mismatches where the processed form (array) differs from the schema form (object).

### Entity Type Discovery

The convention processor derives its entity type list from the entity registry, filtered to types that have completed schema integration. The filter set is:

```

const COMPONENT_ENTITY_TYPES = new Set([
  'models', 'controllers', 'services', 'views', 'events', 'commands'
]);

```

When a new entity type completes schema integration (added to root.schema.json AND ComponentSpec type), it's added to this set.

## Engine Discovery

The EngineRegistry discovers engines from a known list of package names (parser, inference, realize, generators, ai). It tries to import each one, and registers any that are installed. Additional engines can be added via the `additionalEngines` constructor option. Each engine package exports a singleton implementing `SpecVerseEngine`:

```
// @specverse/engine-parser/src/index.ts
export const engine = new SpecVerseParserEngine();
export default engine;
```

The registry looks for `mod.default` | `mod.engine` | `mod.createEngine?.()`.

## Circular Dependency Prevention

Parser and entities had a circular dependency (parser imported entity types, entities imported parser processors). This was broken by moving shared interfaces (`ProcessorContext`, `AbstractProcessor`, `EntityModule`) to `@specverse/types`. Both packages import from types, neither imports from the other at compile time.

## Instance Factory Architecture

Instance factories are YAML files that declare:

- What capabilities they provide (`orm.schema`, `api.rest`, etc.)
- What code templates they generate
- What dependencies they require

The manifest maps capabilities to factories. The resolver finds the right factory for each capability. The code generator executes the factory's templates.

## Self-Hosting

SpecVerse is self-hosting: the self-spec (921 lines, 5 components, 9 CLI commands) generates a CLI that produces byte-for-byte identical output to the hand-written CLI. The generated CLI discovers engines via EngineRegistry and calls them through the standard interface.

Hand-written CLI: 4,407 lines (monolithic). Generated CLI: 410 lines (modular, engine-discovered). Same output for validate, infer, realize, and gen commands.

## Current Implementation

---

### Repositories

Repo	Purpose	Branch
specverse-lang	Orchestrator: CLI, build scripts, entity modules, templates	main
specverse-engines	7 engine packages (types, entities, parser, inference, realize, generators, ai)	main
specverse-self	Self-spec realization (generated CLI + full-stack app)	main

## What's Wired

- **Entity modules** → **Parser**: convention processors discovered from registry at parse time
- **Entity modules** → **Schema**: 14 fragments composed into SPECVERSE-SCHEMA.json at build time
- **Entity modules** → **Inference**: rules loaded from entity modules via registry
- **Entity modules** → **Diagrams**: plugins discovered from entity module declarations
- **Entity modules** → **Quint**: 25 .qnt files, 21 invariants verified via Apache, 52 behavioural convention grammars
- **Entity modules** → **L3 Behaviors**: 15 convention patterns + Quint action transpilation + stubs; 117 Quint guards transpiled to TypeScript runtime functions and running in app-demo
- **Controller engine** → **app-demo**: preconditions → steps → postconditions → events; custom operations execute full behavior pipeline
- **Engine packages** → **CLI**: generated CLI discovers engines via EngineRegistry, 9 commands engine-wired

## What's NOT Yet Wired

- Domain-specific entity types (no real domain has added a custom entity yet)
- Meta-specification registry (@specverse/entity-\* npm distribution)
- AI engine (adapter exists but EcosystemPromptManager not fully configured)

See `docs/CURRENT-STATE-AND-PLAN.md` for the full assessment and implementation plan.